Model Checking and Parity Games – Lectures

Tom van Dijk
University of Twente, Quartile 4, 2025-2026

Acknowledgements

These slides were to some extent inspired by earlier lecture slides
by Tim Willemse, Wolfgang Schreiner and Nathanael Fijalkow,
Joost-Pieter Katoen, and Javier Esparza

Introduction and motivation

Why do model checking?

Why model checking

- Modern systems are concurrent, distributed, and highly configurable
- They are ubiquitous and society relies on them
  - embedded systems and controllers
  - communication and security infrastructure
  - databases, cloud services, finance, healthcare, transportation
- The most costly bugs are often rare behaviours that appear only under special timings or failures

Examples of difficult bugs

- Deadlocks and livelocks
- Rare mutual exclusion failures
- Lost updates or inconsistent replicas
- Protocol violations due to unexpected ordering
- Liveness failures

Why do model checking?

Software reliability: Therac-25

- Radiation therapy machine for cancer patients
- Several patients received massive overdoses (1985–1987)
- Problem: unsafe concurrent behaviour and a subtle race condition in the control software

Software reliability: Ariane 5 Flight 501

- ESA launch vehicle
- Destroyed on its first flight (4 June 1996)
- Both inertial reference systems failed
- Problem: an unchecked numeric overflow in reused software

Why do model checking?

Hardware reliability: Pentium FDIV bug

- Rare wrong results in floating-point division
- Problem: a missing entry in a lookup table used by the division hardware
- Cost: $475 million to recall the affected processors
- Even very rare hardware errors can have large financial and practical consequences
- Verification matters for hardware too, not only for software

Why do model checking in the age of LLMs?

Why this matters even more now

- LLMs make it easy to generate and refactor large systems quickly
- Many defects are not simple syntax or typing mistakes
- Correctness often depends on global behaviour across many executions and components
- This increases the need for precise specifications and automated verification

Especially difficult are properties involving

- concurrency and unexpected interleavings
- protocol state machines and message ordering
- retries, timeouts, and crash recovery
- safety: something bad should never happen
- liveness: something good should eventually happen

Why do model checking?

Success stories across the wider field

- **Microsoft SLAM / Static Driver Verifier:** systematic checking of Windows driver API compliance
- **NASA Deep Space 1: Spin** model checking was applied to mission software and known defects could be reproduced mechanically
- **AWS automated reasoning:** cloud security analyses to check access-control and reachability
- **MongoDB: TLA+** and **TLC** for distributed transactions
- **OpenTitan:** formal property verification in open-source security hardware
- **CERN**: use formal modeling techniques to discover ambiguities in the programming language for the control software

Takeaway

- Formal methods are not only for post-mortems of old disasters
- They are active engineering tools in current large-scale systems

A Dutch perspective

Errors leading to major disruption

- **Rijkswaterstaat:** repeated Haringvlietbrug outages were traced to a software issue in the control of the two main motors
- **ProRail:** a rollback after a failover exercise caused a highest-hindrance disruption at Delft due to a network configuration error

Successful application

- **Rijkswaterstaat:** supervisory control synthesis is used for infrastructure such as water locks and bridges; ESCET/CIF material uses a Westsluis lock case, and a field experiment at Linne used generated PLC code
- **Rijkswaterstaat / research partners:** formal modelling and analysis are also used for tunnels and storm-surge barriers, e.g. with mCRL2 and synthesis-based engineering
- **ProRail:** model-based testing and formal analysis have been applied to railway interlockings and interface standards

Why do model checking now?

Model checking in current large-scale systems

- **MongoDB:** compositional **TLA+** models for distributed transactions; **TLC** is used to check isolation guarantees on finite instances
- **Microsoft CCF / Azure Confidential Ledger: TLA+**, model checking, and CI-integrated validation found subtle protocol and consistency bugs before production
- **AWS: TLA+** is still actively used to explore distributed and concurrent protocol designs in cloud systems

Why this has societal impact

- Correct cloud and database protocols protect data, services, and transactions used by millions of people
- These are exactly the kinds of systems where rare interleavings and failure scenarios matter

What techniques are used successfully in practice?

A wider view of the field

- **State-machine specification + model checking**
  - e.g. TLA+, TLC, temporal properties, counterexamples
- **Model-based testing and conformance checking**
  - deriving tests or validating traces against a formal model
- **SMT-based automated reasoning**
  - e.g. security policies, reachability, configuration analysis
- **Formal hardware verification**
  - e.g. assertion-based property checking

In this course

We focus mainly on models, temporal properties, model checking, and counterexamples.

Why do model checking?

Model checking vs testing

- Testing explores selected executions
- Model checking explores all behaviors of a finite model with respect to a property
- This makes it strong at finding rare concurrency and protocol bugs (the most expensive ones)
- It also gives a counterexample: an execution showing how the property fails
- Model checking can also support test generation and conformance checking

Overview

Model Checking

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for that model.

Model Checking Process

- Create a model of the system
- Formalize requirements into a property specification
- Run the model checker
  - property satisfied? done
  - property violated? inspect counterexample and fix bug
  - out of memory? reduce or abstract the model

Transition systems

Transition Systems

The behaviour of a system is modelled by a directed graph consisting of:

- nodes, representing states of the system
  (e.g. the current value of a program counter, variables, registers, etc.)
- edges, representing transitions (state changes) of the system
  (e.g. events, input/output actions, internal computations)

Information may be attached to states, transitions, or both:

- Kripke Structures (KS)
  Information on states, called atomic propositions
- Labelled Transition Systems (LTS)
  Information on edges, called action labels

Transition Systems

Transition system $\mathcal{M}$ is a tuple $\langle S, S_0, \mathcal{A}ct, R, AP, L \rangle$:

- S is a set of states
- S_0 is a set of initial states (or s_0 is a single initial state)
- $\mathcal{A}ct$ is a set of action labels
- R is a transition relation $R \subseteq S \times \mathcal{A}ct \times S$
- AP is a set of atomic propositions
- L is a labelling of the states: $L \in S \to 2^{AP}$

Notation: $s \xrightarrow{a} t$ denotes $(s, a, t) \in R$

Special cases:

- Kripke Structures: transitions are unlabelled, or all transitions share one dummy label
- Labelled Transition Systems: AP is empty

Transition Systems

Given some TS $\mathcal{M} := \langle S, S_0, \mathcal{A}ct, R, AP, L \rangle$

A terminating state is a state $s \in S$ that has no successors

A path is a sequence of states $s_0 s_1 s_2 \ldots$ where $s_0 \in S_0$ and for each $i \geq 1$, $s_{i-1} \rightarrow s_i$. A path is either infinite or ends in a terminating state.

A trace is like a path, but records only the state labels: $L(s_0)\ L(s_1)\ L(s_2)\ \ldots$

The reachable states **explore**$(\mathcal{M})$ are all the states s in paths of $\mathcal{M}$

Exploring a state space

- Often the first step of model checking: compute **explore**($\mathcal{M}$)
- Also called state space exploration or generation
- Maintain a set of visited states V and a queue Q
- Exploration can be depth-first or breadth-first; BFS finds shortest paths in number of transitions
- Can check for deadlocks and invariants on-the-fly

```
1 def explore(M):
2     V ← S_0                                          // visited states
3     Q ← S_0                                          // states still to explore
4     while Q ≠ ∅ :                                    // while there are states to explore
5         s ← pick(Q)                                  // take a state from the frontier
6         T ← get-successors(s) \ V                    // new successors only
7         Q ← (Q ∪ T) \ {s}                            // update frontier
8         V ← V ∪ T                                    // mark as visited
9     return V                                         // return the set of reachable states
```

State space exploration in practice

A generic explorer only needs a next-state interface

- `initialStates() : Set<State>`
- `next(s) : Set<State>` or `Set<(Action, State)>`

Why this is useful

- The model checker does not need to know what kind of system it explores
- Only the representation of states and the successor function are model-specific
- The same BFS/DFS code can then explore many different formalisms, detect deadlocks, etc.
- For property checking, we additionally need access to labels and/or state predicates

Breadth-first versus depth-first exploration

Exploration order

- **BFS:** use a queue, explore states layer by layer
- **DFS:** use a stack or recursion, explore one branch deeply before backtracking
- Both can compute **explore**$(\mathcal{M})$

Breadth-first versus depth-first exploration

Exploration order

- **BFS:** use a queue, explore states layer by layer
- **DFS:** use a stack or recursion, explore one branch deeply before backtracking
- Both can compute **explore**$(\mathcal{M})$

Why does the choice matter?

- **BFS:** useful when we want a shortest counterexample
- **DFS:** often natural for cycle detection and nested search algorithms
- **DFS** is often simpler and may use less memory for the frontier
- The chosen order affects which counterexample is found first
- In practice, the best choice depends on the property and the tool

State space explosion

Transition systems can be huge

- A state with n bits already has up to 2^n possible valuations
- Concurrency often multiplies the number of possible executions
- This rapid growth is called the state space explosion

Model checking can also be algorithmically expensive

More expressive logics often come with more expensive verification algorithms

State space explosion

Why state spaces become enormous

- A state with n bits already has up to 2^n possible valuations
- Concurrency multiplies behaviours through many possible interleavings
- Inputs, data values, buffers, and queues add further combinations

Main strategies

- Reduce the number of states or transitions
- Represent large sets of states more compactly
- Explore only the part that is relevant for the property

Techniques to reduce the explored state space

Reduce what must be explored

- **Abstraction:** remove irrelevant details from the model
  - Example: keep whether a lock is free/busy, ignore exact timestamps
- **CEGAR:** start with a coarse abstraction, and refine it only when a counterexample is spurious
  - Counterexample-guided abstraction refinement
- **Equivalence reduction:** merge states that are behaviourally indistinguishable
  - Example: bisimulation minimisation
- **Partial-order reduction:** avoid exploring interleavings that differ only in the order of independent actions
  - Example: if process P and Q update unrelated variables, PQ and QP may be equivalent

Techniques to handle large state spaces efficiently

Make exploration cheaper

- **On-the-fly model checking:** generate states only while checking the property
  - May stop early as soon as a counterexample is found
- **Symbolic model checking:** represent sets of states and transitions compactly
  - Example: BDDs or SAT/SMT-style encodings
- **Smart data structures:** store states compactly and compare them efficiently
  - Example: hash tables, bit-state hashing, decision diagrams

Important intuition

- Some techniques reduce the model
- Some reduce the number of executions
- Some reduce the memory needed to store them

Petri nets

Petri nets

- Invented by Carl Adam Petri in his 1962 PhD thesis
- A Petri net is a bipartite directed graph of places and transitions
  - also called a place/transition net (P/T net)
- Each place can contain tokens
- A marking is the current number of tokens in each place
- An initial marking M_0 defines the initial state
- In an ordinary P/T net, a transition is enabled if each input place contains at least one token
- Firing removes one token from each input place and adds one token to each output place
- If multiple transitions are enabled, the next firing step is chosen nondeterministically
- Petri nets naturally model concurrency, synchronization, resource sharing, and nondeterministic choice

Petri nets

Markings and firing

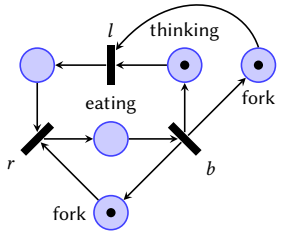
- A marking is a function $M : P \to \mathbb{N}$
- A transition t is enabled in marking M if every input place of t contains enough tokens
- If t fires, the marking changes by removing tokens from the input places of t and adding tokens to the output places of t
- If several transitions are enabled, any of them may fire: this is a source of nondeterminism

Key idea

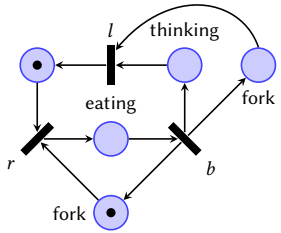
A Petri net together with an initial marking M_0 defines a transition system:

- states = markings
- transitions = firing steps

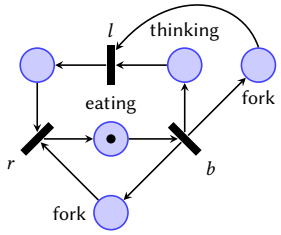
Example: Dining philosophers



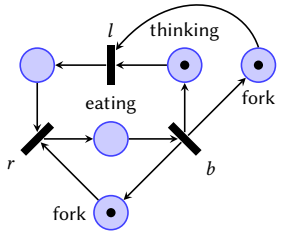
Example: Dining philosophers



Example: Dining philosophers

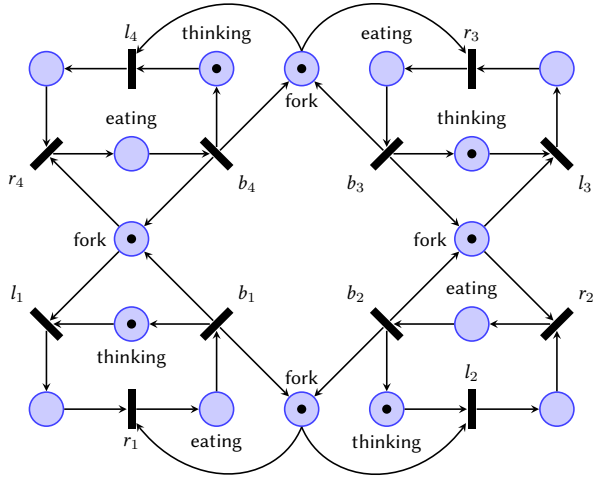


Example: Dining philosophers

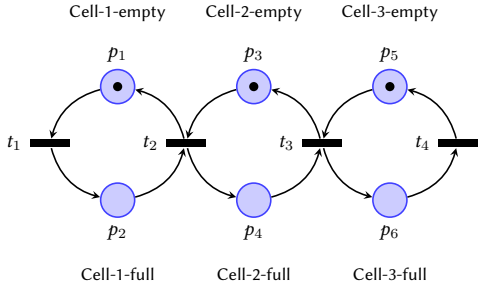


Petri nets

Example: Dining philosophers

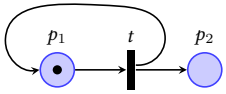


Petri nets



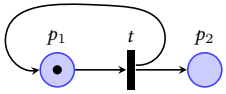
- Simple buffer of capacity $n = 3$
- No cell is simultaneously empty and full
- The net is safe (hence also 1-bounded)
- No deadlocks (each marking has at least one successor marking)
- The reachable markings form a finite state space
- Every reachable marking is reachable again
- A small model checker can explore all reachable markings

Example



Q: what is the set of reachable markings?

Example



Reachable markings:

$$(1, 0), (1, 1), (1, 2), (1, 3), (1, 4), \ldots$$

So there are infinitely many reachable markings: the net is unbounded.

Petri nets

- A Petri net is a tuple $\langle P, T, F \rangle$ where:
  - P and T are disjoint finite sets of places and transitions
  - $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation
- A marking is a function $M : P \to \mathbb{N}$
- Petri nets model concurrency, synchronization, and resource sharing
- A Petri net is k-bounded if no reachable marking puts more than k tokens in any place
- A Petri net is safe if it is 1-bounded
- A Petri net is conservative if the total number of tokens remains constant

Typical verification questions

- starting from an initial marking M_0...
- Termination: are all runs finite?
- Nontermination: does there exist an infinite run?
- Reachability: is a given marking M reachable?
- Coverability: can we reach some marking $M' \geq M$?
- Fireability: is a given transition t enabled in some reachable marking?

Petri nets in practice

- Petri nets are widely used benchmark models in the Model Checking Contest (MCC)
- Standard interchange format: PNML
- The Canvas kit includes optional typed starter parsers for Python, Haskell, and Rust
- In Python, libraries such as `pntools` and `pm4py` can help parse or manipulate nets
- For this course, the important part is not the tool itself, but the modelling and the reachable markings

Petri nets and state space exploration

A Petri net defines a next-state interface

- A state is a marking $M : P \to \mathbb{N}$
- The initial state is the initial marking M_0
- The successors of M are all markings obtained by firing one enabled transition

What do we get for free?

- reachability analysis
- counting reachable markings
- deadlock detection
- shortest deadlock traces with BFS

Temporal properties

What do we want to express about behaviour?

Typical verification questions

Given a transition system, we may want to ask:

- Can the system reach an error state?
- Is a deadlock possible?
- Will every request eventually get a response?
- Is there some execution that eventually reaches a success state?
- Is it always possible to return to a safe state?

Observation

These are all temporal properties: they talk about behaviour over time, not just about one isolated state.

Different kinds of behaviour properties

Properties can talk about different things

- **State properties:** predicates on the current state
- **Trace properties:** properties of one complete execution
- **System properties:** properties of the whole transition system

Examples

- State property: the current state is an error state
- Trace property: every request is eventually followed by a response
- System property: from every reachable state, reset remains possible

One transition system, two temporal views

The same system can be viewed in two ways

- as a set of maximal runs / traces
  - each run is one possible complete behaviour
- as an unwound computation tree
  - each node represents a state together with its possible futures

Why this matters

Some properties talk about individual runs, while others depend on how the future branches.

See also: Moshe Vardi, *Branching vs. Linear Time: Final Showdown* (2001).

Computation tree versus system runs

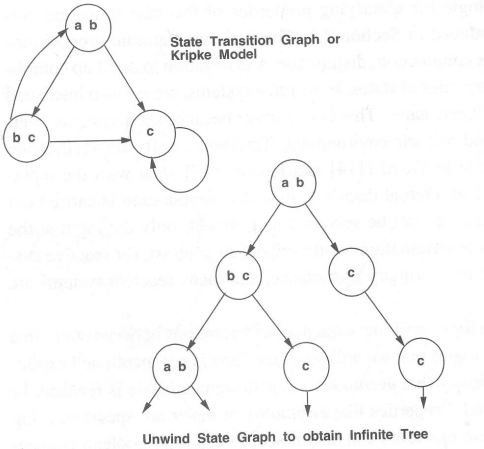


Figure 3.1
Computation trees.

Edmund Clarke et al., *Model Checking*, 1999.

The same transition system can be viewed as:

- a set of maximal runs
- an unwound computation tree

A trace property looks at one run at a time.

A branching property may depend on which futures are possible from a state.

The linear-time viewpoint

Linear-time viewpoint

A linear-time property is a property of a single trace.

How does a system satisfy such a property?

If E is a set of traces, then we can ask:

- **Universal reading:** do all traces of the system belong to E?
- **Existential reading:** does some trace of the system belong to E?

The existential reading is the dual of the universal reading on the complement.

Typical linear-time questions

- mutual exclusion is never violated
- every request eventually gets a response
- process i enters its critical section infinitely often

Branching properties

Some system properties are about the space of futures

These are not just about whether some or all complete runs satisfy a trace property.

Examples

- From every reachable state, reset remains possible
- Whenever a request is pending, both grant and deny remain possible futures
- Every run eventually reaches a state from which safety is guaranteed forever

Key idea

These depend on the branching structure of the transition system.

Linear-time properties

Linear-time (LT) properties

Given a transition system $\mathcal{M} = \langle S, S_0, \mathcal{A}ct, R, AP, L \rangle$:

- A trace is an infinite word over the alphabet $\Sigma = 2^{AP}$
- A linear-time property over AP is a subset of $(2^{AP})^{\omega}$
- So an LT property is a language of infinite traces

Linear-time properties

Two fundamental classes of LT properties

- Safety: “nothing bad ever happens”
- Liveness: “something good eventually happens”

Important intuition

- Safety violations can be detected after a finite bad prefix
- Liveness violations can never be ruled out after finitely many steps

Safety properties

Safety properties

An LT property E is a safety property if for every word $\sigma \notin E$ there exists a finite prefix $\rho \prec \sigma$ such that no infinite extension of ρ belongs to E.

Intuition: once the property is violated, it can never be repaired later.

Examples

- Mutual exclusion
- Whenever y happens, x has happened before
- Two neighbouring philosophers never eat at the same time
- All invariants are safety properties

An invariant is a propositional formula ϕ over AP such that $L(s) \models \phi$ for every reachable state s.

Liveness properties

Liveness properties

An LT property E is a liveness property if every finite word over 2^{AP} can be extended to an infinite word in E.

Equivalently: no finite prefix is enough to prove that the property has been violated.

Examples

- Each process eventually enters its critical section
- Each process enters its critical section infinitely often
- Whenever x happens, eventually y happens afterwards

Which property type?

Determine the type of each LT property: **invariant** or **safety** or **liveness**

- Each philosopher thinks infinitely often.

Which property type?

Determine the type of each LT property: **invariant** or **safety** or **liveness**

- Each philosopher thinks infinitely often.

liveness

Which property type?

Determine the type of each LT property: **invariant** or **safety** or **liveness**

- Each philosopher thinks infinitely often.

liveness

- Two philosophers next to each other never eat at the same time.

Which property type?

Determine the type of each LT property: **invariant** or **safety** or **liveness**

- Each philosopher thinks infinitely often.

liveness

- Two philosophers next to each other never eat at the same time.

invariant

Which property type?

Determine the type of each LT property: **invariant** or **safety** or **liveness**

- Each philosopher thinks infinitely often.

  liveness

- Two philosophers next to each other never eat at the same time.

  invariant

- Whenever a philosopher eats then he has been thinking at some time before.

Which property type?

Determine the type of each LT property: **invariant** or **safety** or **liveness**

- Each philosopher thinks infinitely often.

  liveness

- Two philosophers next to each other never eat at the same time.

  invariant

- Whenever a philosopher eats then he has been thinking at some time before.

  safety

Which property type?

Determine the type of each LT property: **invariant** or **safety** or **liveness**

- Each philosopher thinks infinitely often.

  liveness

- Two philosophers next to each other never eat at the same time.

  invariant

- Whenever a philosopher eats then he has been thinking at some time before.

  safety

- Whenever a philosopher eats then he will think some time afterwards.

Which property type?

Determine the type of each LT property: **invariant** or **safety** or **liveness**

- Each philosopher thinks infinitely often.

  liveness

- Two philosophers next to each other never eat at the same time.

  invariant

- Whenever a philosopher eats then he has been thinking at some time before.

  safety

- Whenever a philosopher eats then he will think some time afterwards.

  liveness

Which property type?

Determine the type of each LT property: **invariant** or **safety** or **liveness**

- Each philosopher thinks infinitely often.

  liveness

- Two philosophers next to each other never eat at the same time.

  invariant

- Whenever a philosopher eats then he has been thinking at some time before.

  safety

- Whenever a philosopher eats then he will think some time afterwards.

  liveness

- Between two eating phases of philosopher i lies at least one eating phase of philosopher $i + 1$.

Which property type?

Determine the type of each LT property: **invariant** or **safety** or **liveness**

- Each philosopher thinks infinitely often.

  liveness

- Two philosophers next to each other never eat at the same time.

  invariant

- Whenever a philosopher eats then he has been thinking at some time before.

  safety

- Whenever a philosopher eats then he will think some time afterwards.

  liveness

- Between two eating phases of philosopher i lies at least one eating phase of philosopher $i + 1$.

  safety

Decomposition theorem

Decomposition of linear-time properties

For every LT property E, there exist a safety property *SAFE* and a liveness property *LIFE* such that

$$E = \mathit{SAFE} \cap \mathit{LIFE}.$$

Proof idea

- Let $\mathit{SAFE} := \mathit{closure}(E)$
  - this keeps exactly the finite prefixes that are still compatible with E
- Let
$$\mathit{LIFE} := E \cup \big((2^{AP})^\omega \setminus \mathit{closure}(E)\big)$$
- Then *SAFE* is a safety property, *LIFE* is a liveness property, and together they reconstruct E

Decomposition theorem: example

Property with both a safety and a liveness part

- Original property:
  "A request is eventually granted, and until then no error occurs."
- **Safety part:**
  "Before the grant happens, no error occurs."
  A bad finite prefix already shows a violation.
- **Liveness part:**
  "Eventually the grant happens."
  No finite prefix can rule this out.
- Together: the original property is the conjunction of these two parts.

Temporal logics

Temporal operators on paths

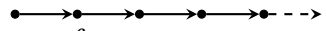
The following temporal operators are used to express **properties along a path**: neXt, Future, Globally, Until, Weak Until, Strong Release (M), Release

$\mathbf{X}\, f$	f holds in the next state	also: $\bigcirc$
$\mathbf{F}\, f$	f holds somewhere (eventually)	also: $\Diamond$
$\mathbf{G}\, f$	f holds everywhere	also: $\Box$
$f\ \mathbf{U}\ g$	g holds eventually, and f in all preceding states	
$f\ \mathbf{W}\ g$	$(\mathbf{G}\, f) \vee (f\ \mathbf{U}\ g)$	
$f\ \mathbf{M}\ g$	$g\ \mathbf{U}\ (f \wedge g)$	
$f\ \mathbf{R}\ g$	$(\mathbf{G}\, g) \vee (f\ \mathbf{M}\ g)$	

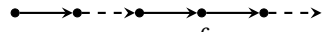
Informal meaning of temporal operators

f

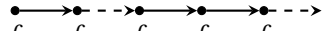
$\mathbf{X}\, f$



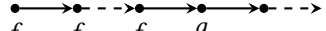
$\mathbf{F}\, f$



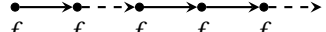
$\mathbf{G}\, f$



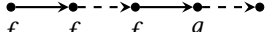
$f\ \mathbf{U}\ g$



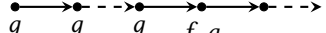
$f\ \mathbf{W}\ g$



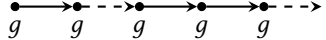
OR



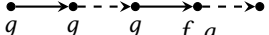
$f\ \mathbf{M}\ g$



$f\ \mathbf{R}\ g$



OR



Linear Temporal Logic

Syntax of LTL

LTL formula φ with a proposition $a \in AP$:

$$\varphi ::= \mathbf{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathbf{X}\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

Propositional logic

- atomic propositions
- negation and conjunction

Temporal modalities

- $\mathbf{X}\varphi$: next state fulfills φ
  also written as $\bigcirc\varphi$
- φ **U** ψ: φ holds until a ψ-state is reached

Linear Temporal Logic

Syntax of LTL

LTL formula φ with a proposition $a \in AP$:

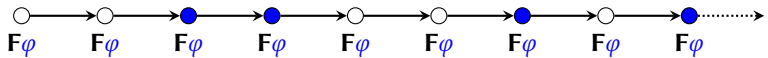
$$\varphi ::= \mathbf{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathbf{X}\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

Derived operators:

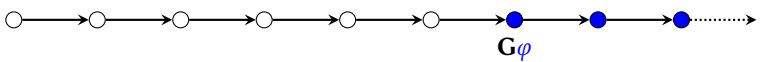
$\mathbf{false}$	$\equiv$	$\neg\,\mathbf{true}$	
$\varphi \vee \psi$	$\equiv$	$\neg\big((\neg\varphi) \wedge (\neg\psi)\big)$	
$\varphi \Rightarrow \psi$	$\equiv$	$\neg\varphi \vee \psi$	
$\mathbf{F}\varphi$	$\equiv$	$\mathbf{true}\ \mathbf{U}\ \varphi$	also written as $\Diamond\varphi$
$\mathbf{G}\varphi$	$\equiv$	$\neg\mathbf{F}\neg\varphi$	also written as $\Box\varphi$
$\varphi\ \mathbf{W}\ \psi$	$\equiv$	$\big(\varphi\ \mathbf{U}\ \psi\big) \vee \mathbf{G}\varphi$	weak until
$\varphi\ \mathbf{R}\ \psi$	$\equiv$	$\big(\psi\ \mathbf{U}\ (\varphi \wedge \psi)\big) \vee \mathbf{G}\psi$	release
$\varphi\ \mathbf{M}\ \psi$	$\equiv$	$\psi\ \mathbf{U}\ (\varphi \wedge \psi)$	strong release

Infinitely often / Eventually always

Infinitely often **GF**φ (recurrence)



Eventually always **FG**φ (persistence)



Duality of **GF** and **FG**

- **GF**φ if and only if $\neg$**FG**$\neg\varphi$
- **FG**φ if and only if $\neg$**GF**$\neg\varphi$

Semantics of LTL

LTL formula φ with a proposition $a \in AP$:

$$\varphi ::= \mathbf{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathbf{X}\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

$\sigma \models \mathbf{true}$	
$\sigma \models a$	iff $a \in AP$ and $a \in A_0$
$\sigma \models \varphi_1 \wedge \varphi_2$	iff $\sigma \models \varphi_1$ and $\sigma \models \varphi_2$
$\sigma \models \neg\varphi$	iff $\sigma \not\models \varphi$
$\sigma \models \mathbf{X}\varphi$	iff $\sigma^1 \models \varphi$
$\sigma \models \varphi \mathbf{U} \psi$	iff $\exists k \colon \sigma^k \models \psi$ and $\sigma^i \models \varphi$ for all $0 \leq i < k$
$\sigma \models \mathbf{F}\varphi$	iff $\exists k \colon \sigma^k \models \varphi$
$\sigma \models \mathbf{G}\varphi$	iff $\forall k \geq 0 \colon \sigma^k \models \varphi$

Examples

Find out whether the sequence

$$\sigma := \{p\}\{q\}\{p\}^{\omega}$$

satisfies the following formulae:

$$p$$
$$q$$
$$\mathbf{X}q$$
$$\mathbf{X}\neg p$$
$$p \mathbf{U} q$$
$$q \mathbf{U} p$$
$$(p \vee q) \mathbf{W} r$$

Linear Temporal Logic

Examples

How do you encode the following properties?

- Directly after red, a traffic light cannot turn green
- The light becomes green eventually
- Once red, the light eventually becomes green
- Once red, the light always becomes green eventually after being yellow for some time in between

- The light will always become red in the future
- If not red, the light can turn red again in the future

Linear Temporal Logic

Examples

How do you encode the following properties?

- Directly after red, a traffic light cannot turn green $\mathbf{G}(red \Rightarrow \neg \mathbf{X} green)$
- The light becomes green eventually
- Once red, the light eventually becomes green
- Once red, the light always becomes green eventually after being yellow for some time in between

- The light will always become red in the future
- If not red, the light can turn red again in the future

Linear Temporal Logic

Examples

How do you encode the following properties?

- Directly after red, a traffic light cannot turn green $\mathbf{G}(red \Rightarrow \neg \mathbf{X} green)$
- The light becomes green eventually $\mathbf{F} green$
- Once red, the light eventually becomes green
- Once red, the light always becomes green eventually after being yellow for some time in between

- The light will always become red in the future
- If not red, the light can turn red again in the future

Linear Temporal Logic

Examples

How do you encode the following properties?

- Directly after red, a traffic light cannot turn green $\mathbf{G}(red \Rightarrow \neg \mathbf{X} green)$
- The light becomes green eventually $\mathbf{F} green$
- Once red, the light eventually becomes green $\mathbf{G}(red \Rightarrow \mathbf{F} green)$
- Once red, the light always becomes green eventually after being yellow for some time in between

- The light will always become red in the future
- If not red, the light can turn red again in the future

Linear Temporal Logic

Examples

How do you encode the following properties?

- Directly after red, a traffic light cannot turn green $\mathbf{G}(red \Rightarrow \neg \mathbf{X} green)$
- The light becomes green eventually $\mathbf{F} green$
- Once red, the light eventually becomes green $\mathbf{G}(red \Rightarrow \mathbf{F} green)$
- Once red, the light always becomes green eventually after being yellow for some time in between $\mathbf{G}(red \Rightarrow \mathbf{X}(red\ \mathbf{U}\ (yellow \wedge \mathbf{X}(yellow\ \mathbf{U}\ green))))$
- The light will always become red in the future
- If not red, the light can turn red again in the future

Linear Temporal Logic

Examples

How do you encode the following properties?

- Directly after red, a traffic light cannot turn green $\mathbf{G}(red \Rightarrow \neg \mathbf{X} green)$
- The light becomes green eventually $\mathbf{F} green$
- Once red, the light eventually becomes green $\mathbf{G}(red \Rightarrow \mathbf{F} green)$
- Once red, the light always becomes green eventually after being yellow for some time in between $\mathbf{G}(red \Rightarrow \mathbf{X}(red\ \mathbf{U}\ (yellow \wedge \mathbf{X}(yellow\ \mathbf{U}\ green))))$
- The light will always become red in the future $\mathbf{GF} red$
- If not red, the light can turn red again in the future

Linear Temporal Logic

Examples

How do you encode the following properties?

- Directly after red, a traffic light cannot turn green $\mathbf{G}(red \Rightarrow \neg \mathbf{X} green)$
- The light becomes green eventually $\mathbf{F} green$
- Once red, the light eventually becomes green $\mathbf{G}(red \Rightarrow \mathbf{F} green)$
- Once red, the light always becomes green eventually after being yellow for some time in between $\mathbf{G}(red \Rightarrow \mathbf{X}(red\ \mathbf{U}\ (yellow \wedge \mathbf{X}(yellow\ \mathbf{U}\ green))))$
- The light will always become red in the future $\mathbf{GF} red$
- If not red, the light can turn red again in the future not expressible in LTL!

Example

If event a occurs, then b must occur before c can occur (a run $\ldots, a, (\neg b)^*, c, \ldots$ is illegal).

Example

If event a occurs, then b must occur before c can occur (a run $\ldots, a, (\neg b)^*, c, \ldots$ is illegal).

- First idea (wrong): $a \Rightarrow \ldots$
  - Every run $d, \ldots$ becomes legal.

Example

If event a occurs, then b must occur before c can occur (a run $\ldots, a, (\neg b)^*, c, \ldots$ is illegal).

- First idea (wrong): $a \Rightarrow \ldots$
  - Every run $d, \ldots$ becomes legal.
- Next idea (correct): $\mathbf{G}\ (a \Rightarrow \ldots)$

Example

If event a occurs, then b must occur before c can occur (a run $\ldots, a, (\neg b)^*, c, \ldots$ is illegal).

- First idea (wrong): $a \Rightarrow \ldots$
  - Every run $d, \ldots$ becomes legal.
- Next idea (correct): $\mathbf{G}\ (a \Rightarrow \ldots)$
- First attempt (wrong): $\mathbf{G}\ (a \Rightarrow \mathbf{X}\ (b\ \mathbf{U}\ c))$
  - Run $a, b, \neg b, c, \ldots$ is illegal.

Example

If event a occurs, then b must occur before c can occur (a run $\ldots, a, (\neg b)^*, c, \ldots$ is illegal).

- First idea (wrong): $a \Rightarrow \ldots$
  - Every run $d, \ldots$ becomes legal.
- Next idea (correct): $\mathbf{G}\ (a \Rightarrow \ldots)$
- First attempt (wrong): $\mathbf{G}\ (a \Rightarrow \mathbf{X}\ (b\ \mathbf{U}\ c))$
  - Run $a, b, \neg b, c, \ldots$ is illegal.
- Second attempt (better): $\mathbf{G}\ (a \Rightarrow (\neg c\ \mathbf{U}\ b))$
  - Run $a, \neg c, \neg c, \neg c, \ldots$ is illegal.

Example

If event a occurs, then b must occur before c can occur (a run $\ldots, a, (\neg b)^*, c, \ldots$ is illegal).

- First idea (wrong): $a \Rightarrow \ldots$
  - Every run $d, \ldots$ becomes legal.
- Next idea (correct): $\mathbf{G}\ (a \Rightarrow \ldots)$
- First attempt (wrong): $\mathbf{G}\ (a \Rightarrow \mathbf{X}\ (b\ \mathbf{U}\ c))$
  - Run $a, b, \neg b, c, \ldots$ is illegal.
- Second attempt (better): $\mathbf{G}\ (a \Rightarrow (\neg c\ \mathbf{U}\ b))$
  - Run $a, \neg c, \neg c, \neg c, \ldots$ is illegal.
- Third attempt (correct): $\mathbf{G}\ (a \Rightarrow (\neg c\ \mathbf{W}\ b))$

Example

If event a occurs, then b must occur before c can occur (a run $\ldots, a, (\neg b)^*, c, \ldots$ is illegal).

- First idea (wrong): $a \Rightarrow \ldots$
  - Every run $d, \ldots$ becomes legal.
- Next idea (correct): $\mathbf{G}\ (a \Rightarrow \ldots)$
- First attempt (wrong): $\mathbf{G}\ (a \Rightarrow \mathbf{X}\ (b\ \mathbf{U}\ c))$
  - Run $a, b, \neg b, c, \ldots$ is illegal.
- Second attempt (better): $\mathbf{G}\ (a \Rightarrow (\neg c\ \mathbf{U}\ b))$
  - Run $a, \neg c, \neg c, \neg c, \ldots$ is illegal.
- Third attempt (correct): $\mathbf{G}\ (a \Rightarrow (\neg c\ \mathbf{W}\ b))$

Think in terms of allowed/prohibited sequences.

Linear Temporal Logic

In practice, many LTL formulas are instances of particular patterns.

Pattern	Pronounced	Name
$\mathbf{G}\ \varphi$	always φ	invariance
$\mathbf{F}\ \varphi$	eventually φ	guarantee
$\mathbf{G}\ \mathbf{F}\ \varphi$	φ holds infinitely often	recurrence
$\mathbf{F}\ \mathbf{G}\ \varphi$	eventually φ holds permanently	stability
$\mathbf{G}\ (\varphi \Rightarrow \mathbf{F}\ \psi)$	always, if φ holds, then eventually ψ holds	response
$\mathbf{G}\ (\varphi \Rightarrow (\psi\ \mathbf{U}\ \chi))$	always, if φ holds, then ψ holds until χ holds	precedence

Path quantifiers: all paths versus there exists a path

Path quantifiers express properties on a system's branching structure:

for All paths versus there Exists a path

Path quantifiers have the following intuitive meaning:

- **A** f: f holds for all paths starting in this state
- **E** f: f holds for at least one path starting in this state
- **A** f and **E** f turn a path property into a state property

Adding branching: from LTL to CTL

LTL: Linear Time Logic

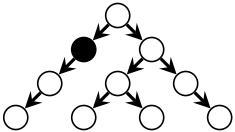
- No path quantifiers
- Formulae are interpreted on traces
- A system satisfies an LTL formula if all runs satisfy it

CTL: Computation Tree Logic

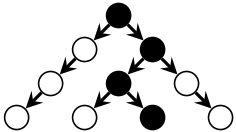
- Every temporal operator is immediately preceded by a path quantifier
- **EX** ϕ, **EG** ϕ, **E**(ϕ **U** ψ), **AX** ϕ, **AF** ϕ, **A**(ϕ **U** ψ), etc.

Examples with path quantifiers

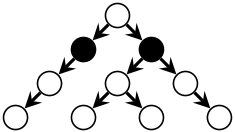
E X black



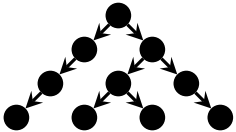
E G black



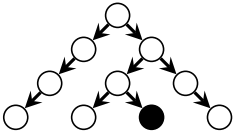
A X black



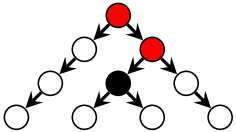
A G black



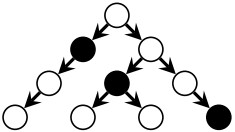
E F black



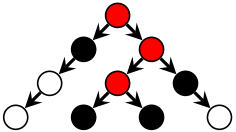
E red **U** black



A F black



A red **U** black



CTL Syntax

CTL: Computation Tree Logic

$\phi, \psi \quad ::= \quad \text{true} \mid \neg\phi \mid AP \mid \phi \wedge \psi \mid \mathbf{EX}\ \phi \mid \mathbf{EG}\ \phi \mid \mathbf{E}(\phi\ \mathbf{U}\ \psi)$

Every temporal operator in CTL is immediately paired with **A** or **E** .

Examples

CTL expressions: **AG EF** p, **E**(p **U** (**EX** q))

Not syntactically in CTL: **A F G** p, **A X X** p, **E**(p **U** (**X** q))

Branching versus Linear Time Logic

We use temporal logic to specify a formula ϕ.

- Model checking question: $\mathcal{M} \models \phi$ ("$\mathcal{M}$ satisfies ϕ").
- Branching time logic (CTL)
  - $\mathcal{M} \models \phi \Leftrightarrow \forall s_0 \in S_0 \,.\, s_0 \models \phi$
  - ϕ is evaluated at states, with respect to the computation tree
- Linear time logic (LTL)
  - $\mathcal{M} \models \phi \Leftrightarrow \pi \models \phi$ for every run π of $\mathcal{M}$.
  - ϕ is evaluated on traces; system satisfies formula iff all runs satisfy it.

Branching versus Linear Time Logic

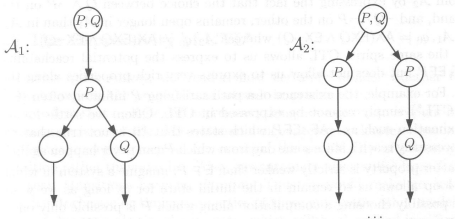


Fig. 2.4. Two automata, indistinguishable for PLTL

B. Berard et al: "Systems and Software Verification", 2001.

- Linear time logic: both systems have the same runs.
  - Thus every formula has same truth value in both systems.
- Branching time logic: the systems have different computation trees.
  - Take formula $\mathrm{AX}(\mathrm{EX}\, Q \wedge \mathrm{EX}\, \neg Q)$.
  - True for left system, false for right system.

The two variants of temporal logic have different expressive power.

Branching versus Linear Time Logic

Is one temporal logic variant more expressive than the other one?

- CTL formula: $\mathrm{AG}(\mathrm{EF}\ \phi)$.
  - "From every reachable state, there exists a path to a ϕ-state".
  - Property cannot be expressed by any LTL logic formula.
- LTL formula: $\mathrm{FG}\ \phi$.
  - "In every run, there is a moment from which on ϕ holds forever.".
  - Naive translation $\mathrm{AFG}\ \phi$ is not a CTL formula.
    - $\mathrm{G}\ \phi$ is a path formula, but F expects a state formula!
  - Translation $\mathrm{AFAG}\ \phi$ expresses a stronger property (see next page).
  - Property cannot be expressed by any CTL formula.

None of the two variants is strictly more expressive than the other one; no variant can express every system property.

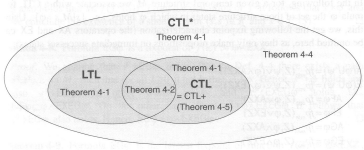
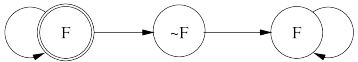


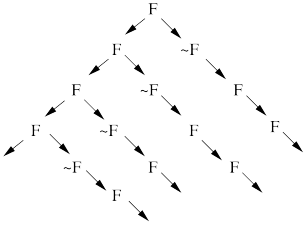
Fig. 4-8. Expressiveness of CTL*, CTL+, CTL and LTL

Thomas Kropf: "Introduction to Formal Hardware Verification", 1999.

Branching versus Linear Time Logic

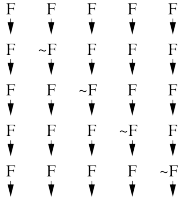
Proof that AFAG F (CTL) is different from FG F (LTL).





AFAG F <=> false

In every run, there is a moment when it is guarantueed that from now on F holds forever.



<>[] F <=> true

In every run, there is a moment from which on F holds forever.

What does “reachability” mean?

Potential vs. inevitability

- **Potential reachability:** p is reachable along some choice: **EF** p
- **Inevitable reachability:** p is reached on every execution: **AF** p

CTL versus LTL

- CTL can express both the potential and the inevitable reading.
- LTL can express only the inevitable reading **F** p

CTL* as a common superlogic

CTL* combines:

- state formulas
- path formulas
- temporal operators on paths
- path quantifiers on states

CTL and LTL are both fragments of CTL*

CTL restricts where temporal operators may occur; LTL omits path quantifiers.

Even CTL* still reasons about one path at a time.

CTL* grammar

CTL* state formulae ($\mathcal{S}$) and path formulae ($\mathcal{P}$) are defined simultaneously by induction:

$$\begin{array}{lcl} \mathcal{S} & ::= & \text{true} \mid \text{false} \mid AP \mid \neg\mathcal{S} \mid \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid \mathbf{E}\,\mathcal{P} \mid \mathbf{A}\,\mathcal{P} \\ \mathcal{P} & ::= & \mathcal{S} \mid \neg\mathcal{P} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P} \vee \mathcal{P} \mid \mathbf{X}\,\mathcal{P} \mid \mathbf{F}\,\mathcal{P} \mid \mathbf{G}\,\mathcal{P} \mid \\ & & \mathcal{P}\ \mathbf{U}\ \mathcal{P} \mid \mathcal{P}\ \mathbf{R}\ \mathcal{P} \mid \mathcal{P}\ \mathbf{W}\ \mathcal{P} \mid \mathcal{P}\ \mathbf{M}\ \mathcal{P} \end{array}$$

Summarising:

- State formulae ($\mathcal{S}$) are:
  - constants true and false and atomic propositions (basis)
  - Boolean combinations of state formulae
  - quantified path formulae
- Path formulae ($\mathcal{P}$) are:
  - state formulae (basis)
  - Boolean combinations of path formulae
  - temporal combinations of path formulae

A property expressible in CTL* but not in CTL or LTL

Example

$$\mathbf{E}\,(\mathbf{G}\,\mathbf{F}\,p \wedge \mathbf{G}\,\mathbf{F}\,q)$$

There exists a single path on which both p and q occur infinitely often.

Why this is interesting

- not in LTL: LTL has no existential path quantifier
- not in CTL: CTL cannot require both recurrence conditions to hold on the same path
- yes in CTL*: path quantification and temporal nesting can be combined freely

Beyond CTL*: hyperproperties

Some interesting specifications compare multiple runs

- LTL, CTL, and CTL* reason about one path at a time
- some security and information-flow properties relate two or more runs

Example: observational determinism / noninterference

If two runs have the same public inputs, then they should have the same public outputs.

Intuition: secret inputs must not influence what an external observer can see.

Why this is beyond CTL*

This property compares multiple traces at once, so it cannot be expressed in LTL, CTL, or CTL*.

Examples of richer logics: HyperLTL, HyperCTL*.

Other temporal and specification logics

Different systems call for different logics

- **TLA+**: temporal logic of actions with rich state variables and next-state relations; widely used for distributed and concurrent algorithms
- **Probabilistic logics** (e.g. PCTL): express probabilities such as "the chance of failure is below 10^{-6}"
- **Timed logics** (e.g. TCTL, MTL, STL): express deadlines and timing constraints using clocks or dense time
- **Process/data logics** (e.g. mCRL2, modal μ-calculus with data): combine actions, communication, and data parameters
- **Hyperlogics** (e.g. HyperLTL): express relations between multiple runs

Why use linear-time logic?

Why LTL is attractive

- LTL talks about runs rather than branching trees
- Formulas are often easier to read, because there are no explicit path quantifiers
- LTL expresses many common specification patterns:
  - invariance
  - eventuality / guarantee
  - response
  - recurrence and persistence
- LTL can also express fairness constraints
  - assumptions that rule out unrealistic infinite behaviour
  - e.g. a process that is enabled infinitely often is eventually scheduled

Intuition

LTL is often a natural language for describing what should happen along an execution of the system.

Limits of linear-time logic

What LTL cannot do as well

- LTL is less expressive than richer temporal logics such as CTL* or the μ-calculus
- LTL cannot express genuinely branching-time properties
  - for example: from every reachable state, it is still possible to recover
- CTL can express such resettability or remaining possibility properties

Model checking complexity

- CTL model checking is linear in the formula size
- LTL model checking is exponential in the formula size
- in practice, the size of the state space is often still the main bottleneck

Ω-automata

From finite words to infinite words

Goals of this chapter

- move from regular languages over finite words to ω-regular languages over infinite words
- introduce ω-regular expressions and Büchi automata
- explain why Büchi automata capture exactly the ω-regular languages
- compare other acceptance conditions for automata over infinite words
- understand the value of parity automata

Finite-word recap

Regular expressions and finite automata

- Regular expressions describe languages over finite words
- Syntax:

$$r ::= \emptyset \mid \epsilon \mid a \mid r + r \mid rr \mid r^*$$

- Semantics: union, concatenation, and finite repetition

ω-regular expressions

From finite repetition to infinite repetition

Regular expressions describe languages over Σ^*. To describe languages over Σ^ω, we add infinite repetition:

$$s ::= r^\omega \mid r.s \mid s + s$$

where r is an ordinary regular expression and $\epsilon \notin \mathcal{L}(r)$.

Finite automata recap

Finite automata

- A finite automaton is $\langle S, S_0, \Sigma, R, F \rangle$
  - S is a finite set of states and S_0 is a set of initial states
  - Σ is an alphabet
  - $R \subseteq S \times \Sigma \times S$ is a transition relation
  - $F \subseteq S$ is a set of accepting (final) states
- NFAs and DFAs recognize exactly the regular languages
- Main constructions:
  - RE $\longrightarrow$ NFA: build from the syntax tree
  - NFA $\longrightarrow$ RE: eliminate states
  - NFA $\longrightarrow$ DFA: subset construction
- Determinization is possible, but may be exponential

Finite automata recap

Finite automata

- A finite automaton is $\langle S, S_0, \Sigma, R, F \rangle$
  - S is a finite set of states and S_0 is a set of initial states
  - Σ is an alphabet
  - $R \subseteq S \times \Sigma \times S$ is a transition relation
  - $F \subseteq S$ is a set of accepting (final) states
- NFAs and DFAs recognize exactly the regular languages
- Main constructions:
  - RE $\longrightarrow$ NFA: build from the syntax tree
  - NFA $\longrightarrow$ RE: eliminate states
  - NFA $\longrightarrow$ DFA: subset construction
- Determinization is possible, but may be exponential
- We now ask: what is the right analogue for infinite words?

Büchi automata

The analogue of NFA for infinite words

A Büchi automaton has the same graph structure as an NFA:

$$\langle S, S_0, \Sigma, R, F \rangle$$

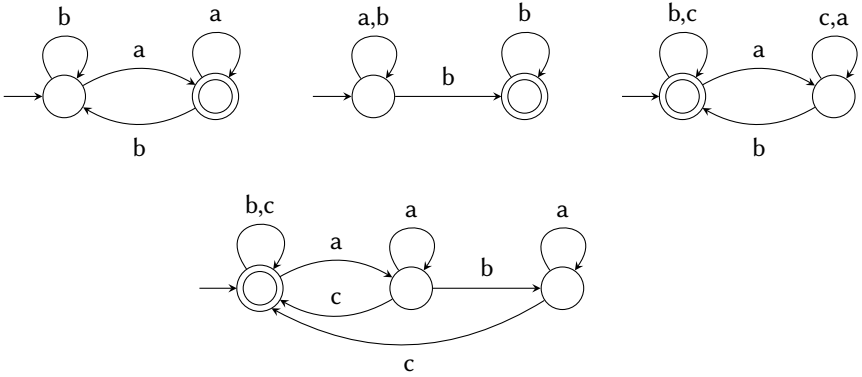
but it runs on infinite words.

Acceptance

A run is accepting if it visits some state in F infinitely often.

Büchi automata

Examples



Büchi automata capture the ω-regular languages

Expressive power

Theorem

- every ω-regular language is recognized by some NBA
- every language recognized by an NBA is ω-regular

Why this matters

Büchi automata play for infinite words the same role that NFAs play for finite words.

From ω-regular expressions to NBAs: idea

Just like for finite words: follow the syntax tree

An ω-regular expression has grammar

$$s ::= r^{\omega} \mid r.s \mid s + s$$

so we only need three constructions.

Three cases

- $\boldsymbol{r^{\omega}}$: start with an NFA for r, and for each accepting state, jump back to an initial state (by merging states or inserting ε-transitions)
- $\boldsymbol{r.s}$: from each accepting state of the automaton for r, continue into an initial state of the automaton for s
- $\boldsymbol{s_1 + s_2}$: take the union by combining the initial states

From NBAs to ω-regular expressions

Intuition: An accepting run of an NBA has two parts:

- a finite prefix that reaches some accepting state q
- an infinite repetition of loops from q back to q

For fixed states p, q, the language of finite words taking the automaton from p to q is regular.

So an accepted ω-word has the shape

$$\text{prefix to } q \;\cdot\; (\text{loop from } q \text{ to } q)^{\omega}$$

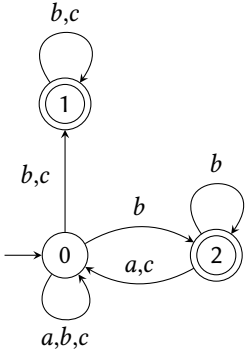
and therefore the accepted language is ω-regular.

Formally:

$$\mathcal{L}_{\omega}(\mathcal{A}) = \bigcup_{p \in S_0} \bigcup_{q \in F} \mathcal{L}(\mathcal{A}_{p,q}) \cdot \left(\mathcal{L}(\mathcal{A}_{q,q}) \setminus \{\epsilon\}\right)^{\omega}$$

From NBAs to ω-regular expressions

Example



$L_{0,1} =$

$L_{1,1} =$

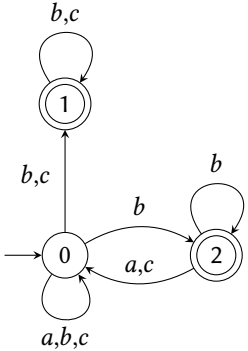
$L_{0,2} =$

$L_{2,2} =$

$L_{\mathcal{A}} =$

From NBAs to ω-regular expressions

Example



$L_{0,1} = (a + b + c)^*(b + c)$

$L_{1,1} =$

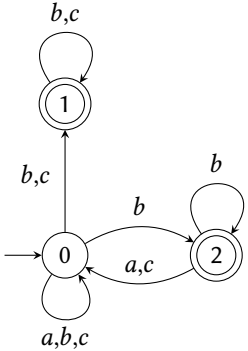
$L_{0,2} =$

$L_{2,2} =$

$L_{\mathcal{A}} =$

From NBAs to ω-regular expressions

Example



$$L_{0,1} = (a + b + c)^*(b + c)$$

$$L_{1,1} = b + c$$

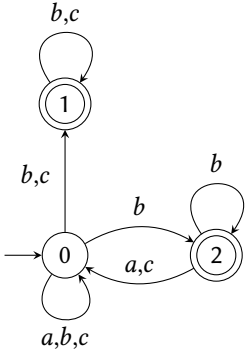
$$L_{0,2} =$$

$$L_{2,2} =$$

$$L_{\mathcal{A}} =$$

From NBAs to ω-regular expressions

Example



$$L_{0,1} = (a + b + c)^*(b + c)$$

$$L_{1,1} = b + c$$

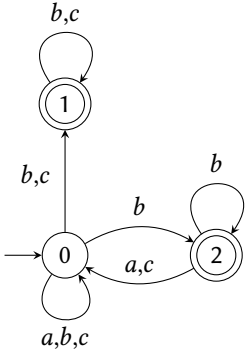
$$L_{0,2} = (a + b + c)^* b$$

$$L_{2,2} =$$

$$L_{\mathcal{A}} =$$

From NBAs to ω-regular expressions

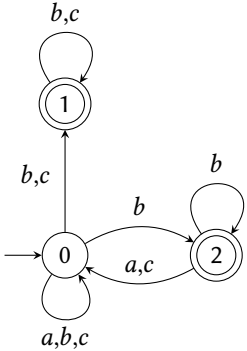
Example



$$L_{0,1} = (a + b + c)^*(b + c)$$
$$L_{1,1} = b + c$$
$$L_{0,2} = (a + b + c)^*b$$
$$L_{2,2} = b + (a + c)(a + b + c)^*b$$
$$L_{\mathcal{A}} =$$

From NBAs to ω-regular expressions

Example



$$L_{0,1} = (a + b + c)^*(b + c)$$
$$L_{1,1} = b + c$$
$$L_{0,2} = (a + b + c)^*b$$
$$L_{2,2} = b + (a + c)(a + b + c)^*b$$
$$L_{\mathcal{A}} = L_{0,1}(L_{1,1})^\omega + L_{0,2}(L_{2,2})^\omega$$

Generalized Büchi automata

Generalized Büchi automaton

Generalized Büchi automaton: $\langle S, S_0, \Sigma, R, \mathcal{F} \rangle$

- $\mathcal{F}$ is a set of sets of accepting states $\{F_1, \ldots, F_n\}$
- A run is accepting if it visits each set of accepting states infinitely often
- Same expressive power as NBAs
- Often more convenient as an intermediate target
  - NBA $\rightarrow$ GNBA: trivial
  - GNBA $\rightarrow$ NBA: multiply states by $|\mathcal{F}|$

Deterministic Büchi is weaker

DBAs and ω-regular languages

- NFAs are as expressive as regular expressions
- DFAs are as expressive as regular expressions
- NBAs are as expressive as ω-regular expressions
- What about DBAs?

Deterministic Büchi is weaker

DBAs and ω-regular languages

- NFAs are as expressive as regular expressions
- DFAs are as expressive as regular expressions
- NBAs are as expressive as ω-regular expressions
- What about DBAs?

Claim: The language $(a + b)^* b^\omega$ is not recognized by any DBA.

Intuition: an NBA can guess the moment after which only b's remain and then move to an accepting mode. A DBA cannot guess this point: it must decide deterministically while reading the word. So it would need to keep unfolding more and more history, which is impossible with finitely many states.

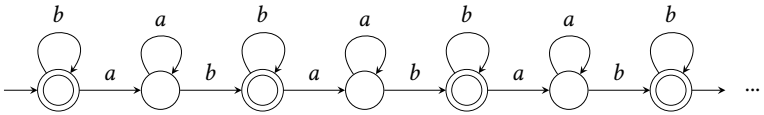
Deterministic Büchi is weaker

DBAs and ω-regular languages

- NFAs are as expressive as regular expressions
- DFAs are as expressive as regular expressions
- NBAs are as expressive as ω-regular expressions
- What about DBAs?

Claim: The language $(a + b)^* b^\omega$ is not recognized by any DBA.

Intuition: an NBA can guess the moment after which only b's remain and then move to an accepting mode. A DBA cannot guess this point: it must decide deterministically while reading the word. So it would need to keep unfolding more and more history, which is impossible with finitely many states.



What makes an acceptance condition useful?

Desired properties

- **Same expressive power as NBAs**
  - otherwise we lose some ω-regular properties
- **Equally expressive deterministic versions**
  - useful for synthesis, games, and some probabilistic verification algorithms
- **Easy complementation**
  - verification often checks emptiness of the complement
  - synthesis frequently alternates assumptions and guarantees
  - language containment and equivalence use complement
- **Compact acceptance condition**
  - otherwise the automaton becomes awkward to manipulate

Co-Büchi automata

Acceptance condition

A run is accepting iff it visits the accepting states only finitely often.

What to remember

- Deterministic and nondeterministic co-Büchi automata have the same expressive power
- But co-Büchi automata are strictly less expressive than Büchi automata
- So co-Büchi is interesting, but not sufficient as a general replacement for Büchi

Muller automata

Acceptance condition

A Muller condition is a set of sets of states $\mathcal{F} = \{F_0, F_1, \dots\}$. A run is accepting if the set of infinitely often occurring states is an element of $\mathcal{F}$.

Key points

- Deterministic Muller automata are as expressive as NBAs
- Very general and elegant in theory
- Complexity is shifted from the graph to the acceptance condition
- Acceptance condition can become large and cumbersome in practice

Rabin automata

Acceptance condition

A Rabin condition is a set of pairs (F_i, G_i). A run is accepting if for some pair:

- F_i is visited infinitely often, and
- G_i is visited only finitely often.

Key points

- NBA $\rightarrow$ NRA: trivial, NBA $\leftarrow$ NRA: quadratic
- Deterministic Rabin automata are as expressive as NBAs
- Historically important as a deterministic target (NBA to DRA)
- Complementation increases size exponentially

Streett automata

Acceptance condition

A Streett condition is a set of pairs (F_i, G_i). A run is accepting if for all pairs:

- either F_i is visited infinitely often,
- or G_i is visited only finitely often.

Key points

- Streett is the dual of Rabin
- Deterministic Streett automata are as expressive as NBAs
- Natural for fairness-style requirements
  If G_i keeps occurring forever, then F_i must also keep occurring forever.
- Complementation increases size exponentially

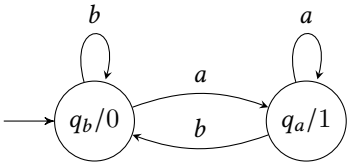
Parity automata

Acceptance condition

A parity automaton assigns priorities to states (or nested accepting sets). A run is accepting if the largest priority seen infinitely often is even.

Key points

- Same expressive power as Büchi automata
- Deterministic parity automata are fully expressive
- Complementation is trivial: swap even and odd priorities
- Easy to manage acceptance condition



Why parity automata matter

Why this acceptance condition is especially useful

- as expressive as Büchi automata
- deterministic parity automata are fully expressive
- complementation is trivial
- priorities are simple to manipulate algorithmically
- parity objectives induce parity games, and those admit memoryless winning strategies
- parity connects naturally to synthesis and to the modal μ-calculus

Takeaway

Büchi automata are central for LTL model checking; parity automata are the key deterministic class for games and synthesis.

Comparing ω-automata

Main picture

- **Büchi:** core class for ω-regular languages
- **Generalized Büchi:** same power, convenient intermediate form
- **Deterministic Büchi:** too weak
- **Co-Büchi:** deterministic = nondeterministic, but weaker than Büchi
- **Muller / Rabin / Streett / parity:** recover full deterministic expressive power
- **Parity:** often the most convenient deterministic target

Why do we care about these automata?

Applications

- **Model checking:** translate LTL to GNBA / NBA and combine with the system
- **Synthesis:** needs deterministic acceptance conditions, often Rabin or parity
- **Games and fixpoints:** parity automata connect naturally to parity games and to model checking of the modal μ-calculus

Practical message

Büchi automata are the core class for LTL model checking; parity automata are the key deterministic class for synthesis and games.

See also: Udi Boker, Why these Automata Types?

LTL rewrite rules

Why rewrite LTL formulas?

Why do we transform formulas?

- to put formulas into a form that is easier to translate to automata
- to simplify formulas before constructing the automaton
- to reduce the number of subformulae and obligations
- to recognize fragments that often yield simpler automata

Positive normal form

Syntax

Standard LTL $\varphi ::= \text{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathbf{X}\,\varphi \mid \varphi_1\ \mathbf{U}\ \varphi_2$

Positive normal form $\varphi ::= \text{true} \mid \text{false} \mid a \mid \neg a \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathbf{X}\,\varphi \mid \varphi_1\ \mathbf{U}\ \varphi_2 \mid \varphi_1\ \mathbf{R}\ \varphi_2$

Positive normal form

Push negations inward

$$
\begin{aligned}
\neg \mathbf{X}\, \varphi &\equiv \mathbf{X}\, \neg\varphi \\
\neg \mathbf{F}\, \varphi &\equiv \mathbf{G}\, \neg\varphi \\
\neg \mathbf{G}\, \varphi &\equiv \mathbf{F}\, \neg\varphi \\
\neg(\varphi\ \mathbf{U}\ \psi) &\equiv \neg\varphi\ \mathbf{R}\ \neg\psi \\
\neg(\varphi\ \mathbf{R}\ \psi) &\equiv \neg\varphi\ \mathbf{U}\ \neg\psi \\
\neg(\varphi\ \mathbf{W}\ \psi) &\equiv \neg\varphi\ \mathbf{M}\ \neg\psi \\
\neg(\varphi\ \mathbf{M}\ \psi) &\equiv \neg\varphi\ \mathbf{W}\ \neg\psi
\end{aligned}
$$

Goal

After rewriting, negations occur only in front of atomic propositions.

Positive normal form

Eliminate derived temporal operators

$$
\begin{aligned}
\mathbf{F}\,\varphi &\equiv \text{true}\ \mathbf{U}\ \varphi \\
\mathbf{G}\,\varphi &\equiv \text{false}\ \mathbf{R}\ \varphi \\
\varphi\ \mathbf{W}\ \psi &\equiv \psi\ \mathbf{R}\ (\varphi \vee \psi) \\
\varphi\ \mathbf{M}\ \psi &\equiv \psi\ \mathbf{U}\ (\varphi \wedge \psi)
\end{aligned}
$$

Goal

After rewriting, formulas use only boolean connectives, atomic propositions, and the temporal operators $\mathbf{X}$, $\mathbf{U}$, $\mathbf{R}$.
Negations occur only in front of atomic propositions.

Useful simplifications

Common equivalences

$$
\begin{aligned}
\mathbf{G}\,\mathbf{G}\,\varphi &\equiv \mathbf{G}\,\varphi \\
\mathbf{F}\,\mathbf{F}\,\varphi &\equiv \mathbf{F}\,\varphi \\
\mathbf{F}\,\mathbf{G}\,\mathbf{F}\,\varphi &\equiv \mathbf{G}\,\mathbf{F}\,\varphi \\
\mathbf{G}\,\mathbf{F}\,\mathbf{G}\,\varphi &\equiv \mathbf{F}\,\mathbf{G}\,\varphi \\
\mathbf{X}\,(\varphi \vee \psi) &\equiv \mathbf{X}\,\varphi \vee \mathbf{X}\,\psi \\
\mathbf{X}\,(\varphi \wedge \psi) &\equiv \mathbf{X}\,\varphi \wedge \mathbf{X}\,\psi \\
\mathbf{X}\,(\varphi\ \mathbf{U}\ \psi) &\equiv (\mathbf{X}\varphi)\ \mathbf{U}\ (\mathbf{X}\psi) \\
\mathbf{F}\,(\varphi \vee \psi) &\equiv \mathbf{F}\,\varphi \vee \mathbf{F}\,\psi \\
\mathbf{G}\,(\varphi \wedge \psi) &\equiv \mathbf{G}\,\varphi \wedge \mathbf{G}\,\psi
\end{aligned}
$$

But not everything distributes

$$\mathbf{F}\,(\varphi \wedge \psi) \not\equiv \mathbf{F}\,\varphi \wedge \mathbf{F}\,\psi \qquad \mathbf{G}\,(\varphi \vee \psi) \not\equiv \mathbf{G}\,\varphi \vee \mathbf{G}\,\psi$$

Recursive view of temporal operators

Expansion laws

$$
\begin{aligned}
\mathbf{F}\,\varphi &\equiv \varphi \vee \mathbf{X}\,\mathbf{F}\,\varphi \\
\mathbf{G}\,\varphi &\equiv \varphi \wedge \mathbf{X}\,\mathbf{G}\,\varphi \\
\varphi\,\mathbf{U}\,\psi &\equiv \psi \vee (\varphi \wedge \mathbf{X}\,\varphi\,\mathbf{U}\,\psi) \\
\varphi\,\mathbf{W}\,\psi &\equiv \psi \vee (\varphi \wedge \mathbf{X}\,\varphi\,\mathbf{W}\,\psi) \\
\varphi\,\mathbf{R}\,\psi &\equiv \psi \wedge (\varphi \vee \mathbf{X}\,\varphi\,\mathbf{R}\,\psi) \\
\varphi\,\mathbf{M}\,\psi &\equiv \psi \wedge (\varphi \vee \mathbf{X}\,\varphi\,\mathbf{M}\,\psi)
\end{aligned}
$$

Notice that the same subformula appears again on the right-hand side.

Recursive view of temporal operators

Expansion laws

$$
\begin{array}{rcll}
\mathbf{F}\,\varphi: & \chi & = & [\![\varphi]\!] \cup \mathbf{X}\,\chi \\
\mathbf{G}\,\varphi: & \chi & = & [\![\varphi]\!] \cap \mathbf{X}\,\chi \\
\varphi\,\mathbf{U}\,\psi: & \chi & = & [\![\psi]\!] \cup ([\![\varphi]\!] \cap \mathbf{X}\,\chi) \\
\varphi\,\mathbf{W}\,\psi: & \chi & = & [\![\psi]\!] \cup ([\![\varphi]\!] \cap \mathbf{X}\,\chi) \\
\varphi\,\mathbf{R}\,\psi: & \chi & = & [\![\psi]\!] \cap ([\![\varphi]\!] \cup \mathbf{X}\,\chi) \\
\varphi\,\mathbf{M}\,\psi: & \chi & = & [\![\psi]\!] \cap ([\![\varphi]\!] \cup \mathbf{X}\,\chi)
\end{array}
$$

We write $[\![\varphi]\!]$ for the set of states where φ holds.
So temporal operators can be viewed as recursive equations over sets of states.

Recursive view of temporal operators

Expansion laws

$$
\begin{array}{rcll}
\mathbf{F}\,\varphi: & \chi & = & [\![\varphi]\!] \cup \mathbf{X}\,\chi \\
\mathbf{G}\,\varphi: & \chi & = & [\![\varphi]\!] \cap \mathbf{X}\,\chi \\
\varphi\,\mathbf{U}\,\psi: & \chi & = & [\![\psi]\!] \cup ([\![\varphi]\!] \cap \mathbf{X}\,\chi) \\
\varphi\,\mathbf{W}\,\psi: & \chi & = & [\![\psi]\!] \cup ([\![\varphi]\!] \cap \mathbf{X}\,\chi) \\
\varphi\,\mathbf{R}\,\psi: & \chi & = & [\![\psi]\!] \cap ([\![\varphi]\!] \cup \mathbf{X}\,\chi) \\
\varphi\,\mathbf{M}\,\psi: & \chi & = & [\![\psi]\!] \cap ([\![\varphi]\!] \cup \mathbf{X}\,\chi)
\end{array}
$$

We write $[\![\varphi]\!]$ for the set of states where φ holds.
So temporal operators can be viewed as recursive equations over sets of states.

Later we will revisit these equations as least and greatest fixed points.

What rewrites help in practice?

Automata-friendly rewriting

- push negations down to atomic propositions
- simplify nested modalities such as **GG**, **FF**, **GFG**, **FGF**
- reduce the number of **U**-subformulae when possible
- share repeated subformulae using a DAG representation
- fewer subformulae in the closure usually means fewer automaton states
- fewer **U**-subformulae means fewer GNBA acceptance sets

LTL to NBA

Warm-up: guess the automaton

Warm up exercise: Guess the NBA/GNBA...

$$AP = \{p, q\}$$

- $\mathbf{F}\, p$
- $\mathbf{G}\, p$
- $p\ \mathbf{U}\ q$
- $\mathbf{GF} p$

From LTL to GNBA: the pipeline

Construction idea

LTL formula $\longrightarrow$ positive normal form $\longrightarrow$ closure $\longrightarrow$ consistent states $\longrightarrow$ GNBA $\longrightarrow$ NBA

Intuition

A state records which subformulae are currently required to hold. Transitions update these obligations one step later. Acceptance ensures that eventualities are eventually fulfilled.

From LTL to GNBA: what each step does

Construction idea

1. Rewrite the formula to positive normal form
2. Compute the closure: the relevant subformulae
3. Build candidate states as subsets of the closure
4. Keep only locally consistent states
5. Keep only transitions that satisfy the **X** , **U** , and **R** rules
6. Add acceptance sets so pending **U** -obligations cannot be postponed forever

Intuition

A state records which subformulae hold now. Transitions describe what must hold one step later. Acceptance ensures that eventualities are eventually fulfilled.

Positive normal form for LTL

Syntax

Standard LTL $\varphi ::= \text{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathbf{X}\,\varphi \mid \varphi_1\ \mathbf{U}\ \varphi_2$

Positive normal form $\varphi ::= \text{true} \mid \text{false} \mid a \mid \neg a \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathbf{X}\,\varphi \mid \varphi_1\ \mathbf{U}\ \varphi_2 \mid \varphi_1\ \mathbf{R}\ \varphi_2$

Normalization theorem

For every LTL formula φ, there exists an equivalent formula ψ in positive normal form such that

$$|\psi| = O(|\varphi|)$$

Idea: push negations inward using dualities, until negations occur only in front of atomic propositions.

Examples:

$$\neg\mathbf{X}\,\varphi \equiv \mathbf{X}\,\neg\varphi \qquad \neg\varphi\ \mathbf{U}\ \psi \equiv \neg\varphi\ \mathbf{R}\ \neg\psi$$

Closure of a formula

Idea

$\texttt{cl}(\varphi)$ contains one representative of each subformula, up to negation. A state will record which of these formulas are true now, so we do not store both ψ and $\neg\psi$ separately.

Example

Let

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b).$$

Closure of a formula

Idea

$\texttt{cl}(\varphi)$ contains one representative of each subformula, up to negation. A state will record which of these formulas are true now, so we do not store both ψ and $\neg\psi$ separately.

Example

Let

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b).$$

Start with the whole formula:

$$a \,\mathbf{U}\, (\neg a \wedge b)$$

Closure of a formula

Idea

$\texttt{cl}(\varphi)$ contains one representative of each subformula, up to negation. A state will record which of these formulas are true now, so we do not store both ψ and $\neg\psi$ separately.

Example

Let

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b).$$

Add immediate subformulae:

$$a \,\mathbf{U}\, (\neg a \wedge b), \quad a, \quad \neg a \wedge b$$

Closure of a formula

Idea

$\texttt{cl}(\varphi)$ contains one representative of each subformula, up to negation. A state will record which of these formulas are true now, so we do not store both ψ and $\neg\psi$ separately.

Example

Let

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b).$$

Add remaining subformulae:

$$a \,\mathbf{U}\, (\neg a \wedge b), \quad a, \quad \neg a \wedge b, \quad b$$

Closure of a formula

Idea

$\texttt{cl}(\varphi)$ contains one representative of each subformula, up to negation. A state will record which of these formulas are true now, so we do not store both ψ and $\neg\psi$ separately.

Example

Let

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b).$$

So the closure is:

$$\texttt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ a \,\mathbf{U}\, (\neg a \wedge b)\}$$

Closure of a formula

Idea

$\texttt{cl}(\varphi)$ contains one representative of each subformula, up to negation. A state will record which of these formulas are true now, so we do not store both ψ and $\neg\psi$ separately.

Example

Let

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b).$$

So the closure is:

$$\texttt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ a \,\mathbf{U}\, (\neg a \wedge b)\}$$

Not stored separately: $\neg a$, $\neg b$, $\neg(\neg a \wedge b)$, $\neg\varphi$.
Their truth is determined by the current state.

States as consistent temporal obligations

Idea

A state of the GNBA is a subset of $\texttt{cl}(\varphi)$.

For each formula $\psi \in \texttt{cl}(\varphi)$:

$$\psi \in S \quad \text{means “}\psi\text{ holds now”}, \qquad \psi \notin S \quad \text{means “}\psi\text{ does not hold now”}.$$

Propositional consistency

- true must hold; false cannot hold
- if $\psi_1 \wedge \psi_2 \in \texttt{cl}(\varphi)$, then
$$\psi_1 \wedge \psi_2 \in S \iff \psi_1 \in S \text{ and } \psi_2 \in S$$
- if $\psi_1 \vee \psi_2 \in \texttt{cl}(\varphi)$, then
$$\psi_1 \vee \psi_2 \in S \iff \psi_1 \in S \text{ or } \psi_2 \in S$$

States as consistent temporal obligations

Idea

A state of the GNBA is a subset of $\mathtt{cl}(\varphi)$.

For each formula $\psi \in \mathtt{cl}(\varphi)$:

$$\psi \in S \quad \text{means “}\psi\text{ holds now”}, \qquad \psi \notin S \quad \text{means “}\psi\text{ does not hold now”}.$$

Local consistency for temporal operators

- if $\chi \in S$, then $\psi \mathbf{U} \chi \in S$
- if $\psi \mathbf{U} \chi \in S$ and $\chi \notin S$, then $\psi \in S$
- if $\psi \in S$ and $\chi \in S$, then $\psi \mathbf{R} \chi \in S$
- if $\psi \mathbf{R} \chi \in S$, then $\chi \in S$

States as consistent temporal obligations

Idea

A state of the GNBA is a subset of $\mathtt{cl}(\varphi)$.

For each formula $\psi \in \mathtt{cl}(\varphi)$:

$$\psi \in S \quad \text{means “}\psi \text{ holds now”}, \qquad \psi \notin S \quad \text{means “}\psi \text{ does not hold now”}.$$

Local consistency for temporal operators

- if $\chi \in S$, then $\psi \mathbf{U} \chi \in S$
- if $\psi \mathbf{U} \chi \in S$ and $\chi \notin S$, then $\psi \in S$
- if $\psi \in S$ and $\chi \in S$, then $\psi \mathbf{R} \chi \in S$
- if $\psi \mathbf{R} \chi \in S$, then $\chi \in S$

So what do we keep?

We keep only those subsets of $\mathtt{cl}(\varphi)$ that satisfy all of these local consistency requirements.

Applying the consistency rules

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \texttt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

A candidate state assigns a truth value to each formula in $\texttt{cl}(\varphi)$.

a	b	$\neg a \wedge b$	φ	remark
T	T	T	T	
T	T	T	F	
T	T	F	T	
T	T	F	F	
T	F	T	T	
T	F	T	F	
T	F	F	T	
T	F	F	F	

a	b	$\neg a \wedge b$	φ	remark
F	T	T	T	
F	T	T	F	
F	T	F	T	
F	T	F	F	
F	F	T	T	
F	F	T	F	
F	F	F	T	
F	F	F	F	

Applying the consistency rules

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \mathtt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

A candidate state assigns a truth value to each formula in $\mathtt{cl}(\varphi)$.

a	b	$\neg a \wedge b$	φ	remark
T	T	T	T	$a,\ \neg a \wedge b$
T	T	T	F	$a,\ \neg a \wedge b$
T	T	F	T	
T	T	F	F	
T	F	T	T	$a,\ \neg a \wedge b$
T	F	T	F	$a,\ \neg a \wedge b$
T	F	F	T	
T	F	F	F	

a	b	$\neg a \wedge b$	φ	remark
F	T	T	T	
F	T	T	F	
F	T	F	T	
F	T	F	F	
F	F	T	T	
F	F	T	F	
F	F	F	T	
F	F	F	F	

Checking: $\varphi \wedge \psi$ holds iff both φ and ψ hold

Applying the consistency rules

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \mathtt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

A candidate state assigns a truth value to each formula in $\mathtt{cl}(\varphi)$.

a	b	$\neg a \wedge b$	φ	remark
T	T	T	T	$a,\ \neg a \wedge b$
T	T	T	F	$a,\ \neg a \wedge b$
T	T	F	T	
T	T	F	F	
T	F	T	T	$a,\ \neg a \wedge b$
T	F	T	F	$a,\ \neg a \wedge b$
T	F	F	T	
T	F	F	F	

a	b	$\neg a \wedge b$	φ	remark
F	T	T	T	
F	T	T	F	
F	T	F	T	
F	T	F	F	
F	F	T	T	$b,\ \neg a \wedge b$
F	F	T	F	$b,\ \neg a \wedge b$
F	F	F	T	
F	F	F	F	

Checking: $\varphi \wedge \psi$ holds iff both φ and ψ hold

Applying the consistency rules

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \mathtt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

A candidate state assigns a truth value to each formula in $\mathtt{cl}(\varphi)$.

a	b	$\neg a \wedge b$	φ	remark
T	T	T	T	$a,\ \neg a \wedge b$
T	T	T	F	$a,\ \neg a \wedge b$
T	T	F	T	
T	T	F	F	
T	F	T	T	$a,\ \neg a \wedge b$
T	F	T	F	$a,\ \neg a \wedge b$
T	F	F	T	
T	F	F	F	

a	b	$\neg a \wedge b$	φ	remark
F	T	T	T	
F	T	T	F	
F	T	F	T	$a,\ b,\ \neg a \wedge b$
F	T	F	F	$a,\ b,\ \neg a \wedge b$
F	F	T	T	$b,\ \neg a \wedge b$
F	F	T	F	$b,\ \neg a \wedge b$
F	F	F	T	
F	F	F	F	

Checking: $\varphi \wedge \psi$ holds iff both φ and ψ hold

Applying the consistency rules

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \texttt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

A candidate state assigns a truth value to each formula in $\texttt{cl}(\varphi)$.

a	b	$\neg a \wedge b$	φ	remark
T	T	T	T	$a,\ \neg a \wedge b$
T	T	T	F	$a,\ \neg a \wedge b$
T	T	F	T	
T	T	F	F	
T	F	T	T	$a,\ \neg a \wedge b$
T	F	T	F	$a,\ \neg a \wedge b$
T	F	F	T	
T	F	F	F	

a	b	$\neg a \wedge b$	φ	remark
F	T	T	T	
F	T	T	F	$\varphi,\ \neg a \wedge b$
F	T	F	T	$a,\ b,\ \neg a \wedge b$
F	T	F	F	$a,\ b,\ \neg a \wedge b$
F	F	T	T	$b,\ \neg a \wedge b$
F	F	T	F	$b,\ \neg a \wedge b$
F	F	F	T	
F	F	F	F	

Checking: if ψ already holds now, then $\varphi \,\mathbf{U}\, \psi$ holds now

Applying the consistency rules

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \mathtt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

A candidate state assigns a truth value to each formula in $\mathtt{cl}(\varphi)$.

a	b	$\neg a \wedge b$	φ	remark
T	T	T	T	$a,\ \neg a \wedge b$
T	T	T	F	$a,\ \neg a \wedge b$
T	T	F	T	
T	T	F	F	
T	F	T	T	$a,\ \neg a \wedge b$
T	F	T	F	$a,\ \neg a \wedge b$
T	F	F	T	
T	F	F	F	

a	b	$\neg a \wedge b$	φ	remark
F	T	T	T	
F	T	T	F	$\varphi,\ \neg a \wedge b$
F	T	F	T	$a,\ b,\ \neg a \wedge b$
F	T	F	F	$a,\ b,\ \neg a \wedge b$
F	F	T	T	$b,\ \neg a \wedge b$
F	F	T	F	$b,\ \neg a \wedge b$
F	F	F	T	$\varphi,\ a,\ \neg a \wedge b$
F	F	F	F	

Checking: if $\varphi \,\mathbf{U}\, \psi$ holds now and ψ does not, then φ must hold now

Applying the consistency rules

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \mathtt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

A candidate state assigns a truth value to each formula in $\mathtt{cl}(\varphi)$.

a	b	$\neg a \wedge b$	φ	remark
T	T	T	T	$a,\ \neg a \wedge b$
T	T	T	F	$a,\ \neg a \wedge b$
T	T	F	T	✓
T	T	F	F	✓
T	F	T	T	$a,\ \neg a \wedge b$
T	F	T	F	$a,\ \neg a \wedge b$
T	F	F	T	✓
T	F	F	F	✓

a	b	$\neg a \wedge b$	φ	remark
F	T	T	T	✓
F	T	T	F	$\varphi,\ \neg a \wedge b$
F	T	F	T	$a,\ b,\ \neg a \wedge b$
F	T	F	F	$a,\ b,\ \neg a \wedge b$
F	F	T	T	$b,\ \neg a \wedge b$
F	F	T	F	$b,\ \neg a \wedge b$
F	F	F	T	$\varphi,\ a,\ \neg a \wedge b$
F	F	F	F	✓

Applying the consistency rules

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \mathtt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

A candidate state assigns a truth value to each formula in $\mathtt{cl}(\varphi)$.

a	b	$\neg a \wedge b$	φ	remark
T	T	T	T	$a,\ \neg a \wedge b$
T	T	T	F	$a,\ \neg a \wedge b$
T	T	F	T	✓
T	T	F	F	✓
T	F	T	T	$a,\ \neg a \wedge b$
T	F	T	F	$a,\ \neg a \wedge b$
T	F	F	T	✓
T	F	F	F	✓

a	b	$\neg a \wedge b$	φ	remark
F	T	T	T	✓
F	T	T	F	$\varphi,\ \neg a \wedge b$
F	T	F	T	$a,\ b,\ \neg a \wedge b$
F	T	F	F	$a,\ b,\ \neg a \wedge b$
F	F	T	T	$b,\ \neg a \wedge b$
F	F	T	F	$b,\ \neg a \wedge b$
F	F	F	T	$\varphi,\ a,\ \neg a \wedge b$
F	F	F	F	✓

Remaining locally consistent states:

$$(T,T,F,T),\ (T,T,F,F),\ (T,F,F,T),\ (T,F,F,F),\ (F,T,T,T),\ (F,F,F,F)$$

Building the GNBA

For an LTL formula φ, we construct a GNBA $B_\varphi = \langle CS(\varphi), S_0, 2^{AP}, \Delta, \mathcal{F} \rangle$

Ingredients

- **States:** all consistent subsets $CS(\varphi)$ of $\mathtt{cl}(\varphi)$
- **Initial states:** those states that contain the formula φ

$$S_0 = \{ S \in CS(\varphi) \mid \varphi \in S \}$$

- **Alphabet:** valuations of the atomic propositions 2^{AP}

Why this alphabet?

The GNBA reads exactly the truth values of the atomic propositions. Later, in the product with a Kripke structure or transition system, these are matched with the state labels.

Building the GNBA: transition rules

Transition relation

A transition $(S, \sigma, S') \in \Delta$ is allowed iff:

- $\sigma = S \cap AP$
- for each $\mathbf{X}\,\psi \in \mathtt{cl}(\varphi)$:

$$\mathbf{X}\,\psi \in S \iff \psi \in S'$$

- for each $\psi\ \mathbf{U}\ \chi \in \mathtt{cl}(\varphi)$:

$$\psi\ \mathbf{U}\ \chi \in S \iff \chi \in S \ \vee\ (\psi \in S \wedge \psi\ \mathbf{U}\ \chi \in S')$$

- for each $\psi\ \mathbf{R}\ \chi \in \mathtt{cl}(\varphi)$:

$$\psi\ \mathbf{R}\ \chi \in S \iff \chi \in S \wedge (\psi \in S \vee \psi\ \mathbf{R}\ \chi \in S')$$

Building the GNBA: transition rules

What this already guarantees

By restricting to consistent states and valid transitions, we already enforce:

- propositional consistency
- all **X**-obligations
- all **R**-obligations
- the local part of each **U**-obligation

So this almost takes care of the whole formula.

What is still missing?

If some φ **U** ψ keeps being postponed forever, then the run should not be accepting. This is handled by the acceptance sets.

Building the GNBA: acceptance sets

Acceptance sets

For each until-subformula $\psi \mathbf{U} \chi \in \mathtt{cl}(\varphi)$, define

$$F_{\psi \mathbf{U} \chi} = \{ S \in CS(\varphi) \mid \psi \mathbf{U} \chi \notin S \ \vee \ \chi \in S \}.$$

So a state is in $F_{\psi \mathbf{U} \chi}$ iff:

- either this until-obligation is no longer pending,
- or it is fulfilled now because χ already holds.

Acceptance condition

A run is accepting iff for every until-subformula $\psi \mathbf{U} \chi$, it visits $F_{\psi \mathbf{U} \chi}$ infinitely often.

Running example: states and acceptance

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \mathtt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

Consistent states

$$S_1 = \{a, b, \varphi\} \quad S_2 = \{a, b\} \quad S_3 = \{a, \varphi\}$$
$$S_4 = \{a\} \quad S_5 = \{b, \neg a \wedge b, \varphi\} \quad S_6 = \emptyset$$

Initial states

$$S_0 = \{S_1, S_3, S_5\}$$

Acceptance set

There is only one until-subformula, namely φ itself, so:

$$F_\varphi = \{\, S \in CS(\varphi) \mid \varphi \notin S \vee (\neg a \wedge b) \in S \,\} = \{S_2, S_4, S_5, S_6\}.$$

The only non-accepting states are S_1 and S_3: there the until-obligation is still pending.

Running example: transition relation

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \mathtt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

Transitions for the six states (just the **U** rule)

$$\psi \,\mathbf{U}\, \chi \in S \iff \chi \in S \ \vee\ (\psi \in S \wedge \psi \,\mathbf{U}\, \chi \in S')$$

$$\begin{aligned}
S_1 &= \{a, b, \varphi\} \xrightarrow{\{a,b\}} \\
S_2 &= \{a, b\} \xrightarrow{\{a,b\}} \\
S_3 &= \{a, \varphi\} \xrightarrow{\{a\}} \\
S_4 &= \{a\} \xrightarrow{\{a\}} \\
S_5 &= \{b, \neg a \wedge b, \varphi\} \xrightarrow{\{b\}} \\
S_6 &= \emptyset \xrightarrow{\emptyset}
\end{aligned}$$

Running example: transition relation

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \mathtt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

Transitions for the six states (just the **U** rule)

$$\psi \,\mathbf{U}\, \chi \in S \iff \chi \in S \ \vee\ (\psi \in S \wedge \psi \,\mathbf{U}\, \chi \in S')$$

$$\begin{aligned}
S_1 &= \{a, b, \varphi\} \xrightarrow{\{a,b\}} \\
S_2 &= \{a, b\} \xrightarrow{\{a,b\}} \\
S_3 &= \{a, \varphi\} \xrightarrow{\{a\}} \\
S_4 &= \{a\} \xrightarrow{\{a\}} \\
S_5 &= \{b, \neg a \wedge b, \varphi\} \xrightarrow{\{b\}} \{S_1, S_2, S_3, S_4, S_5, S_6\} \\
S_6 &= \emptyset \xrightarrow{\emptyset}
\end{aligned}$$

From S_5, the goal $\neg a \wedge b$ already holds, so the obligation is fulfilled and any successor is allowed.

Running example: transition relation

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \mathtt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

Transitions for the six states (just the **U** rule)

$$\psi \,\mathbf{U}\, \chi \in S \iff \chi \in S \ \vee\ (\psi \in S \wedge \psi \,\mathbf{U}\, \chi \in S')$$

$$\begin{aligned}
S_1 = \{a, b, \varphi\} &\xrightarrow{\{a,b\}} \{S_1, S_3, S_5\} \\
S_2 = \{a, b\} &\xrightarrow{\{a,b\}} \\
S_3 = \{a, \varphi\} &\xrightarrow{\{a\}} \{S_1, S_3, S_5\} \\
S_4 = \{a\} &\xrightarrow{\{a\}} \\
S_5 = \{b, \neg a \wedge b, \varphi\} &\xrightarrow{\{b\}} \{S_1, S_2, S_3, S_4, S_5, S_6\} \\
S_6 = \emptyset &\xrightarrow{\emptyset}
\end{aligned}$$

From S_1 and S_3, the **U**-formula is still pending, so the next state must still satisfy φ.

Running example: transition relation

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \mathtt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

Transitions for the six states (just the **U** rule)

$$\psi \,\mathbf{U}\, \chi \in S \iff \chi \in S \ \vee\ (\psi \in S \wedge \psi \,\mathbf{U}\, \chi \in S')$$

$$\begin{aligned}
S_1 = \{a, b, \varphi\} &\xrightarrow{\{a,b\}} \{S_1, S_3, S_5\} \\
S_2 = \{a, b\} &\xrightarrow{\{a,b\}} \{S_2, S_4, S_6\} \\
S_3 = \{a, \varphi\} &\xrightarrow{\{a\}} \{S_1, S_3, S_5\} \\
S_4 = \{a\} &\xrightarrow{\{a\}} \{S_2, S_4, S_6\} \\
S_5 = \{b, \neg a \wedge b, \varphi\} &\xrightarrow{\{b\}} \{S_1, S_2, S_3, S_4, S_5, S_6\} \\
S_6 = \emptyset &\xrightarrow{\emptyset}
\end{aligned}$$

From S_2 and S_4, the **U**-formula is false, and the successor must not revive it.

Running example: transition relation

$$\varphi = a \,\mathbf{U}\, (\neg a \wedge b) \qquad \mathtt{cl}(\varphi) = \{a,\ b,\ \neg a \wedge b,\ \varphi\}$$

Transitions for the six states (just the **U** rule)

$$\psi \,\mathbf{U}\, \chi \in S \iff \chi \in S \ \vee\ (\psi \in S \wedge \psi \,\mathbf{U}\, \chi \in S')$$

$$\begin{aligned}
S_1 = \{a, b, \varphi\} &\xrightarrow{\{a,b\}} \{S_1, S_3, S_5\} \\
S_2 = \{a, b\} &\xrightarrow{\{a,b\}} \{S_2, S_4, S_6\} \\
S_3 = \{a, \varphi\} &\xrightarrow{\{a\}} \{S_1, S_3, S_5\} \\
S_4 = \{a\} &\xrightarrow{\{a\}} \{S_2, S_4, S_6\} \\
S_5 = \{b, \neg a \wedge b, \varphi\} &\xrightarrow{\{b\}} \{S_1, S_2, S_3, S_4, S_5, S_6\} \\
S_6 = \emptyset &\xrightarrow{\emptyset} \{S_1, S_2, S_3, S_4, S_5, S_6\}
\end{aligned}$$

From S_6, both a and $\neg a \wedge b$ are false, so the rule imposes no restriction on the successor.

Linear Temporal Logic

Main theorem (due to Vardi, Wolper)

For any LTL formula φ over AP there exists a GNBA B_φ over 2^{AP} with:

- $\mathit{Words}(\varphi) = \mathcal{L}_\omega(B_\varphi)$;
- B_φ can be constructed in time and space $O(2^{|\varphi|})$;
- $|\mathcal{F}|$ of B_φ is bounded above by $O(|\varphi|)$.

Additional notes

- all transitions leaving some state S carry the same label $S \cap AP$
- each transition is a "nondeterministic guess" about the future
- there exist LTL-formulae of size $O(n)$ for which any Büchi automaton has necessarily exponential size
- In general, the reverse translation BA $\longrightarrow$ LTL is not possible
  The property "p holds in every second step" is not expressible in LTL

Why is LTL model checking exponential in the formula?

Where the blow-up comes from

- the closure $\mathtt{cl}(\varphi)$ has size $O(|\varphi|)$
- a GNBA state is a subset of the closure
- so the number of candidate states is exponential: $2^{O(|\varphi|)}$
- after filtering inconsistent states, the construction is still exponential in $|\varphi|$

Takeaway

The exponential blow-up comes from storing combinations of subformula obligations.

Implementation tips: representing the formula

Formula representation

- Parse the LTL formula into a DAG
- Ensure each subformula is stored only once
- Convert the formula (or in model checking: its negation) to positive normal form
- Number the positive subformulae in post-order (by walking the DAG)
- Represent a GNBA state as a bit vector over $\texttt{cl}(\varphi)$

Computing the state space

- Generate candidate truth assignments for the subformulae
- Keep only locally consistent states
- Initial states are those containing φ

Implementation tips: successors and emptiness

Successors

- For a state S, the label is $S \cap AP$
- Generate successor candidates
- Keep only those satisfying the transition rules for **X** , **U** , and **R**
- Either build the whole GNBA in advance or compute successors on the fly

After building the product

- Match automaton labels with the labels of the Kripke structure
- Compute SCCs
- Check whether an SCC satisfies all acceptance sets

Try it yourself: Spot

Useful examples

- $\mathbf{F}p$
- $\mathbf{G}(request \rightarrow \mathbf{F}grant)$
- $\mathbf{GF}p$

`https://spot.lrde.epita.fr/app/`

From GNBA to NBA

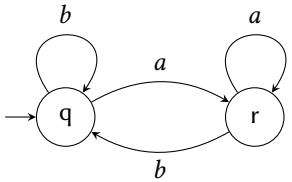
Important fact: the sets $F_1, \ldots, F_n$ are visited infinitely often is equivalent to F_1 is eventually visited and every visit to F_i is eventually followed by a visit to $F_{(i+1) \bmod n}$

Example from GNBA with $\mathcal{F} = \{F_1, F_2, F_3\}$ to NBA

- Create 3 copies of the NBA
- Set as initial states: the initial states of copy 1
- Accepting states: F_1 in copy 1
- Edges from F_1 of copy 1 go to successors in copy 2 instead
- Edges from F_2 of copy 2 go to successors in copy 3 instead
- Edges from F_3 of copy 3 go to successors in copy 1 instead

From GNBA to NBA

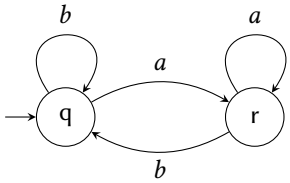
Example from GNBA to NBA



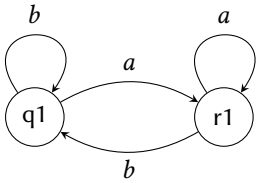
$\mathcal{F} = \{\{q\}, \{r\}\}$

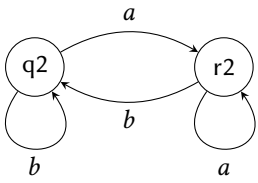
From GNBA to NBA

Example from GNBA to NBA



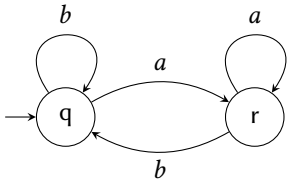
$\mathcal{F} = \{\{q\}, \{r\}\}$



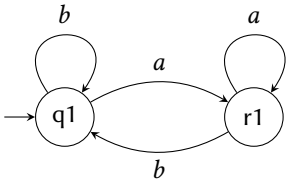


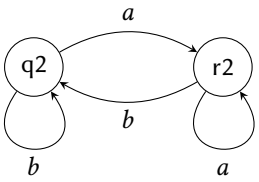
From GNBA to NBA

Example from GNBA to NBA



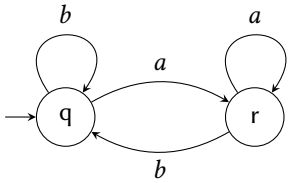
$\mathcal{F} = \{\{q\}, \{r\}\}$



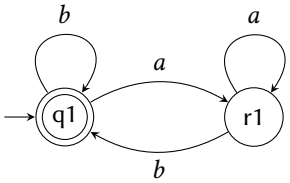


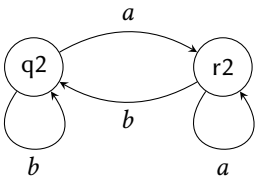
From GNBA to NBA

Example from GNBA to NBA



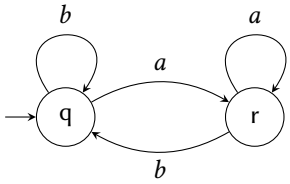
$\mathcal{F} = \{\{q\}, \{r\}\}$



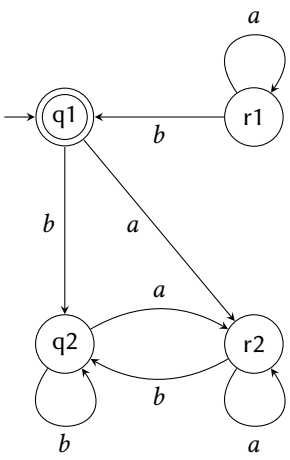


From GNBA to NBA

Example from GNBA to NBA

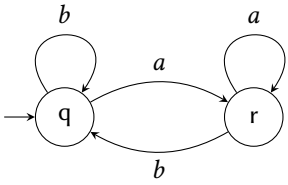


$\mathcal{F} = \{\{q\}, \{r\}\}$

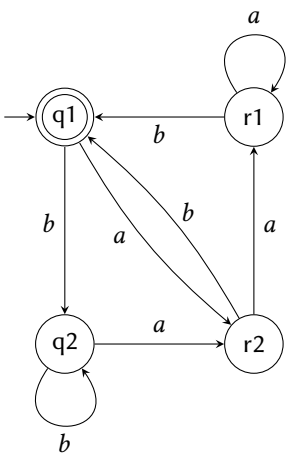


From GNBA to NBA

Example from GNBA to NBA



$\mathcal{F} = \{\{q\}, \{r\}\}$



Implementation tips: GNBA to NBA

Degeneralization in practice

- If the GNBA has acceptance sets $\mathcal{F} = \{F_1, \ldots, F_k\}$, create k copies of each state
- Represent an NBA state as a pair (q, i), where q is a GNBA state and i is the current acceptance-set index
- Initial states are $(q, 1)$ for each GNBA initial state q
- For a transition $q \xrightarrow{\sigma} q'$:
  - stay in copy i if $q \notin F_i$
  - move to copy $i + 1$ (cyclically) if $q \in F_i$
- The accepting states are the states in copy 1 whose GNBA component lies in F_1

Practical notes

- Keep the original GNBA state structure intact; only add the copy counter
- On-the-fly construction is often easy here

Model checking with ω-automata

LTL model checking with automata

Goal

Given a finite transition system TS and an LTL formula ϕ, decide whether

$$\mathrm{TS} \models \phi$$

Automata-theoretic pipeline

$$(\mathrm{TS}, \phi) \;\rightarrow\; \neg\phi \;\rightarrow\; \text{GNBA/NBA for } \neg\phi \;\rightarrow\; \mathrm{TS} \otimes \mathcal{A}_{\neg\phi} \;\rightarrow\; \text{emptiness check}$$

Why do we negate?

A counterexample to $\mathrm{TS} \models \phi$ is exactly a run of TS that satisfies $\neg\phi$.
Emptiness check tests existence: does a bad trace exist? (**not**: does a good trace exist?)

Model checking with ω-automata

We have a transition system TS and a NBA $\mathcal{A}$:

- TS $= \langle S, S_0, \mathcal{A}ct, R, AP, L \rangle$
- $\mathcal{A} = \langle Q, Q_0, 2^{AP}, \delta, F \rangle$

We compute the product $\mathcal{A}_{\text{TS}} = \text{TS} \otimes \mathcal{A} = \langle S \times Q, S'_0, \mathcal{A}ct, R', F' \rangle$

- $F' = \{(s, q) \mid q \in F\}$
- transition relation: if s can go to a s' with labeling $L(s')$, then the automaton may move from q to q' only if it has a transition on the label $L(s')$

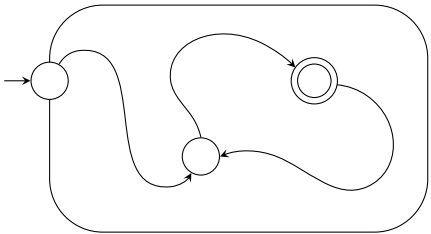
$$\frac{s \xrightarrow{a} s' \quad \wedge \quad q' \in \delta(q, L(s'))}{(s, q) \xrightarrow{a'} (s', q')}$$

- initial states: $S'_0 = \{(s_0, q) \mid s_0 \in S_0,\ \exists q_0 \in Q_0 :\ q \in \delta(q_0, L(s_0))\}$

The automaton reads valuations from the labels of states in the transition system, so in the product we synchronize each TS-step with an automaton step on the matching label.

Model checking with ω-automata

An NBA is nonempty iff it has an accepting lasso



Model checking with ω-automata

Setting

- We want on-the-fly algorithms that search for an accepting lasso of an NBA while constructing it
- Starting in some initial state, we have access to the `successors()` function of the transition system, and we have an NBA of the negation of the property that we want to check
- The resulting state space can be millions of states big, so the algorithm needs to be efficient

Two approaches

- Find an accepting state, then check if it can reach itself: nested depth-first-search
- For every strongly connected component, check if it contains an accepting state.

Model checking with ω-automata

Nested DFS

- The blue search explores the state space
  - All “new” states are colored cyan
  - All “done” states are colored blue, unless it is accepting
  - All “done” accepting states start a red search
- The red search explores the state space (all states are actually blue or cyan) and colors all blue states red.
- If the red search finds a cyan state, then it found an accepting cycle.

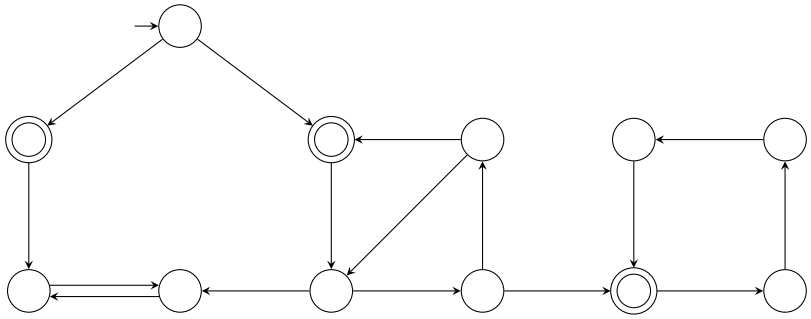
Literature

- Courcoubetis, Vardi, et al: *Memory Efficient Algorithms for the Verification of Temporal Properties.* (CAV 1990)
- Alfons Laarman, Rom Langerak, Jaco van de Pol, Michael Weber, Anton Wijs: *Multi-Core Nested Depth-First Search.* (ATVA 2011)
- Jaco van de Pol: *Automated Verification of Nested DFS.* (FMICS 2015)
- Wytse Oortwijn, Marieke Huisman, Sebastiaan Joosten, Jaco van de Pol: *Automated Verification of Parallel Nested DFS.* (TACAS 2020)

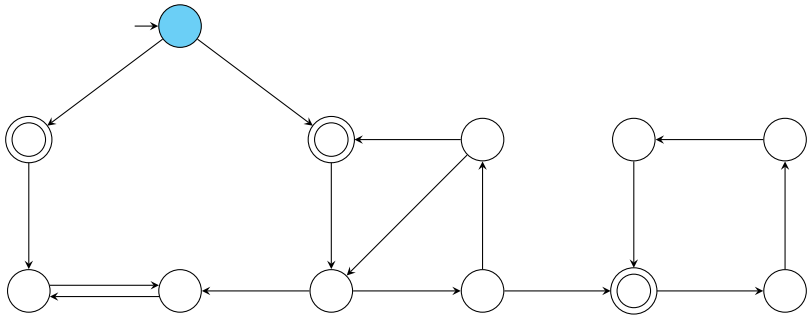
Nested DFS

```
1  def dfsBlue(s):
2      s.color ← cyan                                   // seen in blue search
3      foreach t ∈ get-successors(s) do
4          if t.color = white : dfsBlue(t)
5      if s ∈ F :                                       // if accepting state
6          dfsRed (s)                                   // run the red search
7          s.color ← red                                // finished in red search
8      else: s.color ← blue                             // finished in blue search
9  def dfsRed(s):
10     foreach t ∈ get-successors(s) do
11         if t.color = cyan : report cycle and exit
12         elif t.color = blue :
13             t.color ← red                            // finished in red search
14             dfsRed (t)
```

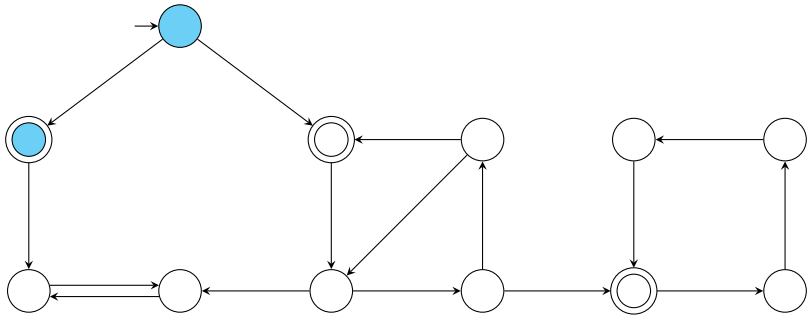
Nested DFS Example



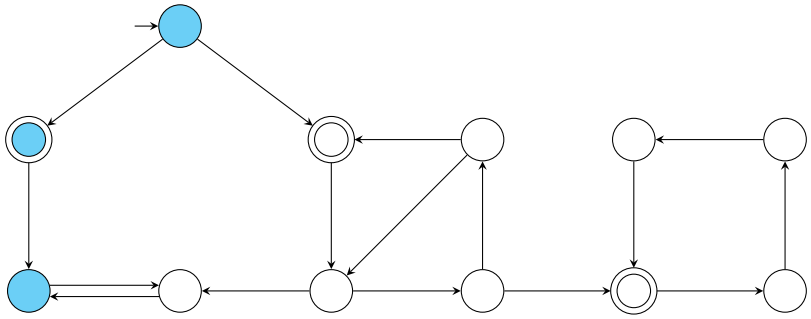
Nested DFS Example



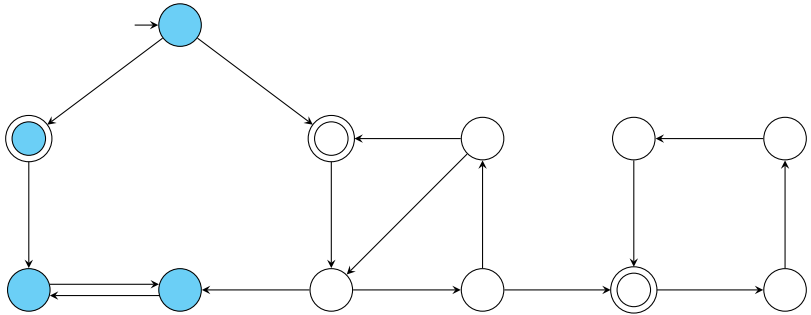
Nested DFS Example



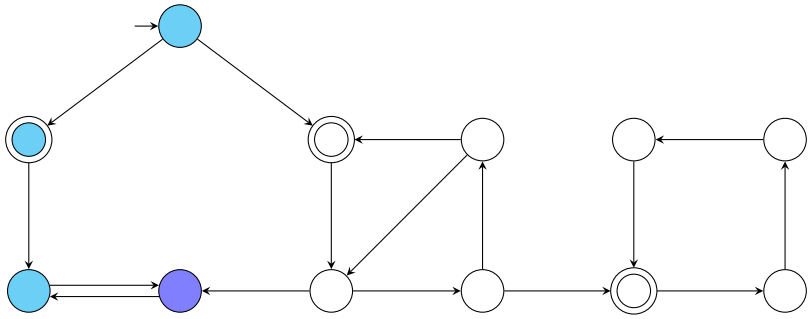
Nested DFS Example



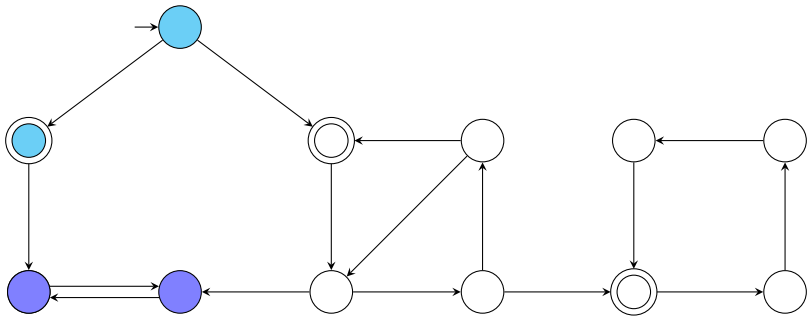
Nested DFS Example



Nested DFS Example

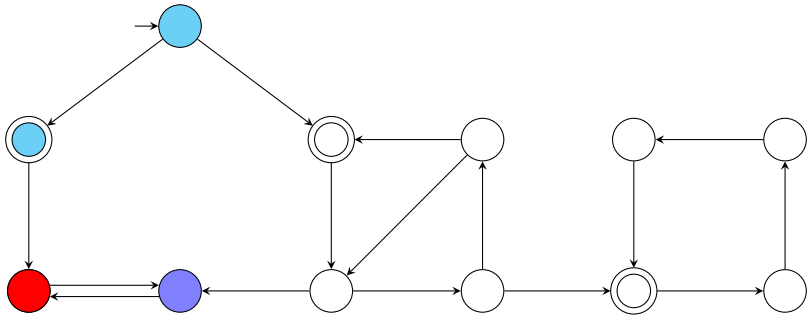


Nested DFS Example

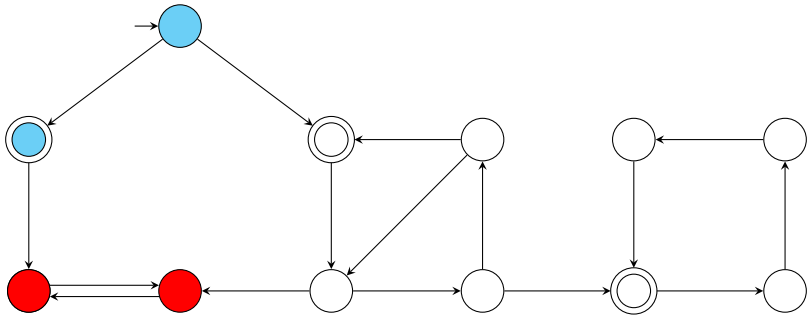


Now we start a red search after finishing the accepting state!

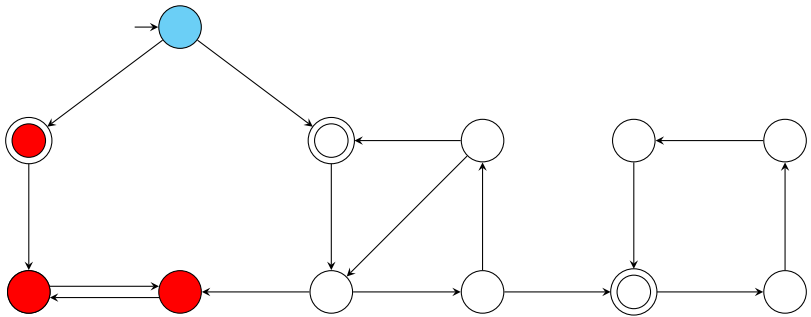
Nested DFS Example



Nested DFS Example

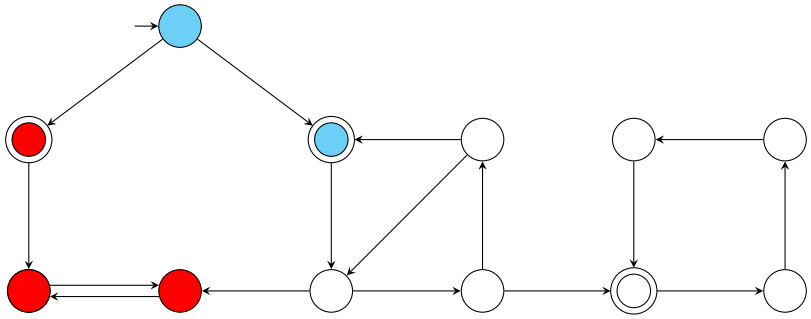


Nested DFS Example

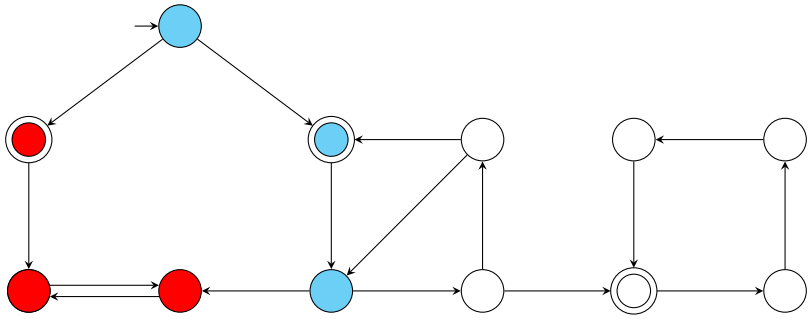


Red search done, continue blue search.

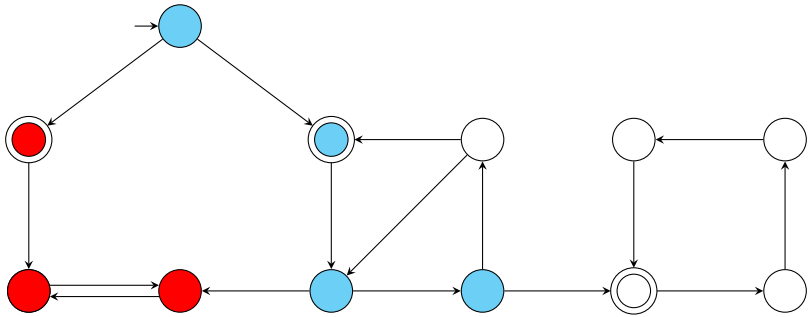
Nested DFS Example



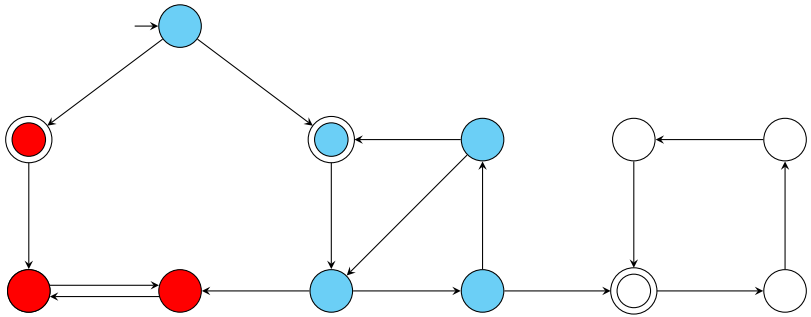
Nested DFS Example



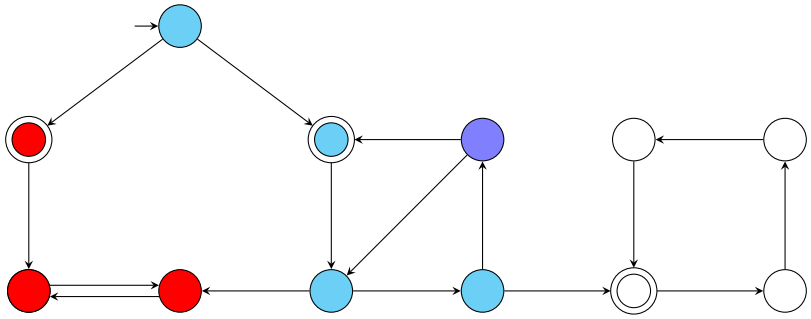
Nested DFS Example



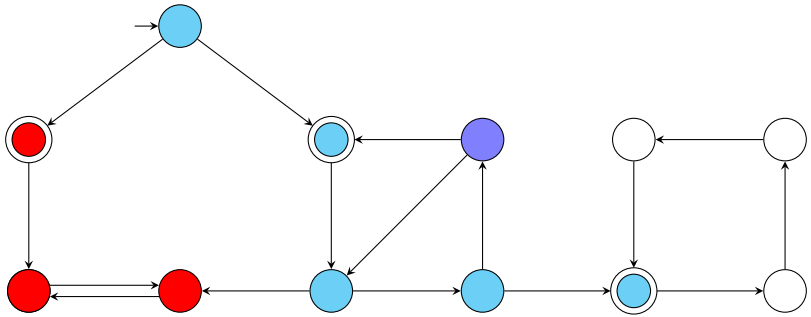
Nested DFS Example



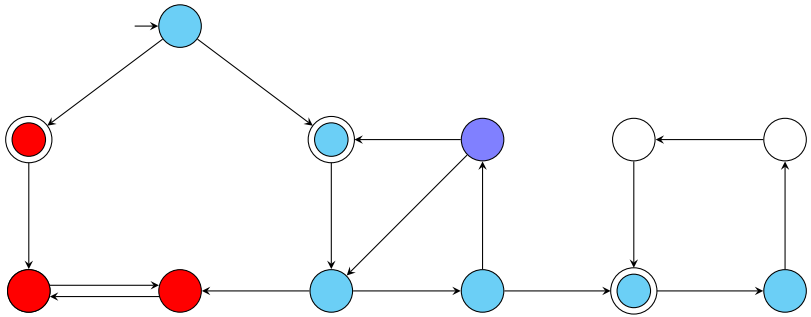
Nested DFS Example



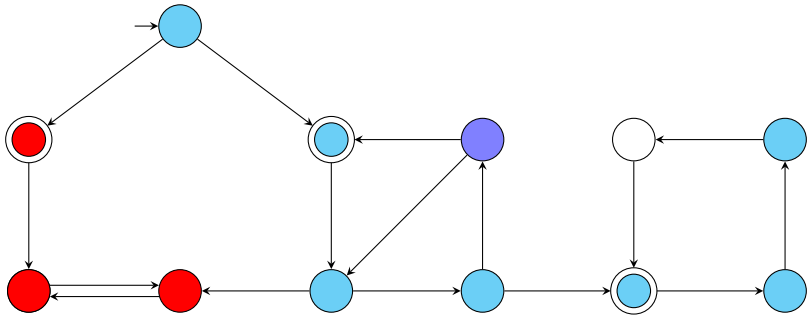
Nested DFS Example



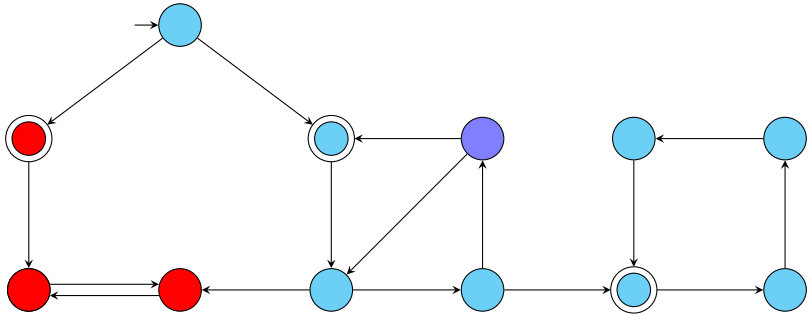
Nested DFS Example



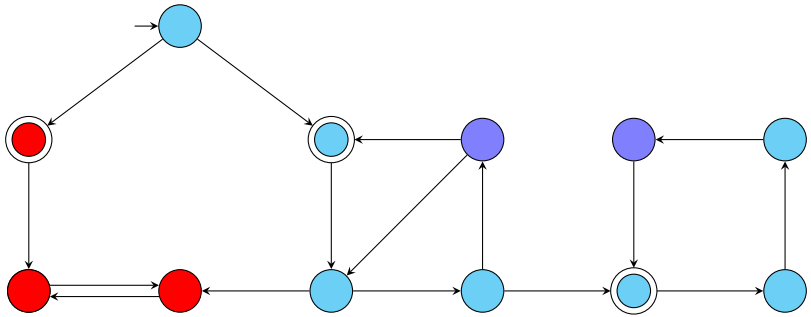
Nested DFS Example



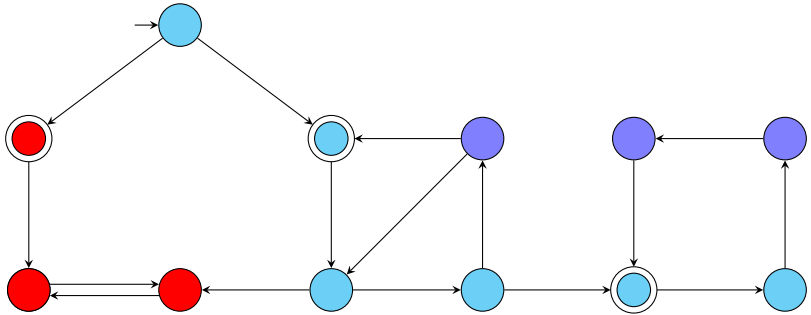
Nested DFS Example



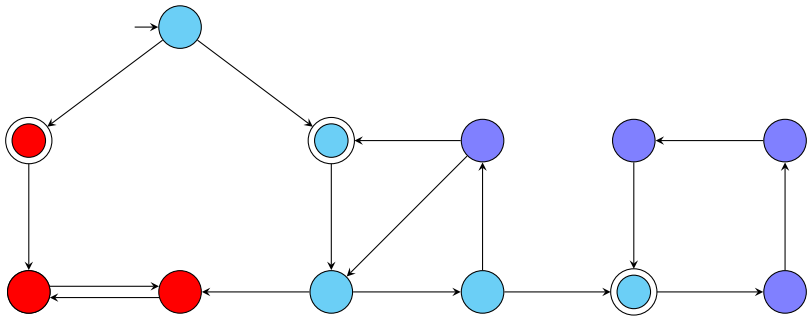
Nested DFS Example



Nested DFS Example

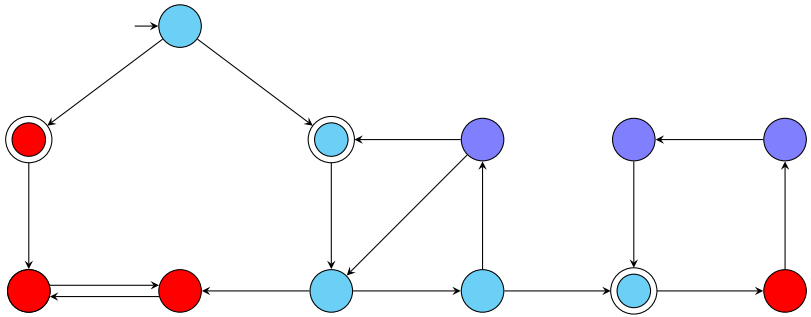


Nested DFS Example

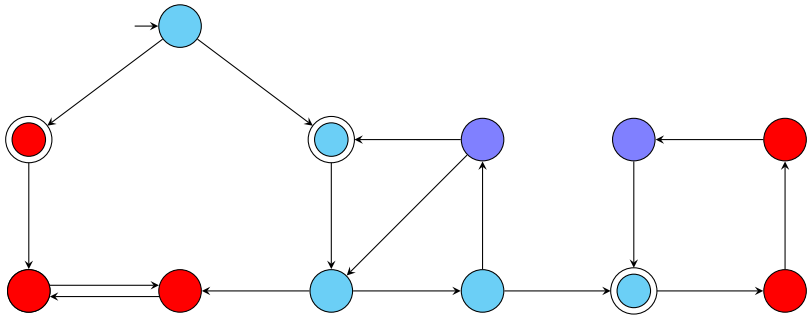


Now we start a red search after finishing the accepting state!

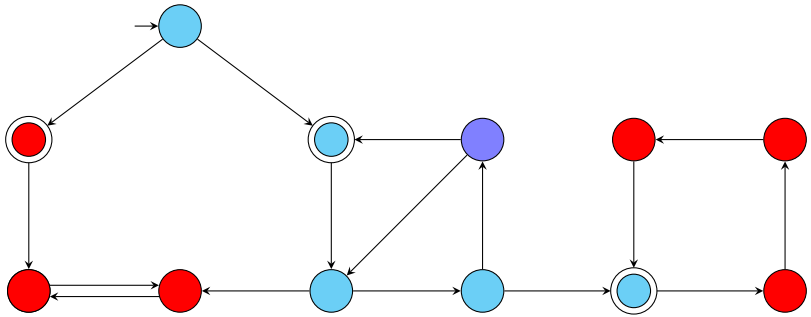
Nested DFS Example



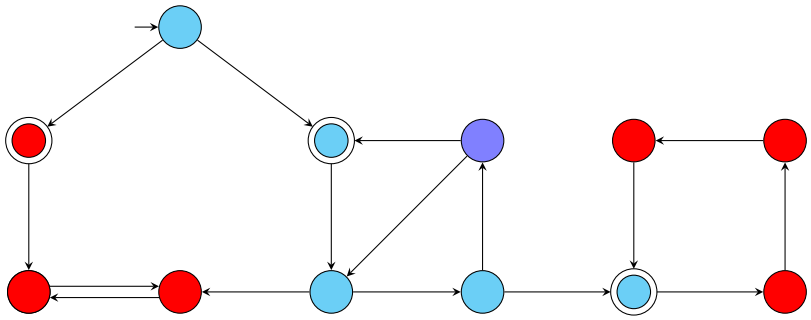
Nested DFS Example



Nested DFS Example

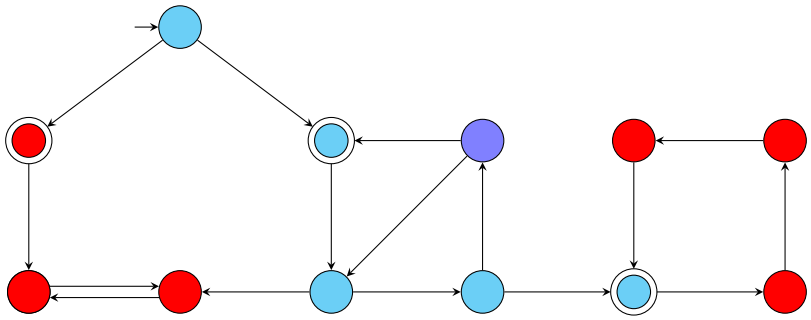


Nested DFS Example



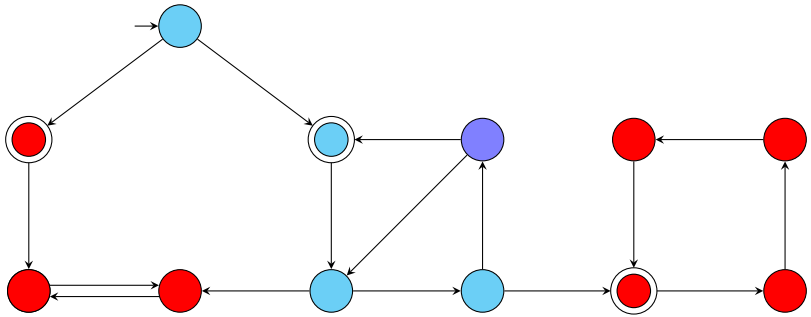
Now we found a cyan state in the red search!

Nested DFS Example

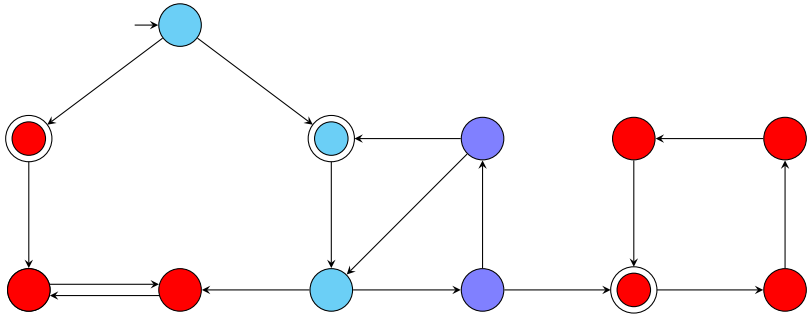


Imagine that we would continue...

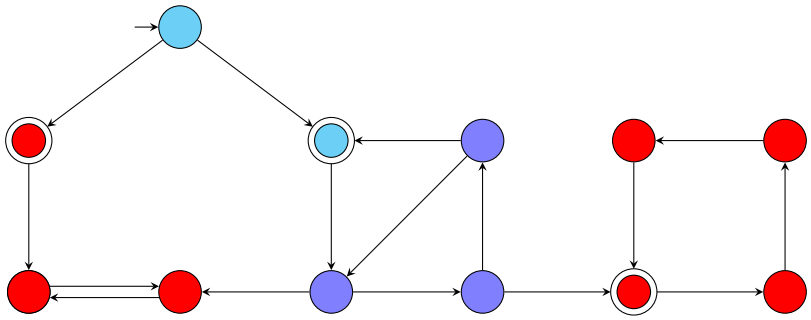
Nested DFS Example



Nested DFS Example

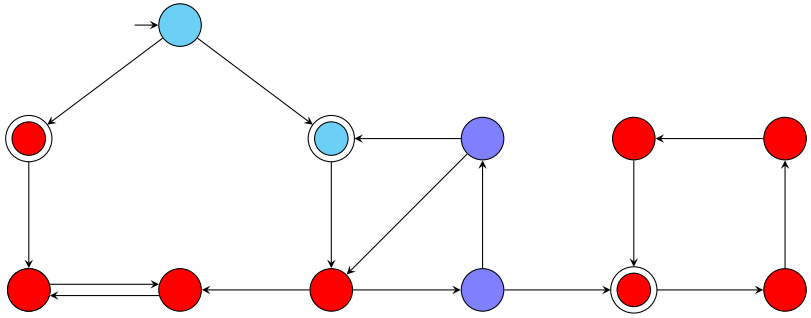


Nested DFS Example

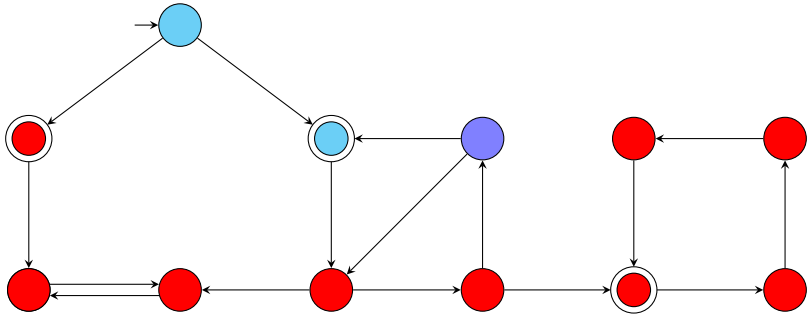


Now we start a red search after finishing the accepting state!

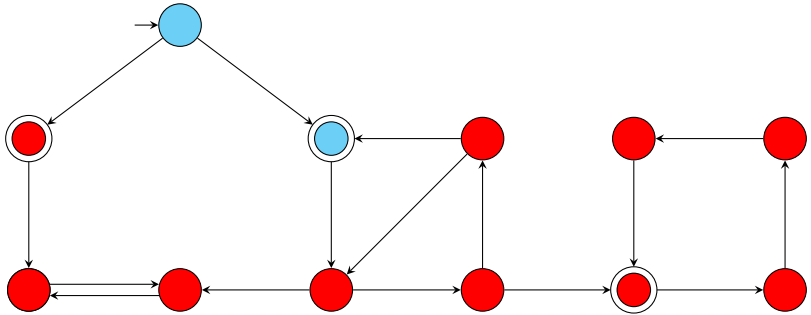
Nested DFS Example



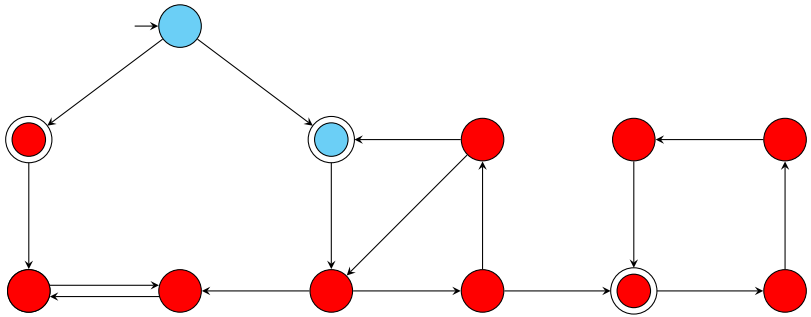
Nested DFS Example



Nested DFS Example

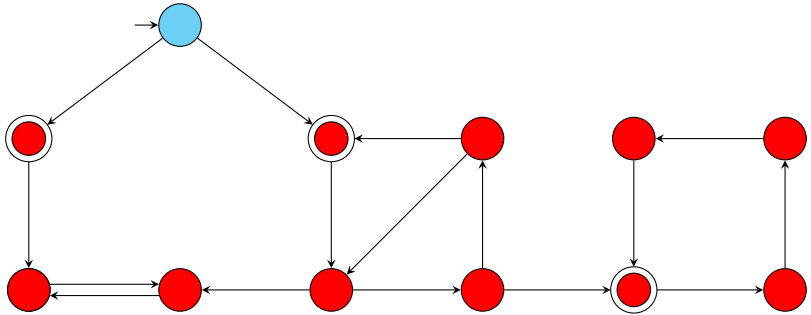


Nested DFS Example

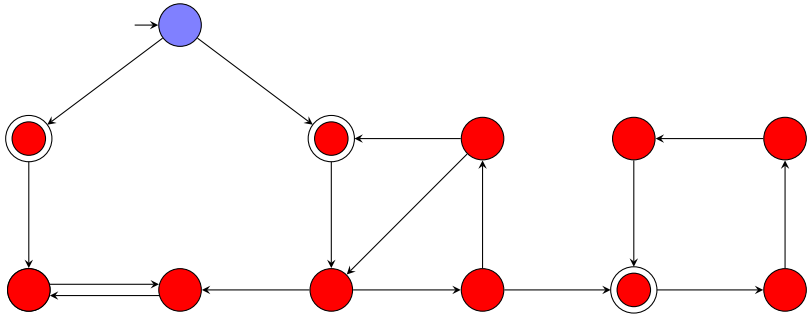


Now we found a cyan state in the red search!

Nested DFS Example



Nested DFS Example



Nested DFS

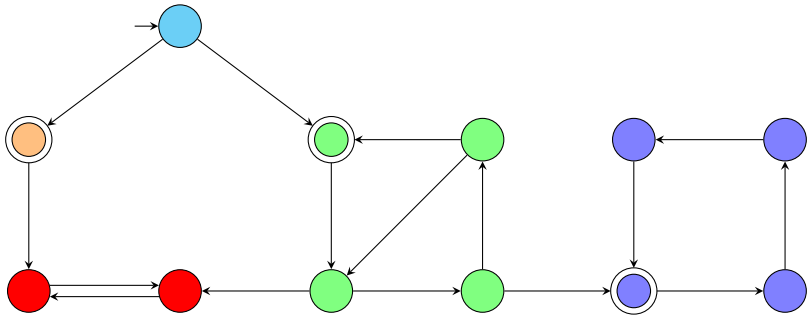
- Disadvantage: cannot handle multiple acceptance sets (GNBA)
- Alternative: find strongly connected components with accepting states

Strongly connected components (SCC)

A strongly connected component $C \subseteq S$ is a maximal subset of states such that $s \in \mathrm{Reach}(s')$ for all $s, s' \in C$.

An SCC is trivial if $|C| = 1$ and $s \not\to s$ (single state without a loop)

Strongly connected components



We are interested in the reachable non-trivial SCCs with accepting states.

Why SCC-based emptiness checking works

Key observation

An NBA is nonempty iff it has a reachable non-trivial SCC that contains an accepting state.

For generalized Büchi automata

A GNBA is nonempty iff it has a reachable non-trivial SCC that contains at least one state from each accepting set.

So what do we need algorithmically?

We need an on-the-fly way to discover SCCs efficiently during search. This leads to Tarjan's algorithm.

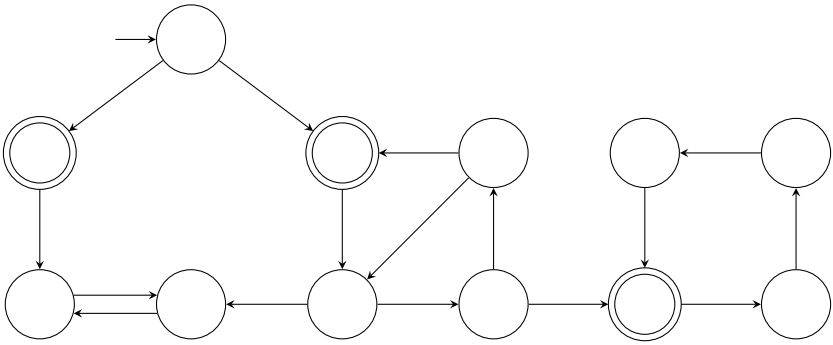
Tarjan's algorithm

- Proposed by Tarjan in 1972.
- Based on a depth-first search
- Number the states in the order they are found
- Record the lowest $\leq$-reachable state via successors

Tarjan's algorithm

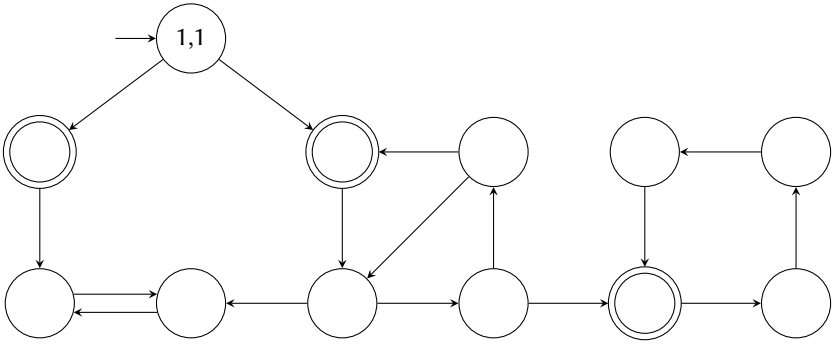
```
1  def tarjan(s):
2      global idx, stack                                   // initialize to 0, empty
3      s.index ← idx
4      s.minreach ← idx
5      idx ← idx+1
6      stack.push(s)                          // all visited states not already in an SCC
7      foreach t ∈ get-successors(s) do
8          if t.index is undefined :                                  // not yet visited
9              tarjan(t)
10             s.minreach ← min(s.minreach, t.minreach)
11         elif t ∈ stack :                                // visited but not yet in SCC
12             s.minreach ← min(s.minreach, t.minreach)
13     if s.index = s.minreach :                                    // root of a new SCC
14         repeat                                       // collect all states of the SCC
15             u ← stack.pop()
16             add u to the current SCC
17         until u = s
```

Strongly connected components



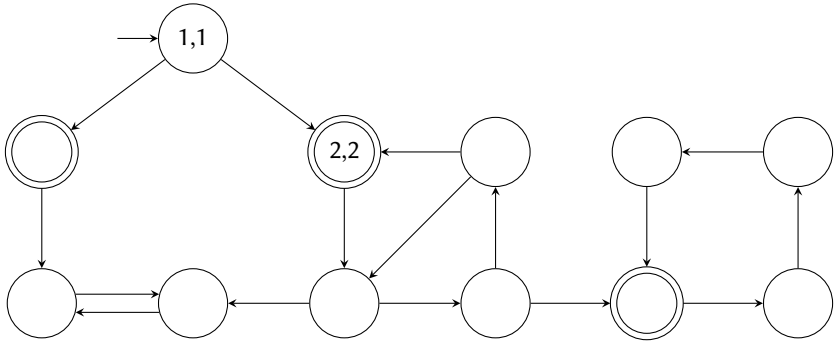
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



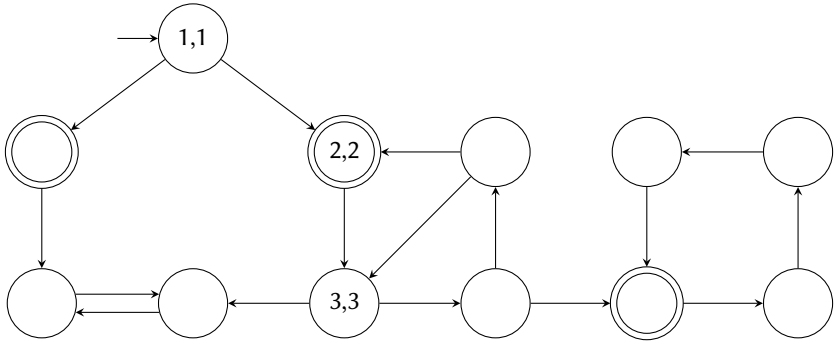
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



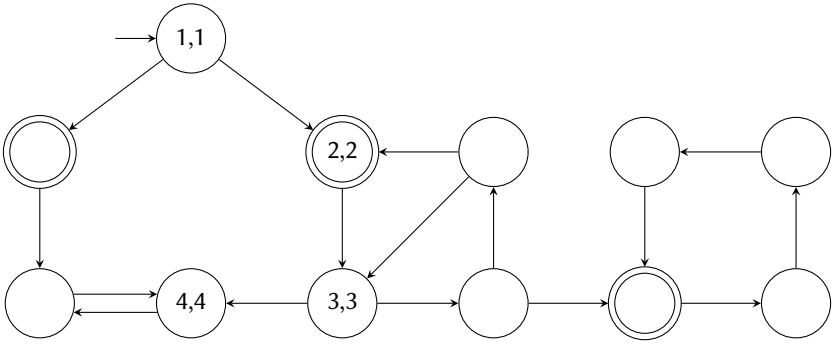
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



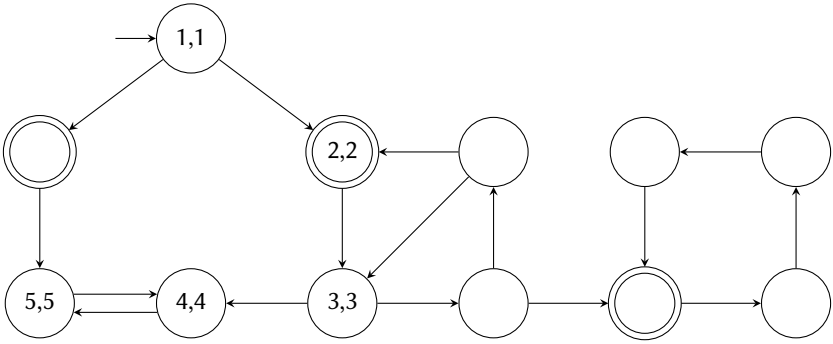
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



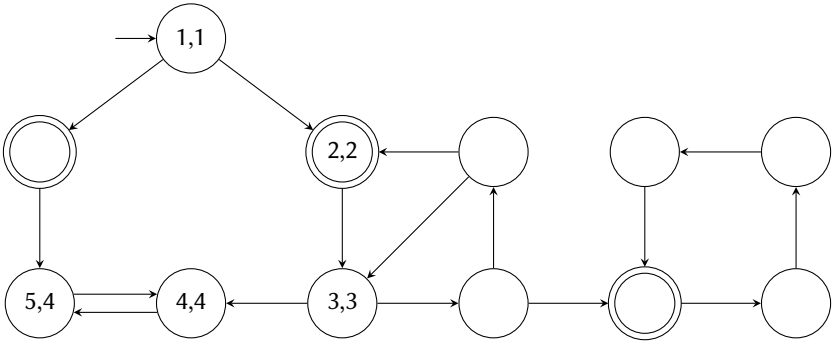
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



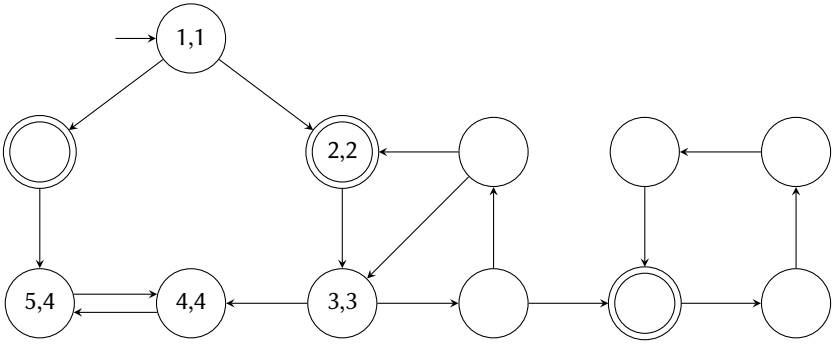
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



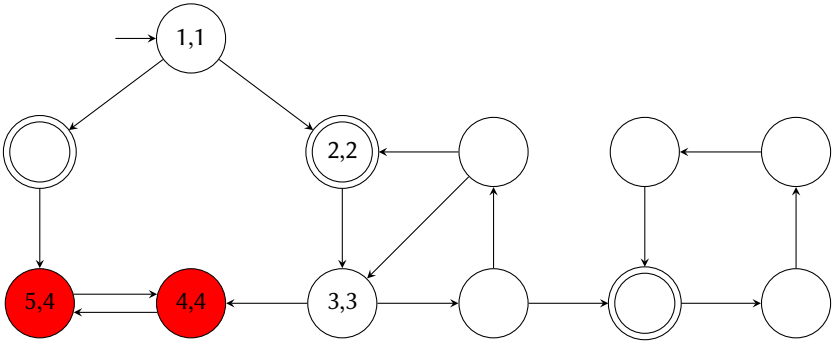
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



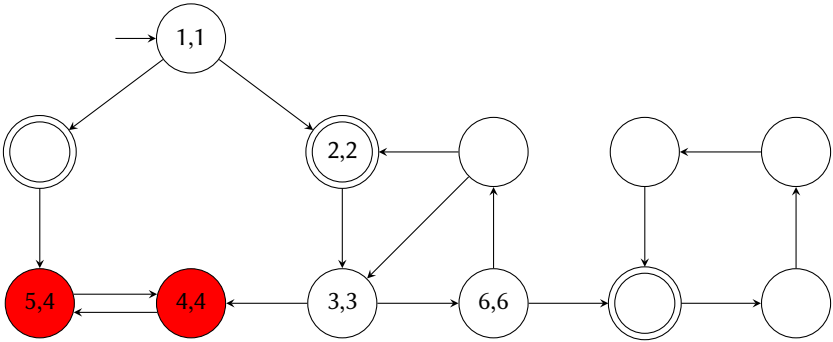
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



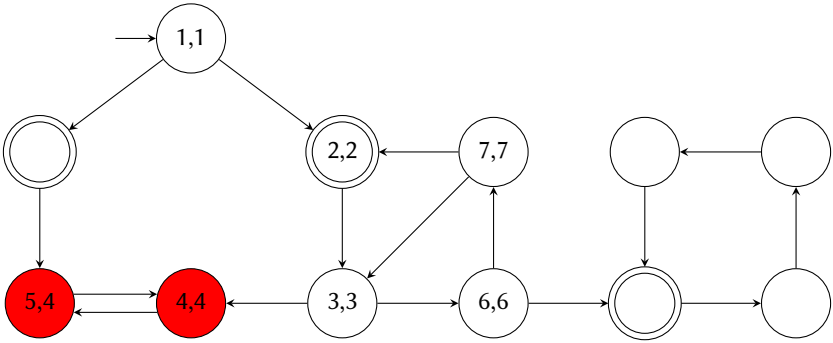
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



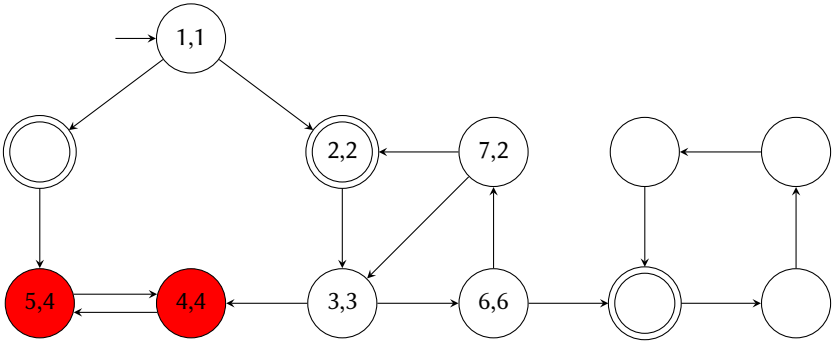
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



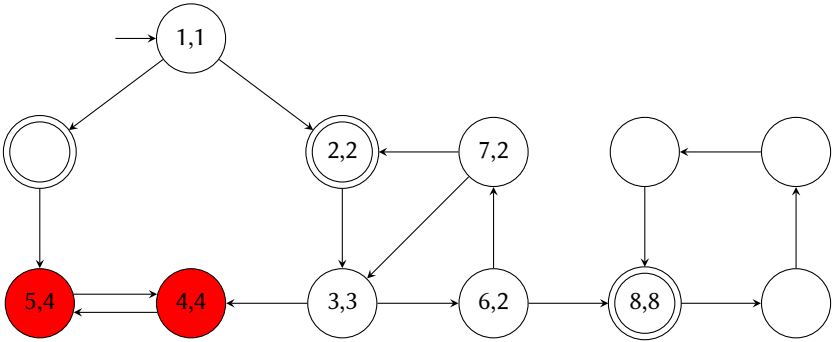
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



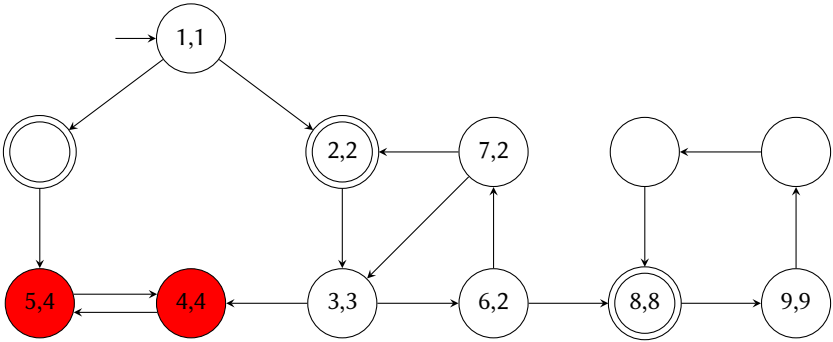
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



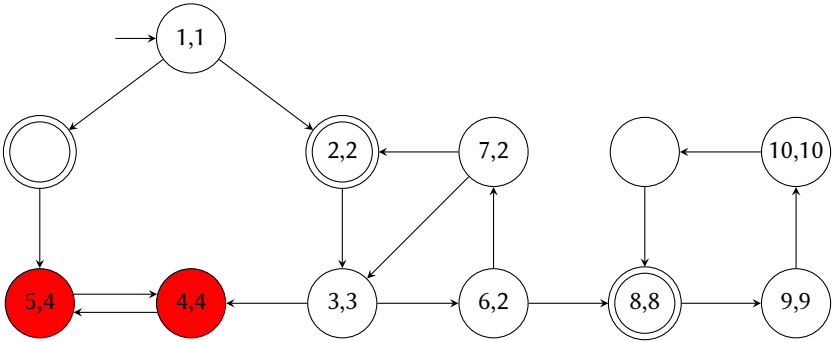
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



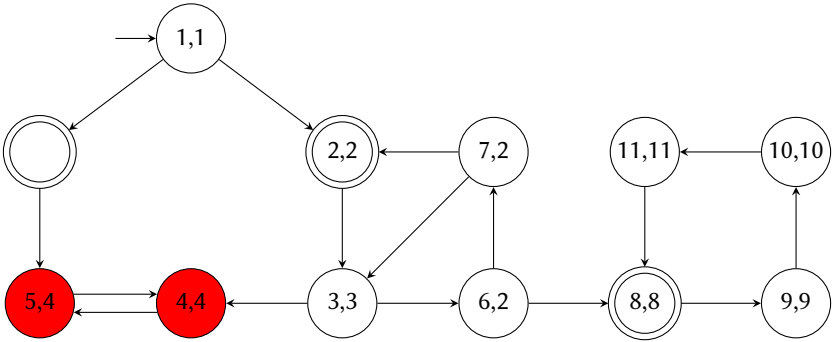
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



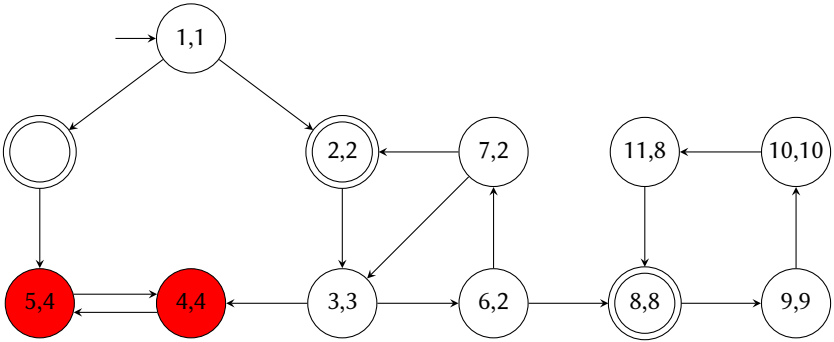
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



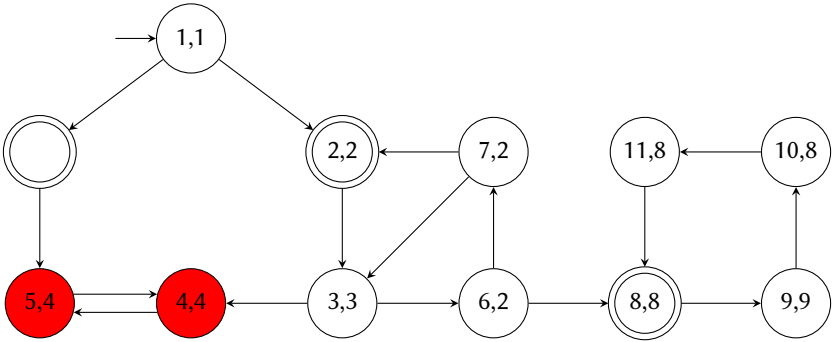
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



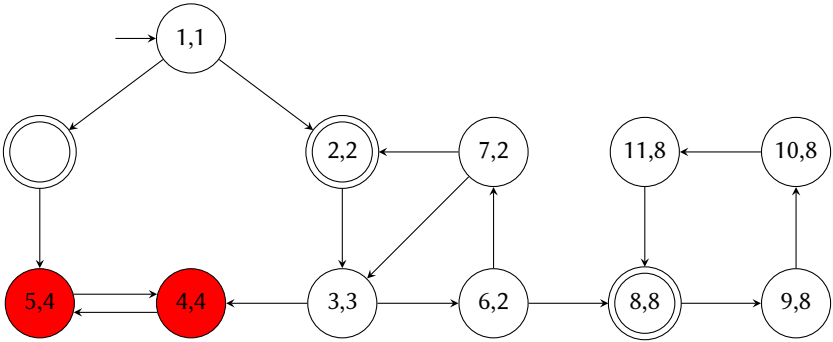
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



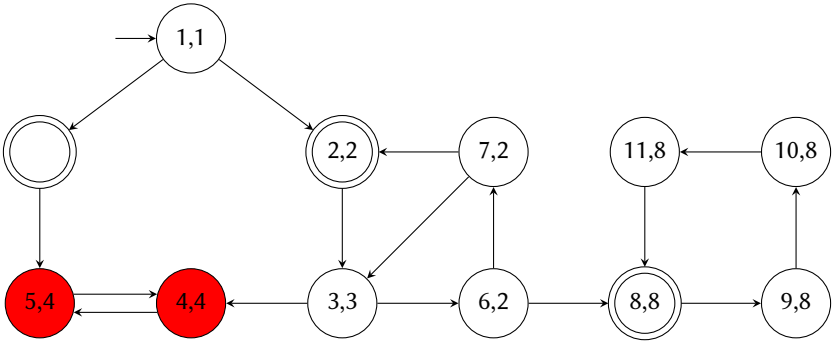
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



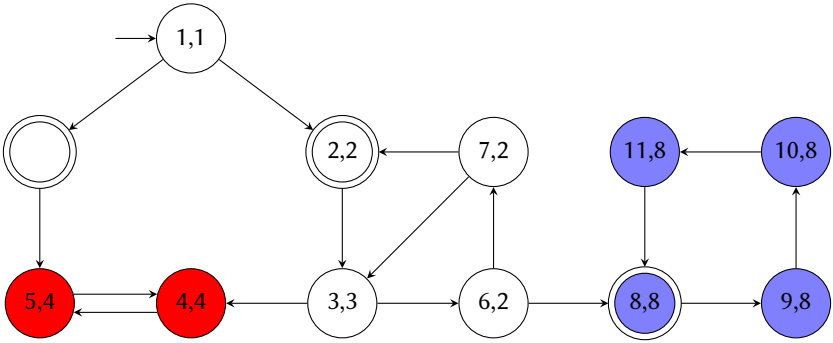
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



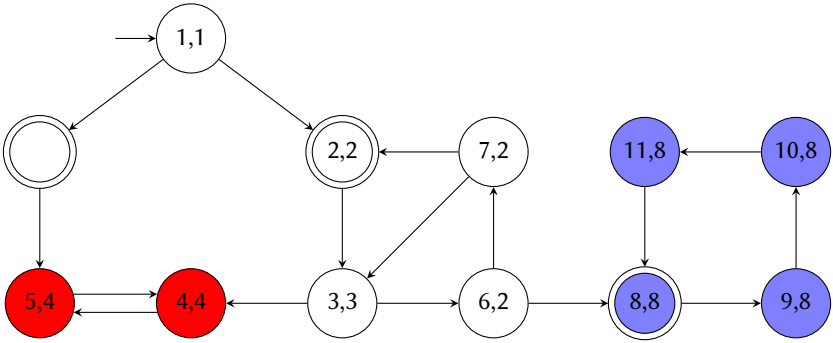
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



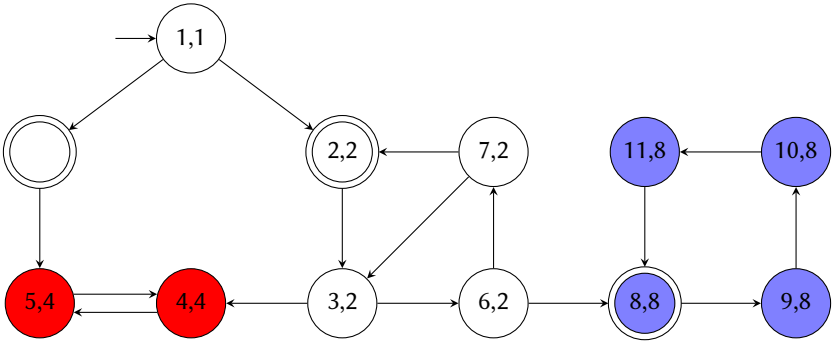
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



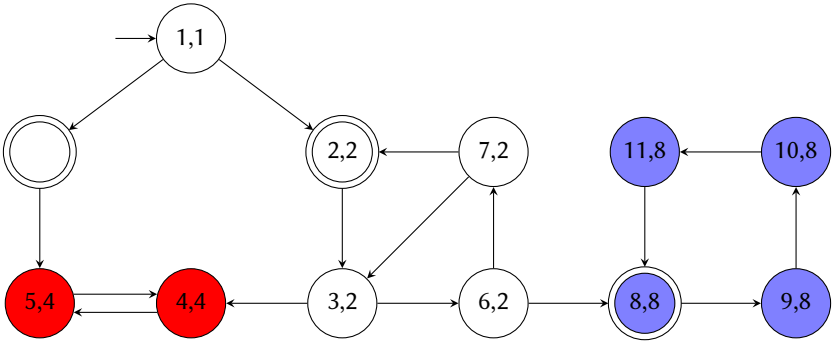
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



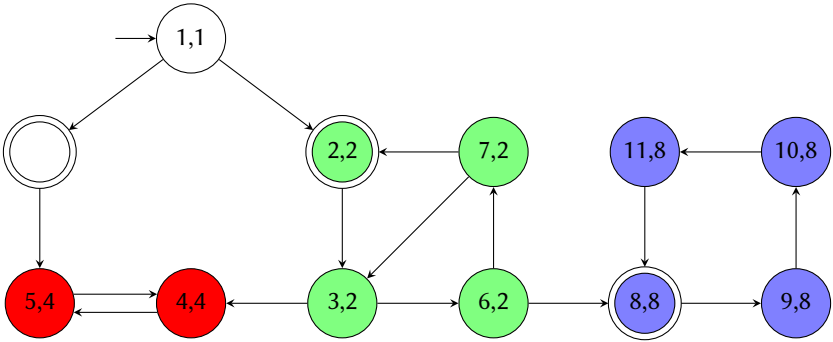
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



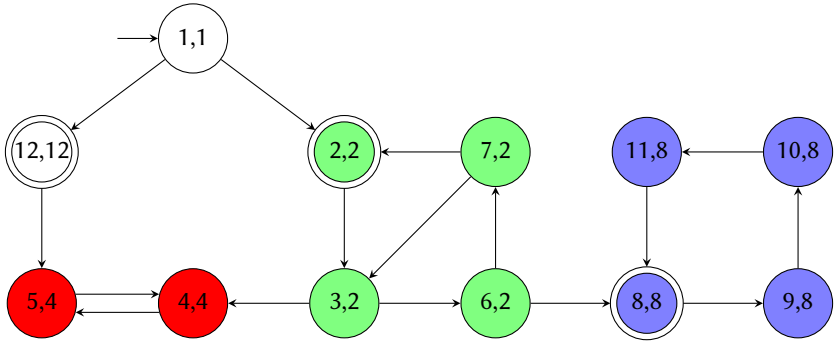
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



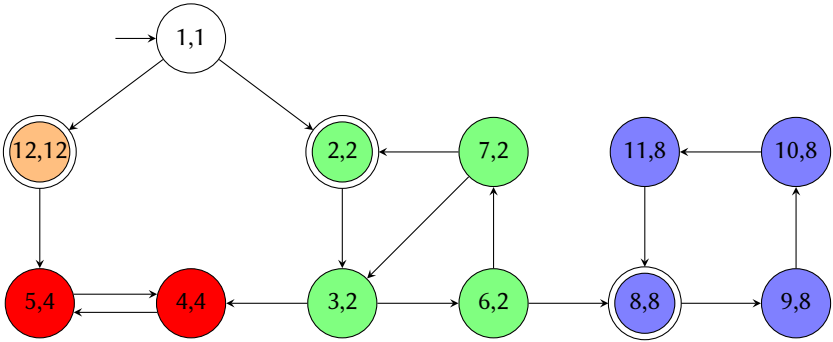
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



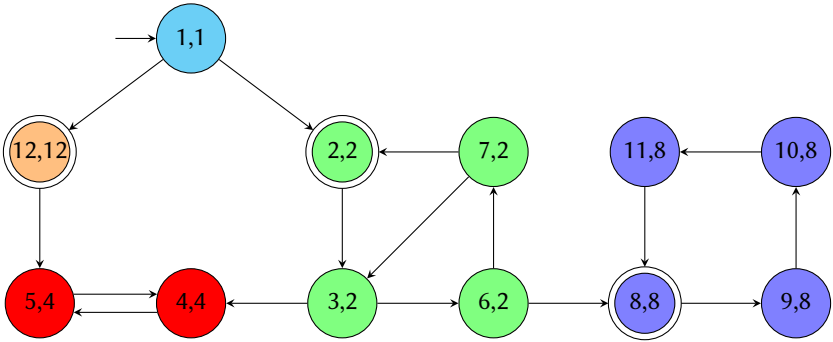
We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



We are interested in the **non-trivial** SCCs with accepting states.

Strongly connected components



We are interested in the **non-trivial** SCCs with accepting states.

Where does the complexity come from?

Formula side versus system side

- the emptiness check itself is linear in the size of the product automaton
- the expensive part is constructing the automaton for $\neg\phi$
- this automaton can have exponentially many states in $|\phi|$
- therefore the product size is roughly

$$O(|\mathrm{TS}| \cdot 2^{|\phi|})$$

Takeaway

LTL model checking is exponential in the formula, but linear in the system size once the automaton has been built.

Implementation tips for model checking

Practical structure

- represent product states lazily as pairs (s, q)
- generate successors on the fly
- do not build the full product unless necessary
- choose either:
  - nested DFS for NBA emptiness, or
  - SCC-based emptiness for NBA/GNBA
- test your result on small examples

Theory of fixed points

Fixed points

Complete lattice

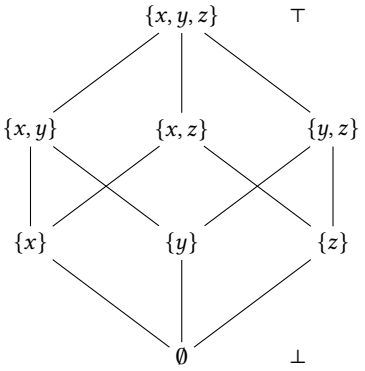
A complete lattice is a tuple $(L, \sqsubseteq)$

- L is a set
- $\sqsubseteq$ is a partial ordering of elements of L

such that every subset $S \subseteq L$ has a join and a meet

- the join of S, $\bigvee S$, is the smallest x such that $s \sqsubseteq x$ for all $s \in S$
- the meet of S, $\bigwedge S$, is the greatest x such that $x \sqsubseteq s$ for all $s \in S$
- the top $\top$ of L is $\bigvee L$
- the bottom $\bot$ of L is $\bigwedge L$

Example of a complete lattice



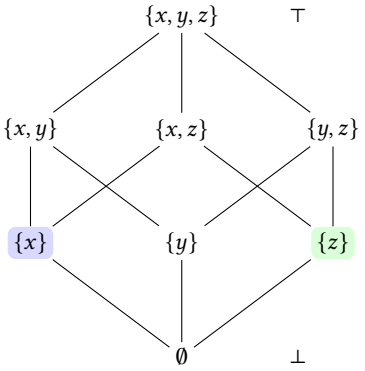
ordered by $\subseteq$

Example

We consider the powerset lattice of

$$\{x, y, z\}.$$

Example of a complete lattice



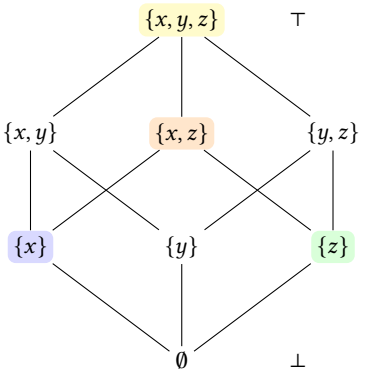
ordered by $\subseteq$

Example

Consider a set with the two elements

$$A = \{x\}, \qquad B = \{z\}.$$

Example of a complete lattice



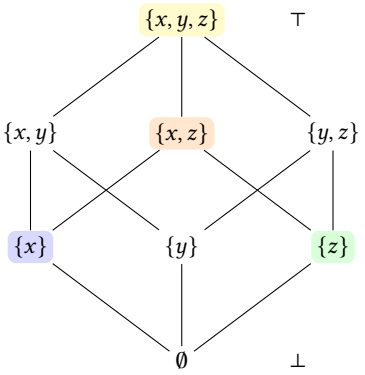
ordered by $\subseteq$

Example

Upper bounds of A and B are the sets above both:

$$\{x, z\},\ \{x, y, z\}.$$

Example of a complete lattice



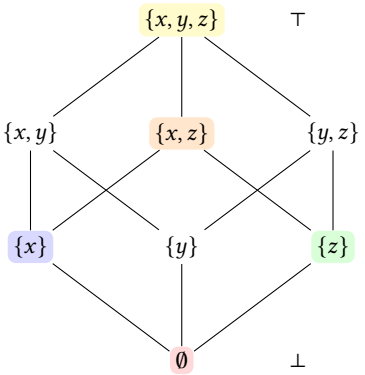
ordered by $\subseteq$

Example

So the least upper bound is

$$A \sqcup B = \{x, z\}.$$

Example of a complete lattice



ordered by $\subseteq$

Example

Lower bounds of A and B are sets below both.
Here the greatest lower bound is

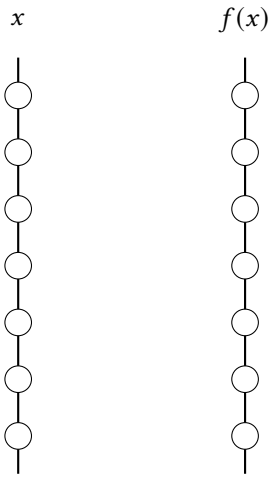
$$A \sqcap B = \emptyset.$$

Monotone functions

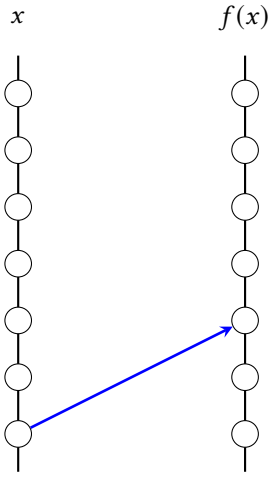
Given a complete lattice $(L, \sqsubseteq)$, a function $f : L \to L$ is called a monotone function (or order-preserving) if $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$ for all $x, y \in L$.

- if $x \sqsubseteq f(x)$, then also $f(x) \sqsubseteq f(f(x))$
- if $f(x) \sqsubseteq x$, then also $f(f(x)) \sqsubseteq f(x)$

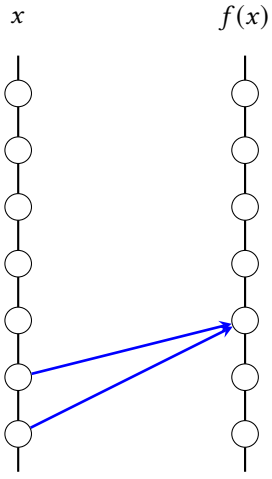
A monotone function on a lattice



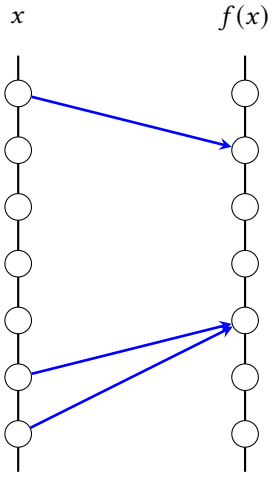
A monotone function on a lattice



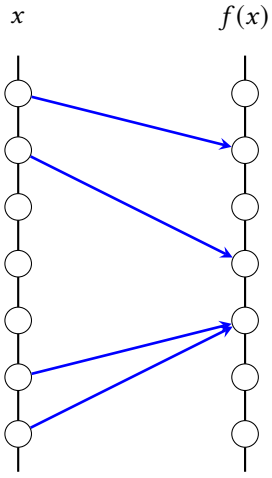
A monotone function on a lattice



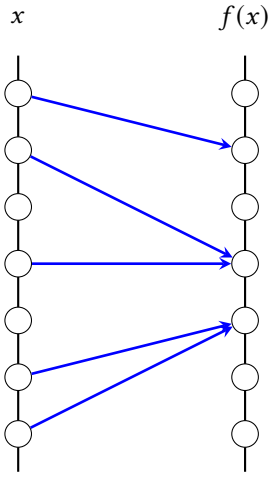
A monotone function on a lattice



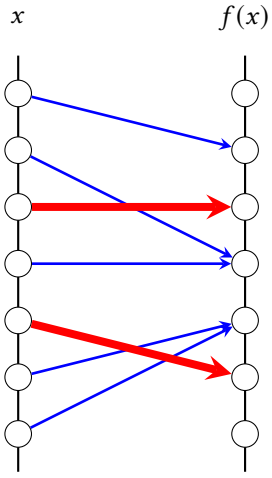
A monotone function on a lattice



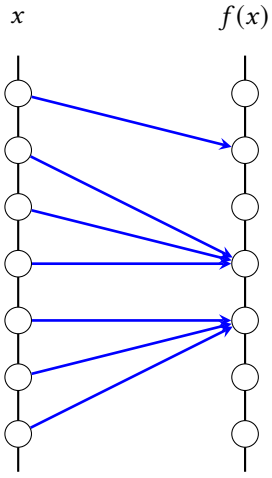
A monotone function on a lattice



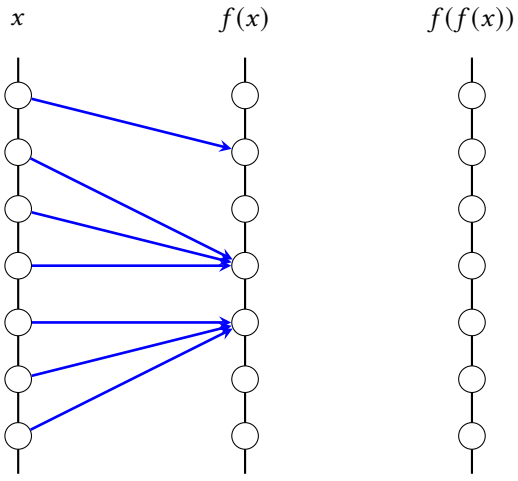
A monotone function on a lattice



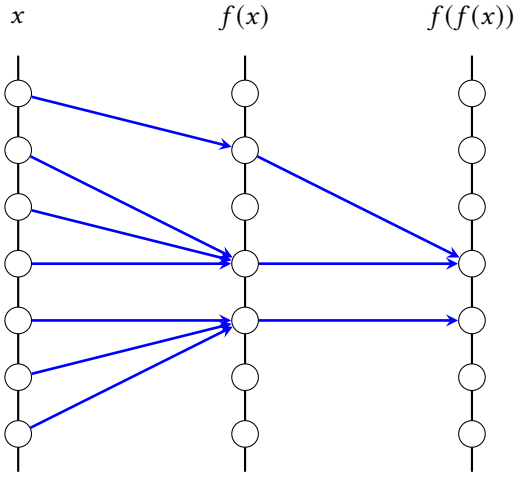
A monotone function on a lattice



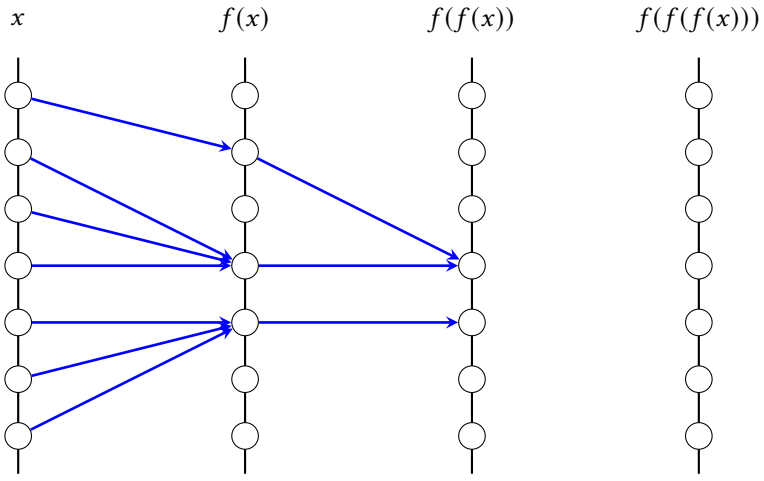
A monotone function on a lattice



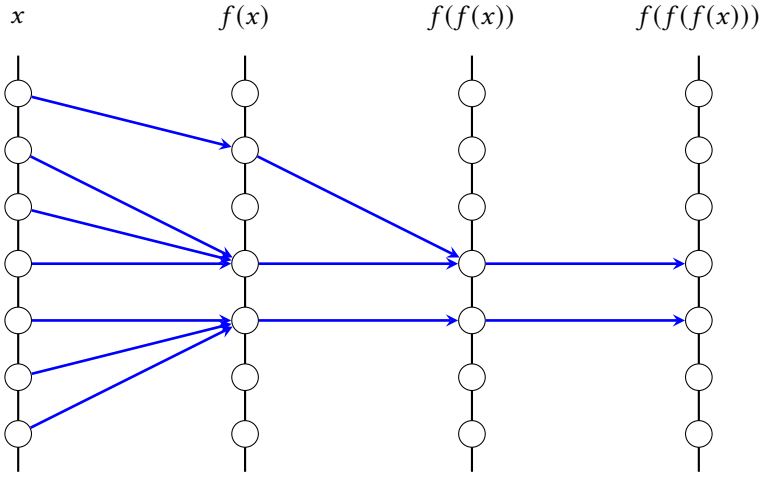
A monotone function on a lattice



A monotone function on a lattice



A monotone function on a lattice



Knaster–Tarski fixed-point theorem

Theorem

Let $(L, \sqsubseteq)$ be a complete lattice and let $f : L \to L$ be monotone. Then f has:

- a unique least fixed point, written $\mu Z.\, f(Z)$
- a unique greatest fixed point, written $\nu Z.\, f(Z)$

Proof idea of Knaster–Tarski

Greatest fixed point

Let $D = \{x \in L \mid x \sqsubseteq f(x)\}$, the set of post-fixed points, and let $u = \bigvee D$.

- for every $x \in D$, we have $x \sqsubseteq u$
- by monotonicity, $f(x) \sqsubseteq f(u)$
- since $x \sqsubseteq f(x)$, this gives $x \sqsubseteq f(u)$
- so $f(u)$ is an upper bound of D, hence $u \sqsubseteq f(u)$
- therefore $u \in D$, so also $f(u) \in D$
- but then $f(u) \sqsubseteq u$, because $u = \bigvee D$

So $u = f(u)$: it is a fixed point. Every fixed point x satisfies $x = f(x)$, hence $x \sqsubseteq f(x)$, so $x \in D$. Therefore $x \sqsubseteq u$. Thus u is the greatest fixed point.

Proof idea of Knaster–Tarski

Greatest fixed point

Let $D = \{x \in L \mid x \sqsubseteq f(x)\}$, the set of post-fixed points, and let $u = \bigvee D$.

- for every $x \in D$, we have $x \sqsubseteq u$
- by monotonicity, $f(x) \sqsubseteq f(u)$
- since $x \sqsubseteq f(x)$, this gives $x \sqsubseteq f(u)$
- so $f(u)$ is an upper bound of D, hence $u \sqsubseteq f(u)$
- therefore $u \in D$, so also $f(u) \in D$
- but then $f(u) \sqsubseteq u$, because $u = \bigvee D$

So $u = f(u)$: it is a fixed point. Every fixed point x satisfies $x = f(x)$, hence $x \sqsubseteq f(x)$, so $x \in D$. Therefore $x \sqsubseteq u$. Thus u is the greatest fixed point.

Least fixed point

Dually, let $E = \{x \in L \mid f(x) \sqsubseteq x\}$, the set of pre-fixed points, and let $w = \bigwedge E$.
Then w is the least fixed point.

What else does Knaster–Tarski say?

Stronger conclusion

Knaster–Tarski does not only say that least and greatest fixed points exist.

It says that all fixed points of a monotone function together form a complete lattice.

Fixed points in verification

Usually we mainly care about:

- least fixed points (μ): build up good states
- greatest fixed points (ν): remove bad states

Why do the fixed points form a complete lattice?

Proof sketch

Let $P = \{x \in L \mid f(x) = x\}$ be the set of fixed points, and take any subset $E \subseteq P$.
We want to show that E has a join and a meet inside P.

- Let $u = \bigvee E$ be the join of E in the original lattice L.
- If $u \in P$, then we are done: u is already the join of E in P.
- Otherwise, consider the set $Z = \{x \in L \mid u \sqsubseteq x\}$, of all elements above u.
- Since every $e \in E$ is a fixed point ($e = f(e)$) and $e \sqsubseteq u \sqsubseteq z$ for every $z \in Z$, monotonicity gives $f(e) \sqsubseteq f(z)$, so $f(z)$ is again an upper bound of E. Hence $f(Z) \subseteq Z$.
- The set Z is itself a complete lattice, with bottom element u and top element $\top$: for every $Y \subseteq Z$, both $\bigvee Y$ and $\bigwedge Y$ are still above u.
- Thus we may restrict f to Z, and apply Knaster–Tarski inside Z: there is a least fixed point $z \in Z$.
- This fixed point z is above all elements of E, and every fixed point above E lies in Z. Therefore z is the smallest fixed point above E, so it is the join of E in P.

Dually, one finds the meet of E in P. Therefore the fixed points themselves form a complete lattice.

Kleene fixed-point theorem

Kleene fixed-point theorem on finite lattices

If L is finite and f is monotone, then:

- $\text{lfp}(f) = f^{\infty}(\bot)$ is obtained by starting from $\bot$ and repeatedly applying f
- $\text{gfp}(f) = f^{\infty}(\top)$ is obtained by starting from $\top$ and repeatedly applying f

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \ldots \sqsubseteq \text{lfp}(f)$$
$$\sqsubseteq \text{gfp}(f) \sqsubseteq \ldots \sqsubseteq f(f(\top)) \sqsubseteq f(\top) \sqsubseteq \top$$

Kleene fixed-point theorem

Kleene fixed-point theorem on finite lattices

If L is finite and f is monotone, then:

- $\text{lfp}(f) = f^{\infty}(\bot)$ is obtained by starting from $\bot$ and repeatedly applying f
- $\text{gfp}(f) = f^{\infty}(\top)$ is obtained by starting from $\top$ and repeatedly applying f

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \ldots \sqsubseteq \text{lfp}(f)$$
$$\sqsubseteq \text{gfp}(f) \sqsubseteq \ldots \sqsubseteq f(f(\top)) \sqsubseteq f(\top) \sqsubseteq \top$$

Why is this so?

Let x be any fixed point. Since $\bot \sqsubseteq x$, monotonicity gives $f(\bot) \sqsubseteq f(x) = x$, and by induction $f^i(\bot) \sqsubseteq x$ for all i.

Intuition: monotonicity forbids the "crossing arrows", so iteration can only move upward from $\bot$ and downward from $\top$.

Why fixed points matter

Fixed points appear everywhere in verification

- reachable states are a least fixed point;
- temporal operators such as **F**, **G**, **U**, and **R** can be seen as least or greatest fixed points;
- the modal μ-calculus is built from nested least and greatest fixed points;
- parity-game algorithms can be understood in fixed-point terms.

Fixed points already appeared in LTL

Recall the expansion laws

$$\mathbf{F}\,\varphi \equiv \varphi \vee \mathbf{X}\,\mathbf{F}\,\varphi \qquad \mathbf{G}\,\varphi \equiv \varphi \wedge \mathbf{X}\,\mathbf{G}\,\varphi$$

$$\varphi\ \mathbf{U}\ \psi \equiv \psi \vee (\varphi \wedge \mathbf{X}\,\varphi\ \mathbf{U}\ \psi) \qquad \varphi\ \mathbf{R}\ \psi \equiv \psi \wedge (\varphi \vee \mathbf{X}\,\varphi\ \mathbf{R}\ \psi)$$

What this means

These operators are defined recursively:

- **F**, **U** and **M** correspond to least fixed points
- **G**, **W** and **R** correspond to greatest fixed points

Why is F p a least fixed point?

$$\mathbf{F}\, p \equiv p \vee \mathbf{X}\, \mathbf{F}\, p$$

A path satisfies **F** p if:

- either p already holds now,
- or **F** p must hold from the next step onward.

Why least fixed point?

We can build the satisfying situations step by step:

- start with paths where p already holds now;
- then add paths whose next step is already known to satisfy **F** p;
- repeat until nothing new can be added.

So **F** p is obtained by starting from nothing and adding justified cases.

$$\mathbf{F}\, p \;=\; \mu X.\big(p \vee \mathbf{X}\, X\big)$$

Why is G p a greatest fixed point?

$$\mathbf{G}\, p \equiv p \wedge \mathbf{X}\, \mathbf{G}\, p$$

A path satisfies **G** p if:

- p holds now, and
- **G** p still holds from the next step onward.

Why greatest fixed point?

We can compute the satisfying situations by elimination:

- start with all paths as candidates;
- remove those where p already fails now;
- remove those whose next step no longer satisfies **G** p;
- repeat until nothing more must be removed.

So **G** p is obtained by starting from everything and removing bad cases.

$$\mathbf{G}\, p \;=\; \nu X.(p \wedge \mathbf{X}\, X)$$

Least versus greatest fixed point intuition

Two different modes of reasoning

- Least fixed points start from nothing and add justified states
  - examples: **F**, **U**, **M**, reachability
- Greatest fixed points start from everything and remove unjustified states
  - examples: **G**, **W**, **R**, invariance

Why this matters

This same distinction comes back in:

- the modal μ-calculus;
- parity-game algorithms;
- verification algorithms based on fixed-point iteration.

Modal μ-calculus

Modal μ-calculus

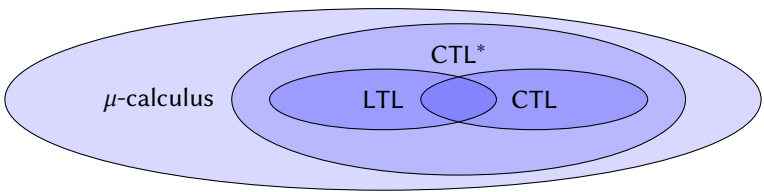
In this chapter

- what the modal μ-calculus is
- how fixed points extend modal logic
- how CTL fits inside the μ-calculus
- why LTL does not embed directly, but via ω-automata
- how automata, parity conditions, and μ-calculus connect
- how μ-calculus model checking is done in practice

μ-calculus: syntax and semantics

Idea of μ-calculus: add fixed point operators to basic modal logic.

- μ-calculus is very expressive
  contains CTL as a fragment; closely connected to temporal logics, automata, and parity games
- μ-calculus is very pure
  "assembly language" for modal logic, cf: λ-calculus for functional programming
- drawback: lack of intuition
- μ-calculus is used in practical model checkers: μCRL, mCRL2, CADP, LTSmin



μ-calculus: syntax and semantics

Elements of the language

- Constants tt and ff
- (Negated) propositions p and $\neg p$
- Operators conjunction $\wedge$ and disjunction $\vee$
- Modalities box / universal $\Box$ and diamond / existential $\Diamond$
- Variables $X \in Var$ representing sets of states
- Fixed point operators lfp $\mu X.\phi$ and gfp $\nu X.\phi$

Some notation and terminology

- The semantics of a μ-calculus formula ϕ is a set of states
- A valuation $\mathcal{V} : Var \to 2^S$ interprets variables as sets of states.
- $\mathcal{V}[X := Q]$ is a valuation like $\mathcal{V}$, but X is set to Q

μ-calculus: syntax and semantics

Syntax

$\phi, \psi ::= tt \mid ff \mid p \mid \neg p \mid \phi \wedge \psi \mid \phi \vee \psi \mid [a]\phi \mid \langle a \rangle \phi \mid X \mid \mu X.\phi \mid \nu X.\phi$

Semantics for a transition system $\mathcal{M} := (S, \rightarrow, L)$

$$
\begin{array}{lcl}
[\![tt]\!]^{\mathcal{M}} & = & S \\
[\![ff]\!]^{\mathcal{M}} & = & \emptyset \\
[\![p]\!]^{\mathcal{M}} & = & \{s \in S \mid p \in L(s)\} \\
[\![\neg p]\!]^{\mathcal{M}} & = & \{s \in S \mid p \notin L(s)\} \\
[\![\phi \vee \psi]\!]^{\mathcal{M}} & = & [\![\phi]\!]^{\mathcal{M}} \cup [\![\psi]\!]^{\mathcal{M}} \\
[\![\phi \wedge \psi]\!]^{\mathcal{M}} & = & [\![\phi]\!]^{\mathcal{M}} \cap [\![\psi]\!]^{\mathcal{M}}
\end{array}
$$

(notice that there is no negation on formulae, only on the propositions)

μ-calculus: syntax and semantics

Syntax

$\phi, \psi ::= tt \mid ff \mid p \mid \neg p \mid \phi \wedge \psi \mid \phi \vee \psi \mid [a]\phi \mid \langle a \rangle \phi \mid X \mid \mu X.\phi \mid \nu X.\phi$

Semantics for a transition system $\mathcal{M} := (S, \rightarrow, L)$

$$
\begin{array}{lcl}
[\![tt]\!]^{\mathcal{M}} & = & S \\
[\![ff]\!]^{\mathcal{M}} & = & \emptyset \\
[\![p]\!]^{\mathcal{M}} & = & \{s \in S \mid p \in L(s)\} \\
[\![\neg p]\!]^{\mathcal{M}} & = & \{s \in S \mid p \notin L(s)\} \\
[\![\phi \vee \psi]\!]^{\mathcal{M}} & = & [\![\phi]\!]^{\mathcal{M}} \cup [\![\psi]\!]^{\mathcal{M}} \\
[\![\phi \wedge \psi]\!]^{\mathcal{M}} & = & [\![\phi]\!]^{\mathcal{M}} \cap [\![\psi]\!]^{\mathcal{M}} \\
[\![[a]\phi]\!]^{\mathcal{M}} & = & \{s \in S \mid \forall t.(s \xrightarrow{a} t) \rightarrow (t \in [\![\phi]\!]^{\mathcal{M}})\} \\
[\![\langle a \rangle \phi]\!]^{\mathcal{M}} & = & \{s \in S \mid \exists t.(s \xrightarrow{a} t) \wedge (t \in [\![\phi]\!]^{\mathcal{M}})\}
\end{array}
$$

μ-calculus: syntax and semantics

Syntax

$\phi, \psi ::= tt \mid ff \mid p \mid \neg p \mid \phi \wedge \psi \mid \phi \vee \psi \mid [a]\phi \mid \langle a \rangle \phi \mid X \mid \mu X.\phi \mid \nu X.\phi$

Semantics for a transition system $\mathcal{M} := (S, \rightarrow, L)$

$$
\begin{array}{lcll}
[\![tt]\!]^{\mathcal{M}} & = & S & \\
[\![ff]\!]^{\mathcal{M}} & = & \emptyset & \\
[\![p]\!]^{\mathcal{M}} & = & \{s \in S \mid p \in L(s)\} & \\
[\![\neg p]\!]^{\mathcal{M}} & = & \{s \in S \mid p \notin L(s)\} & \\
[\![\phi \vee \psi]\!]^{\mathcal{M}}_{\mathcal{V}} & = & [\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}} \cup [\![\psi]\!]^{\mathcal{M}}_{\mathcal{V}} & \\
[\![\phi \wedge \psi]\!]^{\mathcal{M}}_{\mathcal{V}} & = & [\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}} \cap [\![\psi]\!]^{\mathcal{M}}_{\mathcal{V}} & \\
[\![[a]\phi]\!]^{\mathcal{M}}_{\mathcal{V}} & = & \{s \in S \mid \forall t.(s \xrightarrow{a} t) \rightarrow (t \in [\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}})\} & \\
[\![\langle a \rangle \phi]\!]^{\mathcal{M}}_{\mathcal{V}} & = & \{s \in S \mid \exists t.(s \xrightarrow{a} t) \wedge (t \in [\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}})\} & \\
[\![X]\!]^{\mathcal{M}}_{\mathcal{V}} & = & \mathcal{V}(X) & \\
[\![\mu X.\phi]\!]^{\mathcal{M}}_{\mathcal{V}} & = & \bigcap \{S' \subseteq S \mid [\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}[S'/X]} \subseteq S'\} & \text{(lfp)} \\
[\![\nu X.\phi]\!]^{\mathcal{M}}_{\mathcal{V}} & = & \bigsqcup \{S' \subseteq S \mid S' \subseteq [\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}[S'/X]}\} & \text{(gfp)}
\end{array}
$$

where $\mathcal{V}$: Var $\rightarrow 2^S$ assigns a set of states to the variables $X, Y, \ldots$

μ-calculus: syntax and semantics

Syntax

$\phi, \psi ::= tt \mid ff \mid p \mid \neg p \mid \phi \wedge \psi \mid \phi \vee \psi \mid [a]\phi \mid \langle a \rangle \phi \mid X \mid \mu X.\phi \mid \nu X.\phi$

Semantics for a transition system $\mathcal{M} := (S, \rightarrow, L)$

$$
\begin{array}{lcl}
[\![tt]\!]^{\mathcal{M}} & = & S \\
[\![ff]\!]^{\mathcal{M}} & = & \emptyset \\
[\![p]\!]^{\mathcal{M}} & = & \{s \in S \mid p \in L(s)\} \\
[\![\neg p]\!]^{\mathcal{M}} & = & \{s \in S \mid p \notin L(s)\} \\
[\![\phi \vee \psi]\!]^{\mathcal{M}}_{\mathcal{V}} & = & [\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}} \cup [\![\psi]\!]^{\mathcal{M}}_{\mathcal{V}} \\
[\![\phi \wedge \psi]\!]^{\mathcal{M}}_{\mathcal{V}} & = & [\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}} \cap [\![\psi]\!]^{\mathcal{M}}_{\mathcal{V}} \\
[\![[a]\phi]\!]^{\mathcal{M}}_{\mathcal{V}} & = & \{s \in S \mid \forall t.(s \xrightarrow{a} t) \rightarrow (t \in [\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}})\} \\
[\![\langle a \rangle \phi]\!]^{\mathcal{M}}_{\mathcal{V}} & = & \{s \in S \mid \exists t.(s \xrightarrow{a} t) \wedge (t \in [\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}})\} \\
[\![X]\!]^{\mathcal{M}}_{\mathcal{V}} & = & \mathcal{V}(X) \\
[\![\mu X.\phi]\!]^{\mathcal{M}}_{\mathcal{V}} & = & \mathrm{fp}(X := [\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}[X \leftarrow X]}, \emptyset) \\
[\![\nu X.\phi]\!]^{\mathcal{M}}_{\mathcal{V}} & = & \mathrm{fp}(X := [\![\phi]\!]^{\mathcal{M}}_{\mathcal{V}[X \leftarrow X]}, S)
\end{array}
$$

where $\mathcal{V}\colon \mathrm{Var} \rightarrow 2^S$ assigns a set of states to the variables $X, Y, \ldots$

μ-calculus example: $\mu X.[a]X$

$$\mu X.[a]X$$

represents states from which there is no infinite path of a-transitions.

$$\mu^0 X.[a]X = \emptyset$$
$$\mu^1 X.[a]X = [a]\emptyset \quad \text{states with no outgoing } a\text{-transition}$$
$$\mu^2 X.[a]X = [a](\mu^1 X.[a]X) \quad \text{states with no } aa\text{-path}$$
$$\mu^3 X.[a]X = [a](\mu^2 X.[a]X) \quad \text{states with no } aaa\text{-path}$$

So the least fixed point collects exactly the states from which every a-path is finite.

μ-calculus example: $\nu X.(p \wedge [a]X)$

$$\nu X.p \wedge [a]X$$

represents states from which p holds now and continues to hold forever along all a-steps.

$$\nu^0 X.(p \wedge [a]X) = S$$
$$\nu^1 X.(p \wedge [a]X) = p \wedge [a]S = p$$
$$\nu^2 X.(p \wedge [a]X) = p \wedge [a](p)$$
$$\nu^3 X.(p \wedge [a]X) = p \wedge [a](p \wedge [a]p)$$

So the greatest fixed point repeatedly removes states that cannot maintain p forever.

Analogous to **AG** p.

μ-calculus example: $\mu X.(p \vee (\langle a \rangle tt \wedge [a]X))$

$$\mu X.p \vee (\langle a \rangle tt \wedge [a]X)$$

represents states from which p is inevitable along all a-paths.

$$\mu^0 X.(p \vee (\langle a \rangle tt \wedge [a]X)) = \emptyset$$
$$\mu^1 X.(p \vee (\langle a \rangle tt \wedge [a]X)) = p$$
$$\mu^2 X.(p \vee (\langle a \rangle tt \wedge [a]X)) = p \vee (\langle a \rangle tt \wedge [a]p)$$
$$\mu^3 X.(p \vee (\langle a \rangle tt \wedge [a]X)) = p \vee \big(\langle a \rangle tt \wedge [a](p \vee (\langle a \rangle tt \wedge [a]p))\big)$$

So the least fixed point grows from states where p already holds, and then adds states whose a-successors are already safe.

Analogous to **AF** p (on total systems).

μ-calculus: examples

Examples

- $\nu X.p \wedge \Box X$ — analogous to **AG** p
- $\nu X.p \wedge \Diamond X$ — analogous to **EG** p
- $\mu X.p \vee \Diamond X$ — analogous to **EF** p
- $\mu X.p \vee (\Diamond tt \wedge \Box X)$ — analogous to **AF** p
- $\mu X.[a]X$ — states with no infinite a-path
- $\nu X.p \wedge \Diamond\Diamond X$ — example beyond simple CTL/LTL-style patterns

Alternation in the μ-calculus

What matters

The complexity of a μ-calculus formula increases when least and greatest fixed points depend on each other in alternating ways.

Intuition

- μ means: start from nothing and grow
- ν means: start from everything and prune
- mere nesting of μ inside ν, or vice versa, is not yet the problem
- the real issue is whether variables depend back on each other through opposite fixed points

Why this matters

Alternation is reflected later in parity priorities and parity-game solving.

Alternation-free versus alternating formulas

Examples

- $\mu X.(p \vee \langle a \rangle X)$ — alternation-free
- $\nu X.(p \wedge [a]X)$ — alternation-free
- $\mu Y.(\nu Z.(p \wedge [a]Z) \vee [a]Y)$ — still alternation-free
- true alternation appears only when μ- and ν-variables depend back on each other

Important fact

CTL embeds into the alternation-free fragment of the modal μ-calculus.
So even formulas such as **AF AG** p and **AG AF** p remain alternation-free after translation.

μ-calculus example: AF AG p

$$\mu Y.\nu Z.p \wedge [a]Z \vee [a]Y$$

represents states from which one will eventually reach a region where p holds forever.

$$\mu^0 Y.(\nu Z.(p \wedge [a]Z) \vee [a]Y) = \emptyset$$
$$\mu^1 Y.(\nu Z.(p \wedge [a]Z) \vee [a]Y) = \nu Z.(p \wedge [a]Z)$$
$$\mu^2 Y.(\nu Z.(p \wedge [a]Z) \vee [a]Y) = \nu Z.(p \wedge [a]Z) \vee [a](\nu Z.(p \wedge [a]Z))$$
$$\mu^3 Y.(\nu Z.(p \wedge [a]Z) \vee [a]Y) = \nu Z.(p \wedge [a]Z) \vee [a]\big(\nu Z.(p \wedge [a]Z) \vee [a](\nu Z.(p \wedge [a]Z))\big)$$

So we first identify the states satisfying **AG** p, and then grow outward by adding states whose a-successors already lead there.

μ-calculus example: AG AF p

$$\nu Y.\mu Z.p \vee [a]Z \wedge [a]Y$$

represents states from which, along every path, p remains inevitable forever.

$$\nu^0 Y.(\mu Z.(p \vee [a]Z) \wedge [a]Y) = S$$
$$\nu^1 Y.(\mu Z.(p \vee [a]Z) \wedge [a]Y) = \mu Z.(p \vee [a]Z)$$
$$\nu^2 Y.(\mu Z.(p \vee [a]Z) \wedge [a]Y) = \mu Z.(p \vee [a]Z) \wedge [a](\mu Z.(p \vee [a]Z))$$
$$\nu^3 Y.(\mu Z.(p \vee [a]Z) \wedge [a]Y) = \mu Z.(p \vee [a]Z) \wedge [a]\big(\mu Z.(p \vee [a]Z) \wedge [a](\mu Z.(p \vee [a]Z))\big)$$

So we start with all states, and repeatedly remove states that fail to preserve the property that p is always eventually reachable.

From Büchi and parity to fixed points

Key idea

A Büchi condition means: visit an accepting state infinitely often: a repeated reachability condition.

As fixed points

Repeated reachability naturally has the shape

$$\nu X.\, \mu Y.\, \big((F \wedge \Diamond X) \;\vee\; \Diamond Y\big)$$

where:

- μY: reach the accepting set F once
- νX: repeat this forever

So in the generic case, the Büchi condition gives a formula with one ν/μ alternation.

A parity condition with more priorities corresponds to more nested least and greatest fixed points.

State logic versus trace logic

CTL and μ-calculus

Both are state logics: a formula denotes a set of states of a transition system.
So CTL embeds directly into the modal μ-calculus.

LTL is different

LTL is a trace logic: it talks about executions rather than directly about sets of states.

What this means in practice

Although LTL can also be connected to the μ-calculus via ω-automata, this is usually not the practical route for ordinary LTL model checking.

For LTL model checking, the standard route:

$$\text{LTL} \longrightarrow \omega\text{-automata} \longrightarrow \text{product} \longrightarrow \text{emptiness checking}.$$

Evaluating μ-calculus formulas

Basic cases

We evaluate formulas bottom-up:

- propositions denote sets of states;
- Boolean operators become union / intersection;
- modalities compute predecessors:
  - $[a]\phi$: all a-successors satisfy ϕ
  - $\langle a \rangle \phi$: some a-successor satisfies ϕ

Fixed points

- for $\mu X.\phi$, start with $\emptyset$ and iterate until stable
- for $\nu X.\phi$, start with S and iterate until stable

On finite transition systems, this terminates after finitely many steps.

Summary of the modal μ-calculus

Take-home messages

- the modal μ-calculus extends modal logic with least and greatest fixed points
- CTL embeds directly into the alternation-free fragment
- LTL does not embed directly as a state logic
- Büchi and parity acceptance correspond naturally to nested fixed points

See also Chapter 26 of the *Handbook of Model Checking* (2018).

From LTL to GNBA via VWAA

A more specialized route for LTL over words

A general Büchi condition leads naturally to a $\nu\mu$-style fixed-point formula.

For LTL on ω-words, one can first translate to very weak alternating automata.

This gives another route besides the direct tableau-style GNBA construction:

$$\text{LTL} \longrightarrow \text{very weak alternating automaton} \longrightarrow \text{GNBA / NBA}$$

and (only conceptually)

$$\text{LTL} \longrightarrow \text{very weak alternating automaton} \longrightarrow \text{alternation-free } \mu\text{-calculus.}$$

Gastin and Oddoux: *Fast LTL to Büchi Automata Translation.* (CAV 2001)
Babiak et al: *LTL to Büchi automata translation: fast and more deterministic.* (TACAS 2012)
Blahoudek et al: *LTL to self-loop alternating automata with generic acceptance and back.* (TCS 2020)

Very weak automata

A very weak automaton has a partial order $\preceq$ on states such that every transition goes only to states that are lower or equal:

$$q' \text{ occurs as a successor of } q \implies q' \preceq q.$$

- The only cycles are self-loops.
- Like a DAG plus self-loops on states.

Alternating automata

In an alternating automaton, the transition function returns a positive Boolean formula over states for each state q and letter α:

$$\delta(q, \alpha) \in B^+(Q).$$

For example,

$$\delta(q, \alpha) = (q_1 \wedge q_2) \vee q_3$$

means:

- either branch to q_3,
- or branch to both q_1 and q_2.

So:

- $\vee$ represents choice,
- $\wedge$ represents simultaneous obligations.

Alternating automata

In an alternating automaton, the transition function returns a positive Boolean formula over states for each state q and letter α:

$$\delta(q, \alpha) \in B^+(Q).$$

For example,

$$\delta(q, \alpha) = (q_1 \wedge q_2) \vee q_3$$

means:

- either branch to q_3,
- or branch to both q_1 and q_2.

So:

- $\vee$ represents choice,
- $\wedge$ represents simultaneous obligations.
- There is no “current state”, there is a current set of states!!

Very weak alternating co-Büchi automata

Putting the pieces together

A very weak alternating co-Büchi automaton combines:

- a very weak partial-order structure,
- an alternating branching of obligations,
- and a co-Büchi acceptance condition.

From LTL to a VWAA

States

Given an LTL formula φ, the states are "roughly" subformulae of φ.

Transitions

The transition structure is obtained by unfolding the expansion laws:

$$\mathbf{F}\,\psi \equiv \psi \vee \mathbf{X}\,\mathbf{F}\,\psi \qquad \mathbf{G}\,\psi \equiv \psi \wedge \mathbf{X}\,\mathbf{G}\,\psi$$

$$\psi\,\mathbf{U}\,\chi \equiv \chi \vee (\psi \wedge \mathbf{X}\,\psi\,\mathbf{U}\,\chi) \qquad \psi\,\mathbf{R}\,\chi \equiv \chi \wedge (\psi \vee \mathbf{X}\,\psi\,\mathbf{R}\,\chi)$$

A transition rewrites the current formula into a positive Boolean combination of next obligations.

Acceptance

Co-Büchi means that along every branch of a run, rejecting states may occur only finitely often.

U-obligations form rejecting states (staying in the self-loop of a **U**-state).

Alternative view: rejecting transitions (self-loops of the **U**-states).

Tiny VWAA examples

Example 1: $\mathbf{F}\ p$

Using

$$\mathbf{F}\ p \equiv p \vee \mathbf{X}\ \mathbf{F}\ p,$$

we obtain one state $q_{\mathbf{F}p}$ with transition

$$\delta(q_{\mathbf{F}p}, \alpha) = \begin{cases} \text{true} & \text{if } p \in \alpha, \\ q_{\mathbf{F}p} & \text{if } p \notin \alpha. \end{cases}$$

So the automaton keeps waiting until p holds.

State $q_{\mathbf{F}p}$ is rejecting.

Tiny VWAA examples

Example 2: $\mathbf{G}\ p$

Using

$$\mathbf{G}\ p \equiv p \wedge \mathbf{X}\ \mathbf{G} p,$$

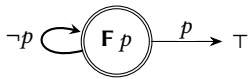
we obtain one state $q_{\mathbf{G}p}$ with transition

$$\delta(q_{\mathbf{G}p}, \alpha) = \begin{cases} q_{\mathbf{G}p} & \text{if } p \in \alpha, \\ \text{false} & \text{if } p \notin \alpha. \end{cases}$$

So the obligation persists forever, unless p fails.

VWAAs for F p and G p

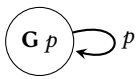
F p



$$\delta(\mathbf{F}\, p, \alpha) = \begin{cases} \top & \text{if } p \in \alpha \\ \mathbf{F}\, p & \text{if } p \notin \alpha \end{cases}$$

If p holds now, the obligation is discharged; otherwise we keep waiting.

G p



$$\delta(\mathbf{G}\, p, \alpha) = \begin{cases} \mathbf{G}\, p & \text{if } p \in \alpha \\ \bot & \text{if } p \notin \alpha \end{cases}$$

p must hold now and continue to hold forever; if p fails, the run dies immediately.

Tiny VWAA examples

Example 3: $p \mathbf{U} q$

Using

$$p \mathbf{U} q \equiv q \vee \big(p \wedge \mathbf{X}\, p \mathbf{U} q\big),$$

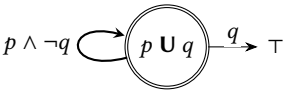
we obtain one state $q_{p\mathbf{U}q}$ with transition

$$\delta(q_{p\mathbf{U}q}, \alpha) = \begin{cases} \text{true} & \text{if } q \in \alpha, \\ q_{p\mathbf{U}q} & \text{if } q \notin \alpha \text{ and } p \in \alpha, \\ \text{false} & \text{if } p, q \notin \alpha. \end{cases}$$

State $q_{p\mathbf{U}q}$ is rejecting.

VWAA for $p \mathbf{U} q$

$p \mathbf{U} q$



if q holds now, the until-obligation is fulfilled;
otherwise, if p holds, we keep waiting

A VWAA example with real alternating structure

Where alternation appears

For formulas with explicit conjunction or disjunction, such as

$$\mathbf{G}\, p \wedge \mathbf{F}\, q,$$

the VWAA uses Boolean branching directly:

$$\delta(q_{\mathbf{G}p \wedge \mathbf{F}q}, \alpha) = \delta(q_{\mathbf{G}p}, \alpha) \ \wedge \ \delta(q_{\mathbf{F}q}, \alpha).$$

So conjunction becomes simultaneous obligations, and disjunction becomes choice.

Another tiny VWAA example: F G p

Expansion law

Using

$$\mathbf{F}\,\mathbf{G}\,p \equiv \mathbf{G}\,p \vee \mathbf{X}\,\mathbf{F}\,\mathbf{G}\,p \qquad \text{and} \qquad \mathbf{G}\,p \equiv p \wedge \mathbf{X}\,\mathbf{G}\,p,$$

we use two states:

$$q_{\mathbf{FG}p} \qquad q_{\mathbf{G}p}.$$

Transitions

For the waiting state:

$$\delta(q_{\mathbf{FG}p}, \alpha) = \begin{cases} q_{\mathbf{G}p} \;\vee\; q_{\mathbf{FG}p} & \text{if } p \in \alpha, \\ q_{\mathbf{FG}p} & \text{if } p \notin \alpha. \end{cases}$$

For the persistent obligation:

$$\delta(q_{\mathbf{G}p}, \alpha) = \begin{cases} q_{\mathbf{G}p} & \text{if } p \in \alpha, \\ \text{false} & \text{if } p \notin \alpha. \end{cases}$$

Computing the VWAA transition function δ

Idea

We compute $\delta(\varphi, \alpha)$ structurally, following the syntax of φ.

We represent $\delta(\varphi, \alpha)$ as a set of subsets (disjunction of conjuncts)

For example, $(q_1 \wedge q_2) \vee q_3$ would be $\{\{q_1, q_2\}, \{q_3\}\}$.

Computing the VWAA transition function δ

Base cases

$$\delta(tt, \alpha) = \{\emptyset\}$$

$$\delta(ff, \alpha) = \emptyset$$

$$\delta(p, \alpha) = \begin{cases} \{\emptyset\} & \text{if } p \in \alpha \\ \emptyset & \text{otherwise} \end{cases} \qquad \delta(\neg p, \alpha) = \begin{cases} \{\emptyset\} & \text{if } p \notin \alpha \\ \emptyset & \text{otherwise} \end{cases}$$

So atomic formulas are checked immediately against the current letter α.

- $\{\emptyset\}$ is "true": contains the successor state $\emptyset$ without obligations
- $\emptyset$ is "false": contains no successor states

Computing the VWAA transition function δ

Boolean connectives

First define the product operation

$$M_1 \otimes M_2 \;=\; \{\, S_1 \cup S_2 \mid S_1 \in M_1,\; S_2 \in M_2 \,\}.$$

Then:

$$\delta(\psi \vee \chi, \alpha) = \delta(\psi, \alpha) \;\cup\; \delta(\chi, \alpha)$$
$$\delta(\psi \wedge \chi, \alpha) = \delta(\psi, \alpha) \;\otimes\; \delta(\chi, \alpha)$$

- $\vee$: choose one branch (non-deterministic)
- $\wedge$: satisfy both branches, so combine their obligations

Computing the VWAA transition function δ

Temporal operators

$$\delta(\mathbf{X}\,\psi, \alpha) = \{\{\psi\}\}$$

$$\delta(\psi\ \mathbf{U}\ \chi, \alpha) = \delta(\chi, \alpha)\ \cup\ \big(\delta(\psi, \alpha) \otimes \{\{\psi\ \mathbf{U}\ \chi\}\}\big)$$

$$\delta(\psi\ \mathbf{R}\ \chi, \alpha) = \delta(\chi, \alpha)\ \otimes\ \big(\delta(\psi, \alpha)\ \cup\ \{\{\psi\ \mathbf{R}\ \chi\}\}\big)$$

That is:

- $\mathbf{X}\,\psi$: postpone ψ to the next step
- $\psi\chi$ $\mathbf{U}$: either χ already holds now, or ψ holds now and the until-obligation continues
- $\psi\chi$ $\mathbf{R}$: χ must hold now, and in addition either ψ already holds now, or the release-obligation continues to the next step.

From VWAA to TGBA: the main idea

Goal

We start from a very weak alternating co-Büchi automaton

$$\mathcal{A} = \langle Q, \Sigma, \delta, I, F \rangle$$

and remove alternation.

Key idea

A state of the target automaton represents a set of simultaneously active VWAA obligations.

So target states are subsets of Q:

$$O \subseteq Q.$$

The alternating branching is replaced by nondeterminism over sets of active obligations.

From VWAA to TGBA: transitions

Combining obligations

- Let $O = \{q_1, \ldots, q_n\}$ be a TGBA state.
- Fix a current valuation $\alpha \in 2^{AP}$.
- For each obligation $q \in O$, $\delta(q, \alpha)$ is the set of possible conjunctions of next obligations.

Successor construction

To construct the TGBA successor(s) under α:

- for each $q \in O$, choose one conjunction $S_q \in \delta(q, \alpha)$,
- and combine all these obligations into

$$O' = \bigcup_{q \in O} S_q.$$

From VWAA to TGBA: acceptance

Acceptance

Co-Büchi condition: rejecting states $f \in F$ are not allowed to remain active forever.

So the TGBA uses one generalized accepting set of transitions

$$T_f \qquad \text{for each } f \in F,$$

requiring that along an accepting run we keep making progress with respect to f.

Intuition

For a fixed $f \in F$, a transition belongs to T_f iff:

- $f \notin O$, or
- the VWAA transition chosen for f is non-looping: it does not put f back into the successor set. (f might still be in the successor set, just not from a f self-loop)

That is: $f \notin O \vee f \notin S_f$.

Example: $G(a \mathbf{U} (\neg a \wedge b))$

Let

$$u = a \mathbf{U} (\neg a \wedge b).$$

The inner U-state has transition shape

$$\delta(u, \alpha) = (\neg a \wedge b) \ \vee \ (a \wedge u).$$

Reading this transition

- if $\neg a \wedge b$ holds, then u is resolved (transition is accepting)
- if a holds now, but $\neg a \wedge b$ does not, then u is postponed (transition is not accepting)

What does the outer G do?

The outer G keeps the property active on every suffix: every transition keeps f active.

Example: $G(a \mathbf{U} (\neg a \wedge b))$

- There are two states in the VWAA:
  - The initial state g: $G(a \mathbf{U} (\neg a \wedge b))$ or with PNF: $ff \mathbf{R} (a \mathbf{U} (\neg a \wedge b)))$
  - The rejecting state u: $a \mathbf{U} (\neg a \wedge b)$
- Transitions from u: $\delta(u, \alpha) = (\neg a \wedge b) \ \vee \ (a \wedge u)$
- Transitions from g: $\delta(g, \alpha) = \delta(u, \alpha) \ \wedge \ g = (\neg a \wedge b \wedge g) \ \vee \ (a \wedge u \wedge g)$

- There are two states in the TGBA:
  - The initial state $\{g\}$
  - The state $\{g, u\}$
- Transitions from $\{g\}$:
  - to $\{g\}$ if $(\neg a \wedge b) \in \alpha$ (accepting)
  - to $\{g, u\}$ if $a \in \alpha$ (accepting)
- Transitions from $\{g, u\}$:
  - to $\{g\}$ if $(\neg a \wedge b) \in \alpha$ (accepting)
  - to $\{g, u\}$ if $a \in \alpha$ (not accepting)

Two routes from LTL to Büchi automata

Route taught earlier: direct tableau / closure construction

$$\text{LTL} \longrightarrow \text{closure of subformulae} \longrightarrow \text{consistent obligation sets} \longrightarrow \text{GNBA} \longrightarrow \text{NBA}.$$

- very direct correctness argument

Alternative route: via VWAA

$$\text{LTL} \longrightarrow \text{VWAA} \longrightarrow \text{TGBA} \longrightarrow \text{NBA}.$$

- closer to practical translation tools
- often more concise intermediate representation, though still exponential!

Why this matters

Practical consequence

The VWAA route gives a concise, state-of-the-art translation from LTL to generalized / ordinary Büchi automata.

Conceptual consequence

Weak alternating automata can also be translated directly into alternation-free modal μ-calculus over words.

LTL synthesis

Beyond verification: constructing correct systems

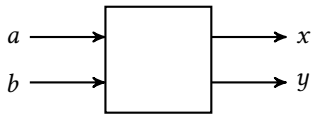
Verification

Check whether a given system satisfies a temporal specification.

Synthesis

Go one step further: automatically construct a system that is guaranteed to satisfy the specification.

In reactive synthesis, the system reacts continuously to input from the environment:



black box with inputs a, b and outputs x, y

LTL synthesis

Problem statement

Given an LTL formula ϕ over AP, and a partition $AP = AP_U \uplus AP_C$ into uncontrollable and controllable propositions, synthesize a controller such that ϕ holds for all choices of the environment on AP_U.

Controller model

Usually we synthesize a Mealy machine:

- the environment chooses the inputs first
- the controller reacts with outputs

Alternative: Moore semantics, where the controller commits its output before seeing the current environment move.

The standard automata-theoretic synthesis pipeline

Main idea

LTL $\longrightarrow$ DPA $\longrightarrow$ 2-player parity game $\longrightarrow$ winning strategy $\longrightarrow$ controller

Reading this pipeline

1. Translate the specification into a deterministic parity automaton.
2. Split each move into an environment choice and a controller choice.
3. Solve the resulting parity game.
4. If the controller wins from the initial state, the specification is realizable.
5. Extracting a controller from that strategy is the actual synthesis step.

Why deterministic parity automata?

Why deterministic?

In synthesis, the controller must react to the actual choices of the environment.

It cannot rely on a hidden nondeterministic guess about which automaton run will be successful later.

So the specification automaton should determine a unique current obligation for every history.

Why parity?

Parity games admit positional winning strategies, which is very convenient for controller extraction: one only needs the current game state, not the full history of the run.

The price is that determinization to parity is one of the main difficult steps in the synthesis pipeline.

Different philosophies for reactive synthesis

Automata-and-games route

LTL $\longrightarrow$ deterministic ω-automaton $\longrightarrow$ parity game $\longrightarrow$ strategy $\longrightarrow$ controller

Bounded synthesis

Instead of constructing a large deterministic automaton first, search directly for a small implementation by imposing a size bound.
The existence of such an implementation is encoded as SAT/QBF/SMT-style constraints.

Practical picture

- **Strix**: direct LTL $\longrightarrow$ DPA plus parity games
- **BoSy**: bounded synthesis via logical encodings
- **ltlsynt**: automata-theoretic synthesis in the Spot ecosystem

Why LTL fragments matter for synthesis

Not all specifications are equally difficult

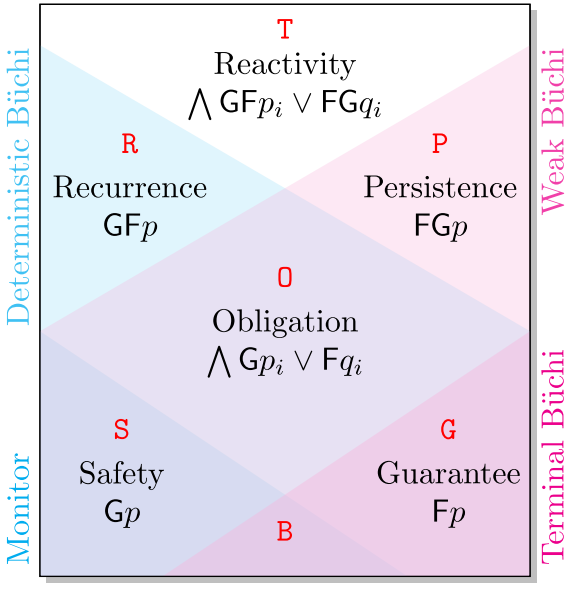
LTL contains important fragments such as:

- safety: “nothing bad ever happens”
- guarantee / co-safety: “something good eventually happens”
- richer fragments such as persistence, recurrence, and full reactivity

Some fragments admit simpler automata constructions, simpler acceptance conditions, and more specialized synthesis procedures.

Modern tools often decompose specifications into easier fragments before applying the full synthesis machinery.

Linear Temporal Logic



LTL synthesis via automata

1. Preprocess the LTL formula
   - simplify easy cases
   - decompose / normalize

LTL synthesis via automata

1. Preprocess the LTL formula
   - simplify easy cases
   - decompose / normalize
2. Construct a deterministic parity (or Rabin) automaton
   - via direct deterministic Emerson–Lei automata (DELA)
   - via transition-based Emerson–Lei automata (TELA)
   - via limit-deterministic automata
   - via very weak alternating automata
   - via nondeterministic automata
   - then determinize / parityize

LTL synthesis via automata

1. Preprocess the LTL formula
   - simplify easy cases
   - decompose / normalize
2. Construct a deterministic parity (or Rabin) automaton
   - via direct deterministic Emerson–Lei automata (DELA)
   - via transition-based Emerson–Lei automata (TELA)
   - via limit-deterministic automata
   - via very weak alternating automata
   - via nondeterministic automata
   - then determinize / parityize
3. Construct a parity game from the parity automaton by splitting inputs and outputs

LTL synthesis via automata

1. Preprocess the LTL formula
   - simplify easy cases
   - decompose / normalize
2. Construct a deterministic parity (or Rabin) automaton
   - via direct deterministic Emerson–Lei automata (DELA)
   - via transition-based Emerson–Lei automata (TELA)
   - via limit-deterministic automata
   - via very weak alternating automata
   - via nondeterministic automata
   - then determinize / parityize
3. Construct a parity game from the parity automaton by splitting inputs and outputs
4. Solve the parity game

LTL synthesis via automata

1. Preprocess the LTL formula
   - simplify easy cases
   - decompose / normalize
2. Construct a deterministic parity (or Rabin) automaton
   - via direct deterministic Emerson–Lei automata (DELA)
   - via transition-based Emerson–Lei automata (TELA)
   - via limit-deterministic automata
   - via very weak alternating automata
   - via nondeterministic automata
   - then determinize / parityize
3. Construct a parity game from the parity automaton by splitting inputs and outputs
4. Solve the parity game
5. Extract a Mealy machine from the winning strategy

LTL synthesis via automata

1. Preprocess the LTL formula
   - simplify easy cases
   - decompose / normalize
2. Construct a deterministic parity (or Rabin) automaton
   - via direct deterministic Emerson–Lei automata (DELA)
   - via transition-based Emerson–Lei automata (TELA)
   - via limit-deterministic automata
   - via very weak alternating automata
   - via nondeterministic automata
   - then determinize / parityize
3. Construct a parity game from the parity automaton by splitting inputs and outputs
4. Solve the parity game
5. Extract a Mealy machine from the winning strategy
6. Encode the Mealy machine as a Boolean circuit

From winning strategy to implementation: a practical view

After solving the parity game

A winning strategy still has to be turned into an implementation.
In practice, this involves choices such as:

- explicit versus symbolic game construction,
- strategy extraction,
- Mealy-machine minimisation,
- circuit encoding and post-processing.

Example

The Knor tool explores this part of the pipeline for the PGAME track of SYNTCOMP, starting from parity automata and ending in AIGER circuits.

Knor in SYNTCOMP

Oink

- State-of-the-art explicit-state parity game solver (TACAS 2018)
- Implements ± 30 algorithms
- Can run some algorithms multi-core

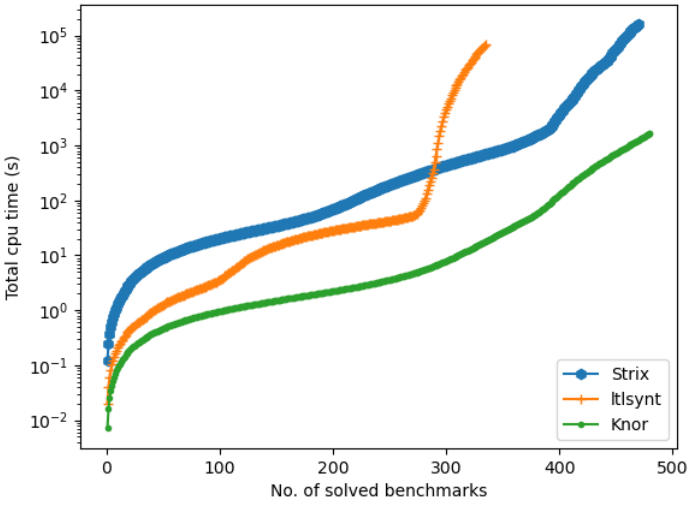
PGAME track in SYNTCOMP

- Input: DPA in the “extended eHOA” format
- Output: Boolean circuit in the “AIGER” format

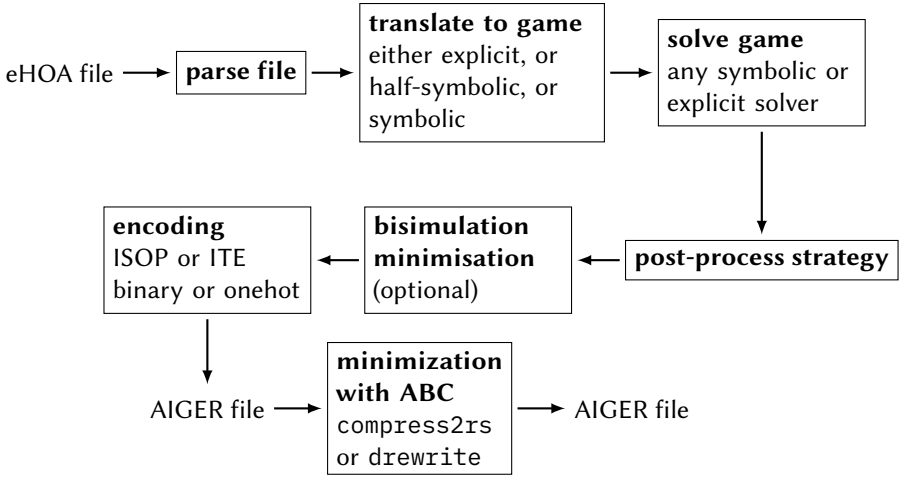
Knor: Oink in SYNTCOMP

- From eHOA input file to parity game in Oink (2020)
- From parity game strategy in Oink to AIGER output file (2021)
- Winner in 2021, 2022 by runtime

Knor in SYNTCOMP 2022



Knor overview

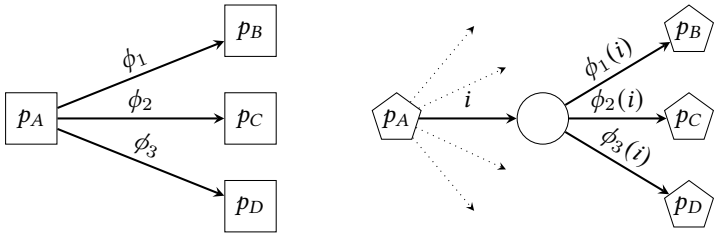


Knobs

- Game construction method
  - explicit, half-symbolic, fully symbolic
- Choice of parity game solver
  - symbolic, or any explicit solver in Oink
- Bisimulation minimisation of Mealy machine
  - yes or no
- Encoding to circuit (latches, AND-gates, inverters)
  - state encoding: binary / onehot
  - logic encoding: ISOP / ITE
- Minimization strategy
  - no minimization, or `compress2rs`, or `drw; b; drf`

Constructing the game

ϕ_1, ϕ_2, ϕ_3 are Boolean formulas over I and O



- Explicit: enumerate every valuation of I/O signals
- Half-symbolic: construct a BDD for each state
- Fully symbolic: represent entire automaton in one BDD
  - Variable order: state, input, output, priority, next state
  - Can construct explicit parity game on the BDD nodes (!!)

Constructing the game

From eHOA to explicit parity game

Technique	**Sum of Vertices**	**Time** (sec)
explicit	622,987,565	1,177.91
half-symbolic	8,491,540	18.28
symbolic	620,510	11.76

Constructing the game

Model	explicit	half-symbolic	symbolic
amba_decomposed_lock_15	T.O.	46	24
amba_decomposed_lock_14	T.O.	46	24
amba_decomposed_lock_13	T.O.	46	24
TwoCountersDisButA9	T.O.	668,065	7,249
amba_decomposed_lock_12	402,997,254	46	24
amba_decomposed_lock_11	100,820,998	46	24
amba_decomposed_lock_10	25,237,510	46	24
TwoCountersGui	21,022,475	256	155
TwoCountersDisButA8	15,254,863	497,310	4,721
full_arbiter_8	11,287,306	1,669,066	177,690
amba_decomposed_lock_9	6,323,718	46	24
amba_decomposed_encode_16	4,981,507	876	330
TwoCountersDisButA7	3,939,305	98,947	2,365
TwoCountersDisButA6	3,806,249	101,175	1,733

Parity game solvers

- Symbolic solver (fpi)
- Explicit solvers in Oink

Parity game solvers

- Symbolic solver (fpi)
- Explicit solvers in Oink

Solver	**Circuit size**		**Time** (sec)
	binary	onehot	
symbolic fpi (`--sym`)	317,403	122,514	18.45
fixpoint with justifications (`--fpj`)	350,035	139,900	0.16
fixpoint with freezing (`--fpi`)	353,120	140,297	0.22
strategy iteration (`--psi`)	334,149	140,916	0.57
priority promotion (`--npp`)	427,048	161,244	0.17
Zielonka (`--zlk`)	480,472	175,427	0.18
tangle learning (`--tl`)	604,044	213,632	0.17

- Symbolic solver best for circuit size, but slower than all others
  - (only slow for a few inputs)
- Future work: design a solver for small strategies? Other symbolic solvers?

Bisimulation minimisation

- Apply bisimulation minimisation on the Mealy machine

Bisimulation minimisation

- Apply bisimulation minimisation on the Mealy machine

Solver	**Circuit size**		**Time** (sec)
	binary	onehot	
symbolic fpi (`--sym`) + minimisation	166,839	106,500	0.19
fixpoint with justifications (`--fpj`) + min.	205,937	124,489	0.15
symbolic fpi (`--sym`)	317,403	122,514	–
fixpoint with justifications (`--fpj`)	350,035	139,900	–

- Takeaway: always apply
- Future work: minimise parity game before solving?
- Future work: other Mealy machine minimisation methods?

Encoding Mealy machine as Boolean circuit

- How to encode states of the Mealy machine as latches
- How to encode logic as circuit (AND-gates and inverters)

Encoding Mealy machine as Boolean circuit

- How to encode states of the Mealy machine as latches
- How to encode logic as circuit (AND-gates and inverters)

Solver	Encoding	Circuit size	Time
symbolic fpi (`--sym`)	ISOP, onehot	102,294	0.69
symbolic fpi (`--sym`)	ITE, onehot	106,500	0.61
fixpoint with justifications (`--fpj`)	ISOP, onehot	113,134	0.72
fixpoint with justifications (`--fpj`)	ITE, onehot	124,489	0.64
symbolic fpi (`--sym`)	ITE, binary	166,839	0.09
fixpoint with justifications (`--fpj`)	ITE, binary	205,937	0.12
symbolic fpi (`--sym`)	ISOP, binary	431,316	1.39
fixpoint with justifications (`--fpj`)	ISOP, binary	476,502	1.61

- Takeaway: onehot encoding is best, ISOP slightly better
- Future work: other latch encoding strategies
- Future work: play with order of encoding from BDD to gates

Postprocessing with ABC

- “Everyone uses post-processing with ABC” e.g. `compress2rs` (Strix)
- What about `drw; b; drf`?

Postprocessing with ABC

- "Everyone uses post-processing with ABC" e.g. `compress2rs` (Strix)
- What about `drw; b; drf`?

Solver	Encoding	Method	Circuit size	Time
symbolic fpi (`--sym`)	ISOP	compress	61,434	149.26
symbolic fpi (`--sym`)	ITE	compress	62,506	121.27
fixpoint with justifications (`--fpj`)	ISOP	compress	71,240	125.29
fixpoint with justifications (`--fpj`)	ITE	compress	72,897	108.10
symbolic fpi (`--sym`)	ISOP	drewrite	80,077	58.72
symbolic fpi (`--sym`)	ITE	drewrite	80,425	53.21
fixpoint with justifications (`--fpj`)	ISOP	drewrite	80,454	60.88
fixpoint with justifications (`--fpj`)	ITE	drewrite	80,903	58.58
symbolic fpi (`--sym`)	ISOP		102,294	44.88
symbolic fpi (`--sym`)	ITE		106,500	39.81
fixpoint with justifications (`--fpj`)	ISOP		113,134	31.66
fixpoint with justifications (`--fpj`)	ITE		124,489	25.77

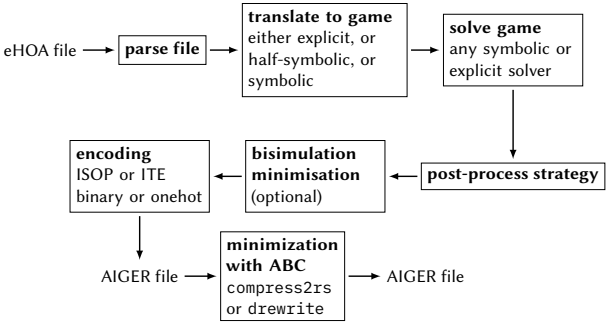
- Takeaway: trading time for space; good initial size: good final size
- Future work: try other ABC scripts; try other libraries e.g. Mockturtle
- Future work: can we find postprocessing-friendly encodings?

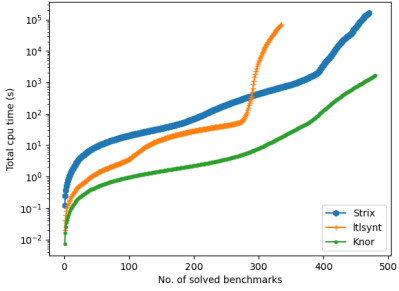
Comparison with other tools

Tool	Circuit size	
	no post-processing	with post-processing
strix	68,550	41,314
sym-bisim-isop-onehot	87,823	50,624
ltlsynt	544,804	98,996

- Knor is better than Strix for many benchmarks, except the `full_arbiter` benchmarks.

Final remarks





- Best result with fully symbolic methods + bisimulation minimisation + onehot encoding with ISOP + `compress2rs` post-processing
- Still many opportunities to improve
- Some SYNTCOMP-PGAME benchmarks are not derived from LTL
- Relying on ABC is not fun, but may be a necessary evil

Introduction to parity games

Games

Arena

- a game is played on an arena $\mathcal{A} = (V, V_0, V_1, E)$
  - (V, E) is a directed graph
  - V_0 is the set of vertices controlled by player 0
  - V_1 is the set of vertices controlled by played 1
  - usually $V = V_0 \uplus V_1$
- a play is a path along the edges, i.e., an infinite sequence $\pi \in V^\omega$
- the controller of a vertex decides the next vertex in the play

Games

Arena

- a game is played on an arena $\mathcal{A} = (V, V_0, V_1, E)$
  - (V, E) is a directed graph
  - V_0 is the set of vertices controlled by player 0
  - V_1 is the set of vertices controlled by played 1
  - usually $V = V_0 \uplus V_1$
- a play is a path along the edges, i.e., an infinite sequence $\pi \in V^\omega$
- the controller of a vertex decides the next vertex in the play

Game

- a game $\mathcal{G} = (\mathcal{A}, \mathrm{Win})$ consists of an arena and a winning condition
- the winning condition $\mathrm{Win} \subseteq V^\omega$ is the set of winning plays

Strategy

- a strategy for player i is a function $\sigma\colon V^*V_i \to V$ that selects a successor for every history of the play ending in a vertex of player i

Strategy

- a strategy for player i is a function $\sigma\colon V^*V_i \to V$ that selects a successor for every history of the play ending in a vertex of player i
- a strategy is memoryless iff $\sigma(wv) = \sigma(v)$ for all $w \in V^*, v \in V_i$

Strategy

- a strategy for player i is a function $\sigma\colon V^*V_i \to V$ that selects a successor for every history of the play ending in a vertex of player i
- a strategy is memoryless iff $\sigma(wv) = \sigma(v)$ for all $w \in V^*, v \in V_i$
- a memoryless strategy is also called a positional strategy $\sigma\colon V_i \to V$

Strategy

- a strategy for player i is a function $\sigma\colon V^*V_i \to V$ that selects a successor for every history of the play ending in a vertex of player i
- a strategy is memoryless iff $\sigma(wv) = \sigma(v)$ for all $w \in V^*, v \in V_i$
- a memoryless strategy is also called a positional strategy $\sigma\colon V_i \to V$
- a set $W \subseteq V$ is closed w.r.t. a strategy σ if for all $v \in W$:
  - if $v \in V_i$, then $\sigma(v) \in W$ (strategy in W)
  - if $v \notin V_i$, then $E(v) \subseteq W$ (opponent's successors in W)

Strategy

- a strategy for player i is a function $\sigma\colon V^*V_i \to V$ that selects a successor for every history of the play ending in a vertex of player i
- a strategy is memoryless iff $\sigma(wv) = \sigma(v)$ for all $w \in V^*, v \in V_i$
- a memoryless strategy is also called a positional strategy $\sigma\colon V_i \to V$
- a set $W \subseteq V$ is closed w.r.t. a strategy σ if for all $v \in W$:
  - if $v \in V_i$, then $\sigma(v) \in W$ (strategy in W)
  - if $v \notin V_i$, then $E(v) \subseteq W$ (opponent's successors in W)
- a strategy σ induces a subgame $\mathcal{G}[\sigma]$ by keeping only the chosen $\sigma(v)$ for each $v \in V_i$

Strategy

- a strategy for player i is a function $\sigma\colon V^*V_i \to V$ that selects a successor for every history of the play ending in a vertex of player i
- a strategy is memoryless iff $\sigma(wv) = \sigma(v)$ for all $w \in V^*, v \in V_i$
- a memoryless strategy is also called a positional strategy $\sigma\colon V_i \to V$
- a set $W \subseteq V$ is closed w.r.t. a strategy σ if for all $v \in W$:
  - if $v \in V_i$, then $\sigma(v) \in W$ (strategy in W)
  - if $v \notin V_i$, then $E(v) \subseteq W$ (opponent's successors in W)
- a strategy σ induces a subgame $\mathcal{G}[\sigma]$ by keeping only the chosen $\sigma(v)$ for each $v \in V_i$
- $\mathcal{G}[\sigma]$ is a one-player game for the opponent (automaton) if σ is total

Winning a game

- a strategy σ is winning from a vertex v if all plays starting in v and consistent with σ are winning for player i
  (that is: player i wins regardless of the choices of the opponent)

Winning a game

- a strategy σ is winning from a vertex v if all plays starting in v and consistent with σ are winning for player i
  (that is: player i wins regardless of the choices of the opponent)
- the winning region $W_i(\mathcal{G})$ is the set of vertices $v \in V$ for which there exists a strategy for player i that is winning from each v

Winning a game

- a strategy σ is winning from a vertex v if all plays starting in v and consistent with σ are winning for player i
  (that is: player i wins regardless of the choices of the opponent)
- the winning region $W_i(\mathcal{G})$ is the set of vertices $v \in V$ for which there exists a strategy for player i that is winning from each v
- a game is determined if $V = W_0(\mathcal{G}) \uplus W_1(\mathcal{G})$

Winning a game

- a strategy σ is winning from a vertex v if all plays starting in v and consistent with σ are winning for player i
  (that is: player i wins regardless of the choices of the opponent)
- the winning region $W_i(\mathcal{G})$ is the set of vertices $v \in V$ for which there exists a strategy for player i that is winning from each v
- a game is determined if $V = W_0(\mathcal{G}) \uplus W_1(\mathcal{G})$
- a game is memoryless determined if for every vertex $v \in V$, there exists a memoryless strategy that is winning for some player from v

Winning a game

- a strategy σ is winning from a vertex v if all plays starting in v and consistent with σ are winning for player i
  (that is: player i wins regardless of the choices of the opponent)
- the winning region $W_i(\mathcal{G})$ is the set of vertices $v \in V$ for which there exists a strategy for player i that is winning from each v
- a game is determined if $V = W_0(\mathcal{G}) \uplus W_1(\mathcal{G})$
- a game is memoryless determined if for every vertex $v \in V$, there exists a memoryless strategy that is winning for some player from v
- a set $D \subseteq V$ is a dominion of player i if D is a closed subset of W_i, i.e., player i has a winning strategy σ such that D is closed w.r.t. strategy σ
  (also: the winning region is a dominion and can contain smaller dominions)

Winning conditions

- the winning condition determines who wins a play
- remember: Win $\subseteq V^\omega$ is the set of winning plays
- safety game: player 0 wins if a set $S \subseteq V$ of vertices can be avoided
- reachability game: player 0 wins if a set $U \subseteq V$ of vertices can be reached
- Büchi game: player 0 wins if some accepting vertex in $F \subseteq V$ can be reached infinitely often
- parity game: player 0 wins if the maximum* priority seen infinitely often is even

*sometimes minimum

Model checking with automata

1. Construct an automaton that accepts “bad” infinite words
2. $\mathrm{TS} \models \varphi \Leftrightarrow \mathcal{L}_\omega(\mathrm{TS} \cap B_{\neg\varphi}) = \emptyset$

Model checking with automata and games

Model checking with automata

1. Construct an automaton that accepts “bad” infinite words
2. $\mathrm{TS} \models \varphi \Leftrightarrow \mathcal{L}_\omega(\mathrm{TS} \cap B_{\neg\varphi}) = \emptyset$

Model checking with games

1. Construct a two-player game of a specification
   - Player 0 tries to prove the specification (“existential” player)
     for example $\exists$, $\vee$: choose a path that satisfies
   - Player 1 tries to violate the specification (“universal” player)
     for example $\forall$, $\wedge$: choose a path that violates
2. $\mathrm{TS} \models \varphi \Leftrightarrow$ player 0 wins the initial vertex of the product game

Parity Games and the modal μ-calculus

Linear from μ-calculus to parity games

Structural decomposition of ϕ.

- Stirling (1995): Local Model Checking Games. In CONCUR.
- Emerson, Jutla & Sistla (2001): On model checking for the mu-calculus and its fragments. In TCS.

Linear from parity games to μ-calculus

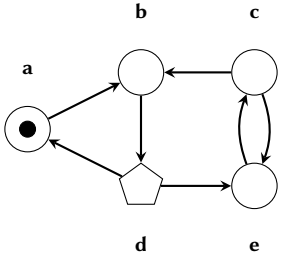
Construction of a μ-calculus formula of the winning condition.

- Walukiewicz (1996): Monadic Second Order Logic on Tree-Like Structures. In STACS.

Hence: the modal μ-calculus is at least as expressive as ω-regular expressions.

Parity Games

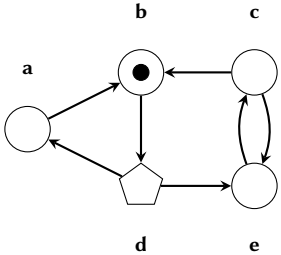
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



The play π: **a**

Parity Games

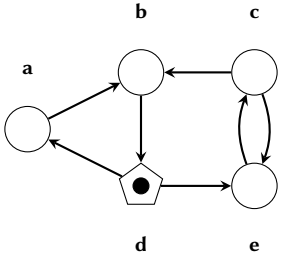
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



The play π: **a b**

Parity Games

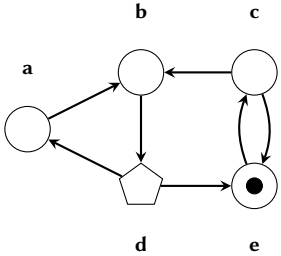
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



The play π: **a b d**

Parity Games

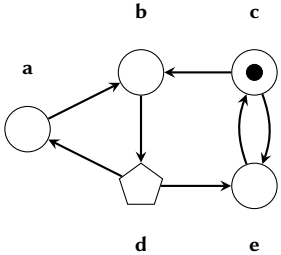
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



The play π: **a b d e**

Parity Games

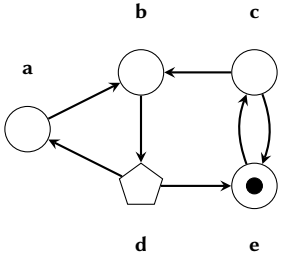
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



The play π: **a b d e c**

Parity Games

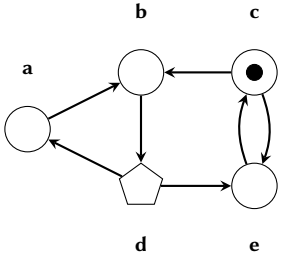
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



The play π: **a b d e c e**

Parity Games

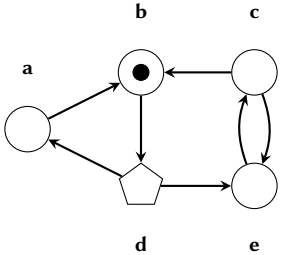
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



The play π: **a b d e c e c**

Parity Games

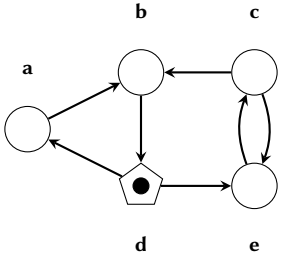
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



The play π: **a b d e c e c b**

Parity Games

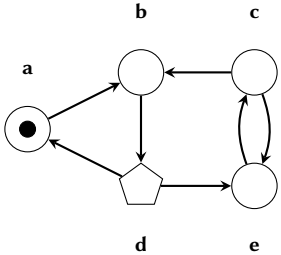
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



The play π: **a b d e c e c b d**

Parity Games

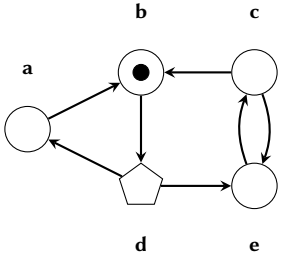
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



The play π: **a b d e c e c b d a**

Parity Games

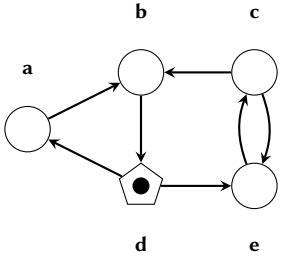
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



The play π: **a b d e c e c b d a b**

Parity Games

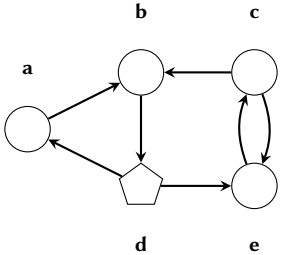
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



The play π: **a b d e c e c b** $(\mathbf{d\ a\ b})^{\omega}$

Parity Games

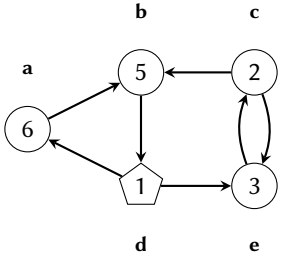
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



Who wins a play?

Parity Games

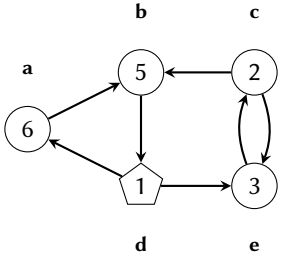
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



- Each vertex has a priority $\{0, 1, 2, \ldots, d\}$
- Player Even wins if the highest priority seen infinitely often is even

Parity Games

- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor

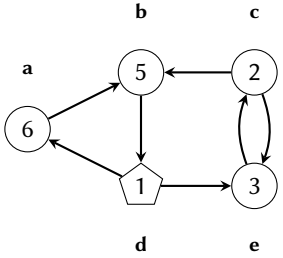


The play π: 6 5 1 3 2 3 2 5 $(1\ 6\ 5)^{\omega}$

Who wins this play?

Parity Games

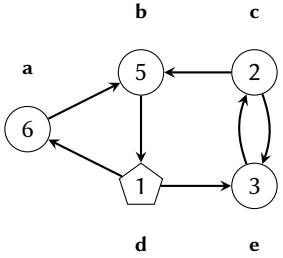
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



How do we determine who wins a vertex?

Parity Games

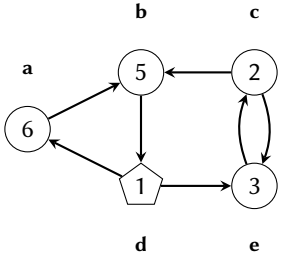
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



A player wins a vertex v if they have a positional strategy σ such that every play from v consistent with σ is won by that player.

Parity Games

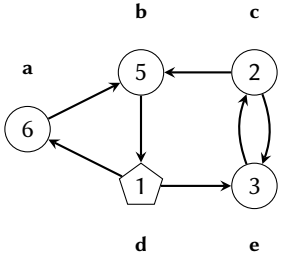
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



A player wins a (subgame) $\mathcal{G}$ if they have a positional strategy σ such that all cycles in the induced graph $\mathcal{G}[\sigma]$ are won by that player

Parity Games

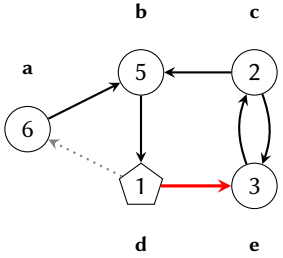
- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



Which vertices are won by which player?

Parity Games

- A parity game is played on a directed graph
- Two players: Even ● and Odd ⬟
- A play is an infinite path along the edges
- The owner of each vertex chooses the successor



Player Odd wins with strategy $\{ \mathbf{d} \rightarrow \mathbf{e} \}$
Only two (Odd) cycles remain

Parity Games

Solving a parity game

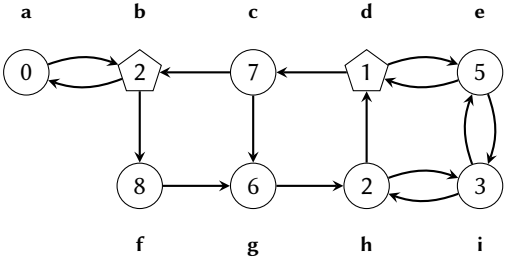
- Compute the winning regions $W_{\bigcirc}$ and $W_{\pentagon}$
- Compute the winning positional strategies $\sigma_{\bigcirc}$ and $\sigma_{\pentagon}$

Complexity status

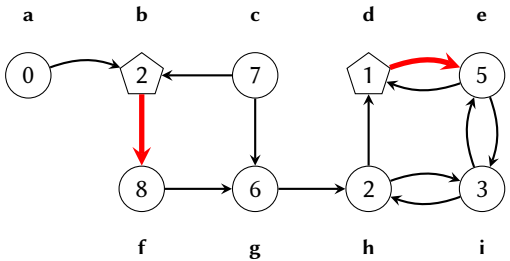
The exact complexity of solving parity games is still open.

- Decision version (who wins from a given vertex?): in $\mathrm{UP} \cap \mathrm{coUP}$
- Search version (find a positional winning strategy): in $\mathrm{CLS} = \mathrm{PLS} \cap \mathrm{PPAD}$

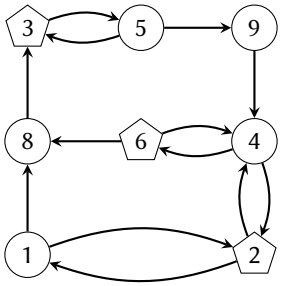
Example parity game



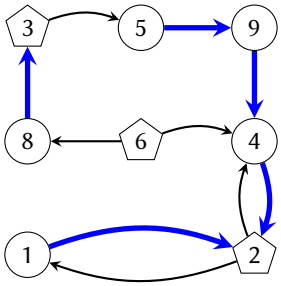
Example parity game



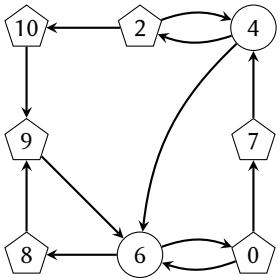
Example parity game



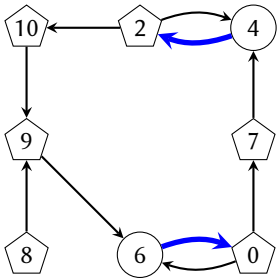
Example parity game



Example parity game



Example parity game



Parity Games

Definitions I

- A parity game $\mathcal{G}$ is a tuple $(V_{\bullet}, V_{\pentagon}, E, \mathrm{pr})$
- Vertices $V = V_{\bullet} \cup V_{\pentagon}$ controlled by players Even and Odd
- Transitions $E \colon V \times V$ such that E is left-total.
  - We write $u \longrightarrow v$ for $(u, v) \in E$
  - $E(u)$ denotes the successors of u: $\{v \mid u \longrightarrow v\}$
  - $E(U)$ denotes all successors of vertices in $\{v \mid u \longrightarrow v \mid u \in U\}$
  - Each vertex has at least one successor.
- Priority function $\mathrm{pr} \colon V \longrightarrow \{0, 1, 2, \ldots, d\}$
- A play is an infinite sequence $v_0 v_1 v_2 \ldots$ consistent with E
- A play π is won by player Even iff $\max(\mathrm{pr}(\inf \pi))$ is even

Definitions II

- A strategy $\sigma\colon V_\alpha \to V$ for player α assigns one successor to each vertex of player α.
- $\mathrm{Plays}(v, \sigma)$ is the set of plays consistent with σ starting in v.
- σ is a winning strategy from v for player α if all plays in $\mathrm{Plays}(v, \sigma)$ are winning for α
- A set W is closed w.r.t. a strategy σ if for all $v \in W$:
  - if v is owned by α, then $\sigma(v) \in W$ (strategy in W)
  - if v is owned by $\overline{\alpha}$, then $E(v) \subseteq W$ (all successors in W)
- A set D is a dominion of player α if α has a strategy σ that is winning for all $v \in D$ and D is closed w.r.t. σ.
- The winning regions of either player are dominions.

Attractor computation

Playing from **A** to **B**

- From which vertices **A** must a play eventually reach **B**?
- What is the highest vertex that player α can reach?
- Which vertices cannot be avoided by the opponent $\overline{\alpha}$?

Attractor computation

Playing from **A** to **B**

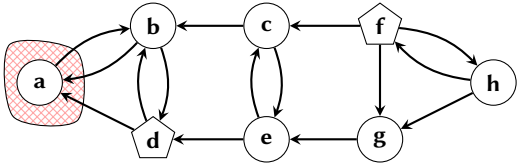
- From which vertices **A** must a play eventually reach **B**?
- What is the highest vertex that player α can reach?
- Which vertices cannot be avoided by the opponent $\overline{\alpha}$?

Attractor computation

- "Backward reachability" with an opponent
- Given target set Z and a player α,
  compute all vertices from which player α can ensure arrival in Z
- Add to Z (exhaustively):
  - all vertices of the player α that *can* play to Z
  - all vertices of opponent $\overline{\alpha}$ that *must* play to Z

Example of attractor computation

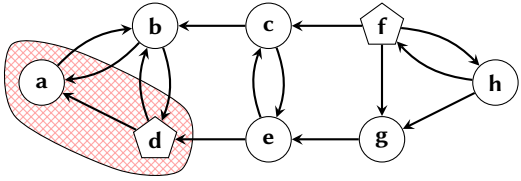
Computing the ⬟-attractor to **a**



Attractor set: {**a**}
Can attract: **d** but not **b**

Example of attractor computation

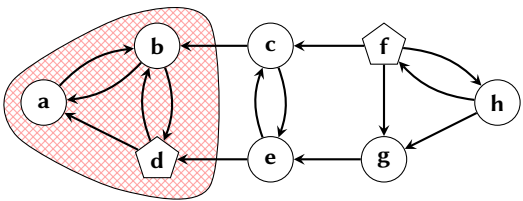
Computing the ⬟-attractor to **a**



Attractor set: $\{\mathbf{a}\}$, $\{\mathbf{a}, \mathbf{d}\}$
Can attract: **b** but not **e**

Example of attractor computation

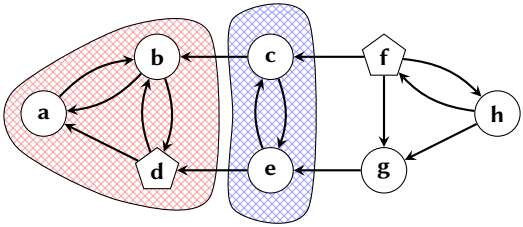
Computing the ⬟-attractor to **a**



Attractor set: {**a**}, {**a**, **d**}, {**a**, **b**, **d**}
Can attract: neither **c** nor **e**

Example of attractor computation

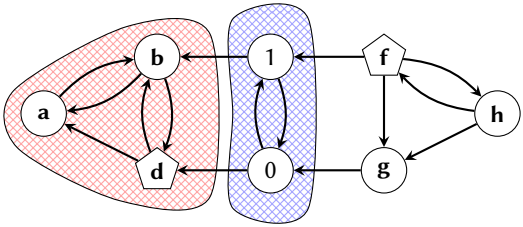
Computing the ⬟-attractor to **a**



Attractor set: {**a**}, { **a**, **d**}, {**a**, **b**, **d**}
What if player ● cannot stay in {**c**, **e**}??

Example of attractor computation

Computing the ⬟-attractor to **a**



Attractor set: {**a**}, {**a**, **d**}, {**a**, **b**, **d**}

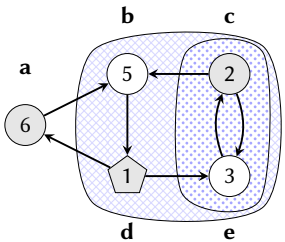
Every play inside the blue area is won by ⬟!

Tangles

Intuition

A tangle is a strongly connected subgame where one player can win all plays that stay inside it.

Equivalently: in the strategy-restricted subgame, all cycles are winning for that player.



A game with a 5-tangle and a 3-tangle

Tangles

Intuition

A tangle is a strongly connected subgame where one player can win all plays that stay inside it.

Equivalently: in the strategy-restricted subgame, all cycles are winning for that player.

Definition

A tangle is a pair $T = (U, \sigma)$ where

- U is a nonempty set of vertices,
- $\sigma \colon U_\alpha \to U$ is a strategy for player α,
- the subgame $\mathcal{G}[U, \sigma]$ is strongly connected,
- and player α wins all plays that remain in $\mathcal{G}[U, \sigma]$.

Tangles

Intuition

A tangle is a strongly connected subgame where one player can win all plays that stay inside it.

Equivalently: in the strategy-restricted subgame, all cycles are winning for that player.

Operational view

- One player has a strategy to win all plays that remain inside the tangle.
- The opponent can still move freely inside the tangle.
- The opponent may also choose any available escape.
- So the opponent must eventually escape, or lose by staying inside the tangle.

Tangles

Intuition

A tangle is a strongly connected subgame where one player can win all plays that stay inside it.

Equivalently: in the strategy-restricted subgame, all cycles are winning for that player.

Relation with dominions

- A closed tangle is a dominion.
- A tangle is a local dominion:
  - the opponent may still escape,
  - but if the opponent stays inside, they lose.

Tangles

Intuition

A tangle is a strongly connected subgame where one player can win all plays that stay inside it.

Equivalently: in the strategy-restricted subgame, all cycles are winning for that player.

Tangle attractor

If all escapes from the tangle lead to Z, then the whole tangle can be attracted to Z at once.

Tangle attractor for player α

Add to Z exhaustively:

- every vertex of player α that *can* move to Z
- every vertex of player $\bar{\alpha}$ that *must* move to Z
- every learned α-tangle whose escapes all lead to Z

Tangles in parity game algorithms

Why tangles matter

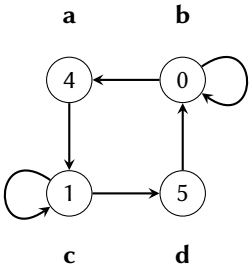
Tangles are the local winning structures inside parity games.

They are often present before a full dominion has become visible.

Why algorithms should care about tangles

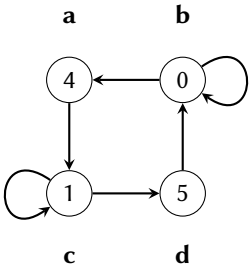
- every dominion decomposes into a structure of tangles
- most algorithms encounter tangles only implicitly
- they may rediscover the same tangle many times
- making tangles explicit can avoid redundant work
- tangle-aware algorithms can learn tangles and reuse them later

Distractions



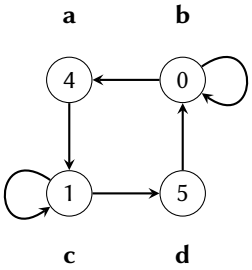
- **Find the winning strategy for player Even.**

Distractions



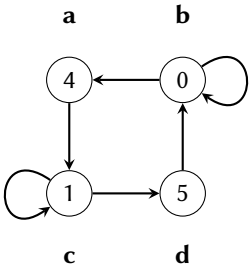
- **Find the winning strategy for player Even.**
- A random vertex inside the winning region?

Distractions



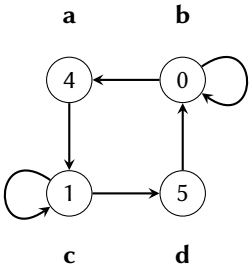
- **Find the winning strategy for player Even.**
- A random vertex inside the winning region?
  If you play from **c** to **c** you lose.

Distractions



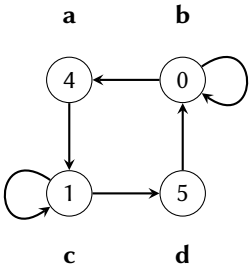
- **Find the winning strategy for player Even.**
- A random vertex inside the winning region?
  If you play from **c** to **c** you lose.
- Always play to the highest reachable even vertex?

Distractions



- **Find the winning strategy for player Even.**
- A random vertex inside the winning region?
  If you play from **c** to **c** you lose.
- Always play to the highest reachable even vertex?
  If you play from **b** to **a** you lose.

Distractions



- **Find the winning strategy for player Even.**
- A random vertex inside the winning region?
  If you play from **c** to **c** you lose.
- Always play to the highest reachable even vertex?
  If you play from **b** to **a** you lose.
- Vertex **a** is a distraction for player Even

Distractions

Intuition

A distraction for α is a vertex with an α-priority that the opponent $\overline{\alpha}$ can win if player α always tries to visit it.

Player α must *sometimes/always* avoid the distraction.

Definition

A distraction for player α is a vertex v with an α-priority p, such that if player α always plays to reach v along paths of priorities $\leq p$, then player $\overline{\alpha}$ wins v and all vertices that reach v.

(Along these paths to v, player $\overline{\alpha}$ either must or chooses to play to v.)

Distractions

A vertex v of parity α is an α-distraction if its priority makes it look attractive for player α, but moving toward v can still be the wrong choice.

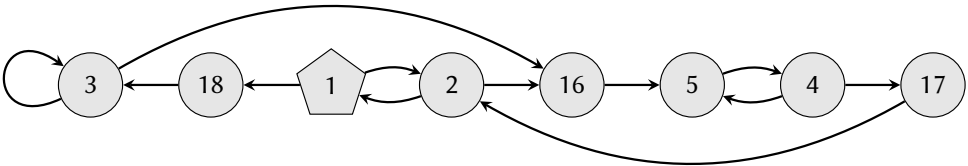
Operational meaning

A distraction can mislead local reasoning:

- some vertices may appear to benefit from reaching v,
- but if play is steered toward v, the opponent can exploit this,
- for example by forcing a dominion, or a higher priority of opposite parity.

Even inside a winning region or a tangle, a vertex may still be a distraction: it may be safe to visit, but not the right target from every predecessor. A distraction is not necessarily a losing vertex; it is a misleading vertex.

Distractions



Vertex **16** is a (non-trivial) distraction.
Play **2** → **1** instead of **2** → **16** to win!
But also play **3** → **16**!

Oink

- Modern (parallel) implementation of parity game algorithms
  - `zlk`: Zielonka's Recursive algorithm
  - `tl`: tangle learning
  - `fpi`: distraction fixed point iteration
  - `psi`: strategy Improvement
  - `spm`: small progress measures
  - `sspm`: succinct progress measures
  - `qpt`: ordered progress measures
  - `pmtl`: progress measures & tangle learning
- Standard preprocessing algorithms
  - inflation and compression
  - removing self-loops
  - detecting winner-controlled winning cycles
  - SCC decomposition
- `https://www.github.com/trolando/oink`
- Simple C++ library for solving parity games

Oink

Easy to use

```
#include "oink.hpp"

pg::Game parity_game;
parity_game.parse_pgsolver(cin);

pg::Oink solver(parity_game);
solver.setSolver("tl");
solver.run();

parity_game.write_sol(cout);
```

Easy to extend

- Implement Solver interface
- Add one line to `solvers.cpp`

Winner-controlled winning cycles

Idea

Some dominions are easy to detect: if player α controls a cycle whose highest priority has parity α, then that cycle may already form a winning region.

Algorithm

For every vertex $v \in V_\alpha$ with $\alpha = \mathrm{pr}(v) \bmod 2$:

- compute the restricted attractor to v through α-vertices of priority at most $\mathrm{pr}(v)$
- if the resulting region Z is closed, then Z is an α-dominion
- attract to Z from the whole game, remove it, and continue

Remark

This is a simple preprocessing rule; stronger variants use SCC-based reductions and related ideas such as fatal attractors (Huth, Kwo, Piterman, 2014).

Overview of parity game algorithms

Decomposition / attractor-based

Compute properties of sets of vertices using attractors, regions, dominions, and tangles.

- Zielonka's recursive algorithm
- tangle learning

Local optimization

Improve information locally, based on direct successors, until a global solution emerges.

- **value iteration**
  - small progress measures
  - succinct / ordered progress measures
- **policy iteration**
  - strategy improvement
- **fixed-point iteration**
  - FPI / FPJ

The fixed-point style is closely connected to the μ-calculus.

Decomposition-based algorithms

Use attractor computation to decompose the game into regions, and refine this decomposition until a dominion is found.

- choose a promising target for some player α and compute the attractor to that target
- continue recursively on the remaining subgame
- repair or refine the decomposition when needed

Region invariant

Inside an α-region:

- player α wins all plays that stay in the region
- player $\bar{\alpha}$ either
  - stays in the region and loses,
  - reaches the target,
  - or escapes to a better α-region

Examples of decomposition-based algorithms

Zielonka's recursive algorithm (1998)

Repeatedly attract to the highest-priority player, recurse on the remaining subgame, and repair the decomposition when the opponent can attract back.

Tangle learning (2018)

Extend attractor computation with tangles.

Each iteration may discover new tangles, and these learned tangles are then reused to refine the decomposition further.

Local optimization algorithms

Instead of decomposing large sets of vertices, these algorithms update local information from successor information until a global solution emerges.

Three styles

- **value iteration:** update a local value or measure until it stabilizes
- **policy iteration:** improve a current strategy step by step
- **fixed-point iteration:** iterate nested least/greatest fixed-point approximations

All three rely on some form of monotone improvement, but they differ in *what* is being improved.

Local optimization: value iteration

Each vertex carries a value or measure describing how favourable it currently looks.

- update each vertex from the values of its successors
- propagate better information backwards through the game
- repeat until the values stabilize

Monotonicity. The values improve monotonically.

Examples:

- small progress measures (1998)
- ordered progress measures (2017)
- succinct progress measures (2017)

Local optimization: policy iteration

The players optimize strategies rather than values.

- start with some strategy for the main player
- let the opponent compute a best response (e.g. nested policy iteration)
- improve the main player's strategy at one or more switchable vertices
- repeat

Monotonicity.

- the opponent's best response improves monotonically for the opponent
- the main player's successive strategies improve monotonically for the main player

So the algorithm alternates between two local optimization steps.

Local optimization: fixed-point iteration

Solve the game by updating local information in a strict priority order, just like evaluating a nested fixed-point formula.

- priorities are handled from low to high in a nested way
- lower-priority regions are iterated until they reach a fixed point
- if something changes at a higher priority, part of the lower-priority work is reset
- then the lower priorities are recomputed until they stabilize again

Monotonicity. Least fixed-point approximations grow monotonically; greatest fixed-point approximations shrink monotonically.

Examples:

- distraction fixed-point iteration (FPI)
- fixed-point iteration with justifications (FPJ).

Decomposition based algorithms

Zielonka's recursive algorithm

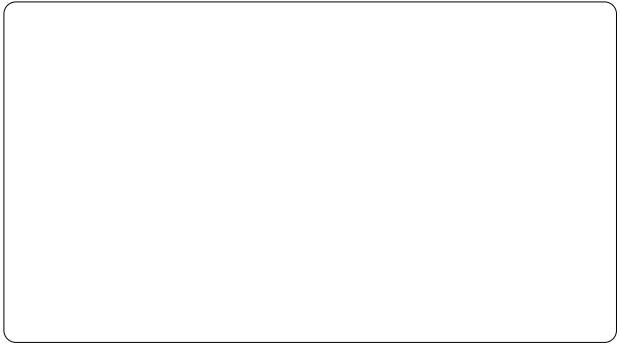
Outline

- Recursive algorithm by McNaughton (1993), Zielonka (1998)
- Decomposition of the game into regions
- Initially assume highest vertex to be "good" for associated player
- Recompute recursively if this was not the case

Recent variations run in quasi-polynomial time (Parys 2019, Lehtinen 2019, Jurdziński 2020).

Recursive structure

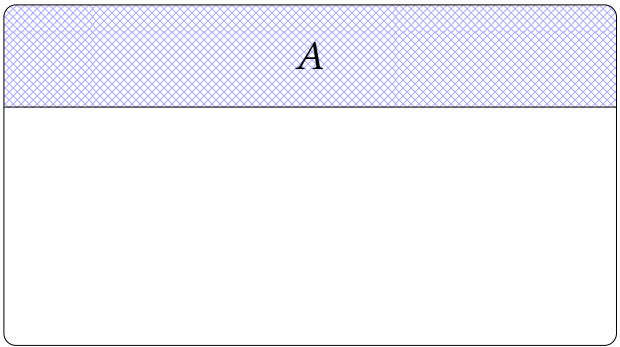
Set A := attractor to highest-priority vertex



Task: solve the game recursively

Recursive structure

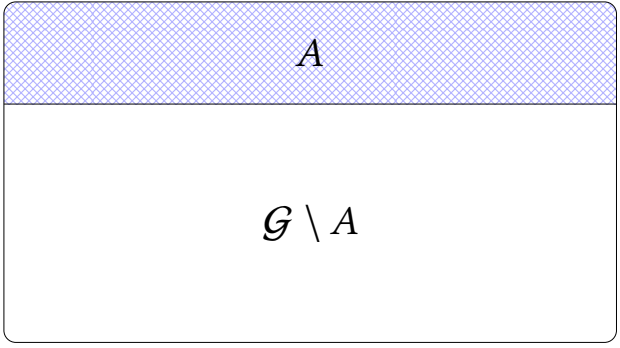
Set A := attractor to highest-priority vertex



Task: solve the game recursively

Recursive structure

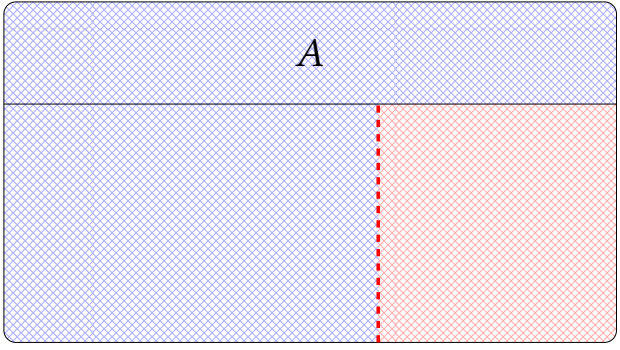
Recursively solve $\mathcal{G} \setminus A$



Task: solve the game recursively

Recursive structure

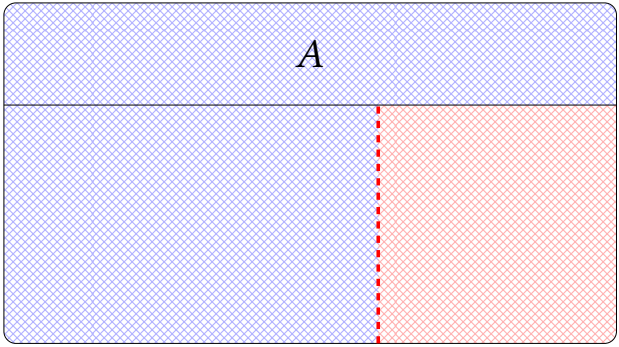
Recursively solve $\mathcal{G} \setminus A$



Task: solve the game recursively

Recursive structure

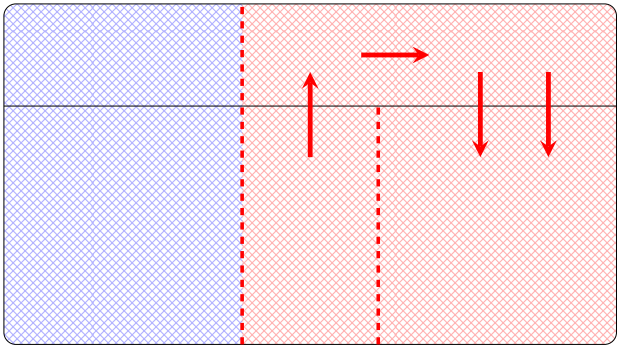
Attract from blue to red (if nothing attracted, then done!)



Task: solve the game recursively

Recursive structure

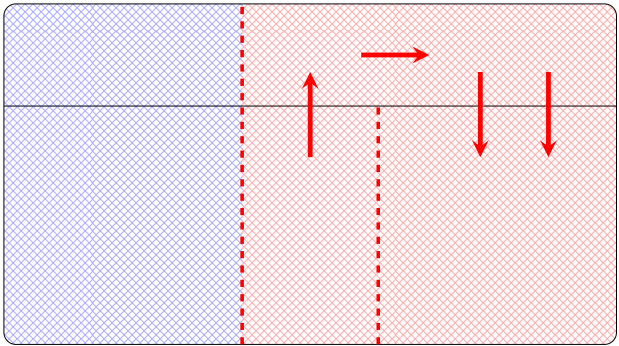
Attract from blue to red (if nothing attracted, then done!)



Task: solve the game recursively

Recursive structure

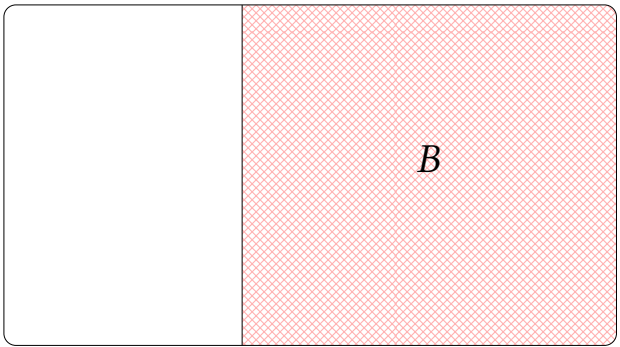
Set B := all vertices attracted to red



Task: solve the game recursively

Recursive structure

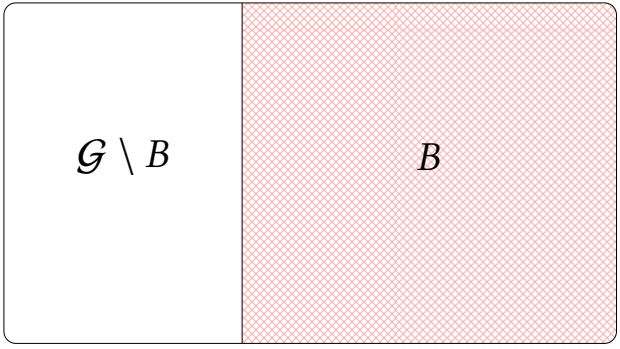
Set B := all vertices attracted to red



Task: solve the game recursively

Recursive structure

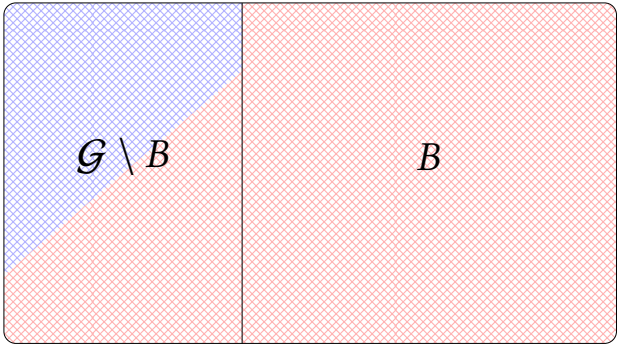
Recursively solve $\mathcal{G} \setminus B$



Task: solve the game recursively

Recursive structure

Recursively solve $\mathcal{G} \setminus B$



Task: solve the game recursively

Recursive structure

Game solved!

Task: solve the game recursively

Zielonka's recursive algorithm

```
1  def zielonka(𝒢):
2      if 𝒢 = ∅ :
3          return ∅, ∅                                          // empty game
4      α ← pr(𝒢) mod 2                                          // winner of highest priority
5      Z ← pr^{-1}(pr(𝒢))                                       // vertices of highest priority
6      A ← attract(𝒢, Z, V, α)                                  // attracted to highest priority
7      W_●, W_⬟ ← zielonka(𝒢 \ A)                               // recursive solution
8      B ← attract(𝒢, W_ᾱ, V, ᾱ)                               // check if opponent attracts
9      if B = W_ᾱ :
10         W_α ← W_α ∪ A                                        // A is won by α
11     else:
12         W_●, W_⬟ ← zielonka(𝒢 \ B)                           // recompute remainder
13         W_ᾱ ← W_ᾱ ∪ B                                        // B is won by ᾱ
14     return W_●, W_⬟
```

Zielonka's recursive algorithm

Computing strategy

- Strategy is computed by attractor
  - For every attracted α-vertex u, store a chosen edge $u \longrightarrow v$ into the attracted set
- Special case: α-vertices of the original target set
  - Pick any successor in winning region as strategy
- Implementation: use only a single strategy array, reset the strategy of target vertices before attracting

Zielonka's recursive algorithm

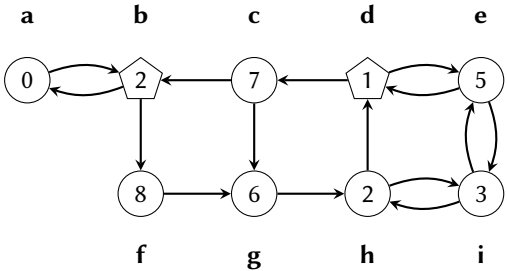
```
1  def zielonka(𝒢):
2      if 𝒢 = ∅ :
3          return ∅, ∅                                          // empty game
4      α ← pr(𝒢) mod 2                                          // winner of highest priority
5      Z ← pr^{-1}(pr(𝒢))                                       // vertices of highest priority
6      A, σ_A ← attract(𝒢, Z, V, α, ∅)                          // attracted to highest priority
7      W_●, W_⬟, σ_●, σ_⬟ ← zielonka(𝒢 \ A)                      // recursive solution
8      B, σ_ᾱ ← attract(𝒢, W_ᾱ, V, ᾱ, σ_ᾱ)                      // check if opponent attracts
9      if B = W_ᾱ :
10         W_α ← W_α ∪ A                                        // A is won by α
11         σ_α ← σ_α ∪ σ_A ∪ ((z ∈ Z) ↦ pick(E(z) ∩ W_α))
12     else:
13         W_●, W_⬟, σ_●, σ_⬟ ← zielonka(𝒢 \ B)                 // recompute remainder
14         W_ᾱ ← W_ᾱ ∪ B                                        // B is won by ᾱ
15         σ_ᾱ ← σ_ᾱ ∪ σ_B
16     return W_●, W_⬟, σ_●, σ_⬟
```

Zielonka's recursive algorithm

Another perspective

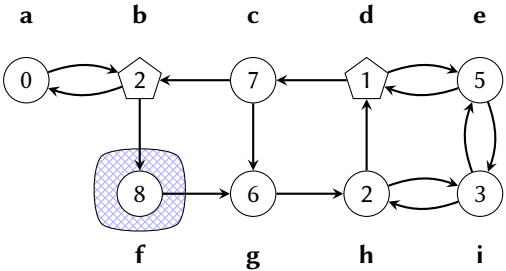
- Decompose top-down into regions
- Going back up, attract from the player's regions to the opponent's lower regions
- If so, reset and recompute the player's lower regions

Zielonka's Algorithm



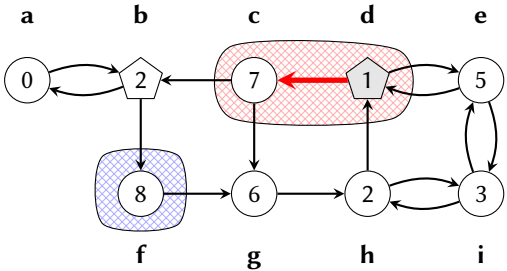
- We start by attracting to 8 for player Even.

Zielonka's Algorithm



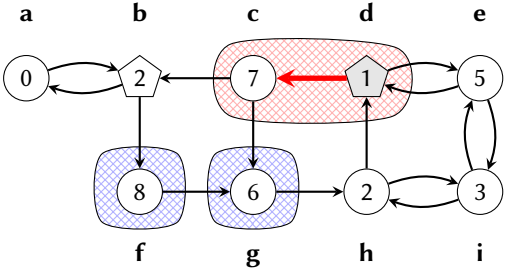
- After region 8 (player Even).
- Continue (recursively) with region 7.

Zielonka's Algorithm



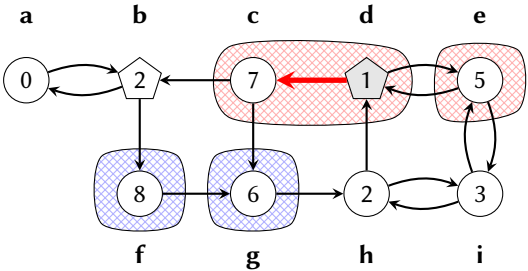
- After regions 8 (player Even) and 7 (player Odd).
- Continue (recursively) with region 6.

Zielonka's Algorithm



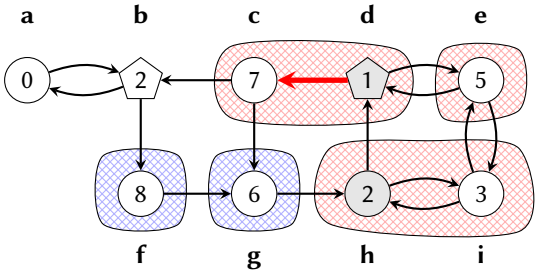
- After regions 8, 7 and 6.
- Continue (recursively) with region 5.

Zielonka's Algorithm



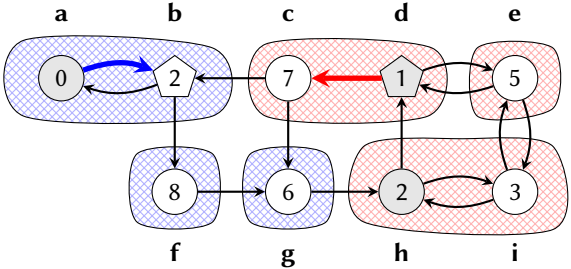
- After regions 8, 7, 6 and 5.
- Continue (recursively) with region 3.

Zielonka's Algorithm



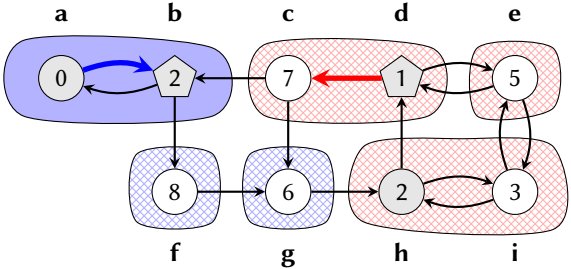
- After regions 8, 7, 6, 5 and 3.
- Now remains just region 2.

Zielonka's Algorithm



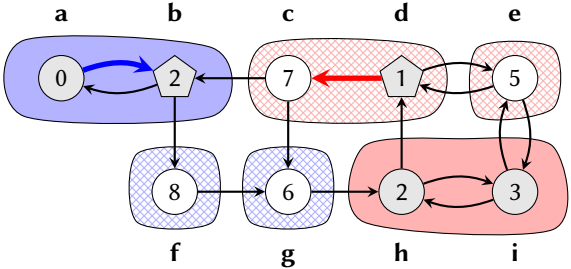
- Game is partitioned fully, now go up in the recursion.
- Up in region 2, does the lower opponent's winning region attract?
- Region 2: no (because the subgame is empty).

Zielonka's Algorithm



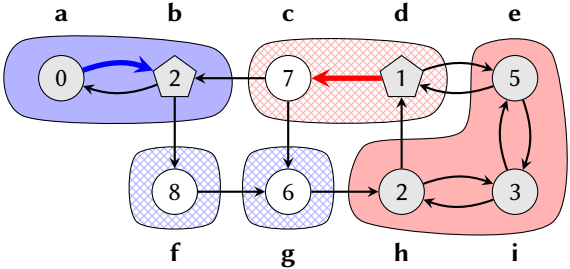
- Up in region 3: does the lower Even region attract from region 3?

Zielonka's Algorithm



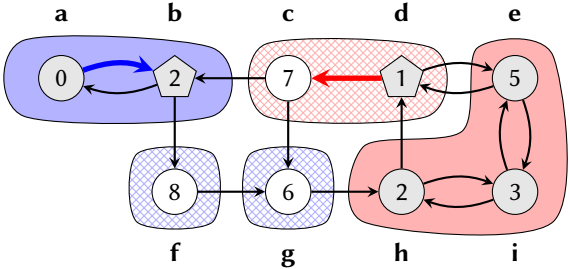
- Up in region 5: does the lower Even region attract from region 5?

Zielonka's Algorithm



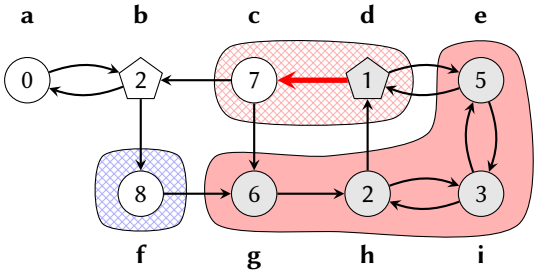
- Up in region 6: does the lower Odd region attract from 6?

Zielonka's Algorithm



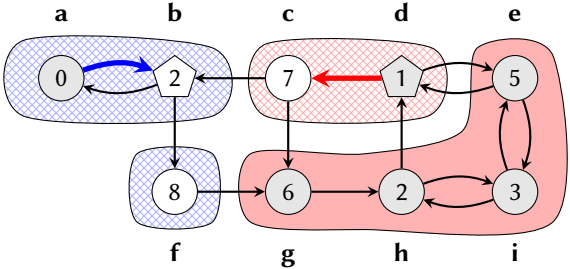
- Up in region 6: does the lower Odd region attract from 6?
- Yes: the lower Odd region attracts vertex **g**.
- So: attract to Odd, reset and recompute lower.

Zielonka's Algorithm



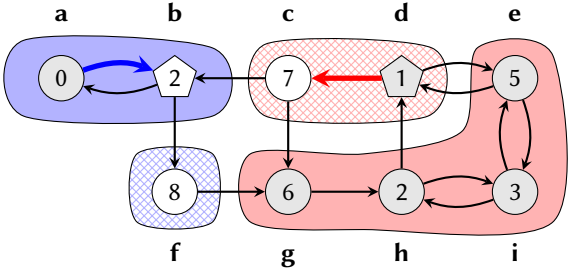
- Vertex **g** is attracted to the Odd region.
- So now recompute the (remainder of the) lower regions of Even.

Zielonka's Algorithm



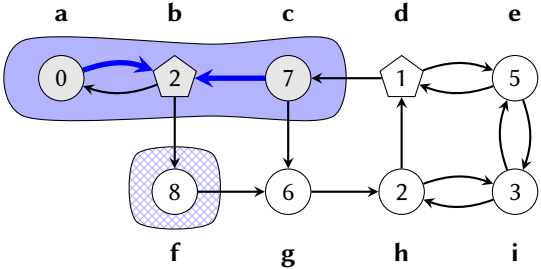
- Vertex **g** is attracted to the Odd region.
- So now recompute the (remainder of the) lower regions of Even.
- Recompute region 2...

Zielonka's Algorithm



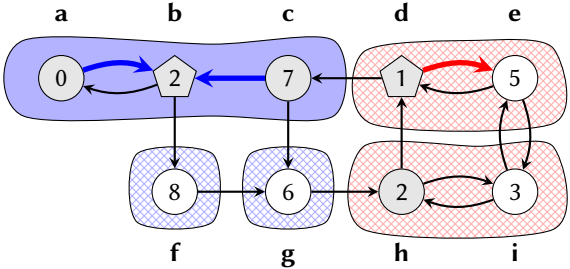
- Vertex **g** is attracted to the Odd region.
- So now recompute the (remainder of the) lower regions of Even.
- Recompute region 2... not attracted to lower Odd...
- Next: region 7: does the lower Even region attract from 7?
- Yes, the lower Even region attracts vertex **c**.
- Attract to Even, reset lower!

Zielonka's Algorithm



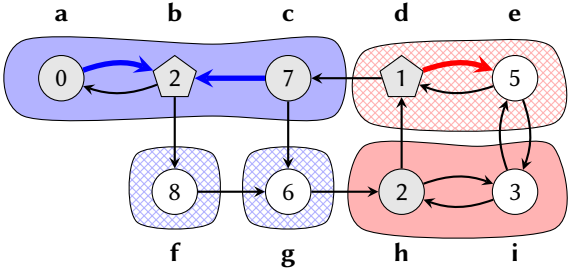
- Vertex **c** is attracted to the Even region.
- Recompute the remainder of the lower regions of Odd.

Zielonka's Algorithm



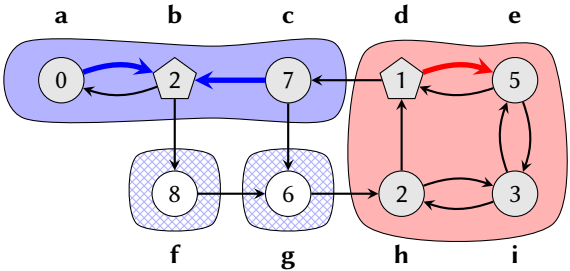
- Partition the remainder into regions 6, 5 and 3.
- Up in region 3: no attraction from Even.
- Up in region 5: no attraction from Even.
- Up in region 6: the lower Odd regions attract **g** again!

Zielonka's Algorithm



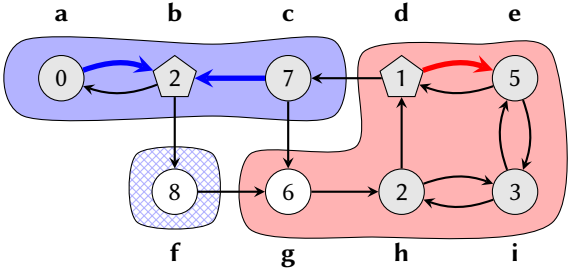
- Partition the remainder into regions 6, 5 and 3.
- Up in region 3: no attraction from Even.
- Up in region 5: no attraction from Even.
- Up in region 6: the lower Odd regions attract **g** again!

Zielonka's Algorithm



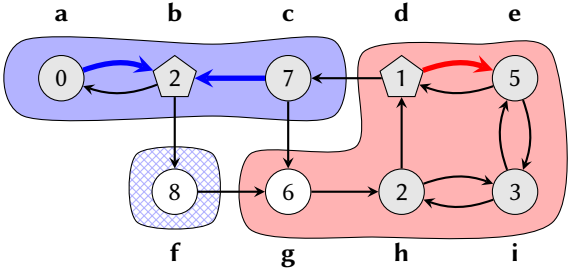
- Region 6: now the Odd region attracts vertex **g** again.
- Attract to Odd, reset the lower game.

Zielonka's Algorithm



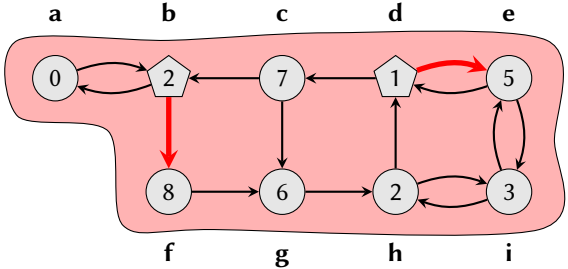
- Vertex **g** is attracted to the Odd region.
- Recursive game of 6 is empty.
- Up to region 8, does Odd now attract 8?
- Yes: vertex **f** is attracted to the Odd region.
- Attracting to Odd now attracts all remaining vertices...

Zielonka's Algorithm



- Vertex **g** is attracted to the Odd region.
- Recursive game of 6 is empty.
- Up to region 8, does Odd now attract 8?

Zielonka's Algorithm



- Final result, entire game won by player Odd.

From Zielonka to tangle learning

If a region in the decomposition is locally closed, then it is a subgame that is winning for one player.

In fact, such a region contains one or more tangles.

So what can we do?

- learn tangles from every locally closed region
- use these tangles inside the attractor computation
- restart the decomposition with the stronger attractor

Idea of tangle learning. Remember and reuse tangles instead of rediscovering them recursively.

Tangle learning

Tangle attractor

Because the opponent $\overline{\alpha}$ **must** escape the tangle, we can use tangles to attract all vertices of a tangle simultaneously.

Add to Z (exhaustively):

- all vertices of the player α that *can* play to Z
- all vertices of opponent $\overline{\alpha}$ that *must* play to Z
- all vertices in an α-tangle where all escapes lead to Z

Escapes:

- All successors from the set of vertices U:
$$E(U) := \{v \in E(u) \mid u \in U\}$$
- All escapes from a tangle (U, σ):
$$E_T(U) := E(U_{\overline{\alpha}}) \setminus U$$

Tangle learning

- Initialize a set of target vertices V_T
  - Can be all vertices in the game
  - Can be all vertices with even/odd priority
  - Can be a selection of vertices

Tangle learning

- Initialize a set of target vertices V_T
  - Can be all vertices in the game
  - Can be all vertices with even/odd priority
  - Can be a selection of vertices
- Partition game into regions with tangle attractor
  - Attract to highest priority vertex in V_T (only $\leq$ priority vertices)
  - Repeat with the remainder until nothing left

Tangle learning

- Initialize a set of target vertices V_T
  - Can be all vertices in the game
  - Can be all vertices with even/odd priority
  - Can be a selection of vertices
- Partition game into regions with tangle attractor
  - Attract to highest priority vertex in V_T (only $\leq$ priority vertices)
  - Repeat with the remainder until nothing left
- Every "locally closed" region contains a new tangle
  - Vertices of α can stay in the region
  - Vertices of $\overline{\alpha}$ can only play to higher α-regions
  - Compute the tangle using a restricted Tarjan's SCC algorithm

Tangle learning

- Initialize a set of target vertices V_T
  - Can be all vertices in the game
  - Can be all vertices with even/odd priority
  - Can be a selection of vertices
- Partition game into regions with tangle attractor
  - Attract to highest priority vertex in V_T (only $\leq$ priority vertices)
  - Repeat with the remainder until nothing left
- Every "locally closed" region contains a new tangle
  - Vertices of α can stay in the region
  - Vertices of $\overline{\alpha}$ can only play to higher α-regions
  - Compute the tangle using a restricted Tarjan's SCC algorithm
- Every closed tangle is a winning region (dominion)
  - After finding a dominion, maximize it with another attractor
  - Remove maximized dominions from the game

Tangle learning

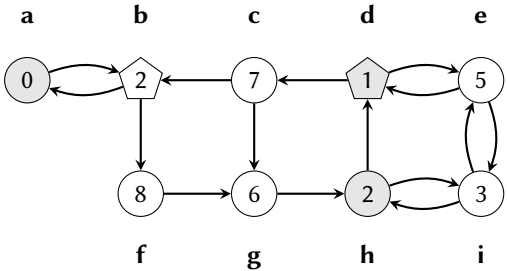
- Initialize a set of target vertices V_T
  - Can be all vertices in the game
  - Can be all vertices with even/odd priority
  - Can be a selection of vertices
- Partition game into regions with tangle attractor
  - Attract to highest priority vertex in V_T (only $\leq$ priority vertices)
  - Repeat with the remainder until nothing left
- Every "locally closed" region contains a new tangle
  - Vertices of α can stay in the region
  - Vertices of $\overline{\alpha}$ can only play to higher α-regions
  - Compute the tangle using a restricted Tarjan's SCC algorithm
- Every closed tangle is a winning region (dominion)
  - After finding a dominion, maximize it with another attractor
  - Remove maximized dominions from the game
- Repeat this procedure until the game is solved

Tangle learning

Summary

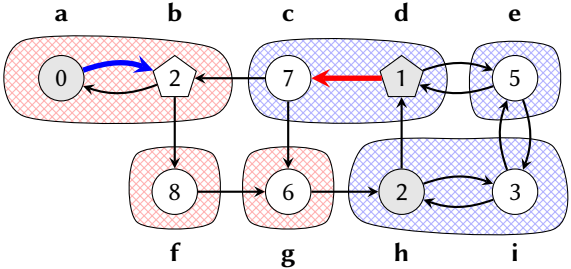
- Decompose into regions with the set of target vertices V_T
- Find new tangles and restart
- Closed tangles solve the parity game

Tangle learning



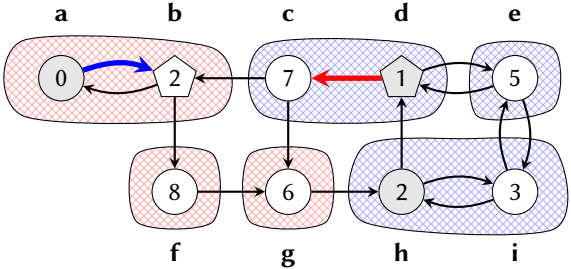
- First partition top-down into α-maximal regions

Tangle learning



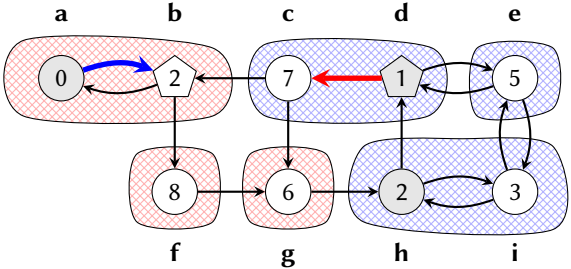
- Which regions are now closed?

Tangle learning



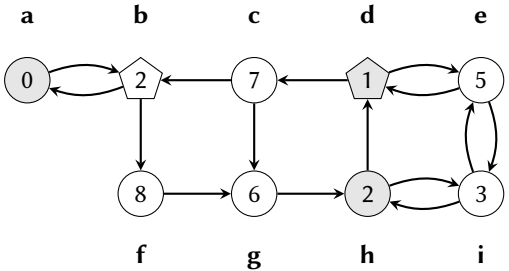
- Which regions are now closed?
- Regions 2 and 3 are locally closed (in their subgame)

Tangle learning



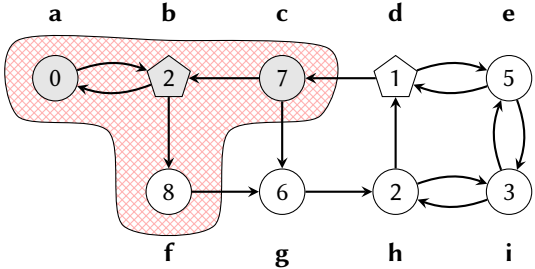
- Which regions are now closed?
- Regions 2 and 3 are locally closed (in their subgame)
- Tangle $(\{\mathbf{a}, \mathbf{b}\}, \{\mathbf{a} \to \mathbf{b}\})$ (attracted to 8)
- Tangle $(\{\mathbf{h}, \mathbf{i}\}, \emptyset)$ (attracted to 5)

Tangle learning



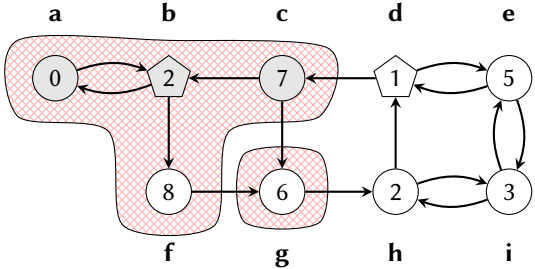
- Again partition top-down into α-maximal regions
- Now we have the two new tangles!

Tangle learning



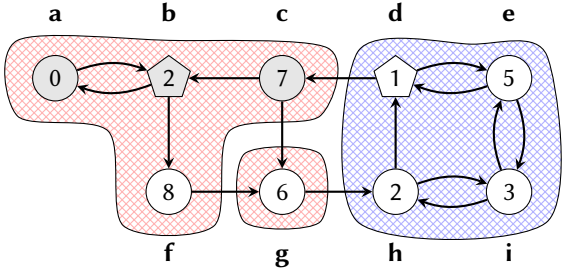
- Tangles: { **a**, **b** } (for Even) and { **h**, **i** } (for Odd)

Tangle learning



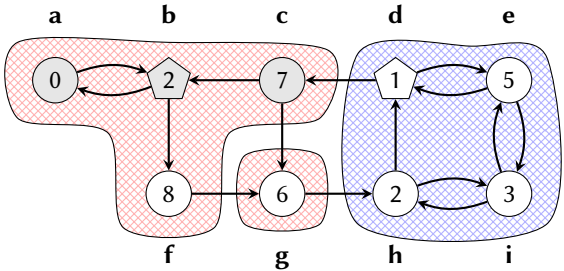
- Tangles: { **a**, **b** } (for Even) and { **h**, **i** } (for Odd)

Tangle learning



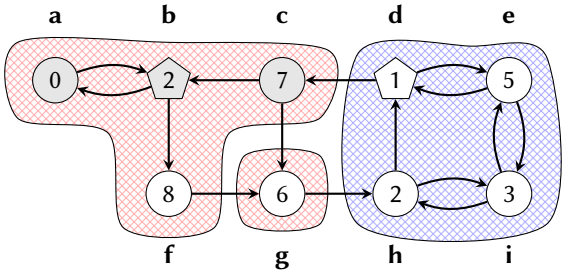
- Tangles: { **a**, **b** } (for Even) and { **h**, **i** } (for Odd)

Tangle learning



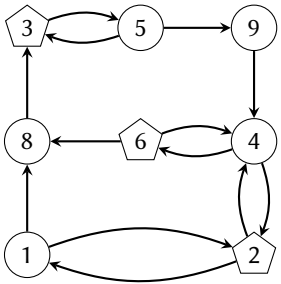
- Tangles: { **a**, **b** } (for Even) and { **h**, **i** } (for Odd)
- Which regions are now closed?

Tangle learning

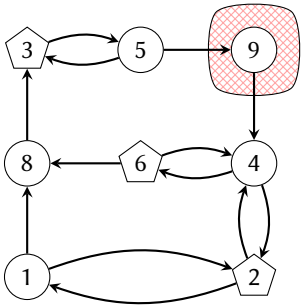


- Tangles: { **a**, **b** } (for Even) and { **h**, **i** } (for Odd)
- Which regions are now closed?
- Only closed region: 5
- One tangle, which is also a dominion

Tangle learning



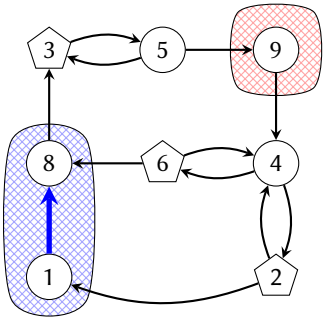
Tangle learning



Regions:

- 9 (open)

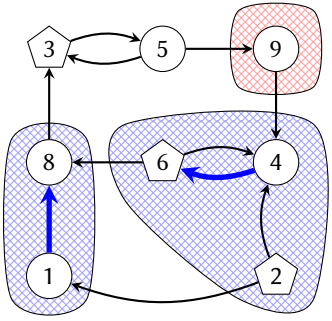
Tangle learning



Regions:

- 9 (open)
- 8 1 (open)

Tangle learning

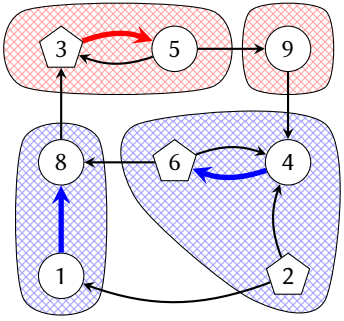


Regions:

- 9 (open)
- 8 1 (open)
- 6 4 2 (closed)

Learned tangles: $\{6, 4 \rightarrow 6\}$

Tangle learning

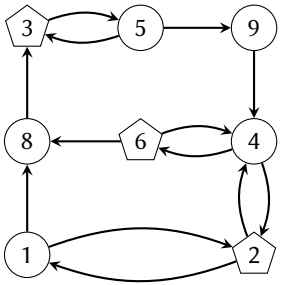


Regions:

- 9 (open)
- 8 1 (open)
- 6 4 2 (closed)
- 5 3 (closed)

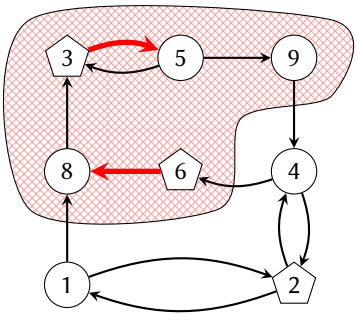
Learned tangles: $\{6, 4 \rightarrow 6\}$, $\{5, 3 \rightarrow 5\}$

Tangle learning



Learned tangles: $\{6, 4 \rightarrow 6\}$, $\{5, 3 \rightarrow 5\}$

Tangle learning

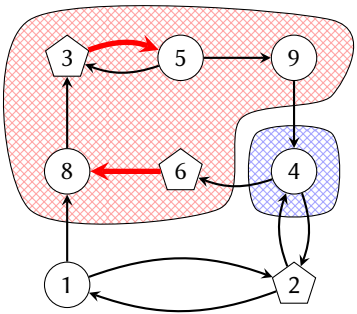


Regions:

- 9 5 3 8 6 (open)

Learned tangles: $\{6, 4 \rightarrow 6\}$, $\{5, 3 \rightarrow 5\}$

Tangle learning

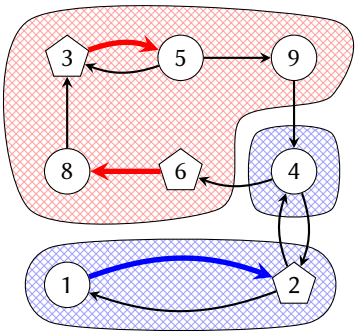


Regions:

- 9 5 3 8 6 (open)
- 4 (open)

Learned tangles: $\{6, 4 \rightarrow 6\}$, $\{5, 3 \rightarrow 5\}$

Tangle learning

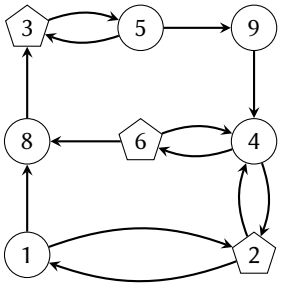


Regions:

- 9 5 3 8 6 (open)
- 4 (open)
- 2 1 (closed)

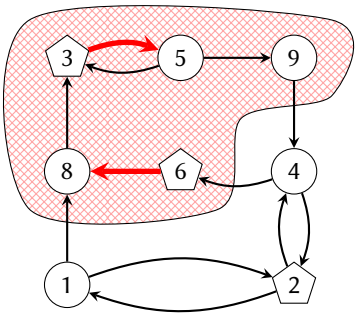
Learned tangles: $\{6, 4 \to 6\}$, $\{5, 3 \to 5\}$, $\{2, 1 \to 2\}$

Tangle learning



Learned tangles: $\{6, 4 \to 6\}$, $\{5, 3 \to 5\}$, $\{2, 1 \to 2\}$

Tangle learning

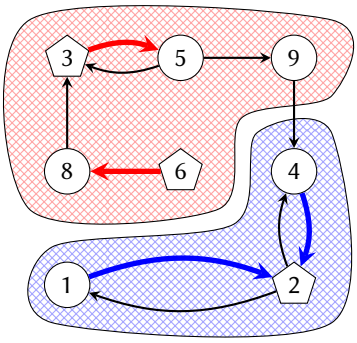


Regions:

- 9 5 3 8 6 (open)

Learned tangles: $\{6, 4 \to 6\}$, $\{5, 3 \to 5\}$, $\{2, 1 \to 2\}$

Tangle learning

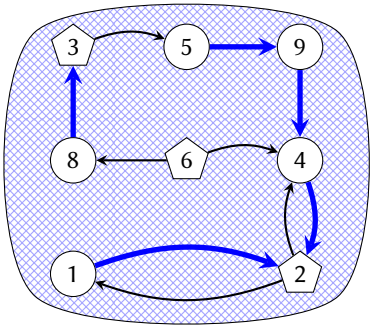


Regions:

- 9 5 3 8 6 (open)
- 4 2 1 (closed)

Learned tangles: $\{6, 4 \to 6\}$, $\{5, 3 \to 5\}$, $\{2, 1 \to 2\}$, $\underline{\{4 \to 2, 2, 1 \to 2\}}$

Tangle learning



Learned tangles: $\{6, 4 \to 6\}$, $\{5, 3 \to 5\}$, $\{2, 1 \to 2\}$, $\underline{\{4 \to 2, 2, 1 \to 2\}}$

Tangle learning

```
1  def tangle-attract(𝒢, 𝒯, Z, R, α, σ):
2      Q ← Z                                          // initialize with all target vertices
3      while Q ≠ ∅ :
4          v ← take(Q)                                                  // take a v ∈ Q
5          foreach u : v ∈ E(u) do                                  // check incoming edges
6              if u ∈ R ∧ u ∉ Z ∧ (u ∈ V_α ∨ E(u) ⊆ Z) :                        // attract?
7                  Z ← Z ∪ {u}
8                  Q ← Q ∪ {u}
9              if u ∈ Z ∩ V_α ∧ u ∉ σ : σ[u] ← v                          // set strategy?
10         foreach (U, τ) ∈ 𝒯_α : v ∈ E(U) do                  // check adjacent tangles
11             if U ⊆ R ∪ Z ∧ E_T(U) ⊆ Z :                                  // attract?
12                 U′ ← U \ Z                          // only vertices not yet in Z
13                 Z ← Z ∪ U
14                 Q ← Q ∪ U
15                 σ ← σ ∪ (τ ∩ U′)
16     return Z, σ
```

Tangle learning (1/2)

- `search` returns new tangles of $\mathcal{G}$, given known tangles $\mathcal{T}$.

```
1 def search(G, T):
2     if G = ∅ : return ∅
3     p ← pr(G)                                          // highest priority
4     α ← pr(G) mod 2                                    // player winning p
5     Z, σ ← tangle-attract(G,T,pr^{-1}(p),V,α,∅)        // region
6     if Z_α = dom(σ) ∧ E(Z_ᾱ) ⊆ Z :                     // no open vertices?
7         return search(G \ Z, T) ∪ bottom-sccs(Z, σ)
8     else:                                              // open vertices?
9         return search(G \ Z, T)
```

Tangle learning (2/2)

1 **def tanglelearning($\mathcal{G}$):**
2 $W_{\bullet} \leftarrow \emptyset$, $\sigma_{\bullet} \leftarrow \emptyset$, $W_{\pentagon} \leftarrow \emptyset$, $\sigma_{\pentagon} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$
3 **while** $\mathcal{G} \neq \emptyset$ **:**
4 $Y \leftarrow$ **search**$(\mathcal{G}, \mathcal{T})$
5 $\mathcal{T} \leftarrow \mathcal{T} \cup \{ (U, \tau) \in Y \mid E_T(U) \neq \emptyset \}$
6 $D \leftarrow \{ (U, \tau) \in Y \mid E_T(U) = \emptyset \}$
7 **if** $D \neq \emptyset$ **:**
8 $D^+_{\bullet}, \sigma \leftarrow$**tangle-attract** $(\mathcal{G}, \mathcal{T}, \bigcup D_{\bullet}, V, \bullet, \bigcup D_{\bullet})$
9 $W_{\bullet} \leftarrow W_{\bullet} \cup D^+_{\bullet}$, $\sigma_{\bullet} \leftarrow \sigma_{\bullet} \cup \sigma$
10 $D^+_{\pentagon}, \sigma \leftarrow$**tangle-attract** $(\mathcal{G}, \mathcal{T}, \bigcup D_{\pentagon}, V, \pentagon, \bigcup D_{\pentagon})$
11 $W_{\pentagon} \leftarrow W_{\pentagon} \cup D^+_{\pentagon}$, $\sigma_{\pentagon} \leftarrow \sigma_{\pentagon} \cup \sigma$
12 $\mathcal{G} \leftarrow \mathcal{G} \setminus (D^+_{\bullet} \cup D^+_{\pentagon})$
13 $\mathcal{T} \leftarrow \mathcal{T} \cap \mathcal{G}$
14 **return** $W_{\bullet}, W_{\pentagon}, \sigma_{\bullet}, \sigma_{\pentagon}$

Tangle learning

Computing strategy

- Compute strategy with the attractor, select any successor in the region for the highest priority vertices of α
- Store the σ of every tangle and use the stored σ as the strategy for α when attracting a tangle
- If a region is closed, then all α's vertices have a strategy
  - Suffices to look at the "target vertices" (highest priority)

Implementation notes

- Store tangle as interleaved list: $\alpha, u_0, \sigma(u_0), u_1, \sigma(u_1), \ldots, u_n, \sigma(u_n)$ (the player winning the tangle, followed by the vertices and the strategy)
- Store a separate list of escapes $E_T(U)$
- Don't get distracted by the mathematical notation... Put aside logic. Do what feels right.

Empirical evaluation

benchmark	fpi	fpj	zlk	tl	rtl	ortl
synthesis	36.51	**0.08**	36.53	**0.08**	**0.16**	**0.16**
model checking	74.10	79.82	50.96	**27.38**	42.22	59.00
equiv. checking	346.89	280.79	150.21	**124.27**	207.87	267.56

Cumulative runtimes in seconds, only on benchmarks solved by all solvers.

Source: Van Dijk, *Avoiding Distractions in Parity Games*, ISoLA 2024.

Fixed-point iteration

Fixed-point iteration

Core idea.

Instead of translating a fixed-point formula into a parity game and then solving the game, we can also go *the other way around*:

interpret the parity game itself as a nested fixed-point computation

Operationally, this gives an algorithm that:

- updates local information, like local optimization methods,
- but in a strict nested order with resets, like fixed-point evaluation.

Nested priorities as nested fixed points

Conceptual view.

Nested priorities correspond to nested fixed points:

$$\nu X_d.\ \mu X_{d-1}.\ \nu X_{d-2}.\ \cdots \quad \underbrace{\mathrm{AV}(X_d, X_{d-1}, X_{d-2}, \ldots)}_{\text{avoid worse priorities, visit better ones}}$$

Intuition from Even's perspective.

- ν-levels (typically even priorities) start optimistic and may shrink: some even-priority vertices turn out not to be safely winning for Even
- μ-levels (typically odd priorities) start pessimistic and may grow: some odd-priority vertices turn out to be good for Even after all

At each level, the AV step says: *avoid a more relevant losing priority, until a more relevant winning priority can be visited.*

So parity solving can be seen as evaluating a nested μ-calculus formula directly on the game.

Rewriting the fixed point using distractions

DFI idea.

Instead of maintaining alternating winning approximations directly, we maintain, for each priority p, a set Z_p of vertices of priority p that are currently known to be distractions.

$$\text{Distraction} := \mu Z_d \ldots \mu Z_1 . \mu Z_0 . \text{OnestepDistraction}\left(\bigvee_{p=0}^{d} (V_p \wedge Z_p)\right)$$

Why this is nice.

- all Z_p only grow
- so the whole computation uses only least fixed points
- operationally: whenever some Z_p changes, reset all lower $Z_{p-1}, \ldots, Z_0$ and recompute
- this is nice for a fast implementation

Different approaches

- (**BFL**) Via μ-calculus translation of Zielonka's recursive algorithm
  - Encoding to μ-calculus: *Walukiewicz, 1996*
  - As parity game solver: *Bruse, Falk, Lange, 2014*
  - Has complicated method to compute winning strategies!
- (**APT**) Via weak alternating automata
  - Encoding to weak alternating automata: *Kupfermann, Vardi, 1998*
  - As parity game solver: *Di Stasio, Murano, Perelli, Vardi, 2016*
  - Does not give winning strategies!
- (**DFI**) As the fixed point of distractions
  - *Van Dijk, Rubbens, 2019*
  - Uses frozen sets to find the winning strategy
  - This also speeds up the computation
- (**FPJ**) Using a justification graph
  - *Lapauw, Bruynooghe, Denecker, 2020*
  - Uses a justification graph to find the winning strategy
  - This also speeds up the computation

Fixed point iteration

Claims in *Van Dijk, Rubbens, 2019*

- The algorithms **BFL** and **APT** are actually equivalent
- Algorithm **DFI** without freezing is equivalent to **BFL** and **APT**
- Freezing enables trivial on-the-fly strategy computation

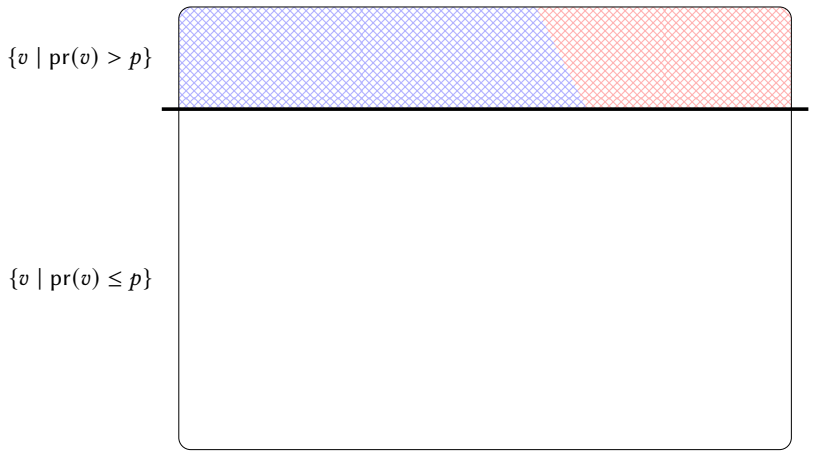
Fixed-point iteration

General idea

- Initially, mark even vertices as won by Even and odd vertices as won by Odd
- Then repair this guess, bottom-up through the priorities
- A vertex is updated whenever its current winner is contradicted by its direct successors
- FPI only looks one step ahead, but repeats this in a strict nested order

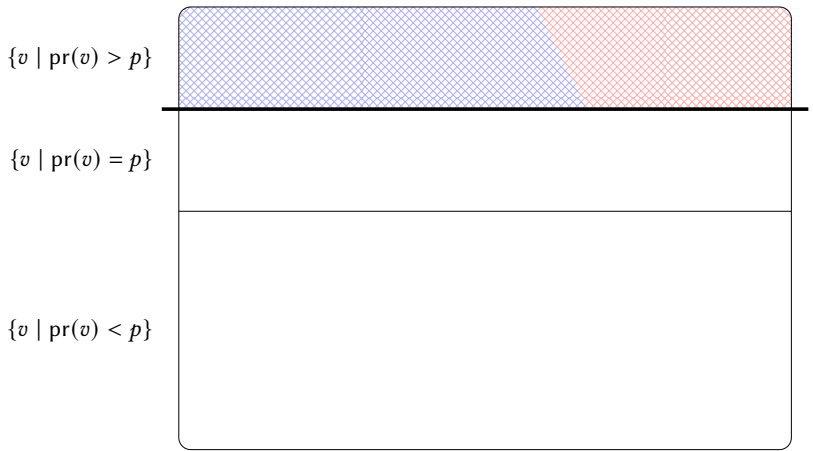
Recursive structure

Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



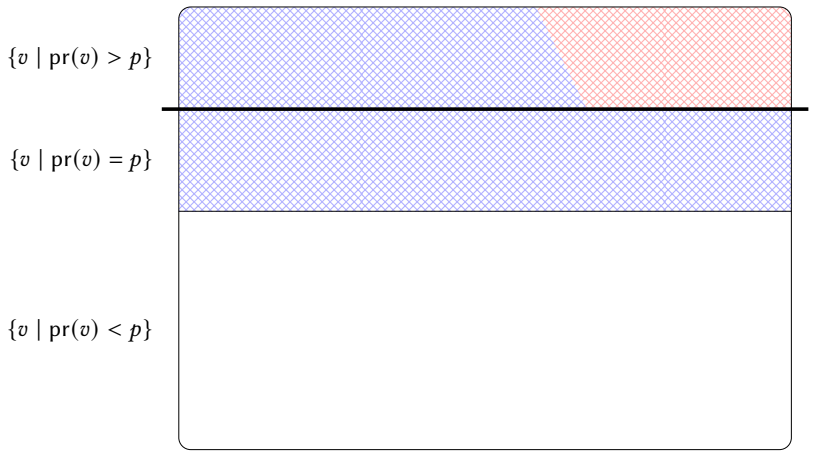
Recursive structure

Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



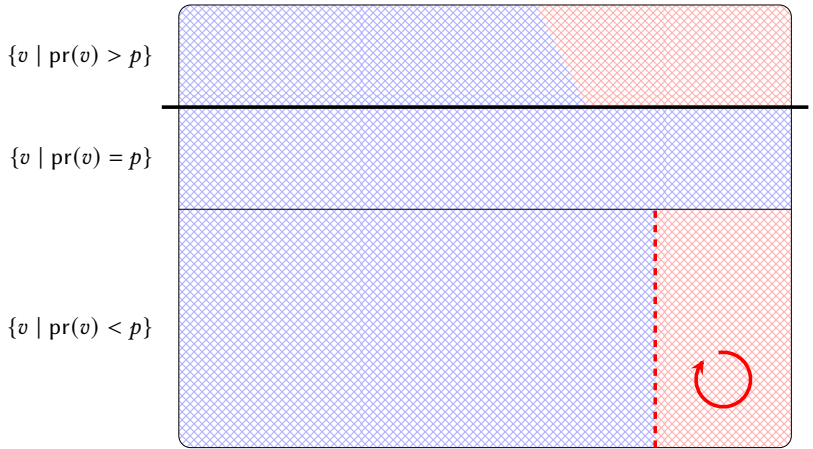
Recursive structure

Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



Recursive structure

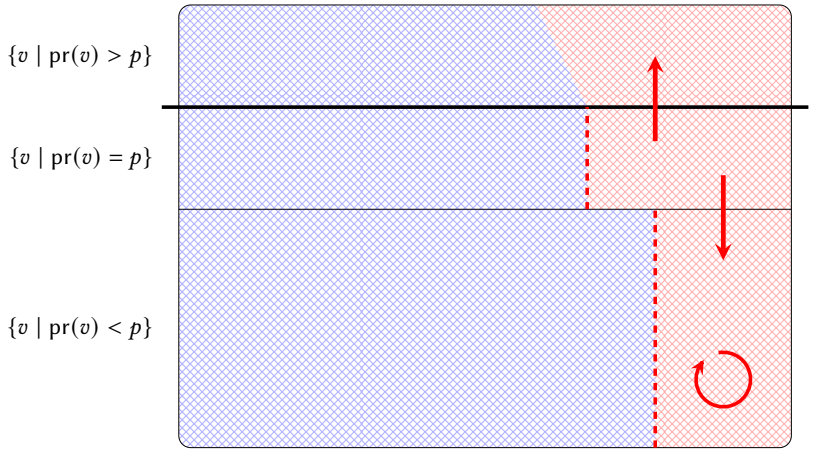
Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



No plays from the red subgame to blue p-vertices

Recursive structure

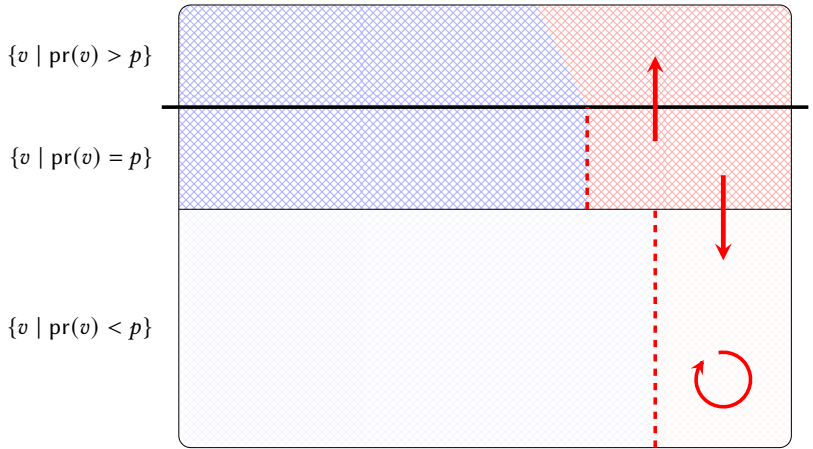
Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



Plays see each red p-vertex **at most once**

Recursive structure

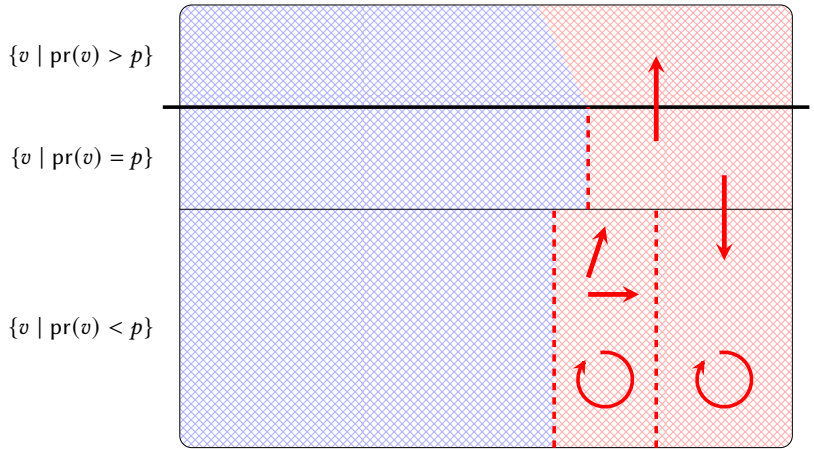
Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



Recompute subgame (only affects blue subgame)

Recursive structure

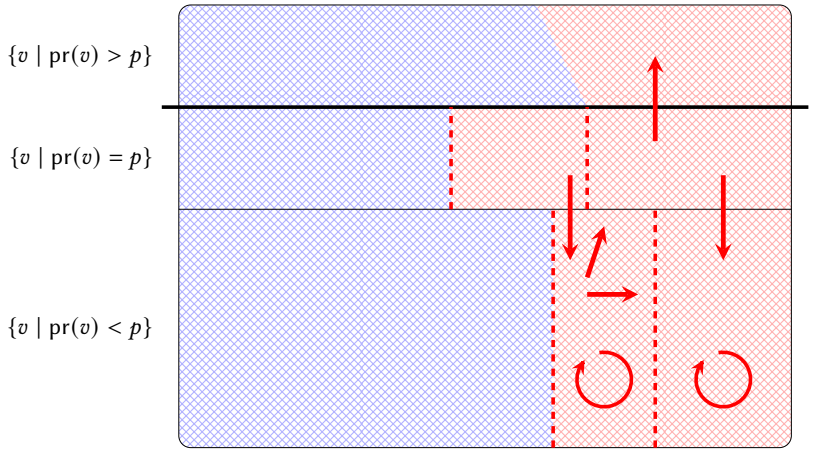
Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



No plays from the red subgame to blue p-vertices

Recursive structure

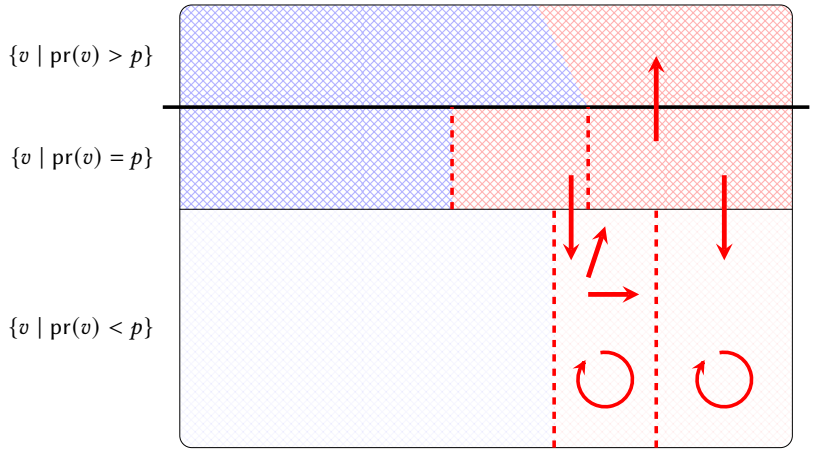
Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



Plays see each red p-vertex **at most once**

Recursive structure

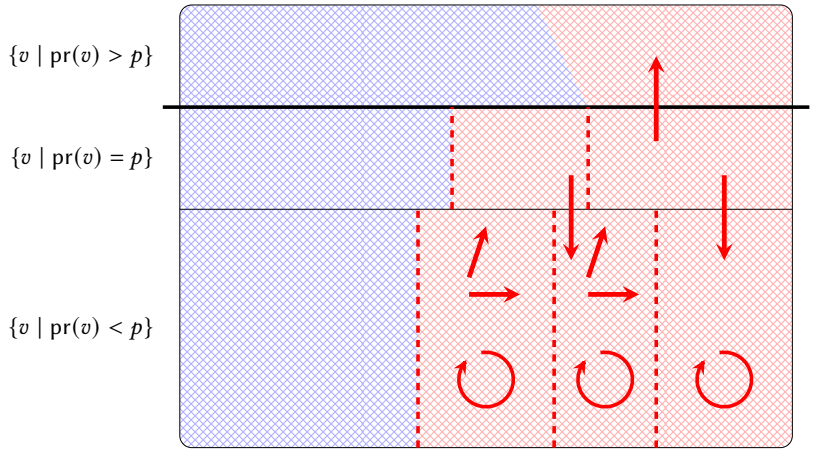
Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



Recompute subgame (only affects blue subgame)

Recursive structure

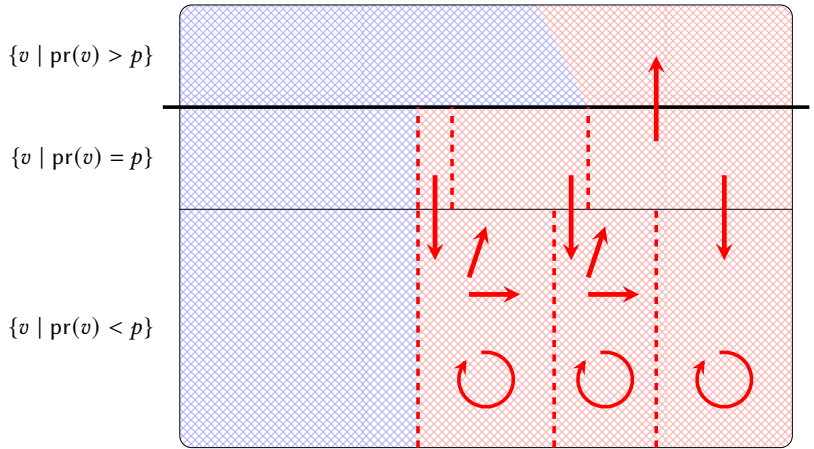
Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



No plays from the red subgame to blue p-vertices

Recursive structure

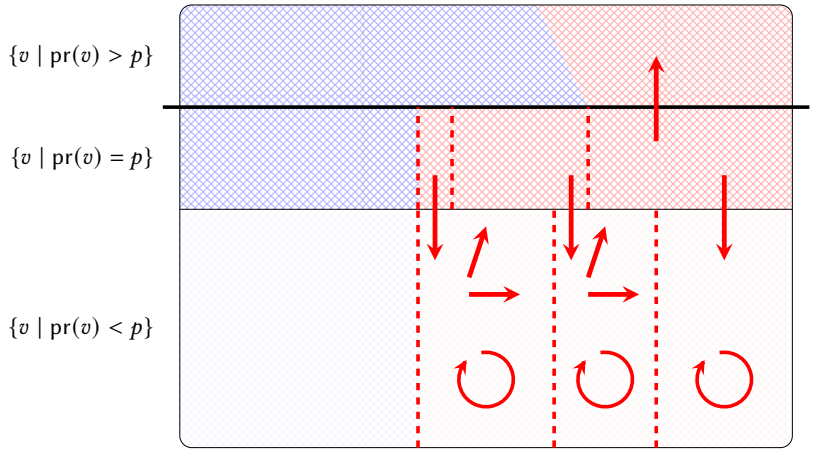
Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



Plays see each red p-vertex **at most once**

Recursive structure

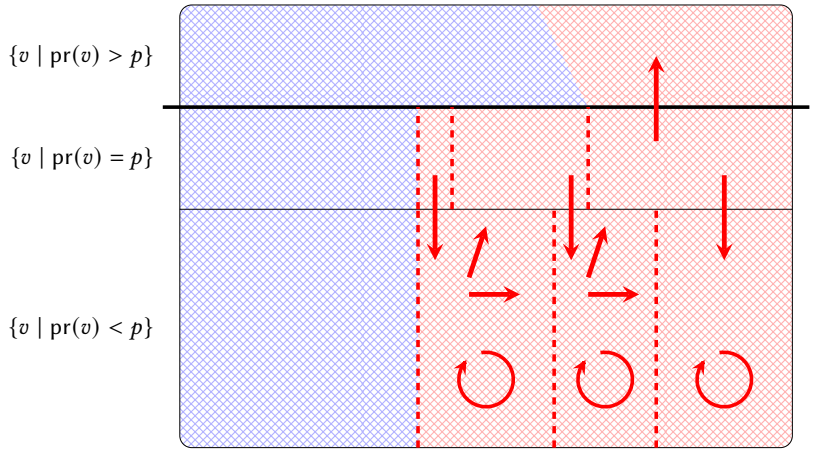
Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



Recompute subgame (only affects blue subgame)

Recursive structure

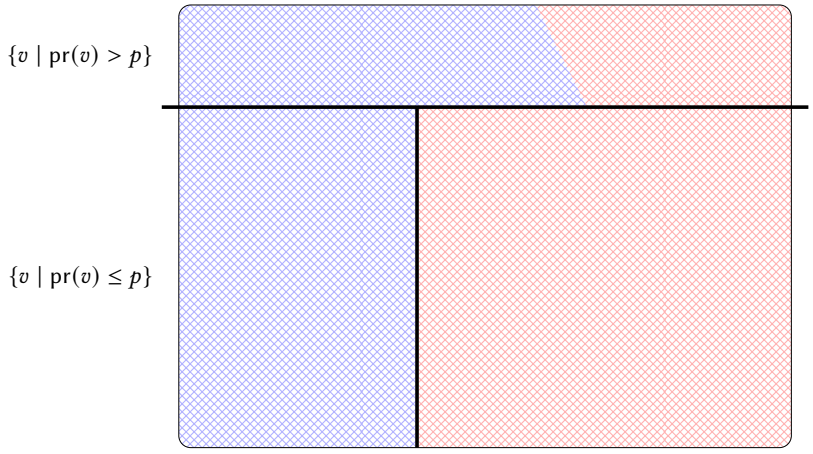
Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



No change: fixpoint

Recursive structure

Task: compute the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



This solves the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$

Fixed-point iteration

Distraction Fixed-point Iteration (DFI)

- Maintain sets $Z_p \subseteq V^p$ of vertices currently identified as distractions
- Initially assume no vertex is a distraction
- A vertex is a one-step distraction if its current winner guess is contradicted by looking one step ahead
- Update the lowest priority where new distractions are found
- Reset all lower priorities and recompute

Fixed point iteration

Given some set of distracting vertices $Z = Z_0 \cup Z_1 \cup \cdots \cup Z_d$,

$$\mathsf{winner}(v, Z) := \begin{cases} \mathsf{pr}(v) \bmod 2 & v \notin Z \\ 1 - (\mathsf{pr}(v) \bmod 2) & v \in Z \end{cases}$$

$$\mathsf{onestep}(v, Z) := \begin{cases} 0 & v \in V_{\bigcirc} \wedge \exists u \in E(v) : \mathsf{winner}(u, Z) = 0 \\ 1 & v \in V_{\bigcirc} \wedge \forall u \in E(v) : \mathsf{winner}(u, Z) = 1 \\ 1 & v \in V_{\pentagon} \wedge \exists u \in E(v) : \mathsf{winner}(u, Z) = 1 \\ 0 & v \in V_{\pentagon} \wedge \forall u \in E(v) : \mathsf{winner}(u, Z) = 0 \end{cases}$$

$$\mathsf{OnestepDistraction}(Z) := \{v \mid \mathsf{onestep}(v, Z) \neq \mathsf{pr}(v) \bmod 2\}$$

Fixed point iteration

```
1  def fpi(𝒢):
2      Z ← ∅                                          // start with no distractions
3      p ← 0                                          // start with lowest priority
4      while p ≤ d :                                  // while ≤ highest priority
5          α ← p mod 2                                // current parity
6          Y ← {v ∈ V_p \ Z | onestep(v, Z) ≠ α}      // new distractions
7          if Y ≠ ∅ :
8              Z ← Z ∪ Y                              // update current fixed point Z_p
9              Z ← Z \ {v | pr(v) < p}                // reset all lower fixed points
10             p ← 0                                  // restart with lowest priority
11         else:
12             p ← p + 1                              // fixed point, continue higher
13     return W_●, W_⬟ where W_● ← {v | winner(v, Z) = 0}, W_⬟ ← V \ W_●
```

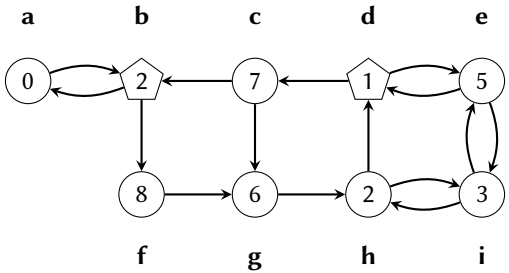
Note: this algorithm does not give a strategy!

Fixed point iteration

So basically, until fixed point:

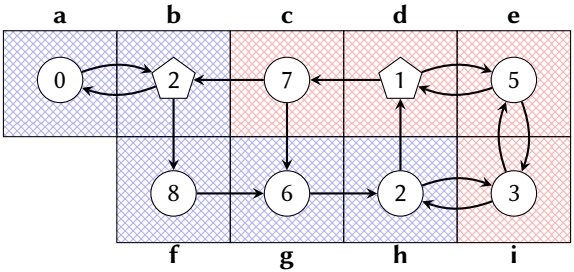
1. Find the lowest-priority vertices where the one-step-winner is not the current winner (meaning they are actually distractions)
2. Update these vertices and reset all lower-priority vertices

Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

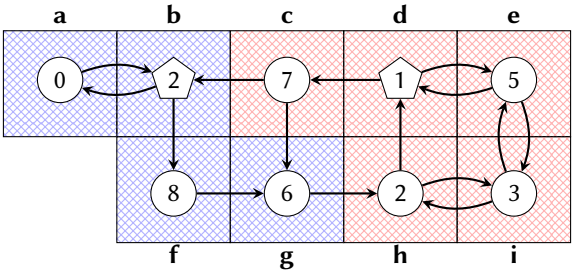
Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\}$

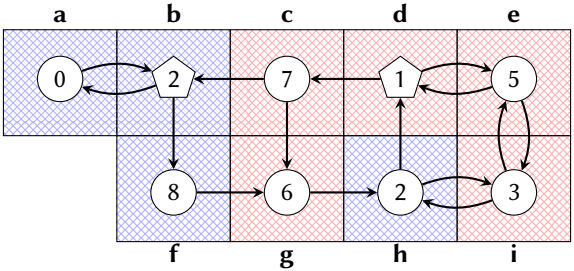
Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\mathbf{h}\}$

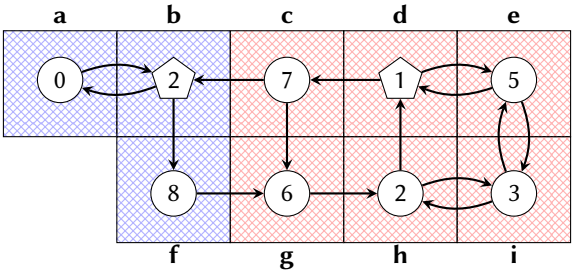
Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\mathbf{g}\}$

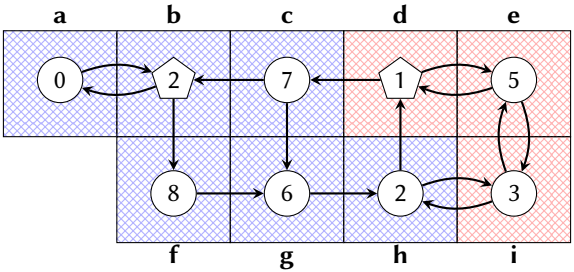
Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\mathbf{g}, \mathbf{h}\}$

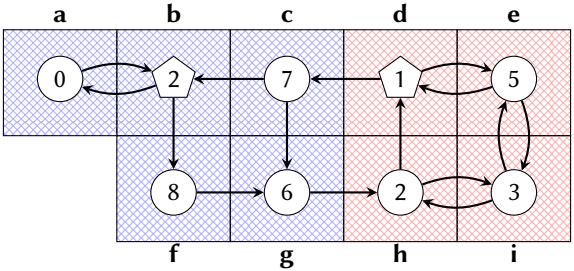
Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\mathbf{c}\}$

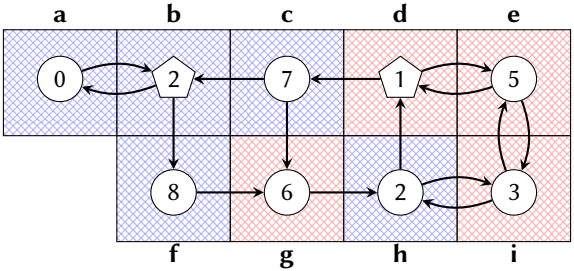
Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\mathbf{c}, \mathbf{h}\}$

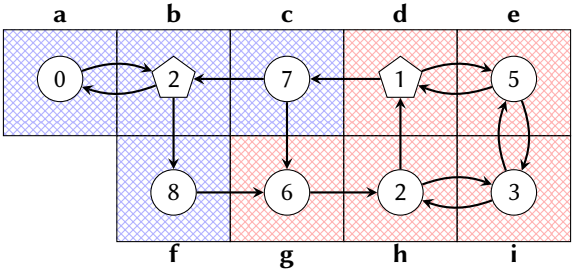
Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\mathbf{c}, \mathbf{g}\}$

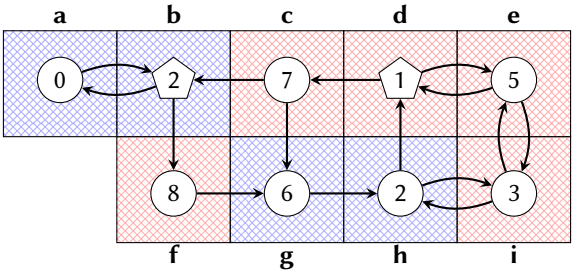
Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\mathbf{c}, \mathbf{g}, \mathbf{h}\}$

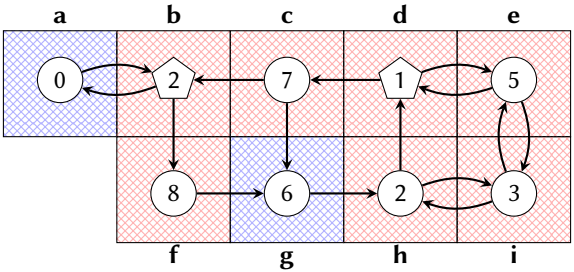
Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\mathbf{f}\}$

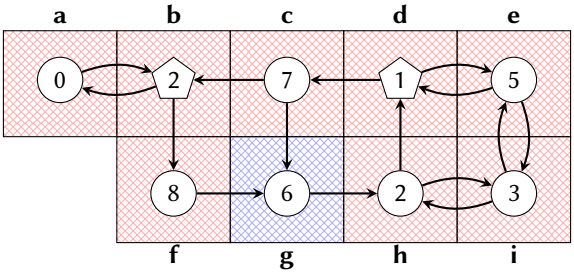
Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\mathbf{b}, \mathbf{f}, \mathbf{h}\}$

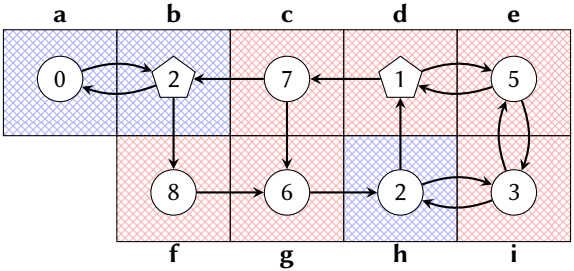
Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{h}\}$

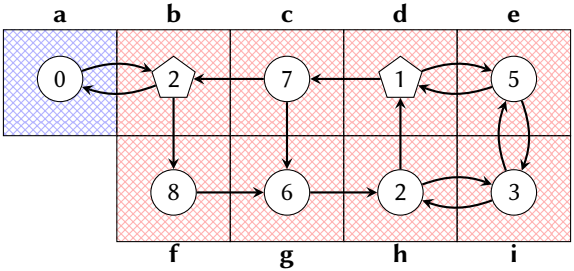
Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\mathbf{f}, \mathbf{g}\}$

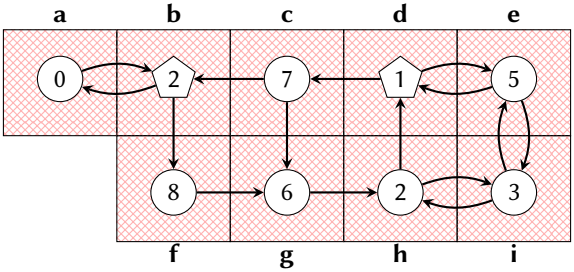
Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\mathbf{b}, \mathbf{f}, \mathbf{g}, \mathbf{h}\}$

Fixed point iteration



Find the lowest-priority vertices that need to be flipped, and reset all lower vertices.

$Z := \{\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{g}, \mathbf{h}\}$

Fixed-point iteration

What about the winning strategy?

- Basic FPI computes the winning regions, but not the strategy
- Extracting a strategy afterwards is almost as hard as solving the game again
- Updating a strategy naively during the run can be unsound
- Two main solutions:
  - **freezing**: temporarily protect parts of the current strategy
  - **justifications**: record why a vertex is currently won

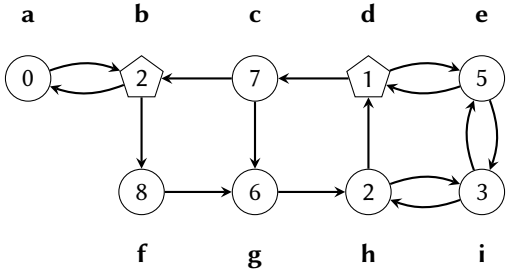
Fixed point iteration

FPI with freezing

Repeat until fixed point:

1. Find the lowest vertex where update the strategy or flip the winner.
   Only look at vertices that are not frozen
2. Whenever we flip vertices of priority p
   2.1 Freeze at priority p all vertices $\leq p$ won by player $\big((1-p) \bmod 2\big)$, that are not frozen at a higher priority
   2.2 Thaw all vertices frozen at lower priorities (frozen by the other player), reset those vertices, and remove their strategy

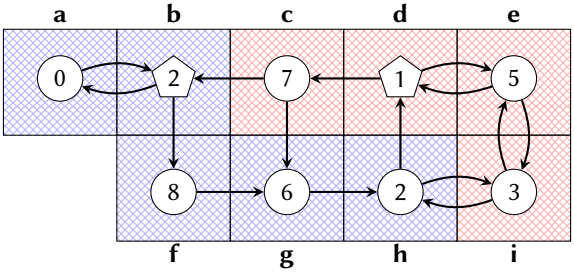
FPI with freezing



Find the lowest-priority vertices to update:

- set the strategy
- flip, freeze, thaw, reset

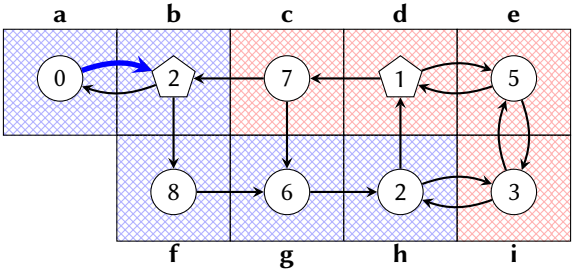
FPI with freezing



Find the lowest-priority vertices to update:

- set the strategy
- flip, freeze, thaw, reset

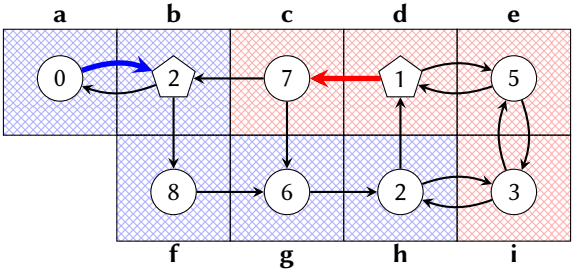
FPI with freezing



Find the lowest-priority vertices to update:

- set the strategy
- flip, freeze, thaw, reset

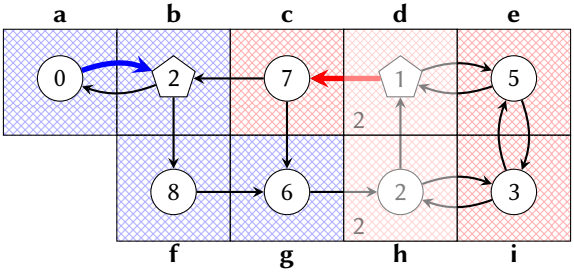
FPI with freezing



Find the lowest-priority vertices to update:

- set the strategy
- flip, freeze, thaw, reset

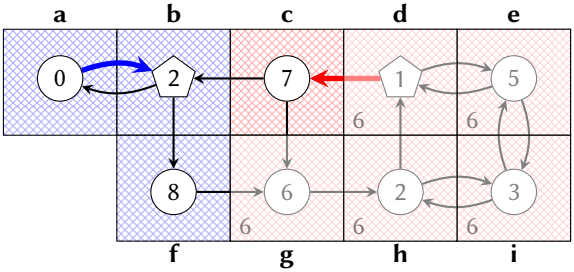
FPI with freezing



Find the lowest-priority vertices to update:

- set the strategy
- flip, freeze, thaw, reset

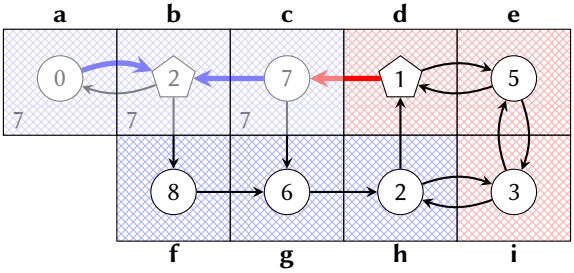
FPI with freezing



Find the lowest-priority vertices to update:

- set the strategy
- flip, freeze, thaw, reset

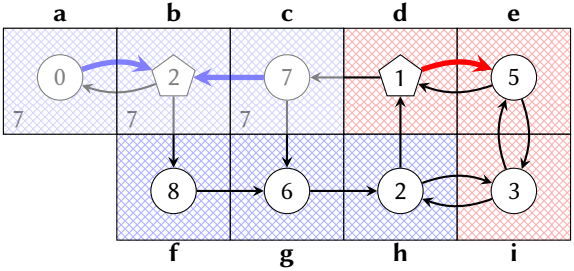
FPI with freezing



Find the lowest-priority vertices to update:

- set the strategy
- flip, freeze, thaw, reset

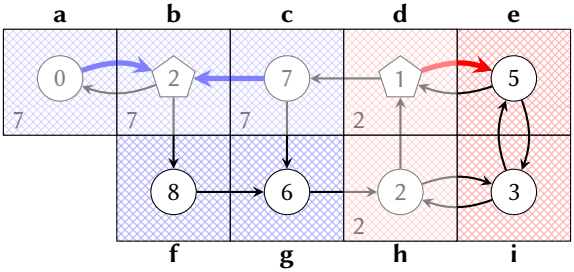
FPI with freezing



Find the lowest-priority vertices to update:

- set the strategy
- flip, freeze, thaw, reset

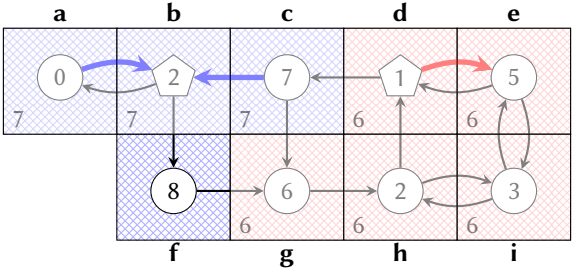
FPI with freezing



Find the lowest-priority vertices to update:

- set the strategy
- flip, freeze, thaw, reset

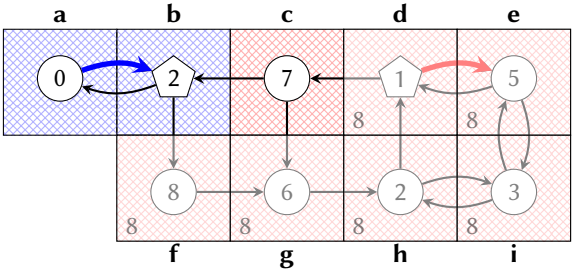
FPI with freezing



Find the lowest-priority vertices to update:

- set the strategy
- flip, freeze, thaw, reset

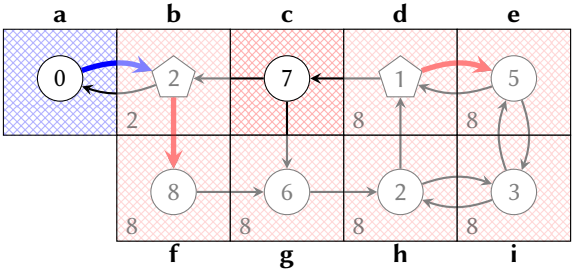
FPI with freezing



Find the lowest-priority vertices to update:

- set the strategy
- flip, freeze, thaw, reset

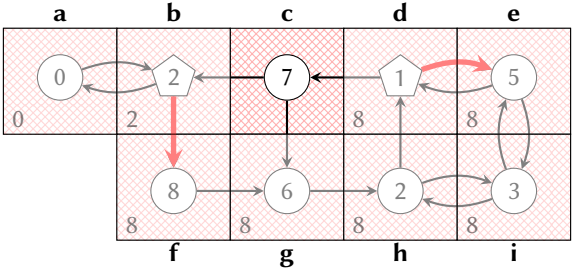
FPI with freezing



Find the lowest-priority vertices to update:

- set the strategy
- flip, freeze, thaw, reset

FPI with freezing



Find the lowest-priority vertices to update:

- set the strategy
- flip, freeze, thaw, reset

Fixed point iteration

FPI with justifications

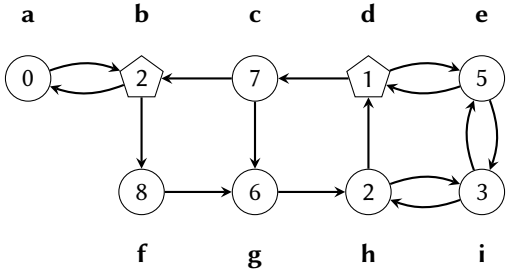
Global idea:

- Maintain "justification graph": strategies for the winner *and the loser*
  - when winning: one edge that stays in the winning region
  - when losing: all edges and they all go to the losing region
- When flipping vertices, update the justification graph

Repeat until fixed point:

1. Find the lowest vertex where we can update the strategy
2. Only consider vertices that are not justified
3. Whenever we flip vertices of priority p, prune the justification graph by removing all vertices that reach flipped vertices (they are no longer justified)
4. Reset "distraction" if no longer justified

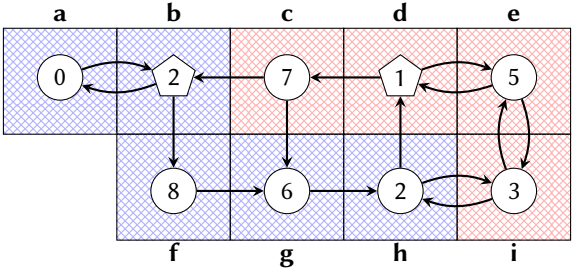
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

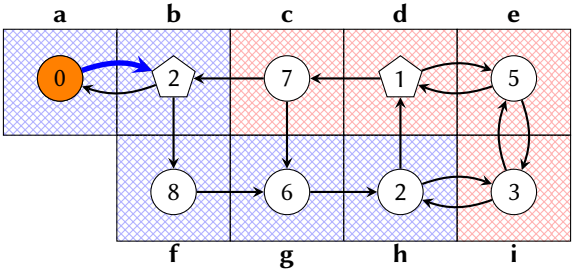
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

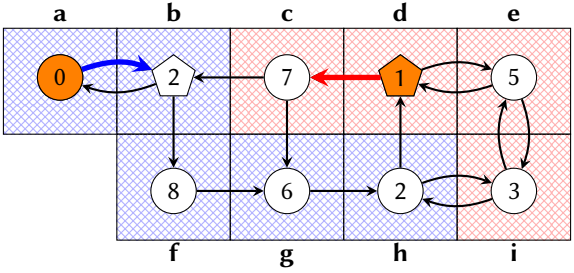
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

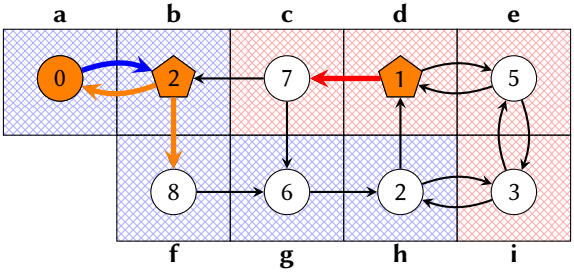
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

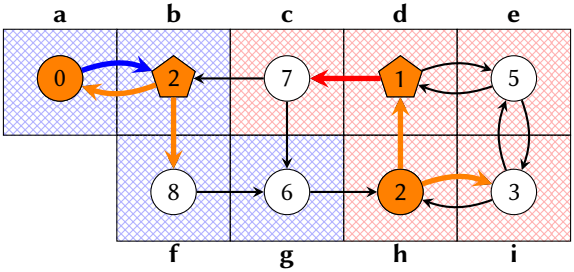
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

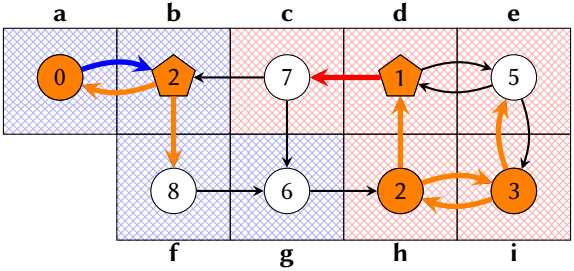
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

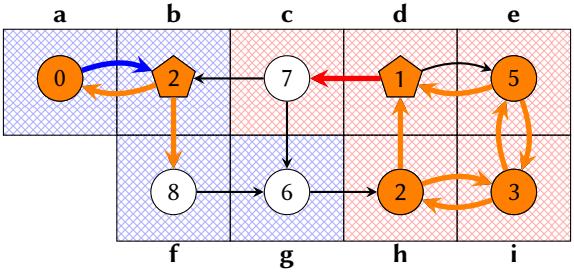
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

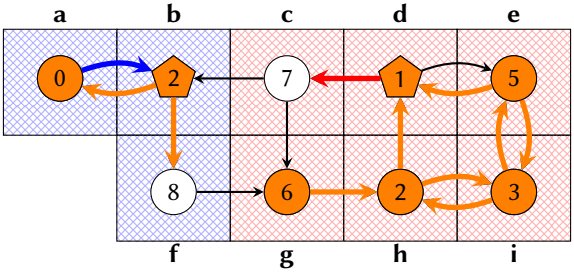
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

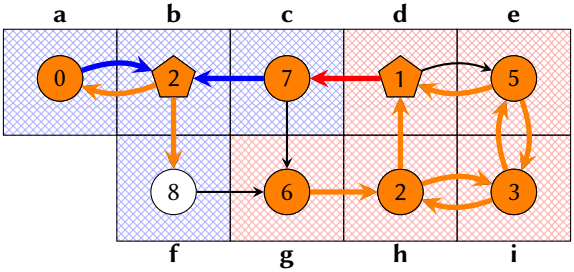
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

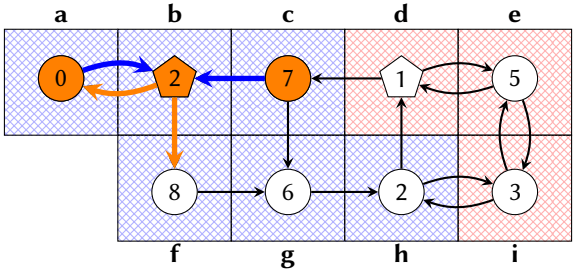
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

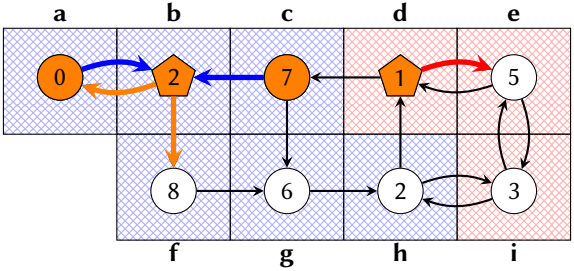
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

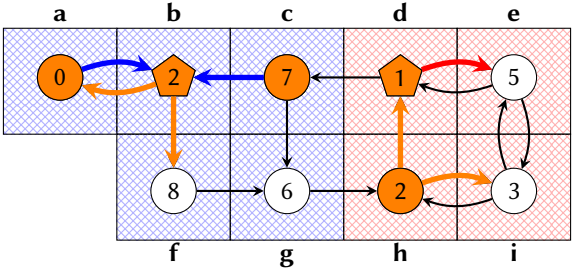
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

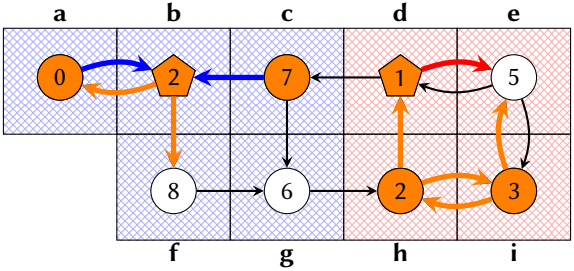
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

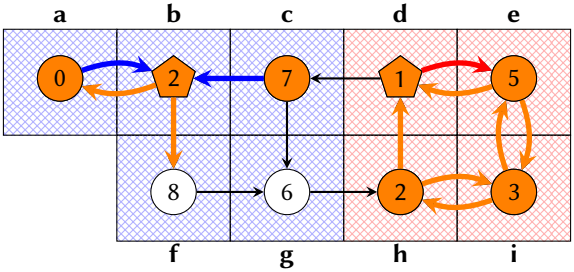
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

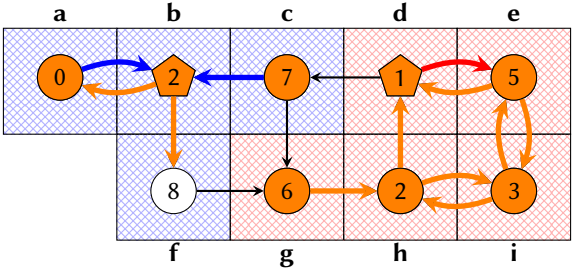
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

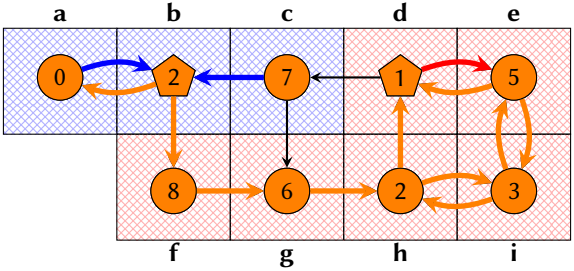
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

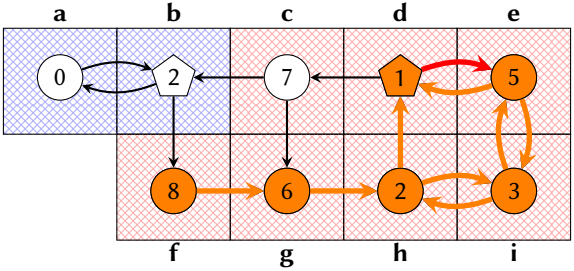
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

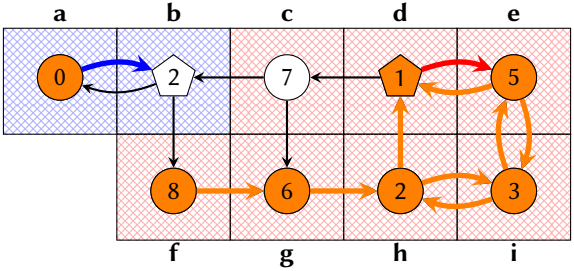
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

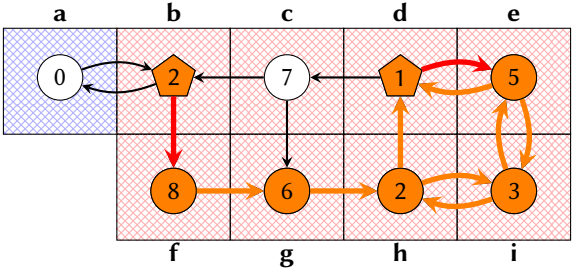
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

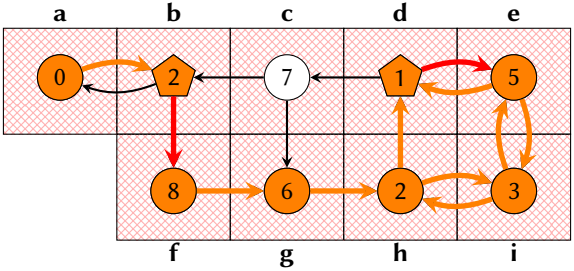
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

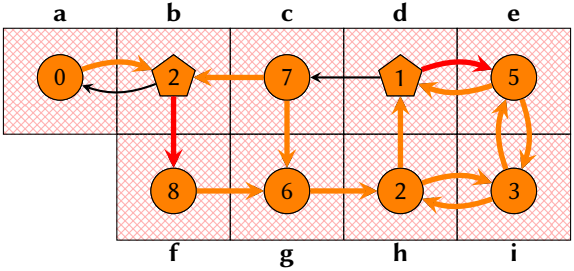
FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

FPI with justifications



Find the lowest-priority vertices to update:

- set the strategy
- if flipped, prune justifications, and reset

Fixed point iteration

What is a correct solution?

We have two winning regions $W_{\bigcirc}$ and $W_{\square}$, such that:

1. Player α must win in W_α
   That is: there must exist a winning strategy $\sigma_\alpha : (V_\alpha \cap W_\alpha) \to W_\alpha$, such that all cycles in $W_\alpha[\sigma_\alpha]$ are won by α
2. Player $\overline{\alpha}$ must not be able to leave W_α

Proof sketch

Prove by induction that the property holds after each fixed point Z_p.
Then it is true after Z_d (entire game).

- Base algorithm: there exists such a strategy
- Freezing: the frozen strategy
- Justifications: the strategy from the justification graph

Fixed point iteration

Lemma

After computing the fixpoint of Z_p, player $\alpha \in \{●, ⬟\}$ has a winning region $W_\alpha \equiv Won(\alpha)_{\leq p}$ and a strategy σ_α for all $v \in V_\alpha \cap W_\alpha$, such that

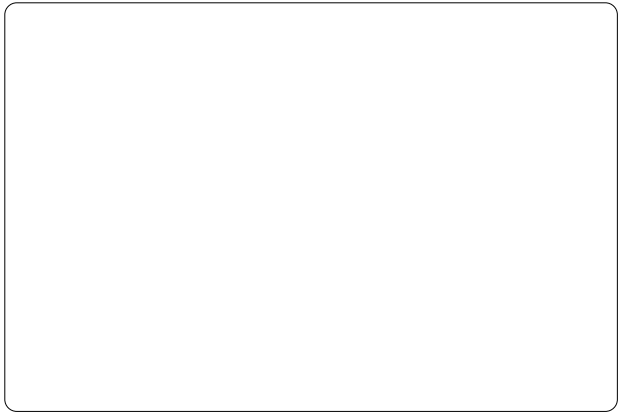
- *player α never plays from W_α to $Won(\overline{\alpha})$*
- *player $\overline{\alpha}$ cannot play from W_α to $Won(\overline{\alpha})$*
- *all cycles consistent with σ_α in W_α are won by α*

Proof by induction

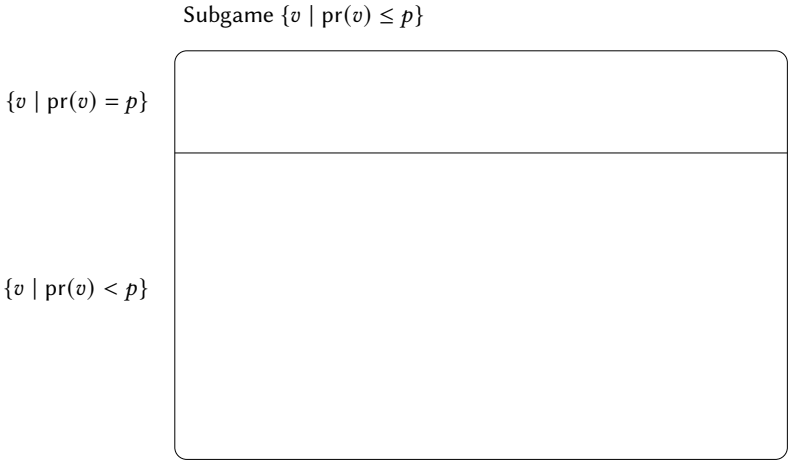
- Lemma is trivially true for the empty game.
- Assume it is true after computing Z_{p-1}.
  - So there exist winning strategies $\sigma_●$ and $\sigma_⬟$ for the subgame $< p$
- Prove for Z_p by induction on each step of the fixed point computation:
  - player α never plays from W_α to Won$(\overline{\alpha})$
  - player $\overline{\alpha}$ cannot play from W_α to Won$(\overline{\alpha})$
  - all cycles consistent with σ_α in W_α are won by α

Fixed point iteration

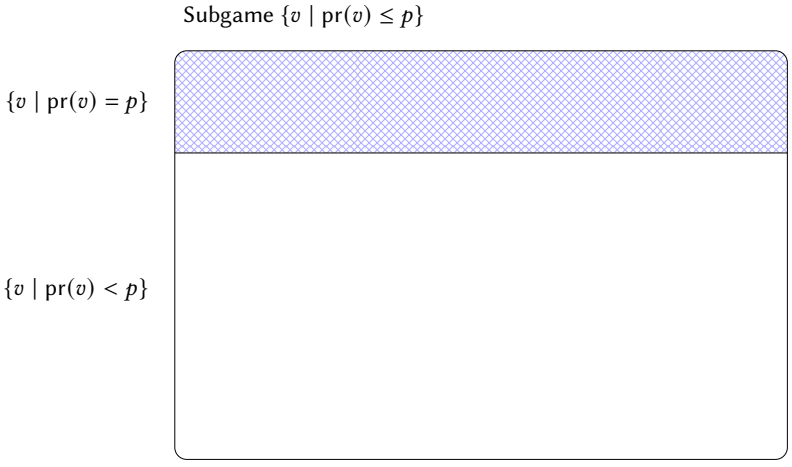
Subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



Fixed point iteration



Fixed point iteration

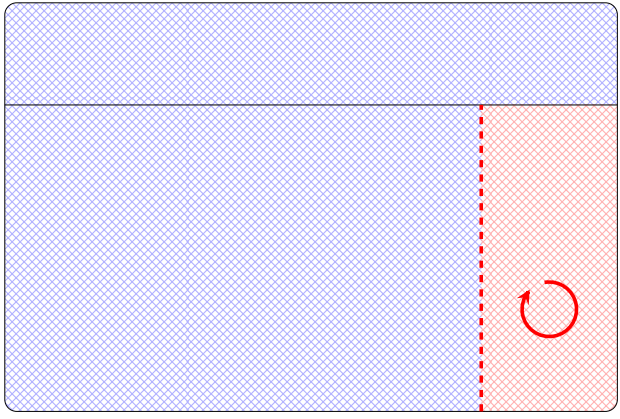


Fixed point iteration

Subgame $\{v \mid \mathrm{pr}(v) \leq p\}$

$\{v \mid \mathrm{pr}(v) = p\}$

$\{v \mid \mathrm{pr}(v) < p\}$



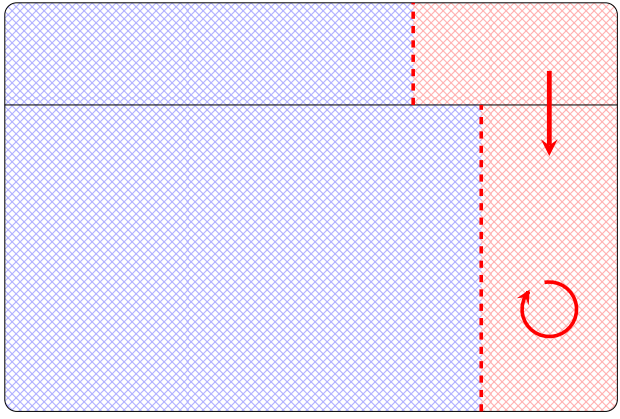
No plays from the red subgame to blue p-vertices

Fixed point iteration

Subgame $\{v \mid \mathrm{pr}(v) \leq p\}$

$\{v \mid \mathrm{pr}(v) = p\}$

$\{v \mid \mathrm{pr}(v) < p\}$



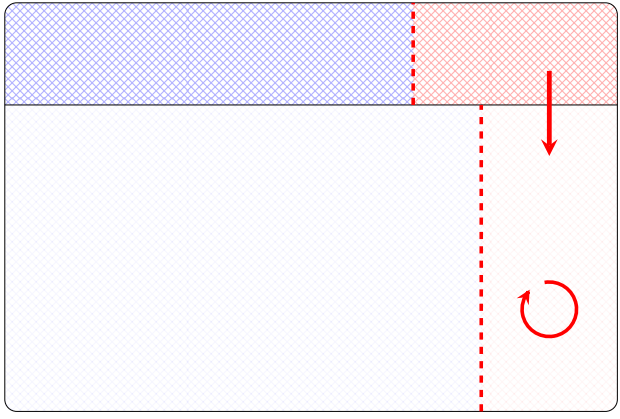
Plays see each red p-vertex **at most once**

Fixed point iteration

Subgame $\{v \mid \mathrm{pr}(v) \leq p\}$

$\{v \mid \mathrm{pr}(v) = p\}$

$\{v \mid \mathrm{pr}(v) < p\}$



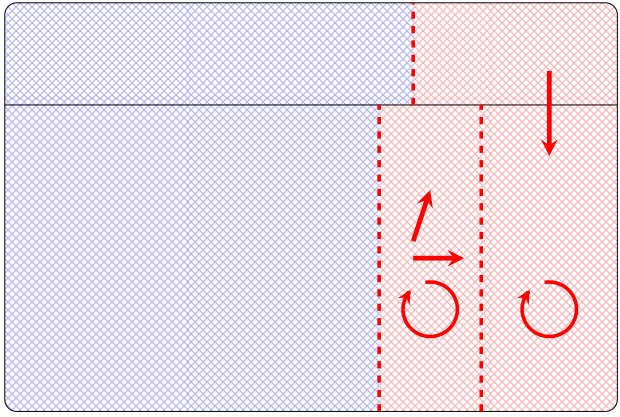
Recompute subgame (only affects blue subgame)

Fixed point iteration

Subgame $\{v \mid \mathrm{pr}(v) \leq p\}$

$\{v \mid \mathrm{pr}(v) = p\}$

$\{v \mid \mathrm{pr}(v) < p\}$



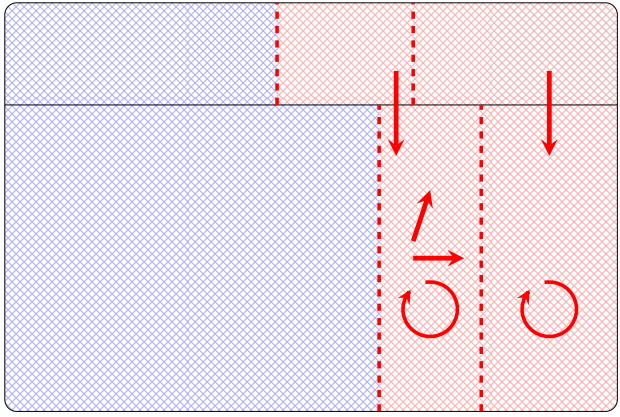
No plays from the red subgame to blue p-vertices

Fixed point iteration

Subgame $\{v \mid \mathrm{pr}(v) \leq p\}$

$\{v \mid \mathrm{pr}(v) = p\}$

$\{v \mid \mathrm{pr}(v) < p\}$



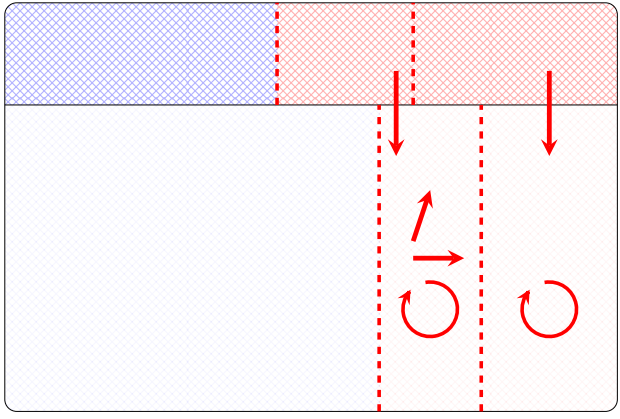
Plays see each red p-vertex **at most once**

Fixed point iteration

Subgame $\{v \mid \mathrm{pr}(v) \leq p\}$

$\{v \mid \mathrm{pr}(v) = p\}$

$\{v \mid \mathrm{pr}(v) < p\}$



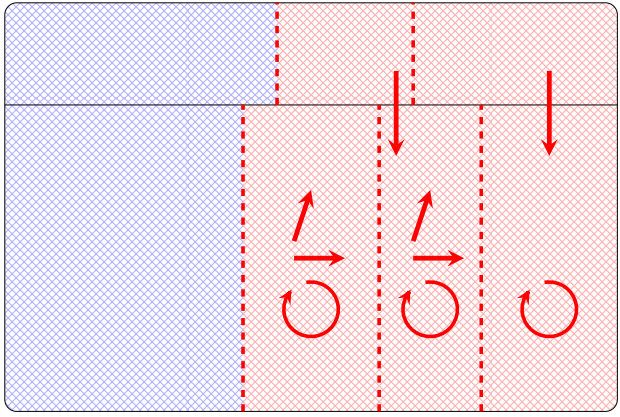
Recompute subgame (only affects blue subgame)

Fixed point iteration

Subgame $\{v \mid \mathrm{pr}(v) \leq p\}$

$\{v \mid \mathrm{pr}(v) = p\}$

$\{v \mid \mathrm{pr}(v) < p\}$



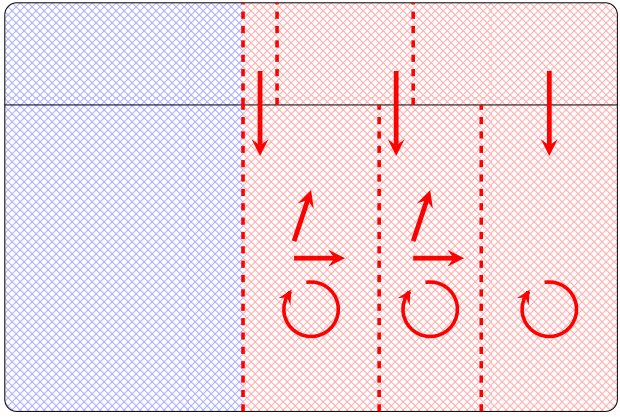
No plays from the red subgame to blue p-vertices

Fixed point iteration

Subgame $\{v \mid \mathrm{pr}(v) \leq p\}$

$\{v \mid \mathrm{pr}(v) = p\}$

$\{v \mid \mathrm{pr}(v) < p\}$



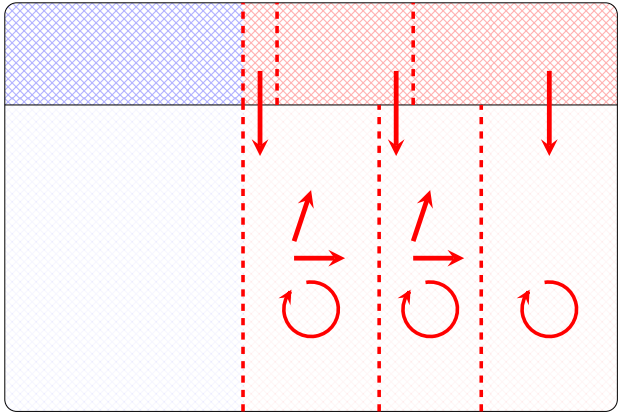
Plays see each red p-vertex **at most once**

Fixed point iteration

Subgame $\{v \mid \mathrm{pr}(v) \leq p\}$

$\{v \mid \mathrm{pr}(v) = p\}$

$\{v \mid \mathrm{pr}(v) < p\}$



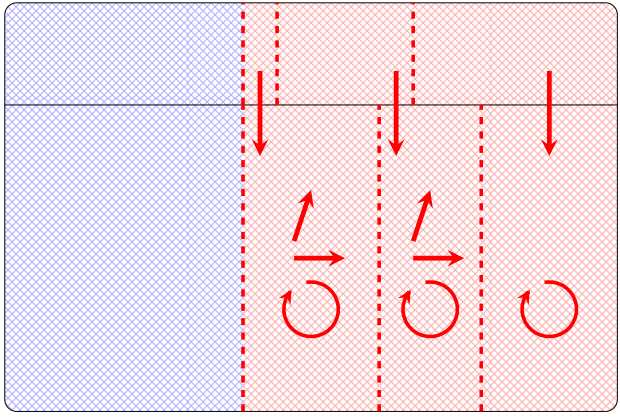
Recompute subgame (only affects blue subgame)

Fixed point iteration

Subgame $\{v \mid \mathrm{pr}(v) \leq p\}$

$\{v \mid \mathrm{pr}(v) = p\}$

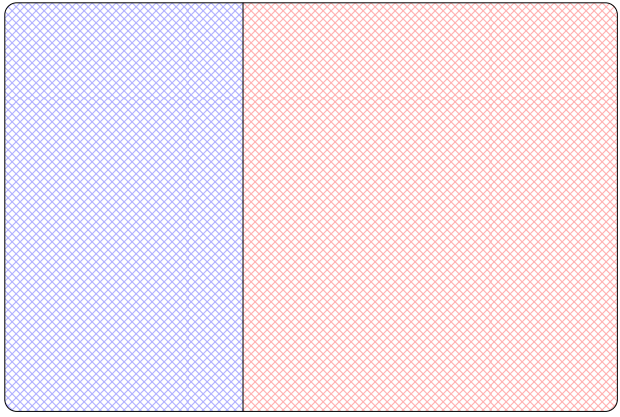
$\{v \mid \mathrm{pr}(v) < p\}$



No change: fixpoint

Fixed point iteration

Subgame $\{v \mid \mathrm{pr}(v) \leq p\}$



This solves the subgame $\{v \mid \mathrm{pr}(v) \leq p\}$

Fixed point iteration

Structure of the final blue area

Prove that all cycles in the blue area are won by blue.

- cycles in the blue subgame $\mathcal{G}_{<p}$ are won by blue (by assumption)
- cycles that leave the blue subgame $\mathcal{G}_{<p}$ are via p-vertices
- assuming some strategy $\sigma_{<p}$, we now have a strategy $\sigma_{\leq p}$

Fixed point iteration

Structure of the final red area

Prove that all cycles in the red area are won by red.

- p-vertices can only be visited once
- first red subgame is a “sink” won by red
- second red subgame
  - blue can stay in losing cycles
  - blue can move directly to the first red subgame
  - blue can visit a p-vertex once and move to the first red subgame
- third red subgame
  - blue can stay in losing cycles
  - blue can move directly to the first red subgame
  - blue can move directly to the second red subgame
  - blue can visit a p-vertex once and move to the first red subgame
  - blue can visit a p-vertex once and move to the second red subgame
- hence: no cycles with p-vertices, all cycles are in red $\mathcal{G}_{<p}$

Implementation

Implementation notes

- Always sort vertices by priority first
- Remove redundant self-loops
- FPI/DFI: record for every vertex:
  - **distraction** (boolean)
  - **freeze priority** (integer)
  - **winstrategy** (successor vertex, or e.g. -1)
- FPJ: record for each vertex:
  - **distraction** (boolean)
  - **justified** (boolean)
  - **winstrategy** (successor vertex, or e.g. -1)
    no need to mark all successors for loser
- Sometimes it's faster to wait with choosing strategies until all vertices of priority p are stable.

Value iteration

Value iteration

Fix a viewpoint where $\top$ means won by player α, and let $\bar{\alpha}$ be the other player.

Complete lattice of measures

A measure $\rho \colon V \to \mathbb{M} \cup \{\top\}$ assigns to each vertex a value from an ordered domain $(\mathbb{M}, \sqsubset)$, together with a special symbol $\top$.

The measure represents: how good is the best continuation from this vertex?

- one player prefers high values
- the other prefers low values
- $\top$ means definitely winning for the player whose viewpoint we fixed

To play from a priority-p vertex to a successor of measure m, we use a monotone update function $\texttt{Prog}(m, p)$.

Value iteration

Fix a viewpoint where $\top$ means won by player α, and let $\bar{\alpha}$ be the other player.

Compute the progress measure (least fixed point)

$$\forall v \in V\colon \rho(v) = \begin{cases} \max_{\sqsubset}\{\texttt{Prog}(\rho(w), \mathrm{pr}(v)) \mid w \in E(v)\} & v \in V_\alpha \\ \min_{\sqsubset}\{\texttt{Prog}(\rho(w), \mathrm{pr}(v)) \mid w \in E(v)\} & v \in V_{\bar{\alpha}} \end{cases}$$

- Start with $\bot$: the lowest measure in $\mathbb{M}$
- Player α chooses the best successor, player $\bar{\alpha}$ chooses the worst successor
- Lifting is monotone: the measure only increases in the order $\sqsubset$
- Repeated local updates compute the least fixed point

This update process is called lifting. The process is like playing an optimal game backwards.

Value iteration

Fix a viewpoint where $\top$ means won by player α, and let $\bar{\alpha}$ be the other player.

Winning regions and strategies

If ρ is the least parity game progress measure, then:

- $W_\alpha = \{v \mid \rho(v) = \top\}$
- $W_{\bar{\alpha}} = \{v \mid \rho(v) \neq \top\}$
- on $W_{\bar{\alpha}}$, the measure directly yields a positional winning strategy: choose the successor u with the lowest $\rho(u)$

So value iteration computes:

- the winning region of player α as the $\top$-vertices
- the winning region *and* a strategy of player $\bar{\alpha}$ on the non-$\top$ vertices

When $\top$ means won by Even, the non-$\top$ vertices come with a strategy for Odd.

Small progress measures

Introduction

Small progress measures instantiate value iteration with a very specific measure domain:

- a measure is a tuple of counters, one for each relevant priority
- the counters track how often each priority is seen along the optimal continuation
- if a counter overflows, the computation must carry into a higher priority
- repeated carry/overflow eventually yields $\top$, corresponding to an unavoidable winning cycle

So the solver keeps track of how local choices accumulate along the best play, and is forced to account for higher priorities when lower ones no longer suffice.

Later we will see succinct and ordered progress measures, which achieve quasi-polynomial complexity.

Small progress measures

Odd measures

- Measures are tuples $\langle r_7, r_5, r_3, r_1 \rangle$ (with highest odd priority 7)
- Each $r_p \in [0 \dots n_p]$, where n_p is the number of vertices with priority p
- Example:

$$\mathbb{M} = \big(\{0\} \times \{0,1,2\} \times \{0\} \times \{0,1\}\big) \cup \{\top\}$$

  if there are 2 vertices of priority 5 and 1 vertex of priority 1

- Measures are ordered lexicographically, with $\top$ as the greatest value

$$\langle 1,0,0,0 \rangle < \langle 1,0,0,1 \rangle, \qquad \langle 4,2,10,5 \rangle < \langle 4,3,0,0 \rangle$$

- Notation: $m_1 >_p m_2$ means: compare the two measures only up to priority p, and ignore everything below p
- That is: below priority p, consider all counters as reset to 0

Small progress measures

The Prog function

Playing from a vertex of priority p to a vertex with measure m gives:

$$\texttt{Prog}(m, p) := \begin{cases} \min\{\, m' \in M \mid m' \geq_p m \,\} & p \text{ even} \\ \min\{\, m' \in M \mid m' >_p m \,\} & p \text{ odd} \end{cases}$$

How to compute $\texttt{Prog}(m, p)$

1. Reset all counters below priority p to 0
2. If p is odd, increase the counter for priority p by 1
3. If that overflows, carry into the next higher odd priority
4. If the highest odd counter overflows, the result is $\top$

Small progress measures

Examples of Prog

Take

$$m = \langle 3, 2, 3, 2 \rangle$$

with every counter counting to 3.

$$\begin{aligned} \texttt{Prog}(m, 1) &= \\ \texttt{Prog}(m, 2) &= \\ \texttt{Prog}(m, 3) &= \\ \texttt{Prog}(m, 7) &= \end{aligned}$$

Small progress measures

Examples of Prog

Take

$$m = \langle 3, 2, 3, 2 \rangle$$

with every counter counting to 3.

$$\begin{aligned} \texttt{Prog}(m, 1) &= \langle 3, 2, 3, 3 \rangle \\ \texttt{Prog}(m, 2) &= \\ \texttt{Prog}(m, 3) &= \\ \texttt{Prog}(m, 7) &= \end{aligned}$$

Priority 1 (odd): reset lower counters, then increment the 1-counter.

Small progress measures

Examples of Prog

Take

$$m = \langle 3, 2, 3, 2 \rangle$$

with every counter counting to 3.

$$\begin{aligned} \texttt{Prog}(m, 1) &= \langle 3, 2, 3, 3 \rangle \\ \texttt{Prog}(m, 2) &= \langle 3, 2, 3, 0 \rangle \\ \texttt{Prog}(m, 3) &= \\ \texttt{Prog}(m, 7) &= \end{aligned}$$

Priority 2 (even): reset all counters below priority 2 to 0.

Small progress measures

Examples of Prog

Take

$$m = \langle 3, 2, 3, 2 \rangle$$

with every counter counting to 3.

$$\begin{aligned} \texttt{Prog}(m, 1) &= \langle 3, 2, 3, 3 \rangle \\ \texttt{Prog}(m, 2) &= \langle 3, 2, 3, 0 \rangle \\ \texttt{Prog}(m, 3) &= \langle 3, 3, 0, 0 \rangle \\ \texttt{Prog}(m, 7) &= \end{aligned}$$

Priority 3 (odd): reset lower counters, increment the 3-counter, and carry if needed.

Examples of Prog

Take

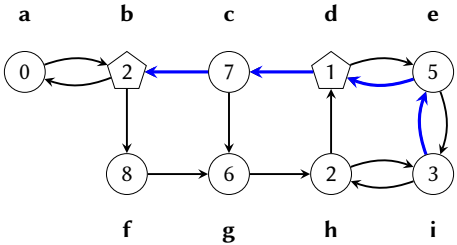
$$m = \langle 3, 2, 3, 2 \rangle$$

with every counter counting to 3.

$$\begin{aligned} \texttt{Prog}(m, 1) &= \langle 3, 2, 3, 3 \rangle \\ \texttt{Prog}(m, 2) &= \langle 3, 2, 3, 0 \rangle \\ \texttt{Prog}(m, 3) &= \langle 3, 3, 0, 0 \rangle \\ \texttt{Prog}(m, 7) &= \top \end{aligned}$$

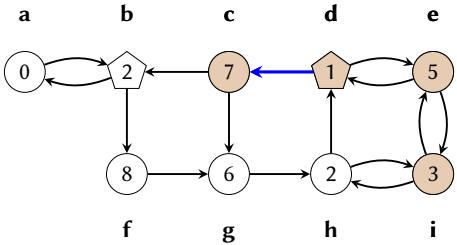
Priority 7 (odd): incrementing the highest odd counter overflows, so the result is $\top$.

Small progress measures



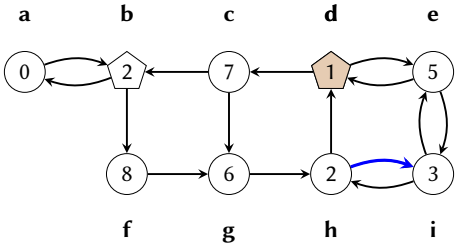
a	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 0 \rangle$	0
b	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
c	$\langle 0, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	7
d	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 1 \rangle$	1
e	$\langle 0, 0, -, - \rangle$	to	$\langle 0, 1, -, - \rangle$	5
f	$\langle -, -, -, - \rangle$	to	$\langle -, -, -, - \rangle$	8
g	$\langle 0, -, -, - \rangle$	to	$\langle 0, -, -, - \rangle$	6
h	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
i	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 1, - \rangle$	3

Small progress measures



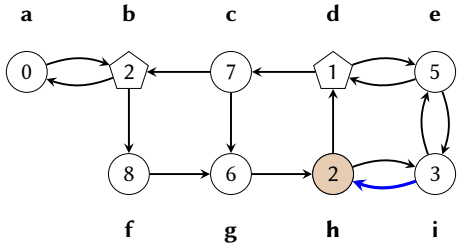
a	$\langle 0,0,0,0\rangle$	to	$\langle 0,0,0,0\rangle$	0
b	$\langle 0,0,0,-\rangle$	to	$\langle 0,0,0,-\rangle$	2
c	$\langle 1,-,-,-\rangle$	to	$\langle 1,-,-,-\rangle$	7
d	$\langle 0,0,0,1\rangle$	to	$\langle 1,0,0,1\rangle$	17
e	$\langle 0,1,-,-\rangle$	to	$\langle 0,1,-,-\rangle$	5
f	$\langle -,-,-,-\rangle$	to	$\langle -,-,-,-\rangle$	8
g	$\langle 0,-,-,-\rangle$	to	$\langle 0,-,-,-\rangle$	6
h	$\langle 0,0,0,-\rangle$	to	$\langle 0,0,0,-\rangle$	2
i	$\langle 0,0,1,-\rangle$	to	$\langle 0,0,1,-\rangle$	3

Small progress measures



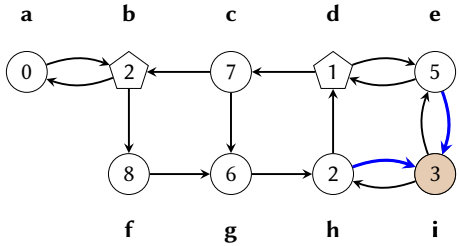
a	$\langle 0,0,0,0\rangle$	to	$\langle 0,0,0,0\rangle$	0
b	$\langle 0,0,0,-\rangle$	to	$\langle 0,0,0,-\rangle$	2
c	$\langle 1,-,-,-\rangle$	to	$\langle 1,-,-,-\rangle$	7
d	$\langle 1,0,0,1\rangle$	to	$\langle 1,0,0,1\rangle$	17
e	$\langle 0,1,-,-\rangle$	to	$\langle 0,1,-,-\rangle$	5
f	$\langle -,-,-,-\rangle$	to	$\langle -,-,-,-\rangle$	8
g	$\langle 0,-,-,-\rangle$	to	$\langle 0,-,-,-\rangle$	6
h	$\langle 0,0,0,-\rangle$	to	$\langle 0,0,1,-\rangle$	23
i	$\langle 0,0,1,-\rangle$	to	$\langle 0,0,1,-\rangle$	3

Small progress measures



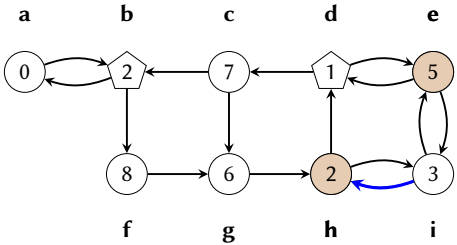
a	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 0 \rangle$	0
b	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
c	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	7
d	$\langle 1, 0, 0, 1 \rangle$	to	$\langle 1, 0, 0, 1 \rangle$	17
e	$\langle 0, 1, -, - \rangle$	to	$\langle 0, 1, -, - \rangle$	5
f	$\langle -, -, -, - \rangle$	to	$\langle -, -, -, - \rangle$	8
g	$\langle 0, -, -, - \rangle$	to	$\langle 0, -, -, - \rangle$	6
h	$\langle 0, 0, 1, - \rangle$	to	$\langle 0, 0, 1, - \rangle$	23
i	$\langle 0, 0, 1, - \rangle$	to	$\langle 0, 1, 0, - \rangle$	323

Small progress measures



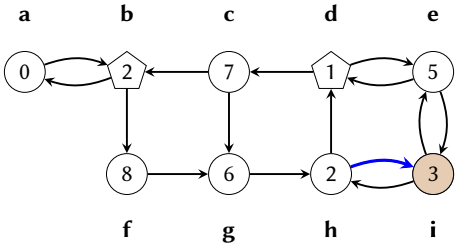
a	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 0 \rangle$	0
b	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
c	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	7
d	$\langle 1, 0, 0, 1 \rangle$	to	$\langle 1, 0, 0, 1 \rangle$	17
e	$\langle 0, 1, -, - \rangle$	to	$\langle 1, 0, -, - \rangle$	5323
f	$\langle -, -, -, - \rangle$	to	$\langle -, -, -, - \rangle$	8
g	$\langle 0, -, -, - \rangle$	to	$\langle 0, -, -, - \rangle$	6
h	$\langle 0, 0, 1, - \rangle$	to	$\langle 0, 1, 0, - \rangle$	2323
i	$\langle 0, 1, 0, - \rangle$	to	$\langle 0, 1, 0, - \rangle$	323

Small progress measures



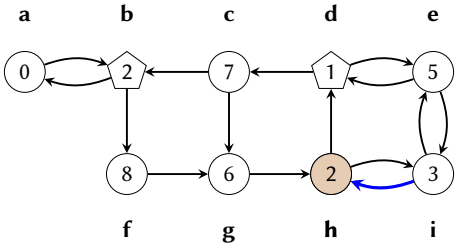
a	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 0 \rangle$	0
b	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
c	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	7
d	$\langle 1, 0, 0, 1 \rangle$	to	$\langle 1, 0, 0, 1 \rangle$	17
e	$\langle 1, 0, -, - \rangle$	to	$\langle 1, 0, -, - \rangle$	5323
f	$\langle -, -, -, - \rangle$	to	$\langle -, -, -, - \rangle$	8
g	$\langle 0, -, -, - \rangle$	to	$\langle 0, -, -, - \rangle$	6
h	$\langle 0, 1, 0, - \rangle$	to	$\langle 0, 1, 0, - \rangle$	2323
i	$\langle 0, 1, 0, - \rangle$	to	$\langle 0, 1, 1, - \rangle$	32323

Small progress measures



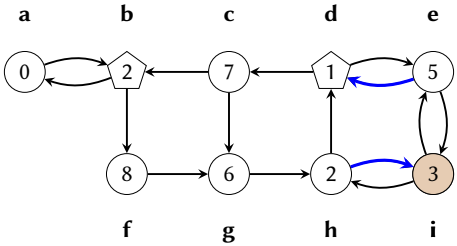
a	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 0 \rangle$	0
b	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
c	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	7
d	$\langle 1, 0, 0, 1 \rangle$	to	$\langle 1, 0, 0, 1 \rangle$	17
e	$\langle 1, 0, -, - \rangle$	to	$\langle 1, 0, -, - \rangle$	5323
f	$\langle -, -, -, - \rangle$	to	$\langle -, -, -, - \rangle$	8
g	$\langle 0, -, -, - \rangle$	to	$\langle 0, -, -, - \rangle$	6
h	$\langle 0, 1, 0, - \rangle$	to	$\langle 0, 1, 1, - \rangle$	232323
i	$\langle 0, 1, 1, - \rangle$	to	$\langle 0, 1, 1, - \rangle$	32323

Small progress measures



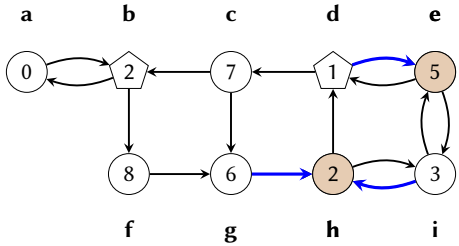
a	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 0 \rangle$	0
b	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
c	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	7
d	$\langle 1, 0, 0, 1 \rangle$	to	$\langle 1, 0, 0, 1 \rangle$	17
e	$\langle 1, 0, -, - \rangle$	to	$\langle 1, 0, -, - \rangle$	5323
f	$\langle -, -, -, - \rangle$	to	$\langle -, -, -, - \rangle$	8
g	$\langle 0, -, -, - \rangle$	to	$\langle 0, -, -, - \rangle$	6
h	$\langle 0, 1, 1, - \rangle$	to	$\langle 0, 1, 1, - \rangle$	232323
i	$\langle 0, 1, 1, - \rangle$	to	$\langle 1, 0, 0, - \rangle$	3232323

Small progress measures



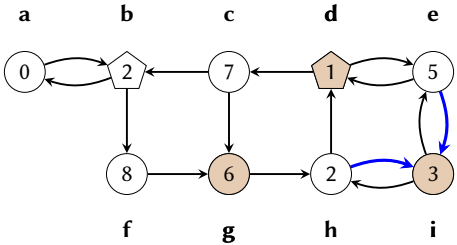
a	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 0 \rangle$	0
b	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
c	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	7
d	$\langle 1, 0, 0, 1 \rangle$	to	$\langle 1, 0, 0, 1 \rangle$	17
e	$\langle 1, 0, -, - \rangle$	to	$\langle 1, 1, -, - \rangle$	517
f	$\langle -, -, -, - \rangle$	to	$\langle -, -, -, - \rangle$	8
g	$\langle 0, -, -, - \rangle$	to	$\langle 0, -, -, - \rangle$	6
h	$\langle 0, 1, 1, - \rangle$	to	$\langle 1, 0, 0, - \rangle$	23232323
i	$\langle 1, 0, 0, - \rangle$	to	$\langle 1, 0, 0, - \rangle$	3232323

Small progress measures



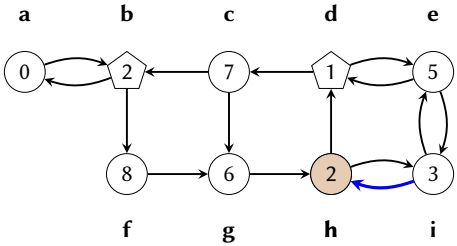
a	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 0 \rangle$	0
b	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
c	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	7
d	$\langle 1, 0, 0, 1 \rangle$	to	$\langle 1, 1, 0, 1 \rangle$	1517
e	$\langle 1, 1, -, - \rangle$	to	$\langle 1, 1, -, - \rangle$	517
f	$\langle -, -, -, - \rangle$	to	$\langle -, -, -, - \rangle$	8
g	$\langle 0, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	623232323
h	$\langle 1, 0, 0, - \rangle$	to	$\langle 1, 0, 0, - \rangle$	23232323
i	$\langle 1, 0, 0, - \rangle$	to	$\langle 1, 0, 1, - \rangle$	323232323

Small progress measures



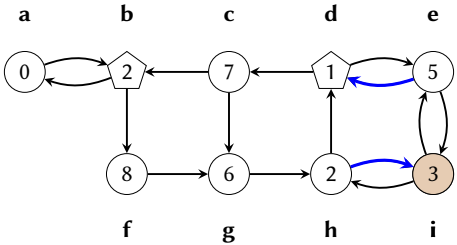
a	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 0 \rangle$	0
b	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
c	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	7
d	$\langle 1, 1, 0, 1 \rangle$	to	$\langle 1, 1, 0, 1 \rangle$	1517
e	$\langle 1, 1, -, - \rangle$	to	$\langle 1, 1, -, - \rangle$	5323232323
f	$\langle -, -, -, - \rangle$	to	$\langle -, -, -, - \rangle$	8
g	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	623232323
h	$\langle 1, 0, 0, - \rangle$	to	$\langle 1, 0, 1, - \rangle$	2323232323
i	$\langle 1, 0, 1, - \rangle$	to	$\langle 1, 0, 1, - \rangle$	323232323

Small progress measures



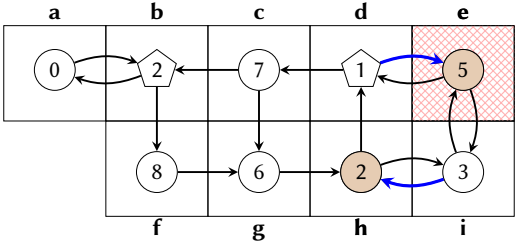
a	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 0 \rangle$	0
b	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
c	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	7
d	$\langle 1, 1, 0, 1 \rangle$	to	$\langle 1, 1, 0, 1 \rangle$	1517
e	$\langle 1, 1, -, - \rangle$	to	$\langle 1, 1, -, - \rangle$	5323232323
f	$\langle -, -, -, - \rangle$	to	$\langle -, -, -, - \rangle$	8
g	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	623232323
h	$\langle 1, 0, 1, - \rangle$	to	$\langle 1, 0, 1, - \rangle$	2323232323
i	$\langle 1, 0, 1, - \rangle$	to	$\langle 1, 1, 0, - \rangle$	32323232323

Small progress measures



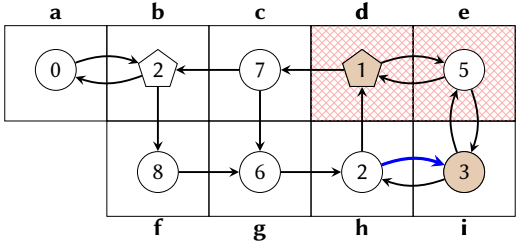
a	$\langle 0,0,0,0\rangle$	to	$\langle 0,0,0,0\rangle$	0
b	$\langle 0,0,0,-\rangle$	to	$\langle 0,0,0,-\rangle$	2
c	$\langle 1,-,-,-\rangle$	to	$\langle 1,-,-,-\rangle$	7
d	$\langle 1,1,0,1\rangle$	to	$\langle 1,1,0,1\rangle$	1517
e	$\langle 1,1,-,-\rangle$	to	$\top$	51517
f	$\langle -,-,-,-\rangle$	to	$\langle -,-,-,-\rangle$	8
g	$\langle 1,-,-,-\rangle$	to	$\langle 1,-,-,-\rangle$	623232323
h	$\langle 1,0,1,-\rangle$	to	$\langle 1,1,0,-\rangle$	232323232323
i	$\langle 1,1,0,-\rangle$	to	$\langle 1,1,0,-\rangle$	32323232323

Small progress measures



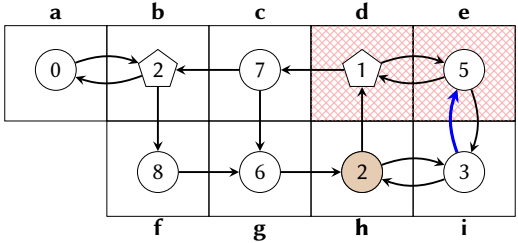
a	$\langle 0,0,0,0\rangle$	to	$\langle 0,0,0,0\rangle$	0
b	$\langle 0,0,0,-\rangle$	to	$\langle 0,0,0,-\rangle$	2
c	$\langle 1,-,-,-\rangle$	to	$\langle 1,-,-,-\rangle$	7
d	$\langle 1,1,0,1\rangle$	to	$\top$	151517
e	$\top$	to	$\top$	51517
f	$\langle -,-,-,-\rangle$	to	$\langle -,-,-,-\rangle$	8
g	$\langle 1,-,-,-\rangle$	to	$\langle 1,-,-,-\rangle$	623232323
h	$\langle 1,1,0,-\rangle$	to	$\langle 1,1,0,-\rangle$	232323232323
i	$\langle 1,1,0,-\rangle$	to	$\langle 1,1,1,-\rangle$	3232323232323

Small progress measures



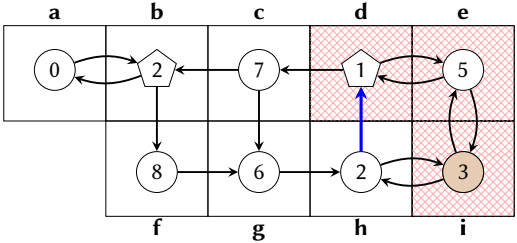
a	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 0 \rangle$	0
b	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
c	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	7
d	$\top$	to	$\top$	151517
e	$\top$	to	$\top$	51517
f	$\langle -, -, -, - \rangle$	to	$\langle -, -, -, - \rangle$	8
g	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	623232323
h	$\langle 1, 0, 1, - \rangle$	to	$\langle 1, 1, 1, - \rangle$	23232323232323
i	$\langle 1, 1, 1, - \rangle$	to	$\langle 1, 1, 1, - \rangle$	3232323232323

Small progress measures



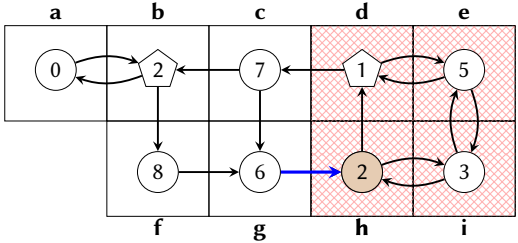
a	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 0 \rangle$	0
b	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
c	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	7
d	$\top$	to	$\top$	151517
e	$\top$	to	$\top$	51517
f	$\langle -, -, -, - \rangle$	to	$\langle -, -, -, - \rangle$	8
g	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	623232323
h	$\langle 1, 1, 1, - \rangle$	to	$\langle 1, 1, 1, - \rangle$	23232323232323
i	$\langle 1, 1, 1, - \rangle$	to	$\top$	351517

Small progress measures



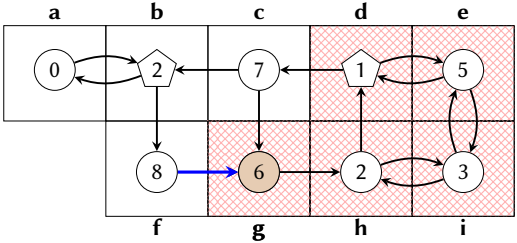
a	$\langle 0,0,0,0\rangle$	to	$\langle 0,0,0,0\rangle$	0
b	$\langle 0,0,0,-\rangle$	to	$\langle 0,0,0,-\rangle$	2
c	$\langle 1,-,-,-\rangle$	to	$\langle 1,-,-,-\rangle$	7
d	$\top$	to	$\top$	151517
e	$\top$	to	$\top$	51517
f	$\langle -,-,-,-\rangle$	to	$\langle -,-,-,-\rangle$	8
g	$\langle 1,-,-,-\rangle$	to	$\langle 1,-,-,-\rangle$	623232323
h	$\langle 1,1,1,-\rangle$	to	$\top$	2151517
i	$\top$	to	$\top$	351517

Small progress measures



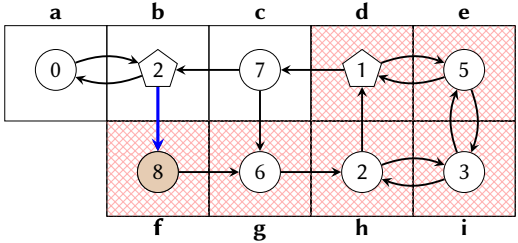
a	$\langle 0,0,0,0\rangle$	to	$\langle 0,0,0,0\rangle$	0
b	$\langle 0,0,0,-\rangle$	to	$\langle 0,0,0,-\rangle$	2
c	$\langle 1,-,-,-\rangle$	to	$\langle 1,-,-,-\rangle$	7
d	$\top$	to	$\top$	151517
e	$\top$	to	$\top$	51517
f	$\langle -,-,-,-\rangle$	to	$\langle -,-,-,-\rangle$	8
g	$\langle 1,-,-,-\rangle$	to	$\top$	62151517
h	$\top$	to	$\top$	2151517
i	$\top$	to	$\top$	351517

Small progress measures



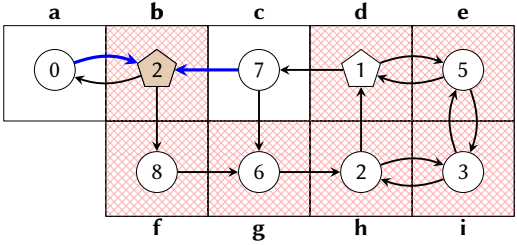
a	$\langle 0, 0, 0, 0 \rangle$	to	$\langle 0, 0, 0, 0 \rangle$	0
b	$\langle 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, - \rangle$	2
c	$\langle 1, -, -, - \rangle$	to	$\langle 1, -, -, - \rangle$	7
d	$\top$	to	$\top$	151517
e	$\top$	to	$\top$	51517
f	$\langle -, -, -, - \rangle$	to	$\top$	862151517
g	$\top$	to	$\top$	62151517
h	$\top$	to	$\top$	2151517
i	$\top$	to	$\top$	351517

Small progress measures



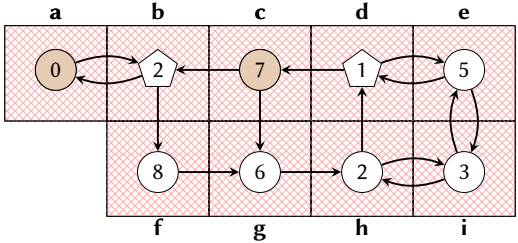
a	$\langle 0,0,0,0\rangle$	to	$\langle 0,0,0,0\rangle$	0
b	$\langle 0,0,0,-\rangle$	to	$\top$	2862151517
c	$\langle 1,-,-,-\rangle$	to	$\langle 1,-,-,-\rangle$	7
d	$\top$	to	$\top$	151517
e	$\top$	to	$\top$	51517
f	$\top$	to	$\top$	862151517
g	$\top$	to	$\top$	62151517
h	$\top$	to	$\top$	2151517
i	$\top$	to	$\top$	351517

Small progress measures



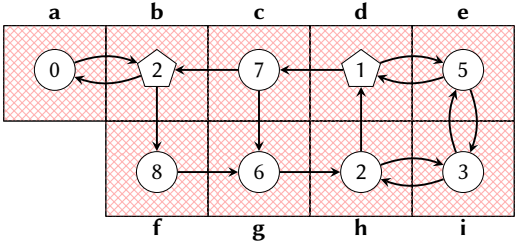
a	$\langle 0, 0, 0, 0 \rangle$	to	$\top$	02862151517
b	$\top$	to	$\top$	2862151517
c	$\langle 1, -, -, - \rangle$	to	$\top$	762151517
d	$\top$	to	$\top$	151517
e	$\top$	to	$\top$	51517
f	$\top$	to	$\top$	862151517
g	$\top$	to	$\top$	62151517
h	$\top$	to	$\top$	2151517
i	$\top$	to	$\top$	351517

Small progress measures



a	⊤	to	⊤	02862151517
b	⊤	to	⊤	2862151517
c	⊤	to	⊤	762151517
d	⊤	to	⊤	151517
e	⊤	to	⊤	51517
f	⊤	to	⊤	862151517
g	⊤	to	⊤	62151517
h	⊤	to	⊤	2151517
i	⊤	to	⊤	351517

Small progress measures



- All vertices are won by Odd
- No strategy for Even

Small progress measures

Problem

The progress measure we just computed gives a winning strategy for one player on the non-$\top$ vertices.

In this example, that is not the strategy we want: Even loses, so we want a strategy for Odd.

Small progress measures

Problem

The progress measure we just computed gives a winning strategy for one player on the non-$\top$ vertices.

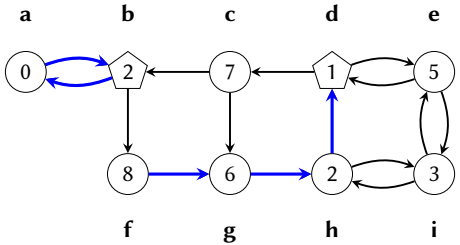
In this example, that is not the strategy we want: Even loses, so we want a strategy for Odd.

Solution

We simply swap the viewpoint:

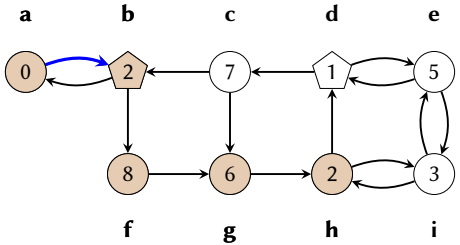
- now the measure records how good a vertex is for Even
- $\top$ means won by Even
- therefore the non-$\top$ vertices come with a strategy for Odd

Small progress measures



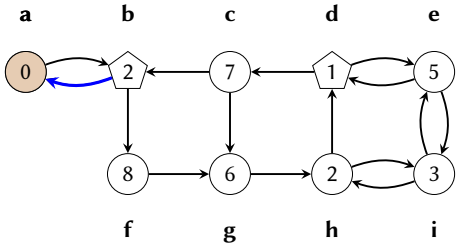
a	$\langle 0,0,0,0,0\rangle$	to	$\langle 0,0,0,0,1\rangle$	0
b	$\langle 0,0,0,0,-\rangle$	to	$\langle 0,0,0,1,-\rangle$	2
c	$\langle 0,-,-,-,-\rangle$	to	$\langle 0,-,-,-,-\rangle$	7
d	$\langle 0,0,0,0,-\rangle$	to	$\langle 0,0,0,0,-\rangle$	1
e	$\langle 0,0,-,-,-\rangle$	to	$\langle 0,0,-,-,-\rangle$	5
f	$\langle 0,-,-,-,-\rangle$	to	$\langle 1,-,-,-,-\rangle$	8
g	$\langle 0,0,-,-,-\rangle$	to	$\langle 0,1,-,-,-\rangle$	6
h	$\langle 0,0,0,0,-\rangle$	to	$\langle 0,0,0,1,-\rangle$	2
i	$\langle 0,0,0,-,-\rangle$	to	$\langle 0,0,0,-,-\rangle$	3

Small progress measures



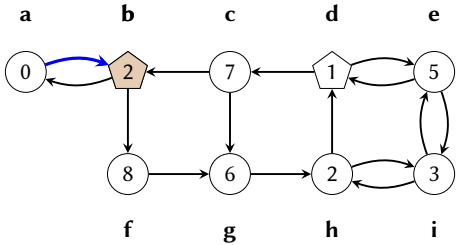
a	$\langle 0,0,0,0,1\rangle$	to	$\langle 0,0,0,1,1\rangle$	02
b	$\langle 0,0,0,1,-\rangle$	to	$\langle 0,0,0,1,-\rangle$	2
c	$\langle 0,-,-,-,-\rangle$	to	$\langle 0,-,-,-,-\rangle$	7
d	$\langle 0,0,0,0,-\rangle$	to	$\langle 0,0,0,0,-\rangle$	1
e	$\langle 0,0,-,-,-\rangle$	to	$\langle 0,0,-,-,-\rangle$	5
f	$\langle 1,-,-,-,-\rangle$	to	$\langle 1,-,-,-,-\rangle$	8
g	$\langle 0,1,-,-,-\rangle$	to	$\langle 0,1,-,-,-\rangle$	6
h	$\langle 0,0,0,1,-\rangle$	to	$\langle 0,0,0,1,-\rangle$	2
i	$\langle 0,0,0,-,-\rangle$	to	$\langle 0,0,0,-,-\rangle$	3

Small progress measures



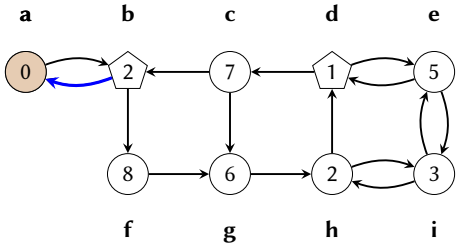
a	$\langle 0,0,0,1,1\rangle$	to	$\langle 0,0,0,1,1\rangle$	02
b	$\langle 0,0,0,1,-\rangle$	to	$\langle 0,0,0,2,-\rangle$	202
c	$\langle 0,-,-,-,-\rangle$	to	$\langle 0,-,-,-,-\rangle$	7
d	$\langle 0,0,0,0,-\rangle$	to	$\langle 0,0,0,0,-\rangle$	1
e	$\langle 0,0,-,-,-\rangle$	to	$\langle 0,0,-,-,-\rangle$	5
f	$\langle 1,-,-,-,-\rangle$	to	$\langle 1,-,-,-,-\rangle$	8
g	$\langle 0,1,-,-,-\rangle$	to	$\langle 0,1,-,-,-\rangle$	6
h	$\langle 0,0,0,1,-\rangle$	to	$\langle 0,0,0,1,-\rangle$	2
i	$\langle 0,0,0,-,-\rangle$	to	$\langle 0,0,0,-,-\rangle$	3

Small progress measures



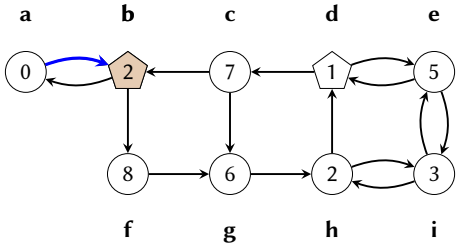
a	$\langle 0,0,0,1,1\rangle$	to	$\langle 0,0,0,2,1\rangle$	0202
b	$\langle 0,0,0,2,-\rangle$	to	$\langle 0,0,0,2,-\rangle$	202
c	$\langle 0,-,-,-,-\rangle$	to	$\langle 0,-,-,-,-\rangle$	7
d	$\langle 0,0,0,0,-\rangle$	to	$\langle 0,0,0,0,-\rangle$	1
e	$\langle 0,0,-,-,-\rangle$	to	$\langle 0,0,-,-,-\rangle$	5
f	$\langle 1,-,-,-,-\rangle$	to	$\langle 1,-,-,-,-\rangle$	8
g	$\langle 0,1,-,-,-\rangle$	to	$\langle 0,1,-,-,-\rangle$	6
h	$\langle 0,0,0,1,-\rangle$	to	$\langle 0,0,0,1,-\rangle$	2
i	$\langle 0,0,0,-,-\rangle$	to	$\langle 0,0,0,-,-\rangle$	3

Small progress measures



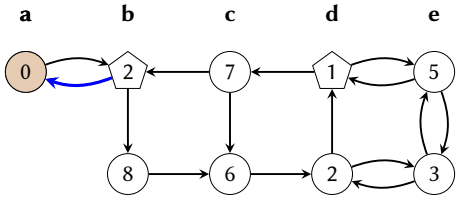
a	$\langle 0,0,0,2,1\rangle$	to	$\langle 0,0,0,2,1\rangle$	0202
b	$\langle 0,0,0,2,-\rangle$	to	$\langle 0,1,0,0,-\rangle$	20202
c	$\langle 0,-,-,-,-\rangle$	to	$\langle 0,-,-,-,-\rangle$	7
d	$\langle 0,0,0,0,-\rangle$	to	$\langle 0,0,0,0,-\rangle$	1
e	$\langle 0,0,-,-,-\rangle$	to	$\langle 0,0,-,-,-\rangle$	5
f	$\langle 1,-,-,-,-\rangle$	to	$\langle 1,-,-,-,-\rangle$	8
g	$\langle 0,1,-,-,-\rangle$	to	$\langle 0,1,-,-,-\rangle$	6
h	$\langle 0,0,0,1,-\rangle$	to	$\langle 0,0,0,1,-\rangle$	2
i	$\langle 0,0,0,-,-\rangle$	to	$\langle 0,0,0,-,-\rangle$	3

Small progress measures



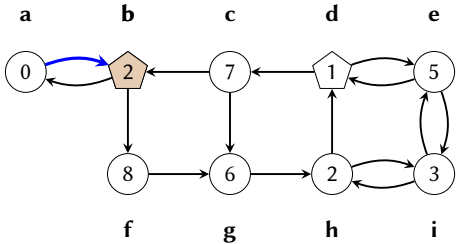
a	$\langle 0,0,0,2,1\rangle$	to	$\langle 0,1,0,0,1\rangle$	020202
b	$\langle 0,1,0,0,-\rangle$	to	$\langle 0,1,0,0,-\rangle$	20202
c	$\langle 0,-,-,-,-\rangle$	to	$\langle 0,-,-,-,-\rangle$	7
d	$\langle 0,0,0,0,-\rangle$	to	$\langle 0,0,0,0,-\rangle$	1
e	$\langle 0,0,-,-,-\rangle$	to	$\langle 0,0,-,-,-\rangle$	5
f	$\langle 1,-,-,-,-\rangle$	to	$\langle 1,-,-,-,-\rangle$	8
g	$\langle 0,1,-,-,-\rangle$	to	$\langle 0,1,-,-,-\rangle$	6
h	$\langle 0,0,0,1,-\rangle$	to	$\langle 0,0,0,1,-\rangle$	2
i	$\langle 0,0,0,-,-\rangle$	to	$\langle 0,0,0,-,-\rangle$	3

Small progress measures



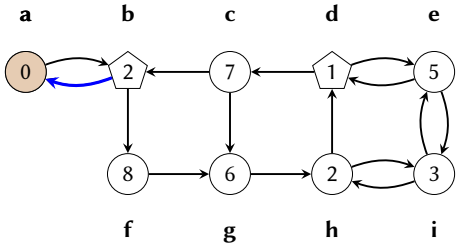
a	$\langle 0, 1, 0, 0, 1\rangle$	to	$\langle 0, 1, 0, 0, 1\rangle$	020202
b	$\langle 0, 1, 0, 0, -\rangle$	to	$\langle 0, 1, 0, 1, -\rangle$	2020202
c	$\langle 0, -, -, -, -\rangle$	to	$\langle 0, -, -, -, -\rangle$	7
d	$\langle 0, 0, 0, 0, -\rangle$	to	$\langle 0, 0, 0, 0, -\rangle$	1
e	$\langle 0, 0, -, -, -\rangle$	to	$\langle 0, 0, -, -, -\rangle$	5
f	$\langle 1, -, -, -, -\rangle$	to	$\langle 1, -, -, -, -\rangle$	8
g	$\langle 0, 1, -, -, -\rangle$	to	$\langle 0, 1, -, -, -\rangle$	6
h	$\langle 0, 0, 0, 1, -\rangle$	to	$\langle 0, 0, 0, 1, -\rangle$	2
i	$\langle 0, 0, 0, -, -\rangle$	to	$\langle 0, 0, 0, -, -\rangle$	3

Small progress measures



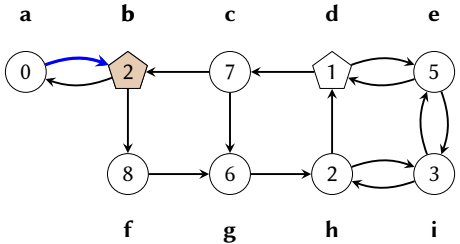
a	$\langle 0, 1, 0, 0, 1 \rangle$	to	$\langle 0, 1, 0, 1, 1 \rangle$	02020202
b	$\langle 0, 1, 0, 1, - \rangle$	to	$\langle 0, 1, 0, 1, - \rangle$	2020202
c	$\langle 0, -, -, -, - \rangle$	to	$\langle 0, -, -, -, - \rangle$	7
d	$\langle 0, 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, 0, - \rangle$	1
e	$\langle 0, 0, -, -, - \rangle$	to	$\langle 0, 0, -, -, - \rangle$	5
f	$\langle 1, -, -, -, - \rangle$	to	$\langle 1, -, -, -, - \rangle$	8
g	$\langle 0, 1, -, -, - \rangle$	to	$\langle 0, 1, -, -, - \rangle$	6
h	$\langle 0, 0, 0, 1, - \rangle$	to	$\langle 0, 0, 0, 1, - \rangle$	2
i	$\langle 0, 0, 0, -, - \rangle$	to	$\langle 0, 0, 0, -, - \rangle$	3

Small progress measures



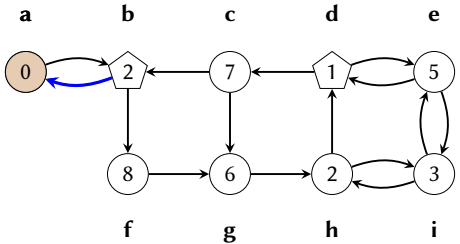
a	$\langle 0,1,0,1,1\rangle$	to	$\langle 0,1,0,1,1\rangle$	02020202
b	$\langle 0,1,0,1,-\rangle$	to	$\langle 0,1,0,2,-\rangle$	202020202
c	$\langle 0,-,-,-,-\rangle$	to	$\langle 0,-,-,-,-\rangle$	7
d	$\langle 0,0,0,0,-\rangle$	to	$\langle 0,0,0,0,-\rangle$	1
e	$\langle 0,0,-,-,-\rangle$	to	$\langle 0,0,-,-,-\rangle$	5
f	$\langle 1,-,-,-,-\rangle$	to	$\langle 1,-,-,-,-\rangle$	8
g	$\langle 0,1,-,-,-\rangle$	to	$\langle 0,1,-,-,-\rangle$	6
h	$\langle 0,0,0,1,-\rangle$	to	$\langle 0,0,0,1,-\rangle$	2
i	$\langle 0,0,0,-,-\rangle$	to	$\langle 0,0,0,-,-\rangle$	3

Small progress measures



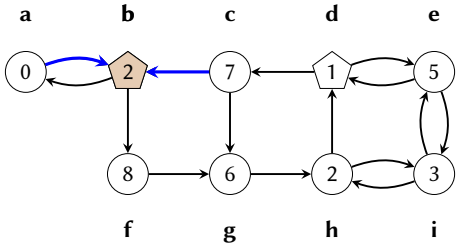
a	$\langle 0, 1, 0, 1, 1 \rangle$	to	$\langle 0, 1, 0, 2, 1 \rangle$	0202020202
b	$\langle 0, 1, 0, 2, - \rangle$	to	$\langle 0, 1, 0, 2, - \rangle$	202020202
c	$\langle 0, -, -, -, - \rangle$	to	$\langle 0, -, -, -, - \rangle$	7
d	$\langle 0, 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, 0, - \rangle$	1
e	$\langle 0, 0, -, -, - \rangle$	to	$\langle 0, 0, -, -, - \rangle$	5
f	$\langle 1, -, -, -, - \rangle$	to	$\langle 1, -, -, -, - \rangle$	8
g	$\langle 0, 1, -, -, - \rangle$	to	$\langle 0, 1, -, -, - \rangle$	6
h	$\langle 0, 0, 0, 1, - \rangle$	to	$\langle 0, 0, 0, 1, - \rangle$	2
i	$\langle 0, 0, 0, -, - \rangle$	to	$\langle 0, 0, 0, -, - \rangle$	3

Small progress measures



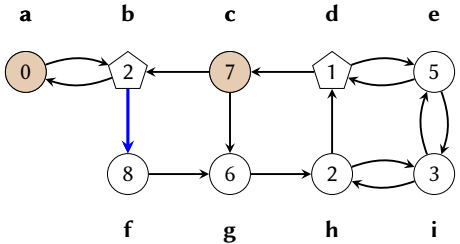
a	$\langle 0, 1, 0, 2, 1 \rangle$	to	$\langle 0, 1, 0, 2, 1 \rangle$	0202020202
b	$\langle 0, 1, 0, 2, - \rangle$	to	$\langle 1, 0, 0, 0, - \rangle$	20202020202
c	$\langle 0, -, -, -, - \rangle$	to	$\langle 0, -, -, -, - \rangle$	7
d	$\langle 0, 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, 0, - \rangle$	1
e	$\langle 0, 0, -, -, - \rangle$	to	$\langle 0, 0, -, -, - \rangle$	5
f	$\langle 1, -, -, -, - \rangle$	to	$\langle 1, -, -, -, - \rangle$	8
g	$\langle 0, 1, -, -, - \rangle$	to	$\langle 0, 1, -, -, - \rangle$	6
h	$\langle 0, 0, 0, 1, - \rangle$	to	$\langle 0, 0, 0, 1, - \rangle$	2
i	$\langle 0, 0, 0, -, - \rangle$	to	$\langle 0, 0, 0, -, - \rangle$	3

Small progress measures



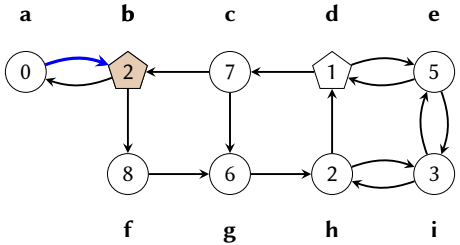
a	$\langle 0, 1, 0, 2, 1\rangle$	to	$\langle 1, 0, 0, 0, 1\rangle$	020202020202
b	$\langle 1, 0, 0, 0, -\rangle$	to	$\langle 1, 0, 0, 0, -\rangle$	20202020202
c	$\langle 0, -, -, -, -\rangle$	to	$\langle 1, -, -, -, -\rangle$	720202020202
d	$\langle 0, 0, 0, 0, -\rangle$	to	$\langle 0, 0, 0, 0, -\rangle$	1
e	$\langle 0, 0, -, -, -\rangle$	to	$\langle 0, 0, -, -, -\rangle$	5
f	$\langle 1, -, -, -, -\rangle$	to	$\langle 1, -, -, -, -\rangle$	8
g	$\langle 0, 1, -, -, -\rangle$	to	$\langle 0, 1, -, -, -\rangle$	6
h	$\langle 0, 0, 0, 1, -\rangle$	to	$\langle 0, 0, 0, 1, -\rangle$	2
i	$\langle 0, 0, 0, -, -\rangle$	to	$\langle 0, 0, 0, -, -\rangle$	3

Small progress measures



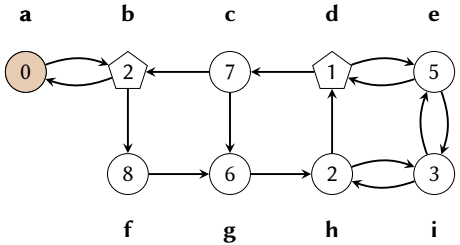
a	$\langle 1,0,0,0,1\rangle$	to	$\langle 1,0,0,0,1\rangle$	020202020202
b	$\langle 1,0,0,0,-\rangle$	to	$\langle 1,0,0,1,-\rangle$	28
c	$\langle 1,-,-,-,-\rangle$	to	$\langle 1,-,-,-,-\rangle$	728
d	$\langle 0,0,0,0,-\rangle$	to	$\langle 0,0,0,0,-\rangle$	1
e	$\langle 0,0,-,-,-\rangle$	to	$\langle 0,0,-,-,-\rangle$	5
f	$\langle 1,-,-,-,-\rangle$	to	$\langle 1,-,-,-,-\rangle$	8
g	$\langle 0,1,-,-,-\rangle$	to	$\langle 0,1,-,-,-\rangle$	6
h	$\langle 0,0,0,1,-\rangle$	to	$\langle 0,0,0,1,-\rangle$	2
i	$\langle 0,0,0,-,-\rangle$	to	$\langle 0,0,0,-,-\rangle$	3

Small progress measures



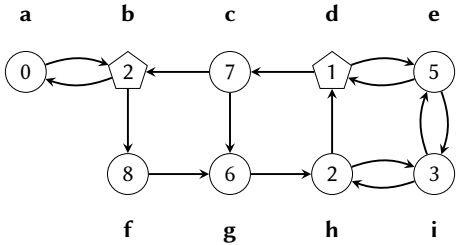
a	$\langle 1,0,0,0,1\rangle$	to	$\langle 1,0,0,1,1\rangle$	028
b	$\langle 1,0,0,1,-\rangle$	to	$\langle 1,0,0,1,-\rangle$	28
c	$\langle 1,-,-,-,-\rangle$	to	$\langle 1,-,-,-,-\rangle$	728
d	$\langle 0,0,0,0,-\rangle$	to	$\langle 0,0,0,0,-\rangle$	1
e	$\langle 0,0,-,-,-\rangle$	to	$\langle 0,0,-,-,-\rangle$	5
f	$\langle 1,-,-,-,-\rangle$	to	$\langle 1,-,-,-,-\rangle$	8
g	$\langle 0,1,-,-,-\rangle$	to	$\langle 0,1,-,-,-\rangle$	6
h	$\langle 0,0,0,1,-\rangle$	to	$\langle 0,0,0,1,-\rangle$	2
i	$\langle 0,0,0,-,-\rangle$	to	$\langle 0,0,0,-,-\rangle$	3

Small progress measures



a	$\langle 1, 0, 0, 1, 1 \rangle$	to	$\langle 1, 0, 0, 1, 1 \rangle$	028
b	$\langle 1, 0, 0, 1, - \rangle$	to	$\langle 1, 0, 0, 1, - \rangle$	28
c	$\langle 1, -, -, -, - \rangle$	to	$\langle 1, -, -, -, - \rangle$	728
d	$\langle 0, 0, 0, 0, - \rangle$	to	$\langle 0, 0, 0, 0, - \rangle$	1
e	$\langle 0, 0, -, -, - \rangle$	to	$\langle 0, 0, -, -, - \rangle$	5
f	$\langle 1, -, -, -, - \rangle$	to	$\langle 1, -, -, -, - \rangle$	8
g	$\langle 0, 1, -, -, - \rangle$	to	$\langle 0, 1, -, -, - \rangle$	6
h	$\langle 0, 0, 0, 1, - \rangle$	to	$\langle 0, 0, 0, 1, - \rangle$	2
i	$\langle 0, 0, 0, -, - \rangle$	to	$\langle 0, 0, 0, -, - \rangle$	3

Small progress measures



a	$\langle 1, 0, 0, 1, 1 \rangle$
b	$\langle 1, 0, 0, 1, - \rangle$
c	$\langle 1, -, -, -, - \rangle$
d	$\langle 0, 0, 0, 0, - \rangle$
e	$\langle 0, 0, -, -, - \rangle$
f	$\langle 1, -, -, -, - \rangle$
g	$\langle 0, 1, -, -, - \rangle$
h	$\langle 0, 0, 0, 1, - \rangle$
i	$\langle 0, 0, 0, -, - \rangle$

- No vertices are won by Even
- Strategy for Odd
  - from **b** to **f**
    (to **a** not the lowest successor)
  - from **d** to **e**
    (to **c** not the lowest successor)

Small progress measures

Operational interpretation [Gazda, Willemse, 2015]

- p-dominated stretches: how often is priority p encountered before a higher priority
- Example: play 00102120232142656201 corresponds to $\langle 2, 1, 3, 2 \rangle$
  - priority 0 is seen 2× before a higher priority
  - priority 2 is seen 3× before a higher priority
  - priority 4 is seen 1× before a higher priority
  - priority 6 is seen 2× before a higher priority

Operational interpretation [Van Dijk, 2018]

- Value represents the "statistic of the best continuation"
- *Interpretation of the "overflow mechanism":* If priority p is seen more than n_p times, there must be a cycle! If priority p overflows, our optimal path contains a cycle of priority p. Keep increasing the measure until the opponent is forced to escape...

Small progress measures

Operational interpretation [Gazda, Willemse, 2015]

- p-dominated stretches: how often is priority p encountered before a higher priority
- Example: play 00102120232142656201 corresponds to $\langle 2, 1, 3, 2 \rangle$
  - priority 0 is seen 2× before a higher priority
  - priority 2 is seen 3× before a higher priority
  - priority 4 is seen 1× before a higher priority
  - priority 6 is seen 2× before a higher priority

Operational interpretation [Van Dijk, 2018]

- Value represents the "statistic of the best continuation"
- *Interpretation of the "overflow mechanism":* If priority p is seen more than n_p times, there must be a cycle! If priority p overflows, our optimal path contains a cycle of priority p. Keep increasing the measure until the opponent is forced to escape...
- (In fact, my implementation in Oink already tries to resolve these cycles, before I had worked out the concept of tangles; now I have a much better solution with PMTL)

Small progress measures

```
1  def spm(𝒢, α):
2      μ ← V ↦ ⟨0, ..., 0⟩
3      Q ← {v ∈ V | pr(v) mod 2 = ᾱ}                    // use a queue or a stack
4      while Q ≠ ∅ :
5          v ← take(Q)                                   // take a v ∈ Q
6          if μ(v) < Lift(μ, v) :                        // can be lifted?
7              μ[v] ← Lift(μ, v)                         // update μ(v)
8              Q ← Q ∪ E^{-1}(v)                         // add predecessors to Q
9      W_ᾱ ← {v | μ(v) = ⊤}
10     W_α ← {v | μ(v) ≠ ⊤}
11     σ_α ← (W_α ∩ V_α) ↦ pick({u ∈ E(v) | μ(v) = Prog(μ(u), pr(v))})
12     return W_α, W_ᾱ, σ_α
```

$$\mathsf{Lift}(\mu, v) := \begin{cases} \min_{(v,w)\in E} \mathsf{Prog}(\mu(w), \mathsf{pr}(v)) & v \in V_\alpha \\ \max_{(v,w)\in E} \mathsf{Prog}(\mu(w), \mathsf{pr}(v)) & v \in V_{\overline{\alpha}} \end{cases}$$

$$\mathsf{Prog}(m, p) := \begin{cases} \min\{m' \in M \mid m' \geq_p m\} & p \text{ is } \alpha\text{'s priority} \\ \min\{m' \in M \mid m' >_p m\} & p \text{ is } \overline{\alpha}\text{'s priority} \end{cases}$$

Small progress measures

Implementation notes

- use a queue or stack to store “to do” vertices
- after lifting a vertex, add its predecessors to the queue (only once!)
  - a simple bit array can take care of this
- when lifting an $\bar{\alpha}$-priority vertex to $\top$, decrease n_p by 1
  - this shrinks the measure domain, because one overflow point has disappeared
- the most delicate part is implementing $<$ and `Prog`

Strategy improvement

Strategy improvement

Core algorithm

1. Start with some strategy σ for player α
2. Compute a best response τ for player $\bar{\alpha}$
3. Together, σ and τ induce exactly one play from every vertex
4. Compute the resulting valuation (also called play profile)
5. Find all switchable edges: local strategy changes that improve the valuation
6. Select some of them according to a switching rule, update σ, and repeat

When no improving switch remains, the current strategies are optimal.

Strategy improvement

Why compute a best response?

If τ is a best response to the current strategy σ, then the valuation tells us how good σ really is against the strongest counterplay.

So if we improve σ against that best response, we know that the new strategy is better.

How can we compute the best response?

Different approaches exist in the literature:

- shortest-path style methods (e.g. Bellman-Ford)
- specialized best-response procedures
- nested strategy improvement

We use the nested SI viewpoint of Fearnley. (CAV 2017)

Strategy improvement

Nested strategy improvement

To compute a best response for player $\bar{\alpha}$, we can run the same idea recursively:

- fix the current strategy σ of player α
- improve τ locally until no improving switch remains
- the resulting τ is a best response to σ

Resulting structure

Strategy improvement has two levels:

- an inner loop computing the best response
- an outer loop improving the main strategy

Strategy improvement

Algorithm sketch (nested SI)

1. Pick some strategy σ for player α
2. Pick some strategy τ for player $\bar{\alpha}$
3. Compute the valuation induced by σ and τ
4. If player $\bar{\alpha}$ has a local improvement, update τ and go back to step 3
5. If player α has a local improvement, update σ and go back to step 3
6. Otherwise, stop

Strategy improvement

Why is the algorithm monotone?

There are two monotone phases:

- for fixed σ, the opponent improves τ to a best response
- against that best response, we update σ using improving switches

What improves?

The induced valuation changes monotonically:

- best-response updates make the valuation worse for the main player
- improving switches make the valuation better for the main player

So SI is a local optimization algorithm on strategies and their induced valuations.

Strategy improvement

Switching rules

A switching rule determines which improving edges are actually used.

Typical examples:

- **greedy all-switches:** at every vertex, if there is a better local choice, switch to a best one
- **single-switch:** perform only one improving switch
- **best-improvement:** choose the locally most profitable switch
- **heuristic rules:** tie-breaking, randomization, round-robin, etc.

Why does the switching rule matter?

- it strongly affects practical performance
- for every fixed game, some switch sequence is best
- the challenge is to find a general local rule that performs well

Strategy improvement

Valuations in the literature

Different strategy-improvement algorithms use different ways to value the induced plays.

- **Vöge–Jurdziński (2000):** a more elaborate profile based on the eventual cycle and the more relevant priorities seen before it
- **Fearnley (2017):** a simpler valuation using halting and counting how often each priority is seen
- **Koh–Loho (2022):** a quasi-polynomial strategy-iteration framework using universal-tree / progress-measure valuations

In this lecture

We use the Fearnley viewpoint:

- the main player can halt the play instead of continuing
- finite plays get a count-vector valuation
- infinite plays that are favorable for the main player get value $\top$

Strategy improvement

Fearnley's valuation with halting

The main player may choose to halt a play early (before each vertex), instead of continuing into an unfavorable continuation.

- let H be the current set of vertices where the main player halts
- initially, the main player halts before all opponent-priority vertices
- finite plays ending in H get a finite valuation
- plays that never halt get value $\top$

Intuition

The main player only continues a play when the continuation is positive for the main player: highest visited vertex is of their priority.
So infinite plays only arise by connecting stretches of the game dominated by the main player.

Strategy improvement

Fearnley's valuation

The valuation of an induced play is:

- $\top$, if the play is infinite
- otherwise a tuple $\langle e_d, e_{d-1}, \ldots, e_0 \rangle$, where e_p counts how often priority p is seen before halting

Comparing valuations

Two valuations are compared by the highest priority where they differ.

- if that priority is good for the main player, the larger count is better
- if that priority is bad for the main player, the smaller count is better
- $\top$ is best

Strategy improvement

Computing valuations in practice

Fix strategies σ and τ, and let $\theta := \sigma \cup \tau$.

Compute valuations by a backward search from the halt set H:

- initialize each vertex in H with the zero tuple
- propagate valuations backwards along θ^{-1}
- when moving backwards through a vertex v, increment the counter for $\mathrm{pr}(v)$

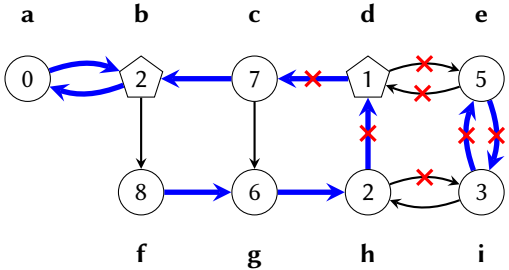
Vertices not reached in this backward search keep value $\top$.

Strategy improvement

Algorithm sketch (nested SI with halt set H)

1. Pick some strategy σ for player α
2. Pick some strategy τ for player $\bar{\alpha}$
3. Pick an initial halt set H
4. Compute the valuation induced by σ, τ, and H
5. If player $\bar{\alpha}$ has a local improvement, update τ and go back to step 4
6. If player α has a local improvement, update σ
7. Update H by removing vertices whose valuation is now positive for player α
8. If neither σ nor H changes, stop; otherwise go back to step 4

Strategy improvement example (even)

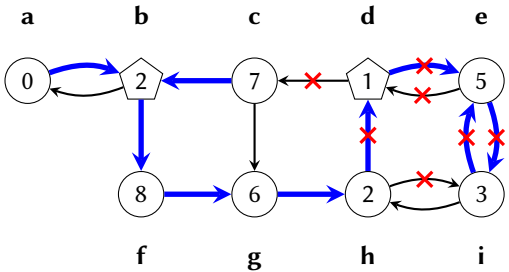


σ: $\mathbf{c} \to \mathbf{b}, \mathbf{e} \to \mathbf{i}, \mathbf{h} \to \mathbf{d}, \mathbf{i} \to \mathbf{e}$, τ: $\mathbf{b} \to \mathbf{a}, \mathbf{d} \to \mathbf{c}$, $H = \{\mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{i}\}$

a: $\top$
b: $\top$
c: $\top$
d: $[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$
e: $[0, 0, 0, 0, 1, 0, 0, 0, 0]$
f: $[0, 0, 1, 0, 0, 0, 1, 0, 1]$
g: $[0, 0, 1, 0, 0, 0, 1, 0, 0]$
h: $[0, 0, 1, 0, 0, 0, 0, 0, 0]$
i: $[0, 0, 0, 1, 0, 0, 0, 0, 0]$

update τ with $\mathbf{b} \to \mathbf{f}$ and $\mathbf{d} \to \mathbf{e}$

Strategy improvement example (even)

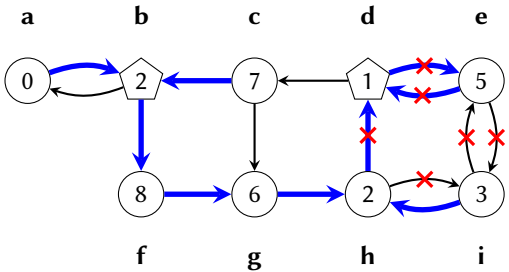


σ: $\mathbf{c} \to \mathbf{b}, \mathbf{e} \to \mathbf{i}, \mathbf{h} \to \mathbf{d}, \mathbf{i} \to \mathbf{e}$, τ: $\mathbf{b} \to \mathbf{f}, \mathbf{d} \to \mathbf{e}$, $H = \{\mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{i}\}$

a: $[1, 0, 2, 0, 0, 0, 1, 0, 1]$
b: $[0, 0, 2, 0, 0, 0, 1, 0, 1]$
c: $[0, 0, 2, 0, 0, 0, 1, 1, 1]$
d: $[0, 1, 0, 0, 0, 0, 0, 0, 0]$
e: $[0, 0, 0, 0, 1, 0, 0, 0, 0]$
f: $[0, 0, 1, 0, 0, 0, 1, 0, 1]$
g: $[0, 0, 1, 0, 0, 0, 1, 0, 0]$
h: $[0, 0, 1, 0, 0, 0, 0, 0, 0]$
i: $[0, 0, 0, 1, 0, 0, 0, 0, 0]$

no update to τ
update σ with $\mathbf{e} \to \mathbf{d}, \mathbf{i} \to \mathbf{h}$ and update H to $\{\mathbf{d}, \mathbf{e}, \mathbf{i}\}$

Strategy improvement example (even)

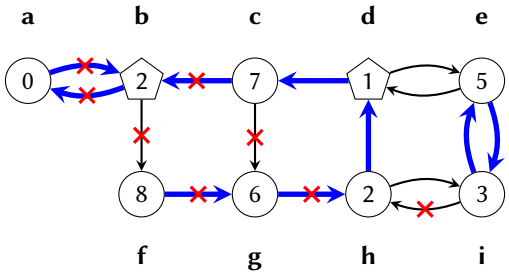


σ: $\mathbf{c} \to \mathbf{b}, \mathbf{e} \to \mathbf{d}, \mathbf{h} \to \mathbf{d}, \mathbf{i} \to \mathbf{h}$, τ: $\mathbf{b} \to \mathbf{f}, \mathbf{d} \to \mathbf{e}$, $H = \{\mathbf{d}, \mathbf{e}, \mathbf{i}\}$

a: $[1, 0, 2, 0, 0, 0, 1, 0, 1]$
b: $[0, 0, 2, 0, 0, 0, 1, 0, 1]$
c: $[0, 0, 2, 0, 0, 0, 1, 1, 1]$
d: $[0, 1, 0, 0, 0, 0, 0, 0, 0]$
e: $[0, 0, 0, 0, 1, 0, 0, 0, 0]$
f: $[0, 0, 1, 0, 0, 0, 1, 0, 1]$
g: $[0, 0, 1, 0, 0, 0, 1, 0, 0]$
h: $[0, 0, 1, 0, 0, 0, 0, 0, 0]$
i: $[0, 0, 0, 1, 0, 0, 0, 0, 0]$

no update to τ
no update to σ or to H
Odd wins all vertices with strategy τ

Strategy improvement example (odd)

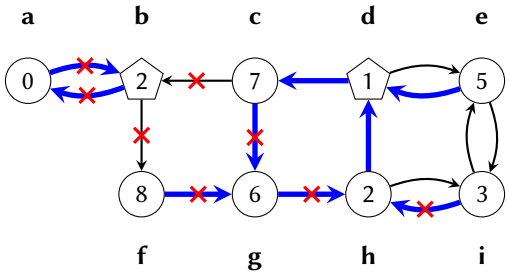


σ: $\mathbf{b} \to \mathbf{a}, \mathbf{d} \to \mathbf{c}$, τ: $\mathbf{c} \to \mathbf{b}, \mathbf{e} \to \mathbf{i}, \mathbf{h} \to \mathbf{d}, \mathbf{i} \to \mathbf{e}$, $H = \{\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{g}, \mathbf{h}\}$

a: $[1, 0, 0, 0, 0, 0, 0, 0, 0]$
b: $[0, 0, 1, 0, 0, 0, 0, 0, 0]$
c: $[0, 0, 0, 0, 0, 0, 0, 1, 0]$
d: $[0, 1, 0, 0, 0, 0, 0, 1, 0]$
e: $\top$
f: $[0, 0, 0, 0, 0, 0, 0, 0, 1]$
g: $[0, 0, 0, 0, 0, 0, 1, 0, 0]$
h: $[0, 1, 1, 0, 0, 0, 0, 1, 0]$
i: $\top$

update τ with $\mathbf{c} \to \mathbf{g}, \mathbf{e} \to \mathbf{d}, \mathbf{i} \to \mathbf{h}$

Strategy improvement example (odd)

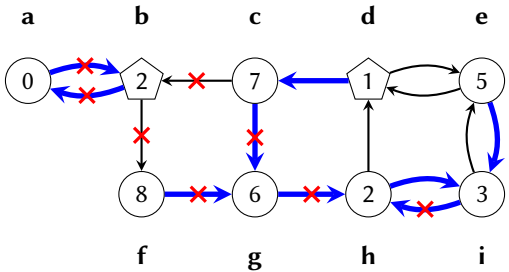


σ: $\mathbf{b} \to \mathbf{a}, \mathbf{d} \to \mathbf{c}$, τ: $\mathbf{c} \to \mathbf{g}, \mathbf{e} \to \mathbf{d}, \mathbf{h} \to \mathbf{d}, \mathbf{i} \to \mathbf{h}$, $H = \{\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{g}, \mathbf{h}\}$

a: $[1, 0, 0, 0, 0, 0, 0, 0, 0]$
b: $[0, 0, 1, 0, 0, 0, 0, 0, 0]$
c: $[0, 0, 0, 0, 0, 0, 0, 1, 0]$
d: $[0, 1, 0, 0, 0, 0, 0, 1, 0]$
e: $[0, 1, 0, 0, 0, 1, 0, 1, 0]$
f: $[0, 0, 0, 0, 0, 0, 0, 0, 1]$
g: $[0, 0, 0, 0, 0, 0, 1, 0, 0]$
h: $[0, 1, 1, 0, 0, 0, 0, 1, 0]$
i: $[0, 0, 0, 1, 0, 0, 0, 0, 0]$

update τ with $\mathbf{e} \to \mathbf{i}$,$\mathbf{h} \to \mathbf{i}$

Strategy improvement example (odd)

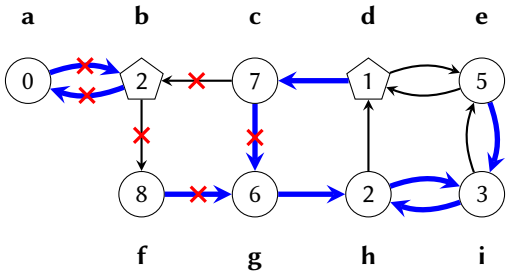


σ: $\mathbf{b} \to \mathbf{a}, \mathbf{d} \to \mathbf{c}$, τ: $\mathbf{c} \to \mathbf{g}, \mathbf{e} \to \mathbf{i}, \mathbf{h} \to \mathbf{i}, \mathbf{i} \to \mathbf{h}$, $H = \{\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{g}, \mathbf{h}\}$

a: $[1, 0, 0, 0, 0, 0, 0, 0, 0]$
b: $[0, 0, 1, 0, 0, 0, 0, 0, 0]$
c: $[0, 0, 0, 0, 0, 0, 0, 1, 0]$
d: $[0, 1, 0, 0, 0, 0, 0, 1, 0]$
e: $[0, 0, 0, 1, 0, 1, 0, 0, 0]$
f: $[0, 0, 0, 0, 0, 0, 0, 0, 1]$
g: $[0, 0, 0, 0, 0, 0, 1, 0, 0]$
h: $[0, 0, 1, 1, 0, 0, 0, 0, 0]$
i: $[0, 0, 0, 1, 0, 0, 0, 0, 0]$

no update to τ
no update to σ, update H to $\{\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{g}\}$

Strategy improvement example (odd)

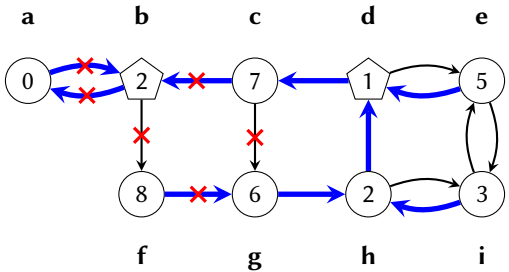


σ: $\mathbf{b} \to \mathbf{a}, \mathbf{d} \to \mathbf{c}$, τ: $\mathbf{c} \to \mathbf{g}, \mathbf{e} \to \mathbf{i}, \mathbf{h} \to \mathbf{i}, \mathbf{i} \to \mathbf{h}$, $H = \{\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{g}\}$

a: $[1,0,0,0,0,0,0,0,0]$	**d**: $[0,1,0,0,0,0,0,1,0]$	**g**: $\top$
b: $[0,0,1,0,0,0,0,0,0]$	**e**: $\top$	**h**: $\top$
c: $[0,0,0,0,0,0,0,1,0]$	**f**: $[0,0,0,0,0,0,0,0,1]$	**i**: $\top$

update τ with $\mathbf{c} \to \mathbf{b}, \mathbf{e} \to \mathbf{d}, \mathbf{h} \to \mathbf{d}$

Strategy improvement example (odd)

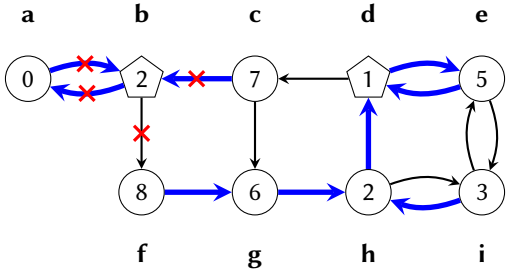


σ: $\mathbf{b} \to \mathbf{a}, \mathbf{d} \to \mathbf{c}$, τ: $\mathbf{c} \to \mathbf{b}, \mathbf{e} \to \mathbf{d}, \mathbf{h} \to \mathbf{d}, \mathbf{i} \to \mathbf{h}$, $H = \{\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{g}\}$

a: $[1, 0, 0, 0, 0, 0, 0, 0, 0]$
b: $[0, 0, 1, 0, 0, 0, 0, 0, 0]$
c: $[0, 0, 0, 0, 0, 0, 0, 1, 0]$
d: $[0, 1, 0, 0, 0, 0, 0, 1, 0]$
e: $[0, 1, 0, 0, 0, 1, 0, 1, 0]$
f: $[0, 0, 0, 0, 0, 0, 0, 0, 1]$
g: $[0, 1, 1, 0, 0, 0, 1, 1, 0]$
h: $[0, 1, 1, 0, 0, 0, 0, 1, 0]$
i: $[0, 1, 1, 1, 0, 0, 0, 1, 0]$

no update to τ
update σ with $\mathbf{d} \to \mathbf{e}$, update H to $\{\mathbf{a}, \mathbf{b}, \mathbf{f}\}$

Strategy improvement example (odd)

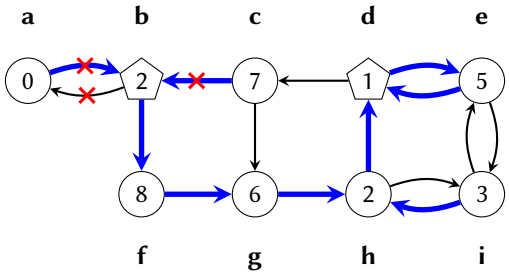


σ: $\mathbf{b} \to \mathbf{a}, \mathbf{d} \to \mathbf{e}$, τ: $\mathbf{c} \to \mathbf{b}, \mathbf{e} \to \mathbf{d}, \mathbf{h} \to \mathbf{d}, \mathbf{i} \to \mathbf{h}$, $H = \{\mathbf{a}, \mathbf{b}, \mathbf{f}\}$

a: $[1,0,0,0,0,0,0,0,0]$	**d**: $\top$	**g**: $\top$
b: $[0,0,1,0,0,0,0,0,0]$	**e**: $\top$	**h**: $\top$
c: $[0,0,0,0,0,0,0,1,0]$	**f**: $\top$	**i**: $\top$

no update to τ
update σ with $\mathbf{b} \to \mathbf{f}$, update H to $\{\mathbf{a}, \mathbf{b}\}$

Strategy improvement example (odd)

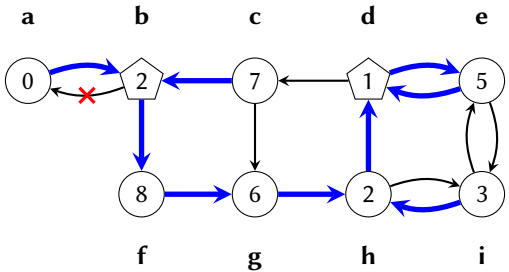


σ: $\mathbf{b} \to \mathbf{f}, \mathbf{d} \to \mathbf{e}$, τ: $\mathbf{c} \to \mathbf{b}, \mathbf{e} \to \mathbf{d}, \mathbf{h} \to \mathbf{d}, \mathbf{i} \to \mathbf{h}$, $H = \{\mathbf{a}, \mathbf{b}\}$

a: $[1, 0, 0, 0, 0, 0, 0, 0, 0]$	**d**: $\top$	**g**: $\top$
b: $\top$	**e**: $\top$	**h**: $\top$
c: $[0, 0, 0, 0, 0, 0, 0, 1, 0]$	**f**: $\top$	**i**: $\top$

no update to τ
no update to σ, update H to $\{\mathbf{a}\}$

Strategy improvement example (odd)

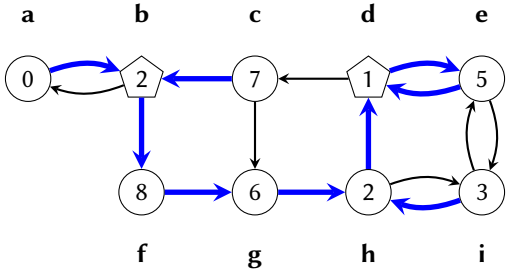


σ: $\mathbf{b} \to \mathbf{f}, \mathbf{d} \to \mathbf{e}$, τ: $\mathbf{c} \to \mathbf{b}, \mathbf{e} \to \mathbf{d}, \mathbf{h} \to \mathbf{d}, \mathbf{i} \to \mathbf{h}$, $H = \{\mathbf{a}\}$

a: $\top$	**d**: $\top$	**g**: $\top$
b: $\top$	**e**: $\top$	**h**: $\top$
c: $\top$	**f**: $\top$	**i**: $\top$

no update to τ
no update to σ, update H to $\emptyset$

Strategy improvement example (odd)



σ: $\mathbf{b} \to \mathbf{f}, \mathbf{d} \to \mathbf{e}$, τ: $\mathbf{c} \to \mathbf{b}, \mathbf{e} \to \mathbf{d}, \mathbf{h} \to \mathbf{d}, \mathbf{i} \to \mathbf{h}$, $H = \emptyset$

a: $\top$	**d**: $\top$	**g**: $\top$
b: $\top$	**e**: $\top$	**h**: $\top$
c: $\top$	**f**: $\top$	**i**: $\top$

no update to τ
no update to σ, no update to H
Odd wins all vertices with strategy σ

Strategy improvement

What did we just see?

In the example, strategy improvement repeatedly did three things:

- compute a best response
- recompute the induced valuation
- apply improving switches

Main lesson

The valuation guides local strategy changes, but the strategy changes can have global effects:

- a single switch may improve the valuation of many vertices
- changing the halt set H can turn long finite plays into $\top$
- the algorithm alternates between improving the opponent response and improving the main strategy

Strategy improvement

Why is strategy improvement interesting?

- conceptually simple: improve strategies until no better switch remains
- often effective in practice
- highly flexible: different valuations, best-response methods, and switching rules
- but difficult to analyze tightly

No polynomial bound on the number of iterations is known in general, and exponential lower bounds are known for several classical switching rules.

Strategy improvement

History and variants

- Originates from policy iteration for Markov decision processes and related stochastic games
- First parity-game specific algorithm by Vöge and Jurdziński (2000)
- Later work improved:
  - best-response computation
  - switching rules
  - valuation schemes
  - extensions such as learning snares / tangles, “symmetric strategy improvement”

Parallel implementations

Strategy improvement is also attractive in practice:

- suitable for parallel computation
- implementations developed in Twente on the Playstation 3 (Kandziora 2009, Van der Berg 2010)
- later work also explored GPU and multi-core implementations

Strategy improvement

Comparison with value iteration

Both are local optimization algorithms, but they optimize different objects.

Value iteration

- maintains a per-vertex value or measure
- updates those values directly
- monotonicity is immediate: values only move upward in the chosen order

Strategy improvement

- maintains strategies for the players
- valuations are recomputed from the current strategies
- monotonicity is indirect: the valuations are monotonically *decreasing* when computing the best response and monotonically *increasing* after each switch for the main player

Adversarial games

Adversarial games

An adversarial game is a family of parity games designed to expose a structural weakness of a solver.

Such examples help us:

- close off overly optimistic routes toward a polynomial-time algorithm
- identify exactly which *structural feature* makes an algorithm vulnerable
- understand what makes parity games *in general* hard and what a robust solver should do

We will look at adversarial games for three types of solvers:

- strategy improvement
- attractor-based decomposition
- progress-measure algorithms / separating-automata barriers

Three kinds of adversarial structure

Strategy improvement algorithms

Force the solver through a long sequence of tiny local strategy updates: counters, deceleration lanes, and long walks through strategy space.

Recursive / attractor-based algorithms

Force repeated decomposition and reset: attracting to the wrong target, re-solving similar subgames, or repeatedly rebuilding regions after new information is discovered.

Progress-measure / separator algorithms

Show that the underlying value domain or separating object must already be large, so this entire algorithmic approach cannot do better than quasi-polynomial size.

Adversarial games for strategy improvement (timeline)

- **2009 – Friedmann:** classical strategy improvement is exponential
- **2010 – Fearnley:** non-oblivious strategy improvement remembers structural information (*snares*)
- **2011 – Friedmann:** exponential lower bounds for two deterministic switching policies
  - switch *all* locally improving vertices
  - switch according to the *globally optimal* improvement
- **2011 – Friedmann:** subexponential lower bound for Zadeh's rule
  - later strengthened to an exponential lower bound
- **2013 – Avis, Friedmann:** exponential lower bound for Cunningham's rule
- **2013 – Friedmann:** super-polynomial lower bound for snare memorisation
- **2015 – Schewe, Trivedi, Varghese:** symmetric strategy improvement, designed to evade Friedmann's traps
- **2023 – van Dijk, Loho, Maat:** symmetric strategy improvement is still exponential
  - even for arbitrary improvement rules

Several of these lower bounds also transfer to MDPs and the simplex method in linear programming.

How Friedmann-style counterexamples work

- A collection of gadgets encodes a binary counter
- Each bit flip corresponds to a local strategy improvement
- When a higher bit becomes switchable, the lower bits are reset
- To prevent shortcutting, deceleration lanes delay the important switch
  - (this works because strategy improvement only makes small local updates)
- Hence the solver is forced through exponentially many intermediate strategies
- Different switching policies require different traps

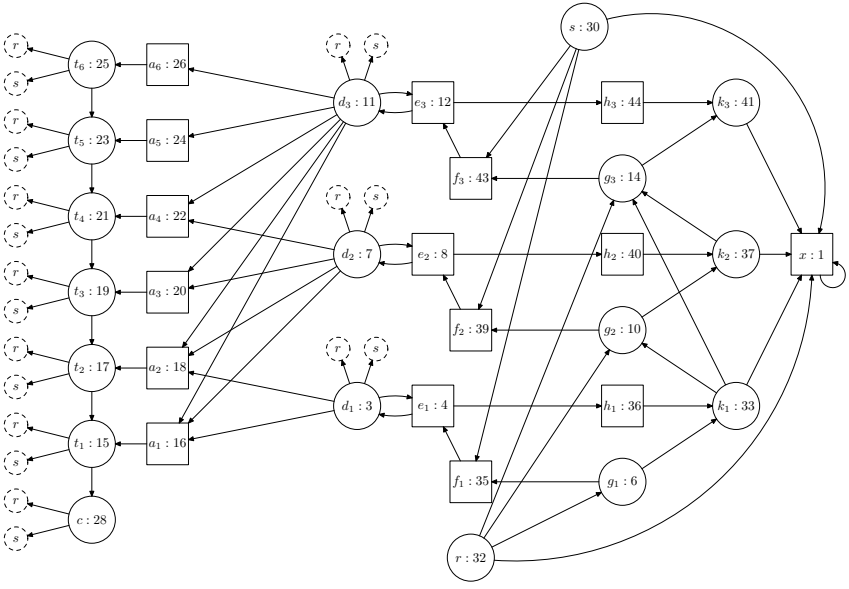
How Friedmann-style counterexamples work

- A collection of gadgets encodes a binary counter
- Each bit flip corresponds to a local strategy improvement
- When a higher bit becomes switchable, the lower bits are reset
- To prevent shortcutting, deceleration lanes delay the important switch
  - (this works because strategy improvement only makes small local updates)
- Hence the solver is forced through exponentially many intermediate strategies
- Different switching policies require different traps

Extensive explanation in:

- O. Friedmann: An Exponential Lower Bound For The Latest Deterministic Strategy Iteration Algorithms. (2011)
- O. Friedmann: A Subexponential Lower Bound for Strategy Iteration based on Snare Memorization. (2013)

Example adversarial game



Animations

Oliver Friedmann's animations

The following animations illustrate strategy-improvement algorithms on adversarial examples.

- Publications page
- Plain strategy improvement, no snares
- Plain strategy improvement, greedy snares
- Enhanced strategy improvement, no snares
- Enhanced strategy improvement, greedy snares

Symmetric strategy improvement

Ordinary strategy improvement is asymmetric: one player improves against the current strategy or best response of the other player.

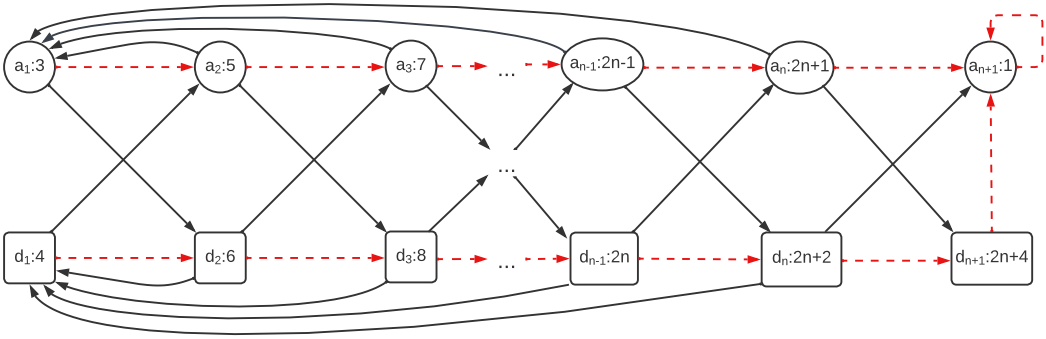
Symmetric strategy improvement instead runs two strategy-improvement processes in parallel.

In each iteration:

- compute the profitable switches for Even
- compute the profitable switches for Odd
- restrict each side using the current best response strategy of the other side
- update both strategies simultaneously

Symmetric strategy improvement avoids the classical traps.

A Gray-code style counterexample



Symmetric strategy improvement can be forced through a Gray-code-like sequence of local changes:

$$000 \to 001 \to 011 \to 010 \to 110 \to 111 \to 101 \to 100$$

Animation / code: Matthew Maat's GitHub SI-animations

From strategy improvement to recursive algorithms

Attractor-based algorithms ignore adversarial SI games

- Strategy-improvement counterexamples rely on locality: changes propagate only slowly through the graph. This is why counters, resets, and deceleration lanes work.
- Attractors skip over deceleration gadgets entirely.
- So adversarial games for attractor-based algorithms must use different principles: misleading decompositions, repeated resets, and repeated re-solving of similar lower regions.

Adversarial games for recursive / attractor-based solvers (timeline)

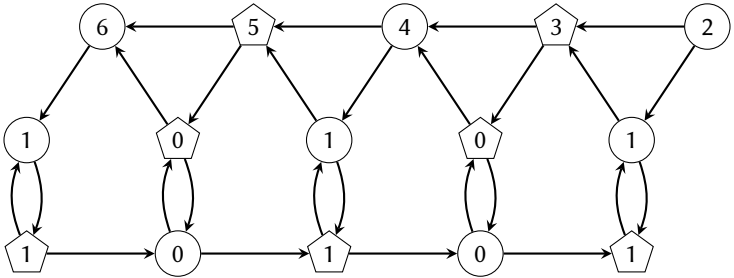
- **2013 – Gazda, Willemse:**
  Zielonka's Recursive Algorithm: dull, weak and solitaire games and tighter bounds
  - a simple family G gives an exponential lower bound for plain Zielonka
  - a second family M is resistant to the SCC optimization
- **2016 – Benerecetti, Dell'Erba, Mogavero:** *Priority promotion*
  - several adversarial games for several priority promotion variants
- **2017 – Benerecetti, Dell'Erba, Mogavero:**
  Robust Exponential Worst Cases for Divide-et-Impera Algorithms for Parity Games
  - robust against memoization and dominion-decomposition style fixes
  - still easy for tangle learning
- **2018 – van Dijk:** *Attracting Tangles to Solve Parity Games*
  - easily solves all known adversarial games
- **2018 – van Dijk:** *A Parity Game Tale of Two Counters*
  - broad exponential lower bound for attractor-based algorithms
  - exponential lower bound for delayed priority promotion and for tangle learning

A simple adversarial game for Zielonka

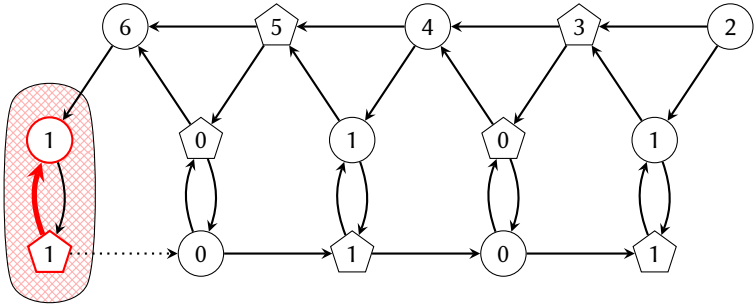
Example

The M family by Gazda and Willemse shows how Zielonka can be forced to re-solve essentially the same lower region again and again.

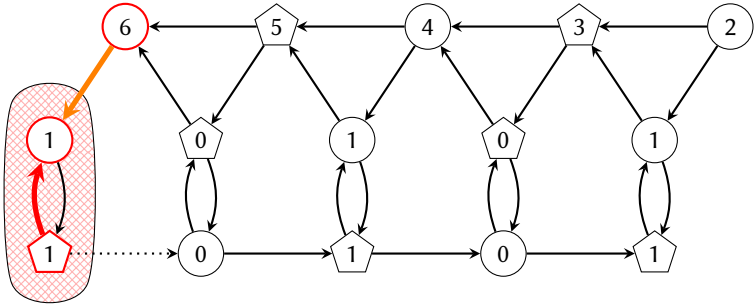
The M family of adversarial games for Zielonka



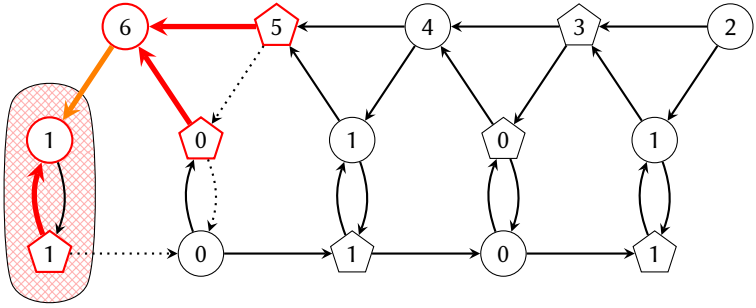
The M family of adversarial games for Zielonka



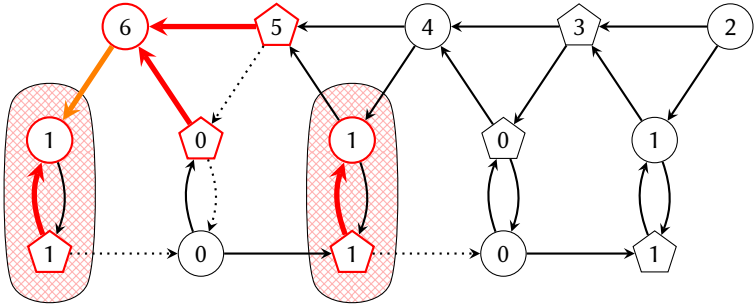
The M family of adversarial games for Zielonka



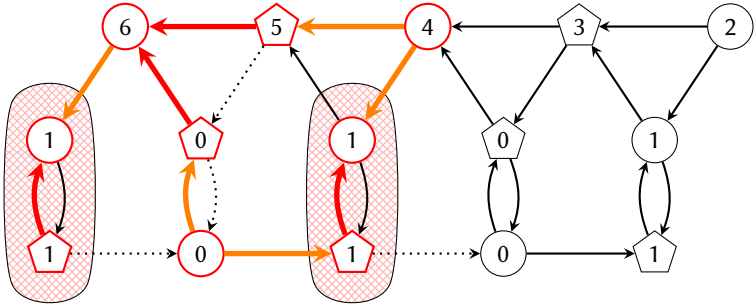
The M family of adversarial games for Zielonka



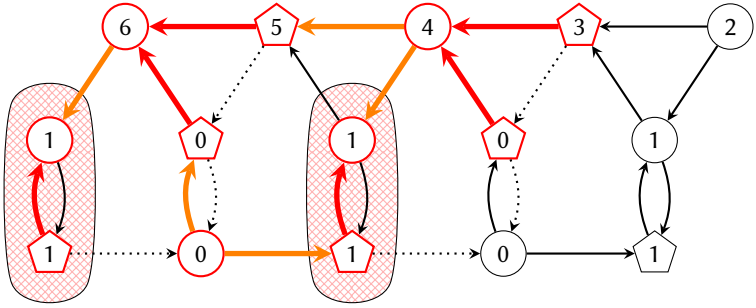
The M family of adversarial games for Zielonka



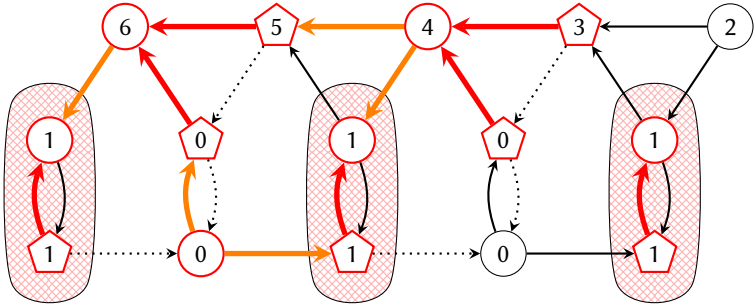
The M family of adversarial games for Zielonka



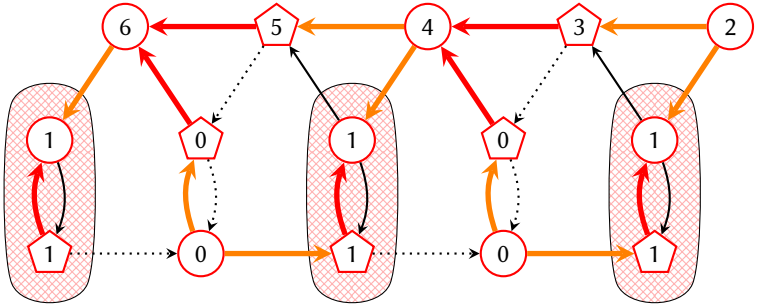
The M family of adversarial games for Zielonka



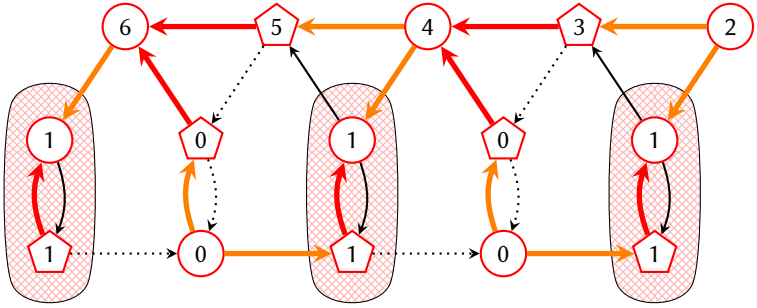
The M family of adversarial games for Zielonka



The M family of adversarial games for Zielonka

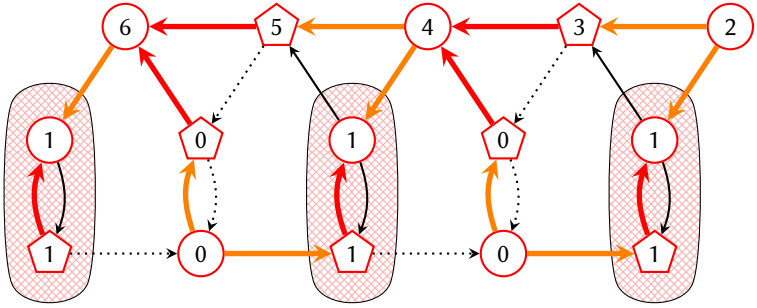


The M family of adversarial games for Zielonka



- Iterations: 5, 9, 19, 30, 56, 88, 155, 245, 419, 666, 1120, 1788, 2975, 4765, 7871, 12638, 20768, . . .

The M family of adversarial games for Zielonka



- Iterations: 5, 9, 19, 30, 56, 88, 155, 245, 419, 666, 1120, 1788, 2975, 4765, 7871, 12638, 20768, ...
- Exact recurrence:

$$T_1 = 5, \quad T_2 = 9, \qquad T_n = T_{n-1} + T_{n-2} + \begin{cases} 2, & n \text{ even}, \\ 2^{(n-1)/2} + 3, & n \text{ odd}. \end{cases}$$

- Thus the family gives $T_n = \Omega(\varphi^n)$, with $\varphi = \frac{1+\sqrt{5}}{2}$; with $|V| = 3n$, this is $\Omega(1.174^{|V|})$.

What goes wrong for Zielonka?

- Zielonka solves a lower subgame recursively
- a small change higher up forces another recursive call
- that second call contains essentially the same lower subgame again

So the adversarial game forces repeated re-solving of similar lower regions.

This is however trivial for tangle learning (as are other adversarial games): once the relevant tangle is learned, the same lower structure need not be solved again.

Two Binary Counters

Main idea

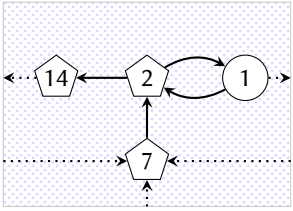
- The family is built from two interleaved binary counters of N bits
- One counter can reset progress made in the other
- Inspired by Friedmann's counterexample to Fearnley's strategy improvement with snares.

It gives exponential behaviour for several solver families:

- fixed-point iteration (`fpi`, `fpj`)
- Zielonka's recursive algorithm
- priority promotion and variants
- tangle learning

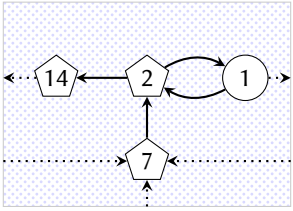
It delays all attractor-based algorithms that require opponent-priority regions to recognize distractions.

A bit in the game



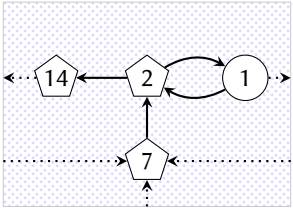
- This is one bit of the counter
- The bit is *set* if the tangle $\{ 2, 1 \}$ for Even is found

A bit in the game



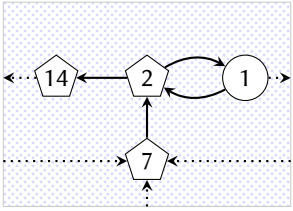
- This is one bit of the counter
- The bit is *set* if the tangle $\{ 2, 1 \}$ for Even is found
- 7 is a distraction for Odd
- Initially, the solver believes 7 is good for Odd and bad for Even
- Once the bit is set, the solver believes 7 is bad for Odd and good for Even

A bit in the game



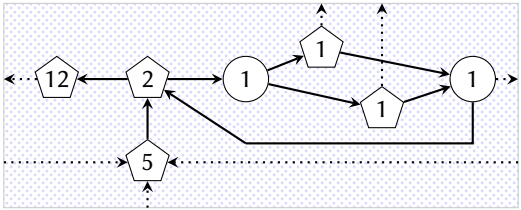
- This is one bit of the counter
- The bit is *set* if the tangle $\{ 2, 1 \}$ for Even is found
- 7 is a distraction for Odd
- Initially, the solver believes 7 is good for Odd and bad for Even
- Once the bit is set, the solver believes 7 is bad for Odd and good for Even
- Paths from 1 lead to distractions for Even, so it does not play to 2

A bit in the game



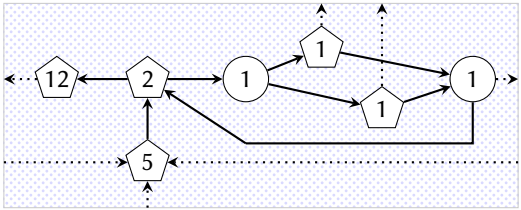
- This is one bit of the counter
- The bit is *set* if the tangle $\{ 2, 1 \}$ for Even is found
- 7 is a distraction for Odd
- Initially, the solver believes 7 is good for Odd and bad for Even
- Once the bit is set, the solver believes 7 is bad for Odd and good for Even
- Paths from 1 lead to distractions for Even, so it does not play to 2
- Only after all distractions (for vertex 1) are attracted by Odd, the solver can play from 1 to 2

A bit in the game



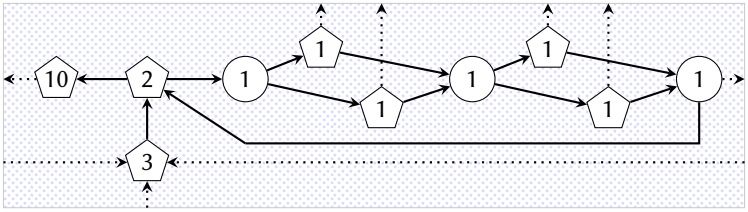
- This is a bit with *two variations*.
- One odd-controlled 1 vertex plays to a distraction for Even
- One odd-controlled 1 vertex plays to a distraction for Odd

A bit in the game



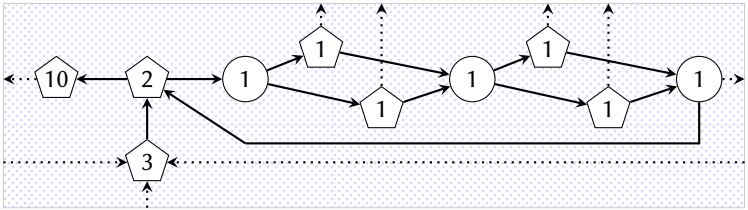
- This is a bit with *two variations*.
- One odd-controlled 1 vertex plays to a distraction for Even
- One odd-controlled 1 vertex plays to a distraction for Odd
- First, an “unstable” tangle is found that escapes to the Even distraction
- Later, a “stable” tangle is found that escapes to the Odd distraction (ultimately won by Even)

A bit in the game



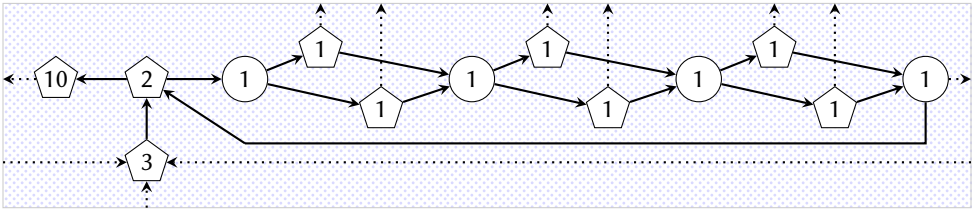
- This is a bit with *four variations*
- Two odd-controlled vertices play to distractions for Even
- Two odd-controlled vertices play to distractions for Odd

A bit in the game



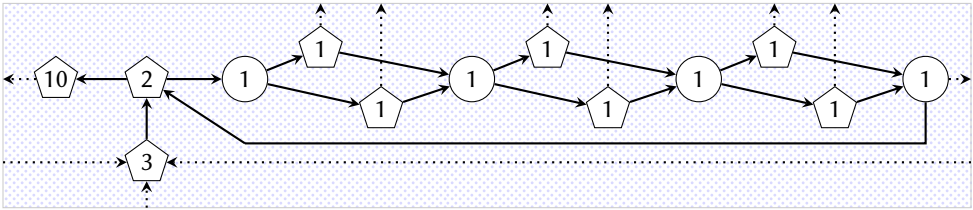
- This is a bit with *four variations*
- Two odd-controlled vertices play to distractions for Even
- Two odd-controlled vertices play to distractions for Odd
- First, seven “unstable” tangles are learned, with escapes to one or more Even distractions
- Finally, the “stable” tangle is found, with escapes to the three Odd distractions

A bit in the game



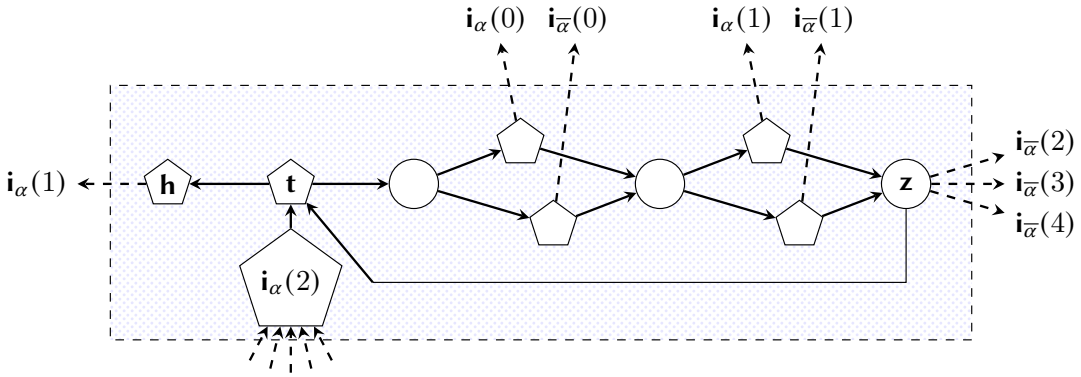
- This is a bit with *eight variations*
- Three odd-controlled vertices play to distractions for Even
- Three odd-controlled vertices play to distractions for Odd

A bit in the game



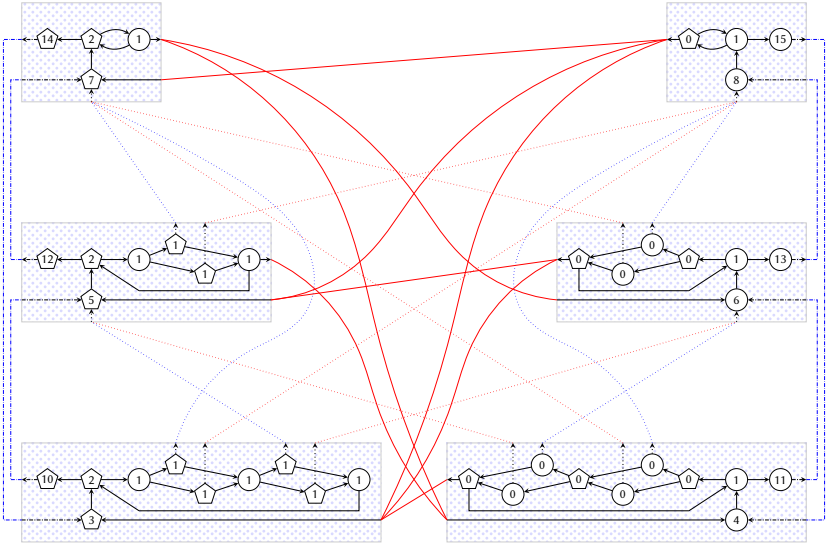
- This is a bit with *eight variations*
- Three odd-controlled vertices play to distractions for Even
- Three odd-controlled vertices play to distractions for Odd
- First, three “unstable” tangles are learned, with escapes to one or two Even distractions
- Finally, the “stable” tangle is found, with escapes to the two Odd distractions

A bit in the game



- A bit is set when the solver believes $\mathbf{i}_\alpha$ is a distraction
- If higher bits (0, 1) of the opponent counter are set (they are now bad), the bit is "reset" because the opponent can now escape
- A bit is not set until **z** plays to **t** instead of the distractions
  That is: first learn lower bits of the opponent counter

Example 3-bit Game



Example 3-bit game

Even	Odd	Event
000	000	Initial state
001	000	Set Even 3
001	001	Set Odd 3
011	000	Set Even 2 (reset Odd 3)
010	010	Set Odd 2 (reset Even 3)
011	010	Set Even 3
011	011	Set Odd 3
111	000	Set Even 1 (reset Odd 2, 3)
100	100	Set Odd 1 (reset Even 2, 3)
101	100	Set Even 3
101	101	Set Odd 3
111	100	Set Even 2 (reset Odd 3)
110	110	Set Odd 2 (reset Even 3)
111	110	Set Even 3
111	111	Set Odd 3

Summary

- Defeats all algorithms that rely only on opponent attraction to identify distractions.
- Until opponent attraction, the vertex is distracted and finding the tangle is delayed.

Summary

- Defeats all algorithms that rely only on opponent attraction to identify distractions.
- Until opponent attraction, the vertex is distracted and finding the tangle is delayed.

- Other methods: play to the best path, not just the best endpoint
  This is like progressive path values in progress measures

Summary

- Defeats all algorithms that rely only on opponent attraction to identify distractions.
- Until opponent attraction, the vertex is distracted and finding the tangle is delayed.

- Other methods: play to the best path, not just the best endpoint
  This is like progressive path values in progress measures

- Alternatives such as “recursive tangle learning”, “distraction-free tangle learning” and “progress measures with tangle learning” circumvent the Two Counters construction.

Adversarial games for progress measures

A different kind of adversarial phenomenon

For progress-measure algorithms, the issue is different.

- Strategy improvement is slowed down by a long walk through strategy space, using deceleration lanes to slowly increase bit counters
- Recursive / attractor-based algorithms are slowed down by repeated re-solving of similar subgames, driven by distraction deception
- Progress-measure / separator algorithms are limited by the size of the witness space itself

Universal trees and running time

Nathanaël Fijalkow: *An Optimal Value Iteration Algorithm for Parity Games* (2018)

> *"A vertex of priority 1 with a self-loop is evidently losing but the algorithm will lift the vertex $|T|$ times to get this information. It follows that the complexity of the algorithm is proportional to the size of the underlying universal tree."*

(more on universal trees next week...)

Progress measures: a different kind of barrier

Fijalkow's viewpoint

In the universal-tree framework:

- every universal tree yields a value-iteration algorithm
- the running time is proportional to the size of that tree

Progress measures: a different kind of barrier

Fijalkow's viewpoint

In the universal-tree framework:

- every universal tree yields a value-iteration algorithm
- the running time is proportional to the size of that tree

All known witness spaces in the value-iteration line are universal-tree-like.

- small progress measures use an exponential universal tree
- succinct progress measures: an explicit universal tree
- Strahler-universal trees
- separating automata with a hidden universal ordered tree
  - ordered progress measures (Fearnley et al) / play summaries (Calude et al)
  - smaller progress measures (Dell'Erba and Schewe)
  - register-game style approaches

Progress measures: a different kind of barrier

Fijalkow's viewpoint

In the universal-tree framework:

- every universal tree yields a value-iteration algorithm
- the running time is proportional to the size of that tree

All known witness spaces in the value-iteration line are universal-tree-like.

- small progress measures use an exponential universal tree
- succinct progress measures: an explicit universal tree
- Strahler-universal trees
- separating automata with a hidden universal ordered tree
  - ordered progress measures (Fearnley et al) / play summaries (Calude et al)
  - smaller progress measures (Dell'Erba and Schewe)
  - register-game style approaches

Result: the *smallest* universal tree is *quasi-polynomial* in size.

Separating automata: a structural lower bound

Main theorem

Every separating safety automaton for parity games has quasi-polynomial size.

- play summaries, progress measures, and register games fit this framework
- every separating automaton has a universal ordered tree hidden inside it
- so this whole separator-based design route inherits a quasi-polynomial barrier

Czerwiński, Daviaud, Fijalkow, Jurdziński, Lazić, Parys (2018):
Universal trees grow inside separating automata: Quasi-polynomial lower bounds for parity games

Takeaway for progress-measure algorithms

This is a real barrier

A large witness space does *not* by itself imply a slow algorithm: in principle, one could hope to jump to the fixed point quickly.

But for plain value iteration over universal-tree-like witnesses, the running time is proportional to the size of the witness space.

Takeaway for progress-measure algorithms

This is a real barrier

A large witness space does *not* by itself imply a slow algorithm: in principle, one could hope to jump to the fixed point quickly.

But for plain value iteration over universal-tree-like witnesses, the running time is proportional to the size of the witness space.

Beyond plain value iteration

To do better, one needs something structurally different: for example by using tangle-aware progress measures (PMTL) we can take major shortcuts and solve parity games without walking the entire universal tree.

Quasi-polynomial progress measures

Quasi-polynomial parity game algorithms

Polynomial time is still open, but several algorithms achieve quasi-polynomial time.

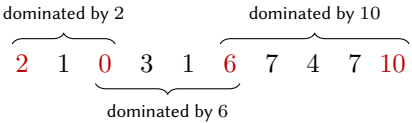
There are two main strands of quasi-polynomial time parity game algorithms:

- compress the witness space
  play summaries, ordered progress measures, succinct encodings, universal trees
- restrict the recursive search
  Parys / Lehtinen style recursive solving, with universal trees in the recursive structure

Dominating stretches

Consider a finite path fragment $v_i \rightsquigarrow v_j$ with priorities p_i and p_j.

This stretch is dominated if every priority seen strictly between v_i and v_j is at most $\max(p_i, p_j)$.

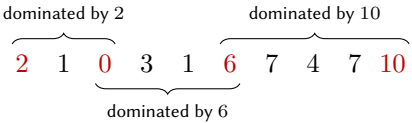


In this example, player Even dominates 4 times.

Dominating stretches

Consider a finite path fragment $v_i \rightsquigarrow v_j$ with priorities p_i and p_j.

This stretch is dominated if every priority seen strictly between v_i and v_j is at most $\max(p_i, p_j)$.



In this example, player Even dominates 4 times.

Also, player Odd dominates 5 times.

2 1 0 3 1 6 7 4 7 10

Calude-style witnesses

Main idea

Can we certify that one player dominates 2^k times along a play?
(With $2^k > |V|$, we know Even can force favorable cycles.)

- A witness is evidence that Even can keep producing more relevant even-than-odd behavior for exponentially long prefixes
- These objects underlie Calude-style play summaries
- Ordered progress measures are a progress-measure style reinterpretation of that idea

Intuition

Instead of recording the exact play, we record enough information to say:

"Even can dominate $1, 2, 4, 8, \ldots, 2^k$ *times."*

A witness as a compressed summary

Store only two things:

$$(p, i)$$

where p is the first selected priority, and i is the number of selected priorities in the witness.

For Even, a 4-witness headed by p has the form

$$p \rightsquigarrow E \rightsquigarrow E \rightsquigarrow E$$

where each arrow is an Even-dominated stretch.

head p	selected priorities	meaning
p even	$E \rightsquigarrow E \rightsquigarrow E \rightsquigarrow E$	4 Even checkpoints
p odd	$p \rightsquigarrow E \rightsquigarrow E \rightsquigarrow E$	3 Even checkpoints after p

Which witness is better for Even?

Suppose two witnesses have the same length. Then their first priority determines which one is better for Even.

$6 \rightsquigarrow E \rightsquigarrow E$	best: high even head
$2 \rightsquigarrow E \rightsquigarrow E$	good: low even head
$1 \rightsquigarrow E \rightsquigarrow E$	bad, but only low odd
$5 \rightsquigarrow E \rightsquigarrow E$	worst: high odd head

For Even:

high even $>$ low even $>$ low odd $>$ high odd.

For priorities $0, \ldots, 8$, this gives the order

$$10 > 8 > 6 > 4 > 2 > 0 > 1 > 3 > 5 > 7 > 9$$

A tuple of witnesses

Instead of storing one witness length, we store powers of two.

$$b = \langle b_8, b_4, b_2, b_1 \rangle$$

entry	meaning
b_8	head priority of a 8-witness
b_4	head priority of a 4-witness
b_2	head priority of a 2-witness
b_1	head priority of a 1-witness

Examples

tuple	one compatible interpretation
$\langle _, _, _, 2 \rangle$	2 a 1-witness headed by 2
$\langle _, _, 3, 2 \rangle$	$2 \rightsquigarrow 3 \rightsquigarrow E$ a 1-witness headed by 2, and a 2-witness headed by 3
$\langle 4, _, 3, 2 \rangle$	$2 \rightsquigarrow 3 \rightsquigarrow E \rightsquigarrow 4 \rightsquigarrow E^7$ also an 8-witness headed by 4
$\langle 5, _, 3, 2 \rangle$	$2 \rightsquigarrow 3 \rightsquigarrow E \rightsquigarrow 5 \rightsquigarrow E^7$ an 8-witness, but now headed by odd 5

Updating a tuple of witnesses

Assume we update witnesses for player Even.

$$b = \langle b_8, b_4, b_2, b_1 \rangle$$

Adding a new 1-witness

Seeing an even priority creates a 1-witness.

$$\langle _, _, _, _ \rangle \xrightarrow{6} \langle _, _, _, 6 \rangle$$

The new 6 is stored as a 1-witness.

Updating a tuple of witnesses

Assume we update witnesses for player Even.

$$b = \langle b_8, b_4, b_2, b_1 \rangle$$

Carrying upward

If there is already an Even 1-witness, the new Even priority combines with it into a 2-witness.

$$\langle _, _, _, 2 \rangle \xrightarrow{6} \langle _, _, 6, _ \rangle$$

$$6 \rightsquigarrow 2$$

The 1-entry is cleared, and the carry lands in the 2-entry.

Updating a tuple of witnesses

Assume we update witnesses for player Even.

$$b = \langle b_8, b_4, b_2, b_1 \rangle$$

Example: seeing another 2

Start with

$$\langle _, 4, 3, 2 \rangle$$

and prepend priority 2.

$$\langle _, 4, 3, 2 \rangle \xrightarrow{2} \langle _, 4, 2, _ \rangle$$

The new 2 carries through the existing Even 1-witness headed by 2, and lands in the 2-entry.

Updating a tuple of witnesses

Assume we update witnesses for player Even.

$$b = \langle b_8, b_4, b_2, b_1 \rangle$$

Example: seeing a 4

Start again with

$$\langle _, 4, 3, 2 \rangle$$

and prepend priority 4.

$$\langle _, 4, 3, 2 \rangle \xrightarrow{4} \langle _, 4, 4, _ \rangle$$

The new 4 carries through the 1-entry and replaces the odd-headed 2-witness.

Updating a tuple of witnesses

Assume we update witnesses for player Even.

$$b = \langle b_8, b_4, b_2, b_1 \rangle$$

Example: seeing a 5

Start again with

$$\langle _, 4, 3, 2 \rangle$$

and prepend priority 5.

$$\langle _, 4, 3, 2 \rangle \xrightarrow{5} \langle _, 4, 5, _ \rangle \rightsquigarrow \langle _, 5, _, _ \rangle$$

First 5 is inserted in the 2-entry. Then it dominates the existing 4-entry, so it moves there and everything below is cleared.

Updating a tuple of witnesses

Assume we update witnesses for player Even.

$$b = \langle b_8, b_4, b_2, b_1 \rangle$$

Dominating a larger entry

Now start with

$$\langle 4, _, 3, 2 \rangle.$$

$$\langle 4, _, 3, 2 \rangle \xrightarrow{5} \langle 4, _, 5, _ \rangle \rightsquigarrow \langle 5, _, _, _ \rangle$$

The new odd priority 5 dominates the 8-entry headed by 4. It replaces it, and the lower entries are forgotten.

Updating a tuple of witnesses

Assume we update witnesses for player Even.

$$b = \langle b_8, b_4, b_2, b_1 \rangle$$

Dominating a larger entry with an even priority

Start again with

$$\langle 4, _, 3, 2 \rangle.$$

and prepend priority 6.

$$\langle 4, _, 3, 2 \rangle \xrightarrow{6} \langle 4, _, 6, _ \rangle \rightsquigarrow \langle 6, _, _, _ \rangle$$

The new even priority 6 dominates the 8-entry headed by 4. This is good for Even: the large witness is now headed by 6.

Updating a tuple of witnesses

Assume we update witnesses for player Even.

$$b = \langle b_8, b_4, b_2, b_1 \rangle$$

Two-phase update

When we prepend a priority p:

1. **Carry phase.** Insert p at the 1-entry. If that entry is already Even-headed, clear it and carry p to the next entry. Repeat while the next entry is Even-headed.
2. **Domination phase.** Starting from the largest witness size, find a stored head dominated by p. Replace it by p, and clear all smaller entries.

Problem: raw update is not monotone

Raw tuple update does not preserve the tuple order.

	before	after seeing 6
b	$\langle _, 4, 2 \rangle$	$\langle 6, _, _ \rangle$
d	$\langle 9, 8, _ \rangle$	$\langle 9, 8, 6 \rangle$

$b < d$ but $\mathrm{up}(b, 6) > \mathrm{up}(d, 6)$.

Why?

The order values the largest stored witness first. Tuple d is higher because its 9-headed large witness would become excellent for Even after seeing 10.

But seeing 6 cannot use that promise: 6 does not dominate 9. It can only carry through the smaller even heads $4, 2$ in b.

Antagonistic update

Raw update:

$$b < d \not\Rightarrow \mathrm{up}(b, c) \leq \mathrm{up}(d, c).$$

Antagonistic update:

$$\mathrm{au}(b, c) = \min\{\mathrm{up}(d, c) \mid d \geq b\}.$$

Reason

If $b \leq d$, then Odd has at least as many choices from b as from d. So the minimum from b cannot be better for Even than the minimum from d.

$$b \leq d \quad \Rightarrow \quad \mathrm{au}(b, c) \leq \mathrm{au}(d, c).$$

This gives the monotone update needed for value iteration.

Antagonistic update in the implementation

For this algorithm, it is enough to also try the next higher tuple whose 1-witness entry is cleared.

$\mathrm{bump}(b)$ is the smallest possible increase in the tuple order that clears the 1-witness entry (if not already cleared).

$$\mathrm{prog}(b, p) = \min\big(\mathrm{up}(b, p), \mathrm{up}(\mathrm{bump}(b), p)\big)$$

	x	$\mathrm{up}(x, 6)$	$\mathrm{bump}(x)$	$\mathrm{up}(\mathrm{bump}(x), 6)$
b	$\langle _, 4, 2\rangle$	$\langle 6, _, _\rangle$	$\langle _, 6, _\rangle$	$\langle _, 6, 6\rangle$
d	$\langle 9, 8, _\rangle$	$\langle 9, 8, 6\rangle$	–	–

$$\mathrm{prog}(b, 6) = \langle _, 6, 6\rangle, \qquad \mathrm{prog}(d, 6) = \langle 9, 8, 6\rangle.$$

$$b < d \qquad \text{and now} \qquad \mathrm{prog}(b, 6) < \mathrm{prog}(d, 6).$$

Why this is quasi-polynomial

Each vertex stores

$$\langle b_{2^k}, \ldots, b_4, b_2, b_1 \rangle, \qquad k = \lceil \log_2 n \rceil.$$

Each entry is either _ or a priority:

$$d + 1 \text{ choices.}$$

So the value space has size at most

$$n \cdot (d+1)^{\lceil \log_2 n \rceil}.$$

Bottom line

We keep only logarithmically many witness lengths:

$$1, 2, 4, 8, \ldots$$

This is the quasi-polynomial compression.

Universal trees as the common structure

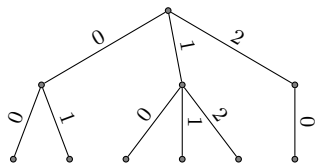
Tuples are paths in a tree

So far, a measure was a tuple:

$$\langle a_1, a_2, \ldots, a_h \rangle.$$

But a tuple can also be read as a navigation path:

$\langle 1, 2 \rangle$ = take branch 1, then branch 2.



Tuples are paths in an ordered tree

Progress measures do not really need the actual values on the edges. They need a way to record a hierarchical ordering between vertices. Lifting a vertex means adjusting its order w.r.t other vertices. The ordered tree just needs to allow for every possible ordering.

Which tree should we use?

Progress measures need a value space large enough to represent all possible progress of a winning strategy.

Naive choice

Use a full tree with many branches at every level. That works, but it is too large.

This is actually the small progress measures algorithm.

Better choice

Use one tree that is large enough to contain every tree we might need, but no larger than necessary.

large enough for all cases + as small as possible = universal tree.

Universal trees

Definition

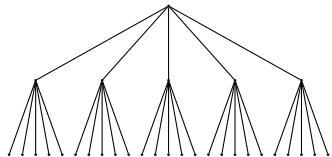
An (n, h)-universal tree is an ordered tree that can embed every ordered tree of height h with at most n leaves.

- n corresponds to the number of vertices / leaves needed;
- h corresponds to the number of relevant priority levels;
- the leaves of the universal tree form the progress-measure domain.

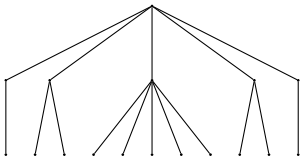
Universal trees can be much smaller

Naive tree versus universal tree

A (n,h)-universal tree can embed all trees of height h and width n.



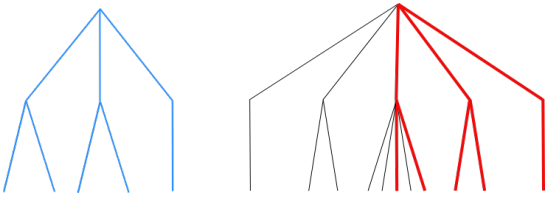
The naive (5,2)-universal tree of size 25



A (5,2)-universal tree of size 11

Universal trees

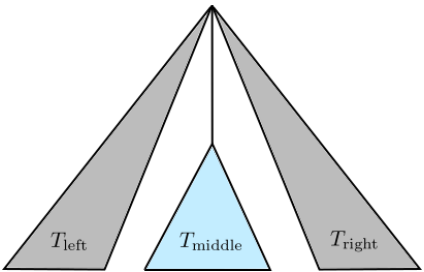
A (n,h)-universal tree can embed all trees with height h and width n.



The tree on the left is embedded into the universal tree

The recursive construction

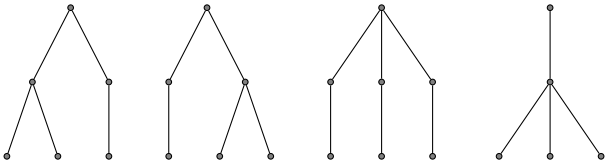
Universal trees are built recursively by splitting the leaves:



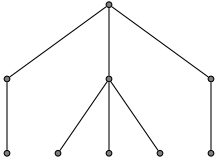
- Left: universal tree of height h, width $\lfloor n/2 \rfloor$;
- Middle: universal tree of height $h-1$, width n;
- Right: universal tree of height h, width $n-1-\lfloor n/2 \rfloor$.

Universal trees

All possible trees of height 2 and width 3:

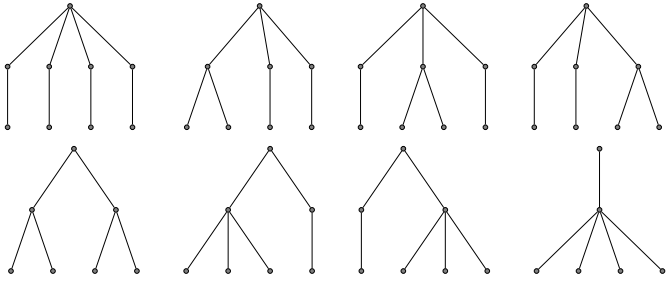


A universal tree of height 2 and width 3:

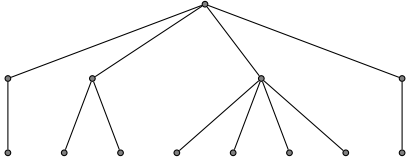


Universal tree

All possible trees of height 2 and width 4:

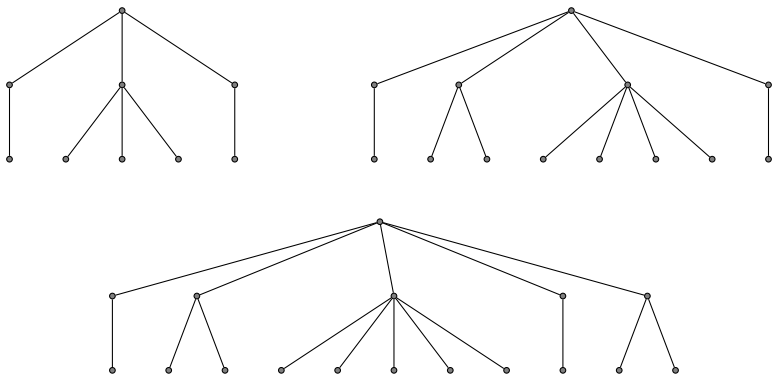


A universal tree of height 2 and width 4:



Universal trees

Universal trees for height 2 and widths 3, 4, 5



Universal trees

Size of universal trees

$$\begin{aligned} f(0,h) &= 0 && \text{for } h \geq 0 \\ f(n,0) &= 1 && \text{for } n \geq 1 \\ f(n,h) &= f(n,h-1) + f(\lfloor n/2 \rfloor, h) + f(n-1-\lfloor n/2 \rfloor, h) && \text{for } n,h \geq 1 \\ &\leq O\left(n^{2.45-\varepsilon+\log_2\left(\frac{h}{\log_2(n)}\right)}\right) \end{aligned}$$

With priorities on vertices, for $d = \omega(\log n)$, one can find a winner in a parity game with n vertices, m edges, and d priorities in time

$$O\left(m \cdot n^{1.45+\log_2\left(\frac{d}{\log_2(n)}\right)} \cdot \frac{1}{d}\right)$$

(source: arXiv, Parys et al 2023)

Encoding the universal tree

Succinct tree encoding

Succinct idea

Describe each child by its position in a recursive split:

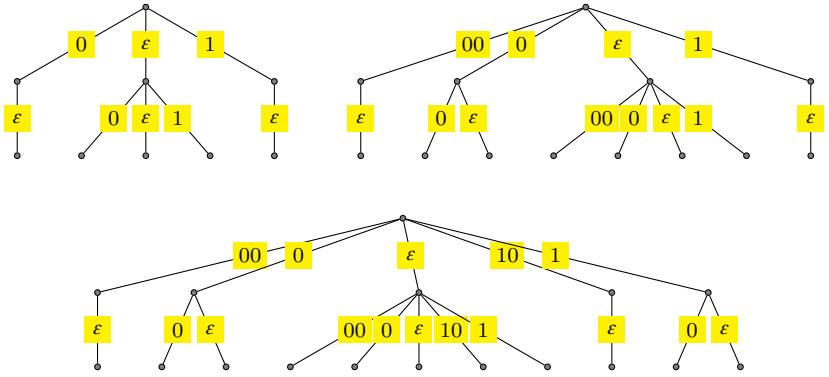
$$\text{Left} = 0, \qquad \text{Middle} = \varepsilon, \qquad \text{Right} = 1.$$

Repeated splits produce bitstrings.

Encoding children by recursive splitting

Succinct tree encoding for height 2 and widths $3, 4, 5$

Edge labels name branches using recursive $0, \varepsilon, 1$-splits.



Succinct progress measures

Jurdziński and Lazić observe that progress-measure values can be seen as paths in ordered trees.

Tree-coding result

In an ordered tree of height h with at most n leaves, every navigation path can be encoded as a tuple of bitstrings whose total bit length is at most

$$\lceil \lg n \rceil.$$

To store this tuple explicitly, we also need to remember at which level each bit occurs. This gives about

$$\lg h \cdot \lg n$$

bits.

For parity games, $h \approx d/2$, so the number of encoded values per vertex is roughly

$$2^{\lg d \cdot \lg n} = n^{\lg d}.$$

Quasi-polynomial recursive algorithm

Zielonka as shaving off opponent dominions

Let p be the current highest priority, won by player α.

Zielonka's loop, rephrased

1. remove the α-attractor to priority p;
2. solve the remaining lower-priority subgame;
3. if the opponent has a dominion W, remove

$$\mathrm{Attr}_{\bar{\alpha}}(W);$$

4. repeat on the remaining game.

So Zielonka repeatedly shaves off opponent dominions whose attractors may eat into the current priority-p region.

Parys' key observation: the shavings are disjoint

Each time we find an opponent dominion, we remove its attractor.

$$B_1,\ B_2,\ B_3, \ldots \qquad B_i = \mathrm{Attr}_{\bar{\alpha}}(W_i)$$

Observation

These removed regions are disjoint:

$$B_i \cap B_j = \emptyset \qquad (i \neq j).$$

Consequence

At most one removed region can have size $> n/2$.

Precision: how hard do we search?

A recursive call does not always need to find every opponent dominion.

Precision m

Search only for opponent dominions whose attractor has size at most m.

By disjointness, all but at most one removed region have size $\leq n/2$.

Parys' strategy

first	:	repeatedly search with precision $n/2$,
middle	:	search once with precision n,
last	:	repeatedly search with precision $n/2$.

Only one recursive call needs full precision.

Parys' algorithm: half–full–half

1 **def** $\mathbf{parys}_{●}(\mathcal{G}, p, n_{●}, n_{⬟})$**:**
2 **if** $\mathcal{G} = \emptyset \vee n_{●} = 0$ **: return** $\emptyset$ // Even wins nothing
3 **repeat**
4 $H \leftarrow \mathcal{G} \backslash \mathbf{attract}_{●}(\mathcal{G}, \mathrm{pr}^{-1}(p))$ // compute subgame H
5 $W_{⬟} \leftarrow \mathbf{parys}_{⬟}(H, p-1, n_{●}, n_{⬟}/2)$ // half precision
6 $\mathcal{G} \leftarrow \mathcal{G} \backslash \mathbf{attract}_{⬟}(\mathcal{G}, W_{\overline{\alpha}})$ // let opponent attract
7 **until** $W_{⬟} = \emptyset$
8 $H \leftarrow \mathcal{G} \backslash \mathbf{attract}_{●}(\mathcal{G}, \mathrm{pr}^{-1}(p))$ // compute subgame H
9 $W_{⬟} \leftarrow \mathbf{parys}_{⬟}(H, p-1, n_{●}, n_{⬟})$ // full precision
10 $\mathcal{G} \leftarrow \mathcal{G} \backslash \mathbf{attract}_{⬟}(\mathcal{G}, W_{\overline{\alpha}})$ // let opponent attract
11 **repeat**
12 $H \leftarrow \mathcal{G} \backslash \mathbf{attract}_{●}(\mathcal{G}, \mathrm{pr}^{-1}(p))$ // compute subgame H
13 $W_{⬟} \leftarrow \mathbf{parys}_{⬟}(H, p-1, n_{●}, n_{⬟}/2)$ // half precision
14 $\mathcal{G} \leftarrow \mathcal{G} \backslash \mathbf{attract}_{⬟}(\mathcal{G}, W_{\overline{\alpha}})$ // let opponent attract
15 **until** $W_{⬟} = \emptyset$
16 **return** $V_{\mathcal{G}}$ // return remaining nodes as won by Even

From Parys to Lehtinen

Parys shows that full precision is only needed once:

$$\text{many small searches} \quad + \quad \text{one full search} \quad + \quad \text{many small searches}$$

Lehtinen et al. turn this into a cleaner recursive pattern:

$$\text{half} \longrightarrow \text{remove highest priority} \longrightarrow \text{full lower-priority call} \longrightarrow \text{half.}$$

What improves?

The result is conceptually simpler and gives a better refined complexity bound.

Lehtinen et al.: recursive half–full–half

```
1 def lehtinen●(𝒢, p, n●, n🔺):
2     if 𝒢 = ∅ ∨ n🔺 = 0 : return V_𝒢                          // Even wins all
3     𝒢₁ ←lehtinen●(𝒢, p, n●, ⌊n🔺/2⌋)                  // left part of universal tree
4     Z ← {v ∈ 𝒢₁ | pr(v) = p}                          // find all nodes of priority p
5     H ← 𝒢₁\attract●(●, 𝒢₁, Z)                          // compute remaining subgame
6     W🔺 ←lehtinen🔺(H, p − 1, n●, n🔺)                  // solve subgame recursively
7     𝒢₂ ← 𝒢₁\attract🔺(🔺, 𝒢₁, W🔺)                       // remove won by opponent
8     𝒢₃ ←lehtinen●(𝒢₂, p, n●, ⌊n🔺/2⌋)                  // right part of universal tree
9     return 𝒢₃                                  // return remaining nodes as won by Even
```

Quasi-polynomial recursive algorithm

Universal trees everywhere!

- Observation: Parys' recursive algorithm's call tree is two interleaved universal trees!
- Improvement: a recursive algorithm parameterised by a universal tree
- Complexity: $n^{O\left(\log_2\left(1+\frac{d}{\log_2(n)}\right)\right)}$
- See: Lehtinen K., Parys P., Schewe S., Wojtczak D.: A Recursive Approach to Solving Parity Games in Quasipolynomial Time.
- See also: Jurdziński M., Morvan R., Thejaswini K . S.: Universal Algorithms for Parity Games and Nested Fixpoints

Final remarks on universal trees

Unifying view

Many quasi-polynomial parity-game algorithms are really using some universal tree, explicitly or implicitly.

approach	where the tree appears
ordered progress measures	implicitly, by Czerwiński et al.
succinct progress measures	compact encoding of universal-tree paths
recursive algorithms	recursive calls arranged like two interleaved universal trees

Bottom line

The quasi-polynomial improvement comes from not exploring all exponential progress measures, but only the leaves of a quasi-polynomial universal tree.

Open questions: can we find implicit universal trees in other algorithms, such as strategy improvement or tangle learning?

Beyond standard tangle learning

Basic tangle learning: recap

Tangle learning (two-player basic variation)

- Partition the game into regions using tangle attractors
  - attract to the highest priority;
  - repeat with the remainder until nothing is left.

Basic tangle learning: recap

Tangle learning (two-player basic variation)

- Partition the game into regions using tangle attractors
  - attract to the highest priority;
  - repeat with the remainder until nothing is left.
- Every locally closed region contains a new tangle
  - locally closed: the top vertex is attracted to the region;
  - run Tarjan's SCC algorithm from the top vertex, restricted to the attractor strategy;
  - the resulting SCC is the new tangle.

Basic tangle learning: recap

Tangle learning (two-player basic variation)

- Partition the game into regions using tangle attractors
  - attract to the highest priority;
  - repeat with the remainder until nothing is left.
- Every locally closed region contains a new tangle
  - locally closed: the top vertex is attracted to the region;
  - run Tarjan's SCC algorithm from the top vertex, restricted to the attractor strategy;
  - the resulting SCC is the new tangle.
- Every closed tangle is a winning region (dominion)
  - after finding a dominion, maximize it with another attractor;
  - remove maximized dominions from the game.

Basic tangle learning: recap

Tangle learning (two-player basic variation)

- Partition the game into regions using tangle attractors
  - attract to the highest priority;
  - repeat with the remainder until nothing is left.
- Every locally closed region contains a new tangle
  - locally closed: the top vertex is attracted to the region;
  - run Tarjan's SCC algorithm from the top vertex, restricted to the attractor strategy;
  - the resulting SCC is the new tangle.
- Every closed tangle is a winning region (dominion)
  - after finding a dominion, maximize it with another attractor;
  - remove maximized dominions from the game.
- But: Two Binary Counters still require exponential time
  - the strict top-down decomposition only ignores distractions after learning an opponent tangle that attracts them.

Taking a step back

The essential ideas are:

- learn locally solved subgames: tangles;
- use learned tangles to extend attractor computation;
- avoid re-solving a subgame by reusing the structure in later decompositions.

Taking a step back

The essential ideas are:

- learn locally solved subgames: tangles;
- use learned tangles to extend attractor computation;
- avoid re-solving a subgame by reusing the structure in later decompositions.

The strict top-down decomposition is not a requirement:

$$\text{highest priority} \rightsquigarrow \text{next highest priority in the remainder} \rightsquigarrow \cdots$$

Taking a step back

The essential ideas are:

- learn locally solved subgames: tangles;
- use learned tangles to extend attractor computation;
- avoid re-solving a subgame by reusing the structure in later decompositions.

The strict top-down decomposition is not a requirement:

$$\text{highest priority} \rightsquigarrow \text{next highest priority in the remainder} \rightsquigarrow \cdots$$

We actually have a lot of flexibility:

- use any sequence of target vertices $\langle v_1, v_2, \dots \rangle$ for the decomposition
  - ignore target vertices we don't want
  - change the order of vertices
  - take note: when attracting to v_i, attract only through vertices of priority at most $\mathrm{pr}(v_i)$.
- only attract some vertices, for example with certain priorities or from certain subgames
- add tangle attractors to other algorithms

Four approaches beyond basic TL

approach	change to the decomposition / solver
RTL	recursively decompose open regions
DFTL	mark unproductive attractor targets and skip them
PMTL	lift progress measures toward tangle-attractor region tops
QTL	add tangle memoization to QP recursive attractor solving

Recursive tangle learning

Main idea

Recursive tangle learning keeps the basic TL idea, but also decomposes open regions recursively.

- First decompose the game into regions using tangle attractors.
  - attract lower vertices/tangles to a target vertex;
  - repeat with the remainder until nothing is left.

Recursive tangle learning

Main idea

Recursive tangle learning keeps the basic TL idea, but also decomposes open regions recursively.

- First decompose the game into regions using tangle attractors.
  - attract lower vertices/tangles to a target vertex;
  - repeat with the remainder until nothing is left.
- If a region is open, recursively decompose it:
  - now avoid the top vertex: remove the opponent attractor to the top vertex, inside the region;
  - partition the remainder of the region.

Recursive tangle learning

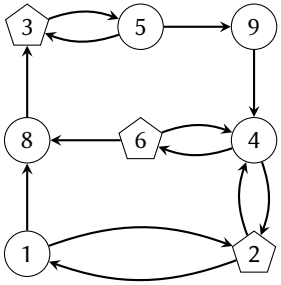
Main idea

Recursive tangle learning keeps the basic TL idea, but also decomposes open regions recursively.

- First decompose the game into regions using tangle attractors.
  - attract lower vertices/tangles to a target vertex;
  - repeat with the remainder until nothing is left.
- If a region is open, recursively decompose it:
  - now avoid the top vertex: remove the opponent attractor to the top vertex, inside the region;
  - partition the remainder of the region.
- Variants:
  - one-player RTL: only use Even targets, or only Odd targets.
  - two-player RTL: use both Even and Odd targets;

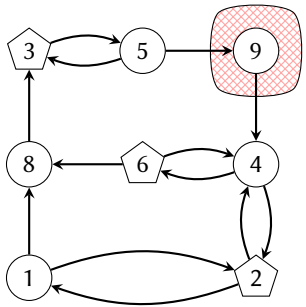
Example game with standard Tangle Learning

We solve the game with standard Two-player Tangle Learning:



Example game with standard Tangle Learning

We solve the game with standard Two-player Tangle Learning:

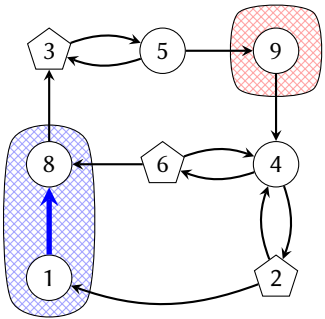


Regions:

- 9 (open)

Example game with standard Tangle Learning

We solve the game with standard Two-player Tangle Learning:

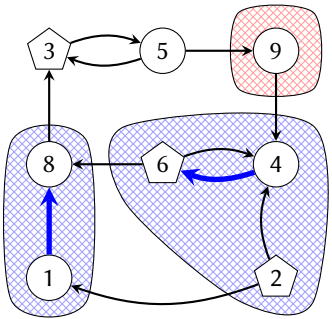


Regions:

- 9 (open)
- 8 1 (open)

Example game with standard Tangle Learning

We solve the game with standard Two-player Tangle Learning:



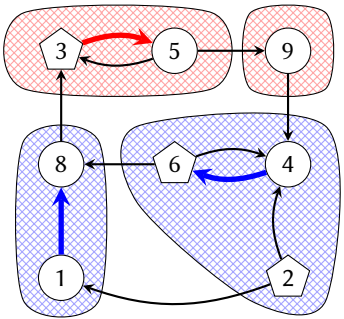
Regions:

- 9 (open)
- 8 1 (open)
- 6 4 2 (closed)

Learned tangles: $\{6, 4 \to 6\}$

Example game with standard Tangle Learning

We solve the game with standard Two-player Tangle Learning:



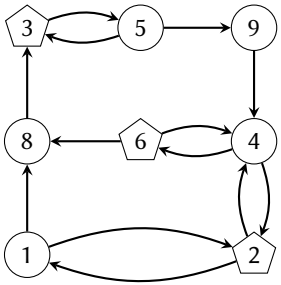
Regions:

- 9 (open)
- 8 1 (open)
- 6 4 2 (closed)
- 5 3 (closed)

Learned tangles: $\{6, 4 \rightarrow 6\}$, $\{5, 3 \rightarrow 5\}$

Example game with standard Tangle Learning

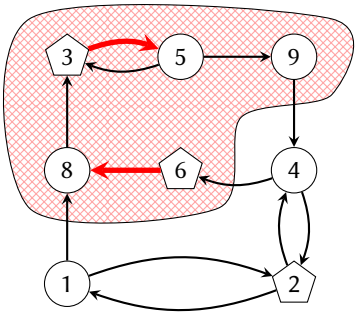
We solve the game with standard Two-player Tangle Learning:



Learned tangles: $\{6, 4 \to 6\}$, $\{5, 3 \to 5\}$

Example game with standard Tangle Learning

We solve the game with standard Two-player Tangle Learning:



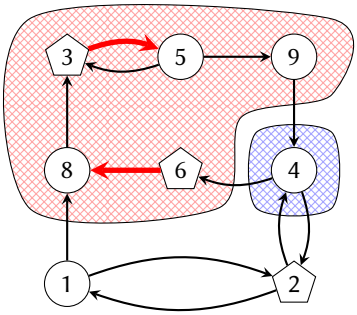
Regions:

- 9 5 3 8 6 (open)

Learned tangles: $\{6, 4 \to 6\}$, $\{5, 3 \to 5\}$

Example game with standard Tangle Learning

We solve the game with standard Two-player Tangle Learning:



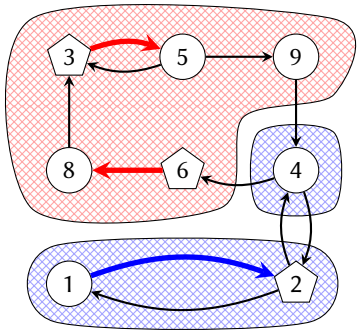
Regions:

- 9 5 3 8 6 (open)
- 4 (open)

Learned tangles: $\{6, 4 \to 6\}$, $\{5, 3 \to 5\}$

Example game with standard Tangle Learning

We solve the game with standard Two-player Tangle Learning:



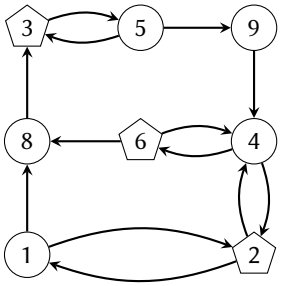
Regions:

- 9 5 3 8 6 (open)
- 4 (open)
- 2 1 (closed)

Learned tangles: $\{6, 4 \to 6\}$, $\{5, 3 \to 5\}$, $\{2, 1 \to 2\}$

Example game with standard Tangle Learning

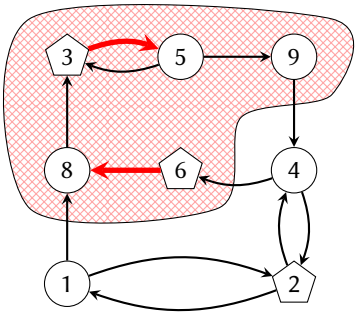
We solve the game with standard Two-player Tangle Learning:



Learned tangles: $\{6, 4 \to 6\}$, $\{5, 3 \to 5\}$, $\{2, 1 \to 2\}$

Example game with standard Tangle Learning

We solve the game with standard Two-player Tangle Learning:



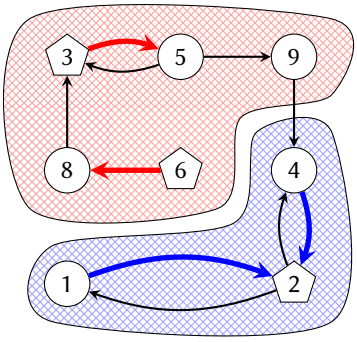
Regions:

- 9 5 3 8 6 (open)

Learned tangles: $\{6, 4 \to 6\}$, $\{5, 3 \to 5\}$, $\{2, 1 \to 2\}$

Example game with standard Tangle Learning

We solve the game with standard Two-player Tangle Learning:



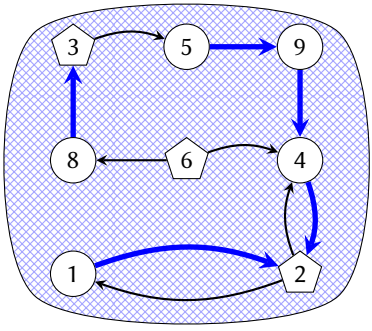
Regions:

- 9 5 3 8 6 (open)
- 4 2 1 (closed)

Learned tangles: $\{6, 4 \to 6\}$, $\{5, 3 \to 5\}$, $\{2, 1 \to 2\}$, $\underline{\{4 \to 2, 2, 1 \to 2\}}$

Example game with standard Tangle Learning

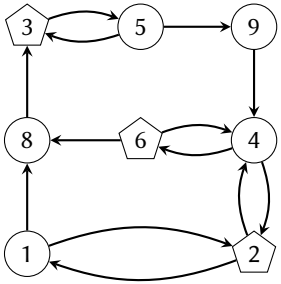
We solve the game with standard Two-player Tangle Learning:



Learned tangles: $\{6, 4 \to 6\}$, $\{5, 3 \to 5\}$, $\{2, 1 \to 2\}$, $\underline{\{4 \to 2, 2, 1 \to 2\}}$

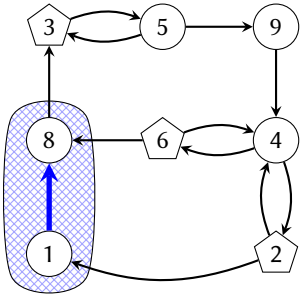
Example game with ORTL

Now we solve the game with One-player Recursive Tangle Learning (for Even):



Example game with ORTL

Now we solve the game with One-player Recursive Tangle Learning (for Even):

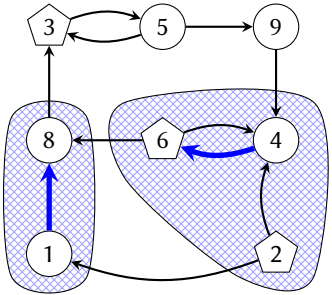


Regions:

- 8 1 (open)
  recursive: -

Example game with ORTL

Now we solve the game with One-player Recursive Tangle Learning (for Even):



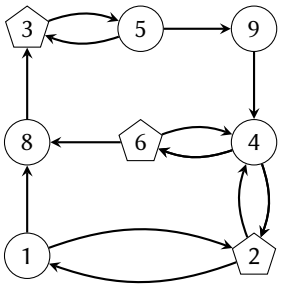
Regions:

- 8 1 (open)
  recursive: -
- 6 4 2 (closed)

Learned tangles: $\{6, 4 \rightarrow 6\}$

Example game with ORTL

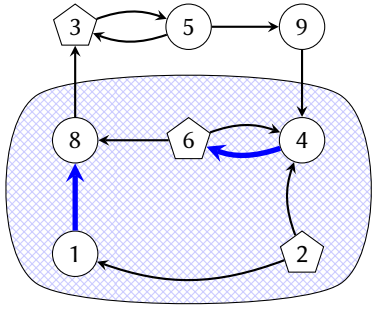
Now we solve the game with One-player Recursive Tangle Learning (for Even):



Learned tangles: $\{6, 4 \rightarrow 6\}$

Example game with ORTL

Now we solve the game with One-player Recursive Tangle Learning (for Even):



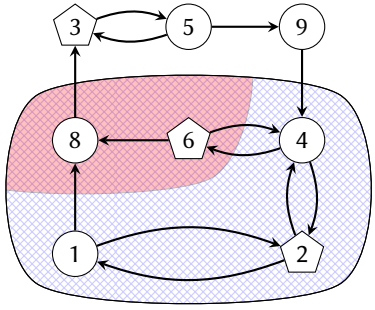
Regions:

- 8 6 4 1 2 (open)
  recursive: 4 2 1

Learned tangles: $\{6, 4 \rightarrow 6\}$

Example game with ORTL

Now we solve the game with One-player Recursive Tangle Learning (for Even):



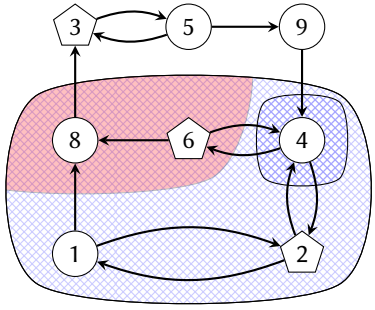
Regions:

- 8 6 4 1 2 (open)
  recursive: 4 2 1

Learned tangles: $\{6, 4 \rightarrow 6\}$

Example game with ORTL

Now we solve the game with One-player Recursive Tangle Learning (for Even):



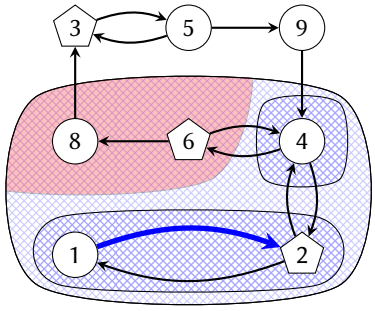
Regions:

- 8 6 4 1 2 (open)
  recursive: 4 2 1
- 4 (open)
  recursive: -

Learned tangles: $\{6, 4 \rightarrow 6\}$

Example game with ORTL

Now we solve the game with One-player Recursive Tangle Learning (for Even):



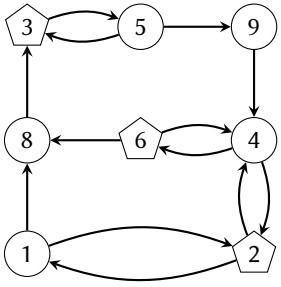
Regions:

- 8 6 4 1 2 (open)
  recursive: 4 2 1
- 4 (open)
  recursive: -
- 2 1 (closed)

Learned tangles: $\{6, 4 \rightarrow 6\}$, $\{2, 1 \rightarrow 2\}$

Example game with ORTL

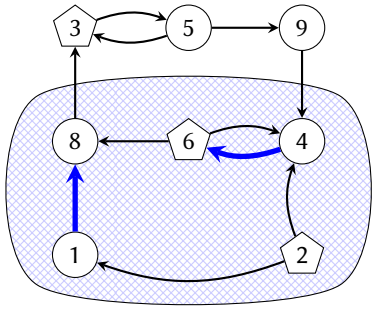
Now we solve the game with One-player Recursive Tangle Learning (for Even):



Learned tangles: $\{6, 4 \to 6\}$, $\{2, 1 \to 2\}$

Example game with ORTL

Now we solve the game with One-player Recursive Tangle Learning (for Even):



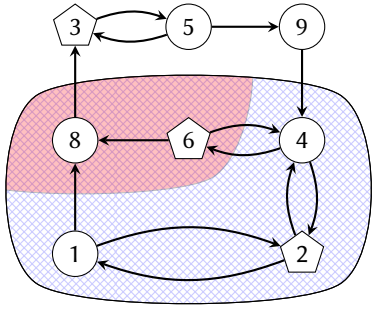
Regions:

- 8 6 4 2 1 (open)
  recursive: 4 2 1

Learned tangles: $\{6, 4 \rightarrow 6\}$, $\{2, 1 \rightarrow 2\}$

Example game with ORTL

Now we solve the game with One-player Recursive Tangle Learning (for Even):



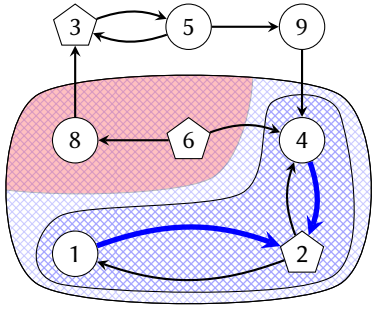
Regions:

- 8 6 4 2 1 (open)
  recursive: 4 2 1

Learned tangles: $\{6, 4 \rightarrow 6\}$, $\{2, 1 \rightarrow 2\}$

Example game with ORTL

Now we solve the game with One-player Recursive Tangle Learning (for Even):



Regions:

- 8 6 4 2 1 (open)
  recursive: 4 2 1
- 4 2 1 (closed)

Learned tangles: $\{6, 4 \to 6\}$, $\{2, 1 \to 2\}$, $\underline{\{4 \to 2, 2, 1 \to 2\}}$

Adversarial games for ORTL

Two Binary Counters and ORTL

One-player recursive tangle learning solves Two Binary Counters quickly.

- ORTL simply ignores the distractions.
- ORTL does not learn opponent tangles and does not need them.
- It immediately learns the highest bit, then the next highest, etc.
- It cannot learn “stable” lower tangles until the necessary higher bits are set.

Adversarial games for ORTL

Two Binary Counters and ORTL

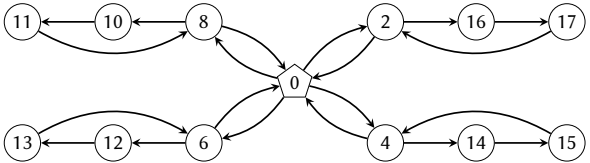
One-player recursive tangle learning solves Two Binary Counters quickly.

- ORTL simply ignores the distractions.
- ORTL does not learn opponent tangles and does not need them.
- It immediately learns the highest bit, then the next highest, etc.
- It cannot learn “stable” lower tangles until the necessary higher bits are set.

ORTL is vulnerable to a different (and very simple) trap:

- Pull distractions into the next higher region, so they distract again!

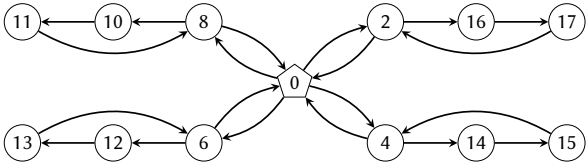
Recursive tangle learning



Regions:

Learned tangles:

Recursive tangle learning

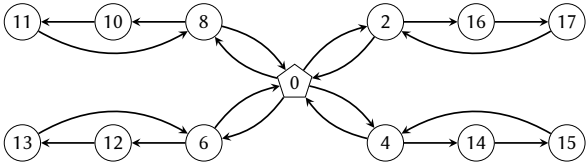


Regions:

- 16 2 (open, -)
- 14 4 (open, -)
- 12 6 (open, -)
- 10 8 0 (open, 8 0)
  - 8 0 (closed)

Learned tangles:
$\{8, 0\}$

Recursive tangle learning

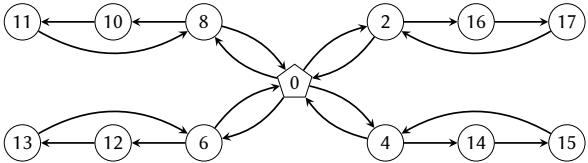


Regions:

- 16 2 (open, -)
- 14 4 (open, -)
- 12 6 8 0 11 10 (open, 11 10 8 6 0)
  - 10 8 (open, -)
  - 6 0 (closed)

Learned tangles:
$\{8, 0\}$, $\{6, 0\}$

Recursive tangle learning

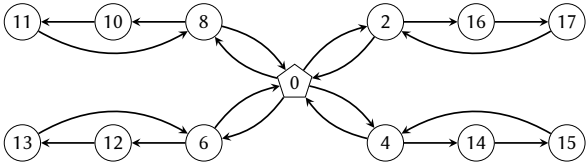


Regions:

- 16 2 (open, -)
- 14 4 (open, -)
- 12 6 8 0 11 10 (open, 11 10 8 6 0)
  - 10 8 6 0 (open, 8 6 0)
    - 8 6 0 (closed)

Learned tangles:
$\{8, 0\}$, $\{6, 0\}$, $\{8, 6, 0\}$

Recursive tangle learning

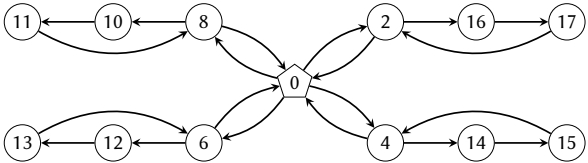


Regions:

- 16 2 (open)
- 14 4 8 6 0 11 10 13 12 (open)
  - 12 6 (open)
  - 10 8 (open)
  - 4 0 (closed)

Learned tangles:
$\{8, 0\}$, $\{6, 0\}$, $\{8, 6, 0\}$, $\{4, 0\}$

Recursive tangle learning

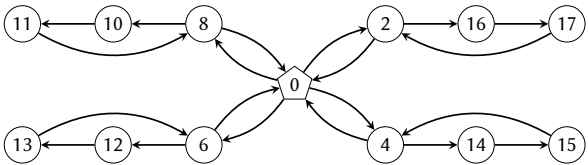


Regions:

- 16 2 (open)
- 14 4 8 6 0 11 10 13 12 (open)
  - 12 6 (open)
  - 10 8 4 0 (open)
    - 8 4 0 (closed)

Learned tangles:
$\{8, 0\}$, $\{6, 0\}$, $\{8, 6, 0\}$, $\{4, 0\}$, $\{8, 4, 0\}$

Recursive tangle learning

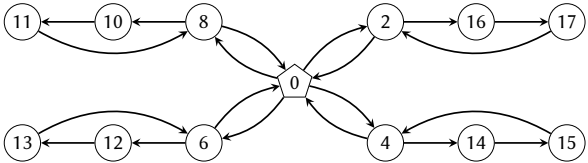


Regions:

- 16 2 (open)
- 14 4 8 6 0 11 10 13 12 (open)
  - 12 6 8 4 0 11 10 (open)
    - 10 8
    - 6 4 0 (closed)

Learned tangles:
$\{8, 0\}$, $\{6, 0\}$, $\{8, 6, 0\}$, $\{4, 0\}$, $\{8, 4, 0\}$, $\{6, 4, 0\}$

Recursive tangle learning



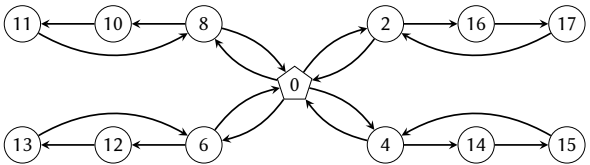
Regions:

- 16 2 (open)
- 14 4 8 6 0 11 10 13 12 (open)
  - 12 6 8 4 0 11 10 (open)
    - 10 8 6 4 0 (open)
      - ▹ 8 6 4 0 (closed)

Learned tangles:

$\{8, 0\}$, $\{6, 0\}$, $\{8, 6, 0\}$, $\{4, 0\}$, $\{8, 4, 0\}$, $\{6, 4, 0\}$, $\{8, 6, 4, 0\}$

Recursive tangle learning



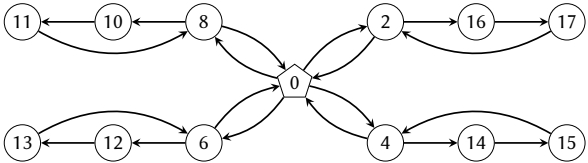
Regions:

- 16 2 8 6 4 0 11 10 13 12 15 14 (open)
  - 14 4 (open)
  - 12 6 (open)
  - 10 8 (open)
  - 2 0 (closed)

Learned tangles:

$\{8, 0\}$, $\{6, 0\}$, $\{8, 6, 0\}$, $\{4, 0\}$, $\{8, 4, 0\}$, $\{6, 4, 0\}$, $\{8, 6, 4, 0\}$, $\{2, 0\}$

Recursive tangle learning



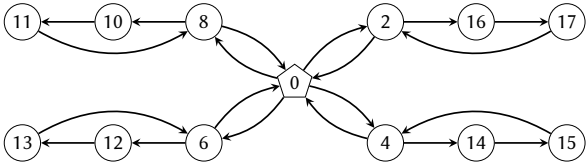
Regions:

- 16 2 8 6 4 0 11 10 13 12 15 14 (open)
  - 14 4 (open)
  - 12 6 (open)
  - 10 8 2 0 (open)
    - 8 2 0 (closed)

Learned tangles:

$\{8, 0\}$, $\{6, 0\}$, $\{8, 6, 0\}$, $\{4, 0\}$, $\{8, 4, 0\}$, $\{6, 4, 0\}$, $\{8, 6, 4, 0\}$, $\{2, 0\}$, $\{8, 2, 0\}$

Recursive tangle learning



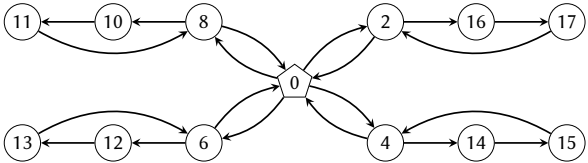
Regions:

- 16 2 8 6 4 0 11 10 13 12 15 14 (open)
  - 14 4 (open)
  - 12 6 8 2 0 11 10 (open)
    - 10 8 (open)
    - 6 2 0 (closed)

Learned tangles:

$\{8, 0\}$, $\{6, 0\}$, $\{8, 6, 0\}$, $\{4, 0\}$, $\{8, 4, 0\}$, $\{6, 4, 0\}$, $\{8, 6, 4, 0\}$, $\{2, 0\}$, $\{8, 2, 0\}$, $\{6, 2, 0\}$

Recursive tangle learning



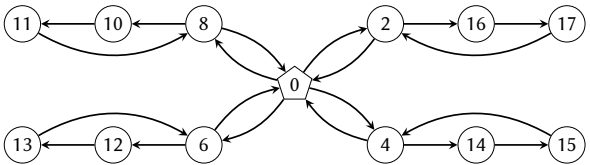
Regions:

- 16 2 8 6 4 0 11 10 13 12 15 14 (open)
  - 14 4 (open)
  - 12 6 8 2 0 11 10 (open)
    - 10 8 6 2 0 (open)
      - ▹ 8 6 2 0 (closed)

Learned tangles:

$\{8, 0\}$, $\{6, 0\}$, $\{8, 6, 0\}$, $\{4, 0\}$, $\{8, 4, 0\}$, $\{6, 4, 0\}$, $\{8, 6, 4, 0\}$, $\{2, 0\}$, $\{8, 2, 0\}$, $\{6, 2, 0\}$, $\{8, 6, 2, 0\}$

Recursive tangle learning



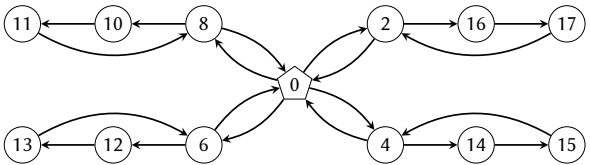
Regions:

- 16 2 8 6 4 0 11 10 13 12 15 14 (open)
  - 14 4 8 6 2 0 11 10 13 12 (open)
    - 12 6 (open)
    - 10 8 (open)
    - 4 2 0 (closed)

Learned tangles:

$\{8, 0\}$, $\{6, 0\}$, $\{8, 6, 0\}$, $\{4, 0\}$, $\{8, 4, 0\}$, $\{6, 4, 0\}$, $\{8, 6, 4, 0\}$, $\{2, 0\}$, $\{8, 2, 0\}$, $\{6, 2, 0\}$, $\{8, 6, 2, 0\}$, $\{4, 2, 0\}$

Recursive tangle learning



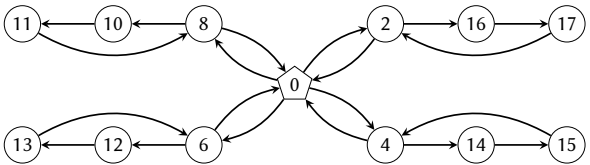
Regions:

- 16 2 8 6 4 0 11 10 13 12 15 14 (open)
  - 14 4 8 6 2 0 11 10 13 12 (open)
    - 12 6 (open)
    - 10 8 4 2 0 (open)
      - ▹ 8 4 2 0 (closed)

Learned tangles:

$\{8,0\}$, $\{6,0\}$, $\{8,6,0\}$, $\{4,0\}$, $\{8,4,0\}$, $\{6,4,0\}$, $\{8,6,4,0\}$, $\{2,0\}$, $\{8,2,0\}$, $\{6,2,0\}$, $\{8,6,2,0\}$, $\{4,2,0\}$, $\{8,4,2,0\}$

Recursive tangle learning



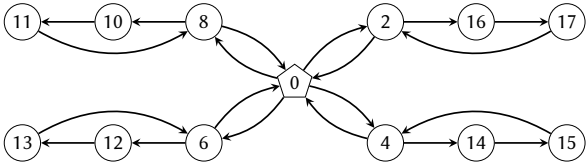
Regions:

- 16 2 8 6 4 0 11 10 13 12 15 14 (open)
  - 14 4 8 6 2 0 11 10 13 12 (open)
    - 12 6 8 4 2 0 11 10 (open)
      - ▹ 10 8 (open)
      - ▹ 6 4 2 0 (closed)

Learned tangles:

$\{8,0\}$, $\{6,0\}$, $\{8,6,0\}$, $\{4,0\}$, $\{8,4,0\}$, $\{6,4,0\}$, $\{8,6,4,0\}$, $\{2,0\}$, $\{8,2,0\}$, $\{6,2,0\}$, $\{8,6,2,0\}$, $\{4,2,0\}$, $\{8,4,2,0\}$, $\{6,4,2,0\}$

Recursive tangle learning



Regions:

- 16 2 8 6 4 0 11 10 13 12 15 14 (open)
  - 14 4 8 6 2 0 11 10 13 12 (open)
    - 12 6 8 4 2 0 11 10 (open)
      - ▹ 10 8 6 4 2 0 (open)
        - ▹ 8 6 4 2 0 (closed)

Learned tangles:

$\{8,0\}$, $\{6,0\}$, $\{8,6,0\}$, $\{4,0\}$, $\{8,4,0\}$, $\{6,4,0\}$, $\{8,6,4,0\}$, $\{2,0\}$, $\{8,2,0\}$, $\{6,2,0\}$, $\{8,6,2,0\}$, $\{4,2,0\}$, $\{8,4,2,0\}$, $\{6,4,2,0\}$, $\underline{\{8,6,4,2,0\}}$

What about two-player Recursive Tangle Learning?

Two-player RTL also solves Two Binary Counters quickly.

What about two-player Recursive Tangle Learning?

Two-player RTL also solves Two Binary Counters quickly.

RTL is harder to attack

Two-player RTL combines two ways of dealing with distractions:

- opponent attraction: distractions are neutralized when they are attracted to opponent tangles;
- recursive decomposition: distractions can also be avoided by decomposing open regions further.

What about two-player Recursive Tangle Learning?

Two-player RTL also solves Two Binary Counters quickly.

RTL is harder to attack

Two-player RTL combines two ways of dealing with distractions:

- opponent attraction: distractions are neutralized when they are attracted to opponent tangles;
- recursive decomposition: distractions can also be avoided by decomposing open regions further.

An adversarial game for RTL therefore needs to delay high bits to protect the counter.

What about two-player Recursive Tangle Learning?

Two-player RTL also solves Two Binary Counters quickly.

RTL is harder to attack

Two-player RTL combines two ways of dealing with distractions:

- opponent attraction: distractions are neutralized when they are attracted to opponent tangles;
- recursive decomposition: distractions can also be avoided by decomposing open regions further.

An adversarial game for RTL therefore needs to delay high bits to protect the counter.

Solution: add anti-ORTL "flowers" to each bit:

- ORTL-like flowers delay finding the higher-bit tangles;
- the Two Binary Counters bit structure resets the lower bits.

Current status: works against RTL and TL and *most* other attractor-based algorithms, but not robust.

Distraction-free tangle learning

Base algorithm: single-player tangle learning

- Run tangle learning using only Even targets, and learn only Even tangles.
- Continue until no new Even tangles are learned.

Distraction-free tangle learning

Base algorithm: single-player tangle learning

- Run tangle learning using only Even targets, and learn only Even tangles.
- Continue until no new Even tangles are learned.

What does it mean if an Even pass stalls?

Suppose no new Even tangle is learned.

- every Even region is open;

Distraction-free tangle learning

Base algorithm: single-player tangle learning

- Run tangle learning using only Even targets, and learn only Even tangles.
- Continue until no new Even tangles are learned.

What does it mean if an Even pass stalls?

Suppose no new Even tangle is learned.

- every Even region is open;
- so every Even target is attracted away to lower regions;

Distraction-free tangle learning

Base algorithm: single-player tangle learning

- Run tangle learning using only Even targets, and learn only Even tangles.
- Continue until no new Even tangles are learned.

What does it mean if an Even pass stalls?

Suppose no new Even tangle is learned.

- every Even region is open;
- so every Even target is attracted away to lower regions;
- some of these targets may be distractions that hide useful tangles in their region;

Distraction-free tangle learning

Base algorithm: single-player tangle learning

- Run tangle learning using only Even targets, and learn only Even tangles.
- Continue until no new Even tangles are learned.

What does it mean if an Even pass stalls?

Suppose no new Even tangle is learned.

- every Even region is open;
- so every Even target is attracted away to lower regions;
- some of these targets may be distractions that hide useful tangles in their region;
- which target should we ignore?

Distraction-free tangle learning

We should ignore the lowest open target.

Distraction-free tangle learning

We should ignore the lowest open target.

Higher targets can still be useful

- A higher Even region may be an escape for a tangle hidden in a lower Even region.
- Marking it too early could skip useful progress.

Distraction-free tangle learning

We should ignore the lowest open target.

Higher targets can still be useful

- A higher Even region may be an escape for a tangle hidden in a lower Even region.
- Marking it too early could skip useful progress.

The lowest target is different

- The lowest target escapes only to non-Even regions.
- So it cannot produce an Even tangle until the game changes: some vertices currently in Odd regions must first be attracted to Even regions.

Distraction-free tangle learning

Safe choice

Mark the lowest open target as unproductive and skip it in later decompositions.

Distraction-free tangle learning

Safe choice

Mark the lowest open target as unproductive and skip it in later decompositions.

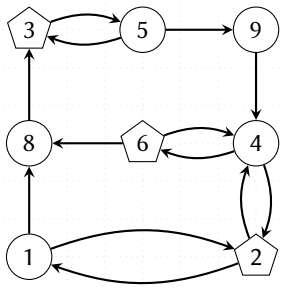
One-player distraction-free tangle learning (Even)

Maintain a set of unproductive vertices (initially empty)

1. decompose the game using unmarked Even targets;
2. learn Even tangles from closed regions;
3. if no Even tangle is learned, mark the lowest remaining Even target as unproductive;
4. if all Even targets are marked, unmark all targets;
5. repeat;

Distraction-free tangle learning

Now we solve the game with One-player Distraction-free Tangle Learning (for Even):

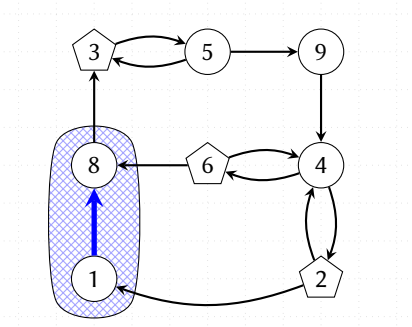


Targets: 8 6 4 2

Regions:

Distraction-free tangle learning

Now we solve the game with One-player Distraction-free Tangle Learning (for Even):



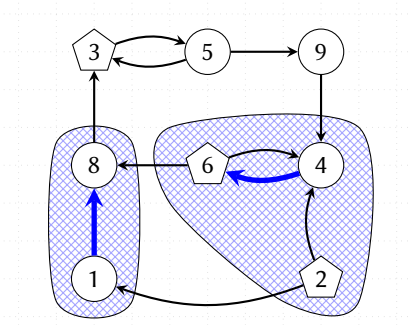
Targets: 8 6 4 2

Regions:

- 8 1 (open)

Distraction-free tangle learning

Now we solve the game with One-player Distraction-free Tangle Learning (for Even):



Targets: 8 6 4 2

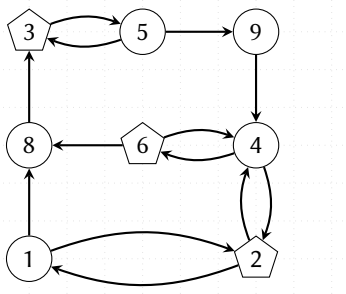
Regions:

- 8 1 (open)
- 6 4 2 (closed)

Learned tangles: $\{6, 4 \rightarrow 6\}$

Distraction-free tangle learning

Now we solve the game with One-player Distraction-free Tangle Learning (for Even):



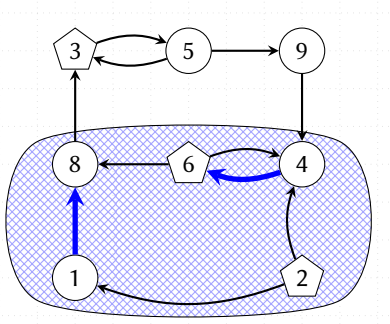
Targets: 8 6 4 2

Regions:

Learned tangles: $\{6, 4 \rightarrow 6\}$

Distraction-free tangle learning

Now we solve the game with One-player Distraction-free Tangle Learning (for Even):



Targets: 8 6 4 2

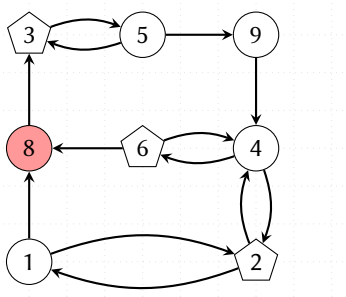
Regions:

- 8 6 4 1 2 (open)
  remove target 8

Learned tangles: $\{6, 4 \to 6\}$

Distraction-free tangle learning

Now we solve the game with One-player Distraction-free Tangle Learning (for Even):



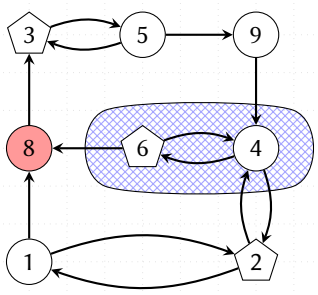
Targets: 6 4 2

Regions:

Learned tangles: $\{6, 4 \rightarrow 6\}$

Distraction-free tangle learning

Now we solve the game with One-player Distraction-free Tangle Learning (for Even):



Targets: 6 4 2

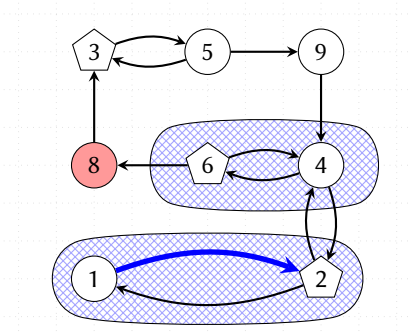
Regions:

- 6 4 (open)

Learned tangles: $\{6, 4 \to 6\}$

Distraction-free tangle learning

Now we solve the game with One-player Distraction-free Tangle Learning (for Even):



Targets: 6 4 2

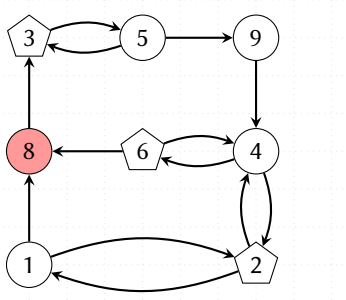
Regions:

- 6 4 (open)
- 2 1 (closed)

Learned tangles: $\{6, 4 \to 6\}$, $\{2, 1 \to 2\}$

Distraction-free tangle learning

Now we solve the game with One-player Distraction-free Tangle Learning (for Even):



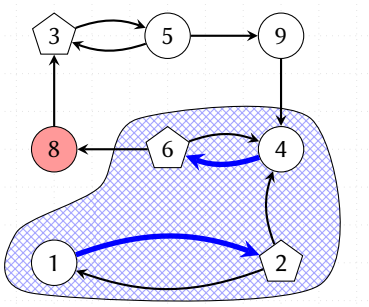
Targets: 6 4 2

Regions:

Learned tangles: $\{6, 4 \to 6\}$, $\{2, 1 \to 2\}$

Distraction-free tangle learning

Now we solve the game with One-player Distraction-free Tangle Learning (for Even):



Targets: 6 4 2

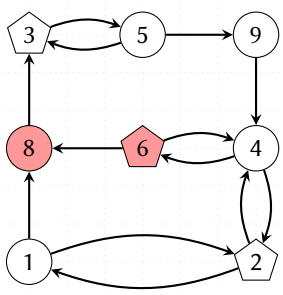
Regions:

- 6 4 2 1 (open)
  remove target 6

Learned tangles: $\{6, 4 \to 6\}$, $\{2, 1 \to 2\}$

Distraction-free tangle learning

Now we solve the game with One-player Distraction-free Tangle Learning (for Even):



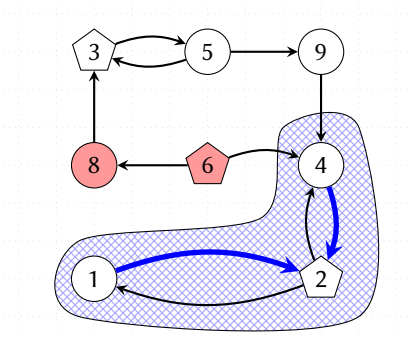
Targets: 4 2

Regions:

Learned tangles: $\{6, 4 \to 6\}$, $\{2, 1 \to 2\}$

Distraction-free tangle learning

Now we solve the game with One-player Distraction-free Tangle Learning (for Even):



Targets: 4 2

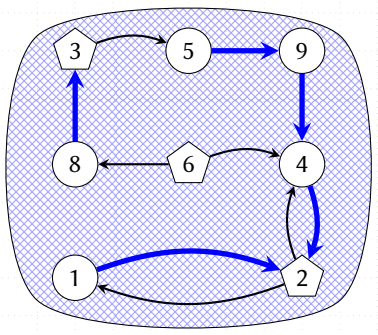
Regions:

- 4 2 1 (closed)

Learned tangles: $\{6, 4 \to 6\}$, $\{2, 1 \to 2\}$, $\underline{\{4 \to 2, 2, 1 \to 2\}}$

Distraction-free tangle learning

Now we solve the game with One-player Distraction-free Tangle Learning (for Even):

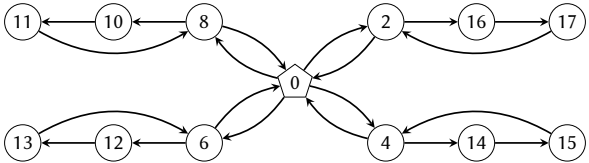


Targets: 4 2

Regions:

Learned tangles: $\{6, 4 \to 6\}$, $\{2, 1 \to 2\}$, $\underline{\{4 \to 2, 2, 1 \to 2\}}$

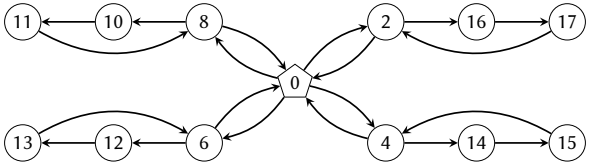
Distraction-free tangle learning



Regions:

Learned tangles:

Distraction-free tangle learning

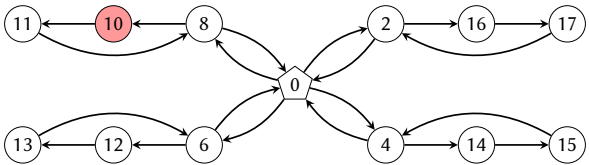


Regions:

- 16 2 (open)
- 14 4 (open)
- 12 6 (open)
- 10 8 0 (open, remove 10)

Learned tangles:

Distraction-free tangle learning



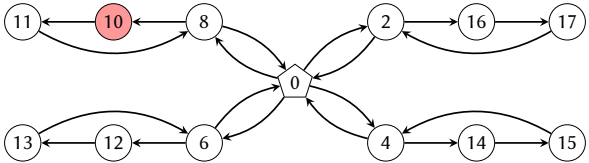
Regions:

- 16 2 (open)
- 14 4 (open)
- 12 6 (open)
- 8 0 (closed)

Learned tangles:

$\{8, 0\}$

Distraction-free tangle learning



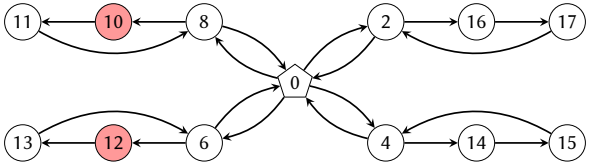
Regions:

- 16 2 (open)
- 14 4 (open)
- 12 6 8 0 11 10 (open, remove 12)

Learned tangles:

$\{8, 0\}$

Distraction-free tangle learning



Regions:

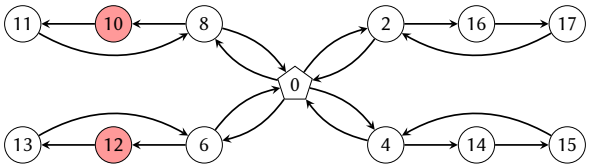
- 16 2 (open)
- 14 4 (open)
- 8 (open)
- 6 0 (closed)

Learned tangles:

$\{8, 0\}$

$\{6, 0\}$

Distraction-free tangle learning



Regions:

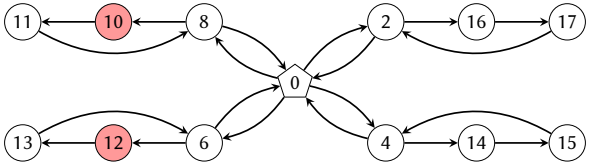
- 16 2 (open)
- 14 4 (open)
- 8 6 0 (closed)

Learned tangles:

$\{8, 0\}$

$\{6, 0\}$, $\{8, 6, 0\}$

Distraction-free tangle learning



Regions:

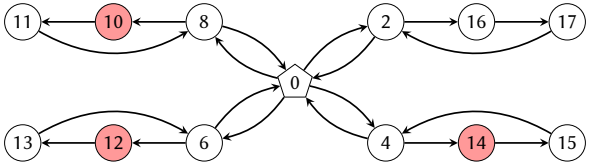
- 16 2 (open)
- 14 4 8 6 0 13 12 11 10 (open, remove 14)

Learned tangles:

$\{8, 0\}$

$\{6, 0\}$, $\{8, 6, 0\}$

Distraction-free tangle learning



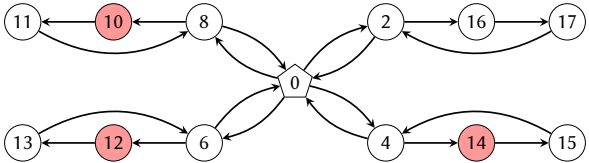
Regions:

- 16 2 (open)
- 8 (open)
- 6 (open)
- 4 0 (closed)

Learned tangles:

$\{8, 0\}$
$\{6, 0\}$, $\{8, 6, 0\}$
$\{4, 0\}$

Distraction-free tangle learning



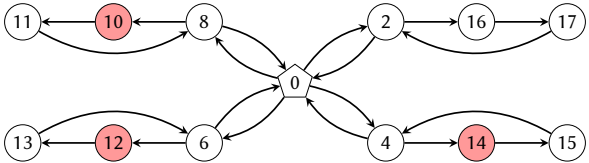
Regions:

- 16 2 (open)
- 8 (open)
- 6 4 0 (closed)

Learned tangles:

$\{8, 0\}$
$\{6, 0\}$, $\{8, 6, 0\}$
$\{4, 0\}$, $\{6, 4, 0\}$

Distraction-free tangle learning



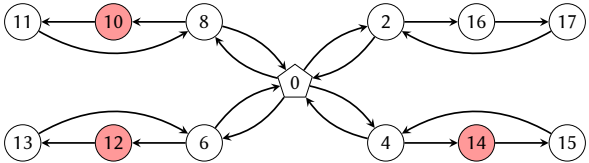
Regions:

- 16 2 (open)
- 8 6 4 0 (closed)

Learned tangles:

$\{8, 0\}$
$\{6, 0\}$, $\{8, 6, 0\}$
$\{4, 0\}$, $\{6, 4, 0\}$, $\{8, 6, 4, 0\}$

Distraction-free tangle learning



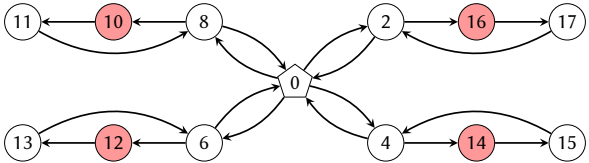
Regions:

- 16 2 8 6 4 0 15 14 13 12 11 (open, remove 16)

Learned tangles:

$\{8, 0\}$
$\{6, 0\}$, $\{8, 6, 0\}$
$\{4, 0\}$, $\{6, 4, 0\}$, $\{8, 6, 4, 0\}$

Distraction-free tangle learning



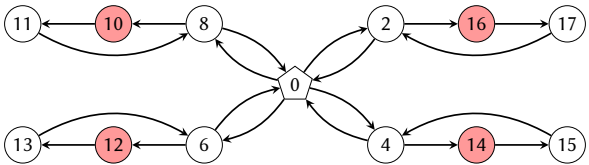
Regions:

- 8 (open)
- 6 (open)
- 4 (open)
- 2 0 (closed)

Learned tangles:

$\{8, 0\}$
$\{6, 0\}$, $\{8, 6, 0\}$
$\{4, 0\}$, $\{6, 4, 0\}$, $\{8, 6, 4, 0\}$
$\{2, 0\}$

Distraction-free tangle learning



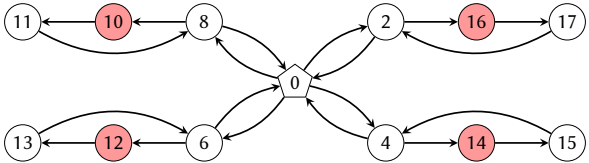
Regions:

- 8 (open)
- 6 (open)
- 4 2 0 (closed)

Learned tangles:

$\{8, 0\}$
$\{6, 0\}$, $\{8, 6, 0\}$
$\{4, 0\}$, $\{6, 4, 0\}$, $\{8, 6, 4, 0\}$
$\{2, 0\}$, $\{4, 2, 0\}$

Distraction-free tangle learning



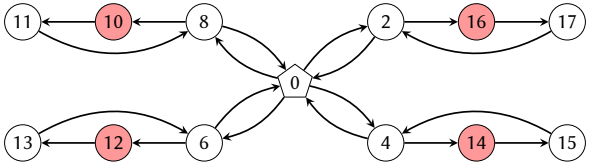
Regions:

- 8 (open)
- 6 4 2 0 (closed)

Learned tangles:

$\{8, 0\}$
$\{6, 0\}$, $\{8, 6, 0\}$
$\{4, 0\}$, $\{6, 4, 0\}$, $\{8, 6, 4, 0\}$
$\{2, 0\}$, $\{4, 2, 0\}$, $\{6, 4, 2, 0\}$

Distraction-free tangle learning



Regions:

- 8 6 4 2 0 (closed)

Learned tangles:

$\{8, 0\}$
$\{6, 0\}$, $\{8, 6, 0\}$
$\{4, 0\}$, $\{6, 4, 0\}$, $\{8, 6, 4, 0\}$
$\{2, 0\}$, $\{4, 2, 0\}$, $\{6, 4, 2, 0\}$, $\underline{\{8, 6, 4, 2, 0\}}$

One-player distraction-free tangle learning

One-player DFTL

There is a counterexample, but it is more involved than the counterexample to ORTL.

The construction must force a useful target to be marked unproductive before the true distractions are marked. It uses two anti-ORTL flowers, a high opponent-priority fence, and an extra distraction for the first flower.

The key is to mark the second flower as unproductive before making real progress on the first flower, as the first flower would attract the fence.

Distraction-free tangle learning

Base algorithm: alternating tangle learning

- Run tangle learning and learn only tangles from Even regions.
- Continue until no new Even tangles are learned.
- Then do the same for Odd.

Two-player distraction-free tangle learning

Maintain a set of unproductive targets, initially empty.
For the current player α:

1. decompose the game using unmarked targets (of both players);
2. learn α-tangles from closed α-regions;
3. if no α-tangle is learned, mark the lowest remaining α-target as unproductive;
4. if all targets (of both players) are marked, unmark all targets;
5. switch player and repeat.

Distraction-free tangle learning: status

Two-player DFTL

No robust adversarial family is known yet.

Research question

Can we construct polynomial-sized games that require exponentially many epochs?

PMTL: Progress measures over tangle attractors

Standard progress measures lift over one edge:

$$u \to v \qquad \Rightarrow \qquad \mathsf{pm}[u] \leftarrow \mathrm{Prog}(\mathsf{pm}[v], \mathsf{pr}(u)).$$

PMTL: Progress measures over tangle attractors

Standard progress measures lift over one edge:

$$u \to v \qquad \Rightarrow \qquad \mathrm{pm}[u] \leftarrow \mathrm{Prog}(\mathrm{pm}[v], \mathrm{pr}(u)).$$

Lift across attractors

Let v be a vertex with priority $p = \mathrm{pr}(v)$. Attract to v only through priorities

$$P_v = \{\, q \mid q \le p \text{ or } q \equiv_2 \alpha \,\}.$$

Every attracted vertex is forced to play to v via paths that avoid higher opponent priorities.

PMTL: Progress measures over tangle attractors

Standard progress measures lift over one edge:

$$u \to v \qquad \Rightarrow \qquad \mathrm{pm}[u] \leftarrow \mathrm{Prog}(\mathrm{pm}[v], \mathrm{pr}(u)).$$

Lift across attractors

Let v be a vertex with priority $p = \mathrm{pr}(v)$. Attract to v only through priorities

$$P_v = \{\, q \mid q \le p \text{ or } q \equiv_2 \alpha \,\}.$$

Every attracted vertex is forced to play to v via paths that avoid higher opponent priorities.

Propagating $\mathrm{pm}[v]$ backwards along such a path cannot decrease it.

$$u \rightsquigarrow v \qquad \Rightarrow \qquad \mathrm{pm}[u] \ge \mathrm{Prog}(\mathrm{pm}[v], \mathrm{pr}(u)).$$

PMA: lifting towards attractor regions

Region decomposition

Sort vertices by current measure and priority. Then repeatedly:

- take the current highest-measure vertex v;
- compute a priority-restricted attractor to v;
- remove that region and continue.

Each region A is represented by its top vertex $\mathrm{top}(A) = v$.

After the decomposition, each vertex u chooses the highest region A it is directly attracted to.

$$\mathrm{target}(u) = \mathrm{top}(A) = v$$

Then lift u as if v were a direct successor:

$$pm[u] \leftarrow \mathrm{Prog}(pm[v], \mathrm{pr}(u)).$$

So one lift can skip over an entire attractor region.

PMTL: add tangle learning

PMA uses ordinary attractors. PMTL uses tangle attractors.

- When a region is locally closed, extract tangles from the region.
- Use tangles in later decompositions to create larger attractor sets: skip over tangles!
- Closed tangles [partially] solve the game!! Maximize closed tangles, then remove them.

PMTL

For player α:

1. compute an α-tangle-attractor decomposition, ordered by current measures;
2. if a locally closed α-region with an α-priority top is found, learn tangles;
3. if a learned tangle is closed in the full game, remove its attractor as a dominion.
4. for each vertex, determine to which region it is attracted and that region's top
5. lift each vertex toward its target;

Main difference from normal tangle learning

The decomposition order is driven by the current progress measures, instead of a fixed top-down priority order.

Two Binary Counters is trivially solved by breaking the strict top-down order, as distracted vertices quickly get a higher measure than the distractions.

Fixed-point semantics

PMA and PMTL compute the same progress-measure fixed point as the underlying progress-measure algorithm.

If the underlying measure domain is quasi-polynomial, then PMTL remains quasi-polynomial.

PMTL

Fixed-point semantics

PMA and PMTL compute the same progress-measure fixed point as the underlying progress-measure algorithm.

If the underlying measure domain is quasi-polynomial, then PMTL remains quasi-polynomial.

Empirical picture

PMTL substantially reduces the number of lifts and the run time on benchmarks.

PMTL

Fixed-point semantics

PMA and PMTL compute the same progress-measure fixed point as the underlying progress-measure algorithm.

If the underlying measure domain is quasi-polynomial, then PMTL remains quasi-polynomial.

Empirical picture

PMTL substantially reduces the number of lifts and the run time on benchmarks.

Open question

Can we construct an adversarial game family where PMTL is forced to explore the full progress-measure state space before learning a useful dominion?

Quasi-polynomial tangle learning

Parys-style algorithms give a quasi-polynomial recursive attractor structure.

Quasi-polynomial tangle learning

Parys-style algorithms give a quasi-polynomial recursive attractor structure.

QTL: add tangle learning to these recursive algorithms

- use tangle attractors instead of ordinary attractors
- if a region is locally closed, extract tangles
- closed tangles partially solve the game, remove and continue the recursive search

Quasi-polynomial tangle learning

Parys-style algorithms give a quasi-polynomial recursive attractor structure.

QTL: add tangle learning to these recursive algorithms

- use tangle attractors instead of ordinary attractors
- if a region is locally closed, extract tangles
- closed tangles partially solve the game, remove and continue the recursive search

The recursive structure gives the quasi-polynomial bound; the tangles improve the practical decomposition.

Research question

Can we delay tangle discovery long enough to force super-polynomial behaviour in practice?

Open directions

Adversarial games

Can we construct robust adversarial families for two-player DFTL, for PMTL, and for QTL?

Fundamental understanding of tangle learning

- What role do universal trees play in tangle learning?
- Can we characterize how much information one iteration or decomposition can learn?
- Can we prove a quasi-polynomial lower bound to tangle learning approaches?

New algorithms

Can we design a tangle-learning algorithm that is naturally quasi-polynomial, without importing the structure of existing quasi-polynomial algorithms like PMTL and QTL?

Smaller strategies

Can we design algorithms that yield smaller winning strategies, for applications like LTL synthesis?