These slides were created in part based on earlier presentations by Tim Willemse, Wolfgang Schreiner and Nathanael Fijalkow.
Overview

- Lecture I
  - Labeled Transition Systems, Kripke Structures
  - The CTL*, CTL and LTL languages
  - The difference between CTL and LTL
    - Different intuition: properties of all runs vs branching structure
    - Incomparable in expressiveness
  - How to express common properties in LTL
  - Fixed points and the modal $\mu$-calculus
  - Naive $\mu$-calculus model checking
  - Translation of $\mu$-calculus to parity games
Overview

• Lecture II
  • Concepts of “attractor computation”, “tangle”, “distraction”
  • Zielonka’s recursive algorithm
  • Priority promotion
  • Tangle learning

• Lecture III
  • Small progress measures algorithm
  • Universal trees and the succinct progress measures algorithm
  • “Ordered” progress measures algorithm

• Lecture IV
  • Strategy iteration
  • Fixed point iteration
1 Temporal logics CTL and LTL
2 The modal $\mu$-calculus
3 Parity games
4 Attractor based algorithms
5 Fixed point based algorithms
The behaviour of a system is modelled by a graph consisting of:

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

Information can be put in states or on transitions (or both):

- **Kripke Structures (KS)**
  Information on states, called **atomic propositions**
- **Labelled Transition Systems (LTS)**
  Information on edges, called **action labels**
Transition System $\mathcal{M} = \langle S, S_0, Act, R, L \rangle$ over set $AP$ of atomic propositions:

- $S$ is a set of states
- $S_0$ is a set of initial states (or $s_0$ is a single initial state)
- $Act$ is a set of action labels
- $R$ is a labelled transition relation: $R \subseteq S \times Act \times S$
- $L$ is a labelling: $L \in S \rightarrow 2^{AP}$

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

Special cases:

- Kripke Structures: $Act$ is a singleton (only one transition relation)
- Labelled Transition Systems: $AP$ is empty
We want to reason about transition systems, i.e., to specify system properties, behavior, etc.

- Reachability graph: starting from \( s_0 \), the system runs evolve
- Consider the reachability graph as an infinite computation tree
  - Different tree nodes may denote occurrences of the same state
  - Every path in this tree is infinite
  - Temporal logic CTL reasons about the computation tree
- Consider the reachability graph as a set of system runs
  - Same state may occur multiple times (in one or in different runs)
  - Temporal logic LTL reasons about each run
Computation Trees versus System Runs

Set of system runs:

\[
\begin{align*}
[a, b] &\rightarrow c \rightarrow c \rightarrow \ldots \\
[a, b] &\rightarrow [b, c] \rightarrow c \rightarrow \ldots \\
[a, b] &\rightarrow [b, c] \rightarrow [a, b] \rightarrow \ldots \\
[a, b] &\rightarrow [b, c] \rightarrow [a, b] \rightarrow \ldots \\
\ldots
\end{align*}
\]

Figure 3.1
Computation trees.

CTL* is the **Full** Computation Tree Logic

- CTL* formulae express properties over states or paths
- CTL* has the following **temporal operators**, which are used to express **properties of paths**: neXt, Future, Globally, Until, Weak Until, Strong Release (M), Release

\[
\begin{align*}
\text{X } f & \quad \text{f holds in the next state} \quad \text{also: } \bigcirc \\
\text{F } f & \quad \text{f holds somewhere (eventually)} \quad \text{also: } \lozenge \\
\text{G } f & \quad \text{f holds everywhere} \quad \text{also: } \Box \\
\text{f U g} & \quad \text{g holds eventually, and f in all preceding states} \\
\text{f W g} & \quad (\text{G } f) \lor (\text{f U g}) \\
\text{f M g} & \quad \text{g U (f \land g)} \\
\text{f R g} & \quad (\text{G } g) \lor (\text{f M g})
\end{align*}
\]

**Example**

\[
\text{F G p \ versus \ G F p: \ almost always \ versus \ infinitely often}
\]
Temporal Logics: CTL

\[
\begin{align*}
  & f \\
  & \text{X } f \\
  & \text{F } f \\
  & \text{G } f \\
  & \text{U } g \\
  & \text{W } g \\
  & \text{M } g \\
  & \text{R } g \\
\end{align*}
\]
Temporal Logics: CTL*

CTL* consists of:

- Atomic propositions \((AP)\)
- Boolean connectives: \(\neg\) (not), \(\lor\) (or), \(\land\) (and)
- Temporal operators (on paths)
- Path quantifiers (on states)

Path quantifiers are capable of expressing properties on a system’s branching structure:

- \(\forall f\): \(f\) holds for all paths from this state
- \(\exists f\): \(f\) holds for at least one path from this state
Temporal Logics: CTL and LTL

\[
\begin{align*}
E \ X \ & \text{black} \\
E \ G \ & \text{black} \\
A \ X \ & \text{black} \\
A \ G \ & \text{black} \\
E \ F \ & \text{black} \\
E \ \text{red} \ U \ & \text{black} \\
A \ F \ & \text{black} \\
A \ \text{red} \ U \ & \text{black}
\end{align*}
\]
CTL* state formulae ($S$) and path formulae ($\mathcal{P}$) are defined simultaneously by induction:

$$S ::= \text{true} \mid \text{false} \mid AP \mid \neg S \mid S \land S \mid S \lor S \mid E \mathcal{P} \mid A \mathcal{P}$$

$$\mathcal{P} ::= S \mid \neg \mathcal{P} \mid \mathcal{P} \land \mathcal{P} \mid \mathcal{P} \lor \mathcal{P} \mid X \mathcal{P} \mid F \mathcal{P} \mid G \mathcal{P} \mid \mathcal{P} \mathcal{U} \mathcal{P} \mid \mathcal{P} \mathcal{R} \mathcal{P} \mid \mathcal{P} \mathcal{W} \mathcal{P} \mid \mathcal{P} \mathcal{M} \mathcal{P}$$

Summarising:

- **State formulae ($S$) 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
The semantics of CTL\(^*\) state formulae and path formulae is defined relative to a fixed Kripke Structure \(\mathcal{M} = \langle S, S_0, R, L \rangle\) over \(AP\):

For state formulae:

\[
\begin{align*}
    s & \models true \\
    s & \not\models false \\
    s & \models p \quad \text{iff} \quad p \in L(s) \\
    s & \models \neg f \quad \text{iff} \quad s \not\models f \\
    s & \models f \land g \quad \text{iff} \quad s \models f \quad \text{and} \quad s \models g \\
    s & \models f \lor g \quad \text{iff} \quad s \models f \quad \text{or} \quad s \models g \\
    s & \models E f \quad \text{iff} \quad \exists \pi \in \text{path}(s) . \pi \models f \\
    s & \models A f \quad \text{iff} \quad \forall \pi \in \text{path}(s) . \pi \models f
\end{align*}
\]
The semantics of CTL* state formulae and path formulae is defined relative to a fixed Kripke Structure $\mathcal{M} = \langle S, S_0, R, L \rangle$ over $AP$:

For path formulae:

- $\pi \models f$ iff $\pi(0) \models f$ (if $f$ is a state formula)
- $\pi \models \neg f$ iff $\pi \not\models f$
- $\pi \models f \land g$ iff $\pi \models f$ and $\pi \models g$
- $\pi \models f \lor g$ iff $\pi \models f$ or $\pi \models g$
- $\pi \models X f$ iff $\pi^1 \models f$
- $\pi \models F f$ iff $\exists i . \pi^i \models f$
- $\pi \models G f$ iff $\forall i . \pi^i \models f$
- $\pi \models f U g$ iff $\exists i . \pi^i \models g$ and $\forall j < i . \pi^j \models f$
- $\pi \models f W g$ iff $\pi \models G f$ or $\pi \models f U g$
- $\pi \models f M g$ iff $\exists i . \pi^i \models g$ and $\forall j \leq i . \pi^j \models f$
- $\pi \models f R g$ iff $\pi \models G f$ or $\pi \models f M g$
Temporal Logics: CTL and LTL

Two simpler sublogics of CTL* are defined

CTL: Computation Tree Logic

\[ \phi, \psi ::= \text{true} | \neg \phi | AP | \phi \land \psi | \text{EX} \phi | \text{EG} \phi | \text{E}(\phi \text{ U } \psi) \]

(derived: false, \lor, EF, EW, EM, ER, AX, AG, AF, AU, AW, AM, AR)

CTL expressions: \text{AG EF } p, \text{ E } p \text{ U (E X q)};
syntactically not in CTL: \text{A F G } p, \text{ A X X } p, \text{ E(p U (X q))}

Question: \text{A X X } p \equiv \text{AX AX } p

LTL: Linear Time Logic

\[ \phi, \psi ::= \text{true} | \neg \phi | AP | \phi \land \psi | \text{X} \phi | (\phi \text{ U } \psi) \]

(derived: false, \lor, F, G, W, M, R)

LTL expressions: \text{F G } p, (\neg(G F p) \lor F q);
syntactically not in LTL: \text{A F A G } p, \text{ A G E F } p

Question: \text{A F G } p \equiv \text{A F A G } p
We use temporal logic to specify a formula $\phi$.

- **Model checking question**: $\mathcal{M} \models \phi$ ("$\phi$ holds in system $\mathcal{M}$").
- **Branching time logic (CTL)**
  - $\mathcal{M} \models \phi \iff \forall s_0 \in S_0 \cdot s_0 \models \phi$
  - $\phi$ is evaluated on the computation tree of $s_0$.
- **Linear time logic (LTL)**
  - $\mathcal{M} \models \phi \iff \pi \models \phi$ for every run $\pi$ of $\mathcal{M}$.
  - $\phi$ is evaluated on all paths of the computation tree originating in $s_0$. 
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 \( \text{AX} (\text{EX} \ Q \land \text{EX} \ \neg Q) \).
- 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: $\text{AG}(\text{EF } \phi)$.
  - “In every run, it is at any time still possible that later $\phi$ will hold”.
  - Property cannot be expressed by any LTL logic formula.

- LTL formula: $\Diamond \Box \phi$ (i.e. $\text{FG } \phi$).
  - “In every run, there is a moment from which on $\phi$ holds forever.”.
  - Naive translation $\text{AFG } \phi$ is not a CTL formula.
    - $\text{G } \phi$ is a path formula, but $\text{F}$ expects a state formula!
  - Translation $\text{AFAG } \phi$ 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.

Branching versus Linear Time Logic

Proof that $\text{AFAG } F$ (CTL) is different from $\Diamond \Box F$ (LTL).

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

In every run, there is a moment from which on $F$ holds forever.
Why using linear time logic (LTL) for system specifications?

- LTL has many advantages:
  - LTL formulas are easier to understand.
  - Reasoning about computation paths, not computation trees.
  - No explicit path quantifiers used.
  - LTL can express most interesting system properties.
    - Invariance, guarantee, response, ... (see later).
  - LTL can express fairness constraints (see later).
    - CTL cannot do this.
    - But CTL can express resettability (which LTL cannot).

- LTL has also some disadvantages:
  - LTL is strictly less expressive than other specification languages.
    - CTL* or \( \mu \)-calculus.
  - Asymptotic complexity of model checking is higher.
    - LTL: exponential in size of formula; CTL: linear in size of formula.
    - In practice the number of system states dominates the checking time.
In practice, most temporal formulas are instances of particular patterns.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Pronounced</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G \phi )</td>
<td>always ( \phi )</td>
<td>invariance</td>
</tr>
<tr>
<td>( F \phi )</td>
<td>eventually ( \phi )</td>
<td>guarantee</td>
</tr>
<tr>
<td>( G F \phi )</td>
<td>( \phi ) holds infinitely often</td>
<td>recurrence</td>
</tr>
<tr>
<td>( F G \phi )</td>
<td>eventually ( \phi ) holds permanently</td>
<td>stability</td>
</tr>
<tr>
<td>( G (\phi \Rightarrow F \psi) )</td>
<td>always, if ( \phi ) holds, then eventually ( \psi ) holds</td>
<td>response</td>
</tr>
<tr>
<td>( G (\phi \Rightarrow (\psi U \chi)) )</td>
<td>always, if ( \phi ) holds, then ( \psi ) holds until ( \chi ) holds</td>
<td>precedence</td>
</tr>
</tbody>
</table>

Typically, there are at most two levels of nesting of temporal operators.
Examples

• Mutual exclusion: $\mathbf{G} \neg (pc_1 = C \land pc_2 = C)$.
  • Alternatively: $\neg \mathbf{F} (pc_1 = C \land pc_2 = C)$.
  • Never both components are simultaneously in the critical region.

• No starvation: $\forall i : \mathbf{G} (pc_i = W \Rightarrow \Diamond pc_i = R)$.
  • Always, if component $i$ waits for a response, it eventually receives it.

• No deadlock: $\mathbf{G} \neg \forall i : pc_i = W$.
  • Never all components are simultaneously in a wait state $W$.

• Precedence: $\forall i : \mathbf{G} (pc_i \neq C \Rightarrow (pc_i \neq C \cup \text{lock} = i))$.
  • Always, if component $i$ is out of the critical region, it stays out until it receives the shared lock variable (which it eventually does).

• Partial correctness: $\mathbf{G} (pc = L \Rightarrow C)$.
  • Always if the program reaches line $L$, the condition $C$ holds.

• Termination: $\forall i : \mathbf{F} (pc_i = T)$.
  • Every component eventually terminates.
Example

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

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

• Next idea (correct): $G(a \Rightarrow ...)$

• First attempt (wrong): $G(a \Rightarrow (b \cup c))$
• Run $a, b, \neg b, c, ...$ is illegal.

• Second attempt (better): $G(a \Rightarrow (\neg c \cup b))$
• Run $a, \neg c, \neg c, \neg c, ...$ is illegal.

• Third attempt (correct): $G(a \Rightarrow (\neg c \wedge b))$

Think in terms of allowed/prohibited sequences.
If event $a$ occurs, then $b$ must occur before $c$ can occur (a run ..., $a, (¬b)^*, c, ...$ is illegal).

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

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

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

- Next idea (correct): $G (a \Rightarrow ...)$
If event $a$ occurs, then $b$ must occur before $c$ can occur ($a$ run ..., $a, (\neg b)^*, c, ...$ is illegal).

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

- **Next idea (correct):** $G (a \Rightarrow ...)$

- **First attempt (wrong):** $G (a \Rightarrow (b \mathbf{U} c))$
  - Run $a, b, \neg b, c, ...$ 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): $\mathbf{G} \ (a \Rightarrow (b \ U \ c))$
  - Run $a, b, \neg b, c, \ldots$ is illegal.
- Second attempt (better): $\mathbf{G} \ (a \Rightarrow (\neg c \ U \ b))$
  - 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 $...,a,(\neg b)^*,c,...$ is illegal).

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

- **Next idea (correct):** $\mathbf{G} (a \Rightarrow ...)$

- **First attempt (wrong):** $\mathbf{G} (a \Rightarrow (b \mathbf{U} c))$
  - Run $a,b,\neg b,c,...$ is illegal.

- **Second attempt (better):** $\mathbf{G} (a \Rightarrow (\neg c \mathbf{U} b))$
  - Run $a,\neg c,\neg c,\neg c,...$ 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 ..., $a, (\neg b)^*, c, ...$ is illegal).

- First idea (wrong): $a \Rightarrow ...$
  - Every run $d, ...$ becomes legal.
- Next idea (correct): $G (a \Rightarrow ...)$
- First attempt (wrong): $G (a \Rightarrow (b \bigcup c))$
  - Run $a, b, \neg b, c, ...$ is illegal.
- Second attempt (better): $G (a \Rightarrow (\neg c \bigcup b))$
  - Run $a, \neg c, \neg c, \neg c, ...$ is illegal.
- Third attempt (correct): $G (a \Rightarrow (\neg c \bigwedge b))$

Think in terms of allowed/prohibited sequences.
Basic LTL expansion laws (e.g. for unfolding)

\[ F \phi \equiv \phi \lor X (F \phi) \]
\[ G \phi \equiv \phi \land X (G \phi) \]
\[ \phi U \psi \equiv \psi \lor (\phi \land X (\phi U \psi)) \]
\[ \phi W \psi \equiv \psi \lor (\phi \land X (\phi W \psi)) \]
\[ \phi M \psi \equiv \psi \land (\phi \lor X (\phi M \psi)) \]
\[ \phi R \psi \equiv \psi \land (\phi \lor X (\phi R \psi)) \]

Notice the recursion
Basic LTL expansion laws (e.g. for unfolding)

\[
\begin{align*}
F \phi & \equiv \phi \lor X (F \phi) \\
G \phi & \equiv \phi \land X (G \phi) \\
\phi U \psi & \equiv \psi \lor (\phi \land X (\phi U \psi)) \\
\phi W \psi & \equiv \psi \lor (\phi \land X (\phi W \psi)) \\
\phi M \psi & \equiv \psi \land (\phi \lor X (\phi M \psi)) \\
\phi R \psi & \equiv \psi \land (\phi \lor X (\phi R \psi))
\end{align*}
\]

Notice the recursion

Think of \(F, G, U, W, M, R\) as specialized recursive operators. What if we could have more powerful (arbitrary) recursions?
Outline

1. Temporal logics CTL and LTL
2. The modal $\mu$-calculus
3. Parity games
4. Attractor based algorithms
5. Fixed point based algorithms
Background: Fixed-points

Tarski-Knaster theorem
A monotonic function $f : L \to L$ on a complete lattice $L$ has a greatest fixed point (gfp) and a least fixed point (lfp).

- **gfp** ($f$) = $\bigcup \{ x \in L \mid x \sqsubseteq f(x) \} = \bigcup \{ Ext(f) \} \in Fix(f)$
- **lfp** ($f$) = $\bigcap \{ x \in L \mid f(x) \sqsubseteq x \} = \bigcap \{ Red(f) \} \in Fix(f)$
Background: Fixed-points

Reductive
\[ f(x) \sqsubseteq x \]

Extensive
\[ x \sqsubseteq f(x) \]

Kleene fixed-point theorem
\[
\begin{align*}
gfp &= f^\infty(\top) = \bigsqcap_{n \geq 0} f^n(\top) \\
lfp &= f^\infty(\bot) = \bigsqcup_{n \geq 0} f^n(\bot)
\end{align*}
\]

\[
\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \ldots \sqsubseteq \lfp(f) \\
\sqsubseteq \gfp(f) \sqsubseteq \ldots \sqsubseteq f(f(\top)) \sqsubseteq f(\top) \sqsubseteq \top
\]
Idea of $\mu$-calculus: add fixed point operators to basic modal logic.

- $\mu$-calculus is **very** expressive (subsumes CTL, LTL, CTL*).
- $\mu$-calculus is very **pure** (“assembly language” for modal logic, cf: $\lambda$-calculus for functional programming).
- drawback: lack of intuition.
- fragments of the $\mu$-calculus are the basis for practical model checkers, such as $\mu$CRL, mCRL2, CADP, LTSmin
Some notation and terminology:

- The $\mu$-calculus introduces variables representing sets of states.
- An occurrence of $X$ is bound by a surrounding fixed point symbol $\mu X$ or $\nu X$. Unbound occurrences of $X$ are called free.
- A formula is closed if it has no free variables, otherwise it is called open.
- A valuation $\mathcal{V} : \text{Var} \rightarrow 2^S$ interprets the free variables as sets of states.
- $\mathcal{V}[X := Q]$ is a valuation like $\mathcal{V}$, but $X$ is set to $Q$.
- The semantics of a $\mu$-calculus formula $\phi$ is a set of states.
\( \mu\)-calculus: syntax and semantics

**Syntax**
\[
\phi, \psi ::= tt \mid ff \mid p \mid \neg p \mid \phi \land \psi \mid \phi \lor \psi \mid [a] \phi \mid \langle a \rangle \phi \mid X \mid \mu X. \phi \mid \nu X. \phi
\]

**Semantics**
\[
\begin{align*}
[tt] & = S \\
[ff] & = S \\
[p] & = \{ s \in S \mid p \in L(s) \} \\
[\neg p] & = \{ s \in S \mid p \notin L(s) \} \\
[\phi \lor \psi] & = [\phi] \cup [\psi] \\
[\phi \land \psi] & = [\phi] \cap [\psi]
\end{align*}
\]

(notice that there is no negation on formulae, only on the propositions)
$\mu$-calculus: syntax and semantics

Syntax

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

Semantics

$\llbracket tt \rrbracket^M = S$
$\llbracket ff \rrbracket^M = \emptyset$
$\llbracket p \rrbracket^M = \{ s \in S \mid p \in L(s) \}$
$\llbracket \neg p \rrbracket^M = \{ s \in S \mid p \notin L(s) \}$
$\llbracket \phi \lor \psi \rrbracket^M = \llbracket \phi \rrbracket^M \cup \llbracket \psi \rrbracket^M$
$\llbracket \phi \land \psi \rrbracket^M = \llbracket \phi \rrbracket^M \cap \llbracket \psi \rrbracket^M$
$\llbracket [a] \phi \rrbracket^M = \{ s \in S \mid \forall t. (s \xrightarrow{a} t) \rightarrow (t \in \llbracket \phi \rrbracket^M) \}$
$\llbracket \langle a \rangle \phi \rrbracket^M = \{ s \in S \mid \exists t. (s \xrightarrow{a} t) \land (t \in \llbracket \phi \rrbracket^M) \}$
Syntax
\[ \phi, \psi ::= tt \mid ff \mid p \mid \neg p \mid \phi \land \psi \mid \phi \lor \psi \mid [a] \phi \mid \langle a \rangle \phi \mid X \mid \mu X. \phi \mid \nu X. \phi \]

Semantics
\[ \begin{align*}
[tt]^M &= S \\
[ff]^M &= \emptyset \\
[p]^M &= \{ s \in S \mid p \in L(s) \} \\
[\neg p]^M &= \{ s \in S \mid p \notin L(s) \} \\
[\phi \lor \psi]^M &= [\phi]^M \cup [\psi]^M \\
[\phi \land \psi]^M &= [\phi]^M \cap [\psi]^M \\
[[a] \phi]^Y &= \{ s \in S \mid \forall t. (s \xrightarrow{a} t) \rightarrow (t \in [\phi]^Y) \} \\
[[\langle a \rangle] \phi]^Y &= \{ s \in S \mid \exists t. (s \xrightarrow{a} t) \land (t \in [\phi]^Y) \} \\
[X]^Y &= \mathcal{V}(X) \\
[\mu X. \phi]^M &= \bigwedge \{ S' \subseteq S \mid [\phi]^M_{[S'/X]} \subseteq S' \} \quad \text{(lfp)} \\
[\nu X. \phi]^M &= \bigvee \{ S' \subseteq S \mid S' \subseteq [\phi]^M_{[S'/X]} \} \quad \text{(gfp)}
\end{align*} \]

where \( \mathcal{V} : \text{Var} \rightarrow 2^S \) assigns a set of states to the variables \( X, Y, \ldots \)
\( \mu X.[a]X \) represent states with no infinite sequences of \( a \)-transitions

\[
\begin{align*}
\mu^0 X.[a]X & = \emptyset \quad \text{false} \\
\mu^1 X.[a]X & = [a]\emptyset \\
& = \{ s \in S \mid \forall t. \ s \xrightarrow{a} t \rightarrow t \models \emptyset \}
\end{align*}
\]

since no \( t \) satisfies \( \emptyset \), the right hand side (RHS) of \( \rightarrow \) is false; thus the left hand side (LHS) of \( \rightarrow \) cannot be true.

This represents states with no outgoing \( a \)-transitions

\[
\mu^2 X.[a]X = [a]T
\]

where \( T = \mu^1 X.[a]X \) are states with no outgoing \( a \)-transitions

Thus \( \mu^2 \) means states with no \( aa \)-paths
\( \nu X.p \land [a]X \) is informally analogous to LTL \( \mathbf{G} \ p \)

\[
\begin{align*}
\nu^0 X.p \land [a]X &= S = \text{true} \\
\nu^1 X.p \land [a]X &= p \land [a]S \\
\nu^2 X.p \land [a]X &= p \land [a]T
\end{align*}
\]

Intersection between all nodes satisfying \( p \) (LHS of \( \land \)) and all nodes (RHS of \( \land \))

Where \( T = \nu^1 X.p \land [a]X \) are all nodes that satisfy \( p \)
Thus \( \mu^2 \) is the intersection between all nodes that satisfy \( p \) and all nodes that have an outgoing edge labeled \( a \) to a node that satisfies \( p \)

All nodes that satisfy \( p \) and whose descendants that are reachable through \( a \)-transitions also satisfy \( p \).
\( \mu X.p \lor (\langle a \rangle True \land [a]X) \) is informally analogous to LTL \( F\ p \)

\[
\begin{align*}
\mu^0 X.p &\lor (\langle a \rangle True \land [a]X) = \emptyset \\
\mu^1 X.p &\lor (\langle a \rangle True \land [a]\emptyset) = p \lor (\langle a \rangle True \land [a]\emptyset) \\
\langle a \rangle True &\text{ is the set of states with an outer } a \text{-transition} \\
[a]\emptyset &\text{ is the set of states with no outgoing } a \text{-transition} \\
\text{Therefore, intersection } \land &\text{ is empty} \\
\text{and the formula boils down to the set of states satisfying } p \\
\mu^2 X.p &\lor (\langle a \rangle True \land [a]T) = p \lor (\langle a \rangle True \land [a]T) \\
\text{where } T = \mu^1 &\text{ which means nodes satisfying } p \\
[a]T &\text{ are nodes whose children reachable via } a \text{-transitions satisfy } p
\end{align*}
\]

Thus either \( p \) is satisfied, or it is satisfied via a node reachable through an \( a \)-transitions, or via an \( aa \)-transition, or via an \( a^n \)-transition.
Note

• Increasing complexity with alternation of fixed point types
  • With one fix-point we talk about termination properties
  • With two fix-points we can write fairness formulas
• See also Chapter 26 of the Handbook of Model Checking
Nesting Depth: maximum number of nested fixed points

\[
\begin{align*}
ND(f) & := 0 & \text{for } f \in \{p, \neg p, X\} \\
ND(\circ f) & := ND(f) & \text{for } \circ \in \{[a], \langle a \rangle\} \\
ND(f \Box g) & := \max(ND(f), ND(g)) & \text{for } \Box \in \{\land, \lor\} \\
ND(\mu_\nu X.f) & := 1 + ND(f) & \text{for } \mu_\nu \in \{\mu, \nu\}
\end{align*}
\]

Example: \(ND\left(\left(\mu X_1 \cdot \nu X_2 \cdot X_1 \lor X_2\right) \land \left(\mu X_3 \cdot \mu X_4 \cdot (X_3 \land \mu X_5 \cdot p \lor X_5)\right)\right)\)
### Alternation Depth

**Nesting Depth**: maximum number of nested fixed points

<table>
<thead>
<tr>
<th>( ND(f) )</th>
<th>:=</th>
<th>0</th>
<th>for ( f \in { p, \neg p, X } )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ND(\circ f) )</td>
<td>:=</td>
<td>( ND(f) )</td>
<td>for ( \circ \in { [a], \langle a \rangle } )</td>
</tr>
<tr>
<td>( ND(f \Box g) )</td>
<td>:=</td>
<td>( \max(ND(f), ND(g)) )</td>
<td>for ( \Box \in { \land, \lor } )</td>
</tr>
<tr>
<td>( ND(\mu \nu X. f) )</td>
<td>:=</td>
<td>( 1 + ND(f) )</td>
<td>for ( \mu, \nu \in { \mu, \nu } )</td>
</tr>
</tbody>
</table>

**Example:**

\[
ND\left( (\mu X_1. \nu X_2. X_1 \lor X_2) \land (\mu X_3. \mu X_4. (X_3 \land \mu X_5. p \lor X_5)) \right) = 3
\]

\( X_3, X_4 \) and \( X_5 \) have no alternation between fixed point signs.
Alternation Depth: number of alternating fixed points

\[
\begin{align*}
AD(f) & := 0 & \text{for } f \in \{p, \neg p, X\} \\
AD(\circ f) & := AD(f) & \text{for } \circ \in \{[a], \langle a \rangle\} \\
AD(f \Box g) & := \max(AD(f), AD(g)) & \text{for } \Box \in \{\&, \lor\} \\
AD(\mu X.f) & := 1 + \max\{AD(g) \mid g \text{ is a } \nu\text{-subformula of } f\} \\
AD(\nu X.f) & := 1 + \max\{AD(g) \mid g \text{ is a } \mu\text{-subformula of } f\}
\end{align*}
\]

Examples:

\[
AD\left((\mu X_1. \nu X_2. \ X_1 \lor X_2) \land (\mu X_3. \mu X_4. \ (X_3 \land \mu X_5.p \lor X_5))\right)
\]

\[
AD\left((\mu X_1. \nu X_2. \ X_1 \lor X_2) \land (\mu X_3. \nu X_4. \ (X_3 \land \mu X_5.p \lor X_5))\right)
\]
Alternation Depth: number of alternating fixed points

\[
\begin{align*}
AD(f) & := 0 & \text{for } f \in \{p, \neg p, X\} \\
AD(\Box f) & := AD(f) & \text{for } \Box \in \{[, ]\} \\
AD(f \Box g) & := \max(AD(f), AD(g)) & \text{for } \Box \in \{\land, \lor\} \\
AD(\mu X.f) & := 1 + \max\{AD(g) | g \text{ is a } \nu\text{-subformula of } f\} \\
AD(\nu X.f) & := 1 + \max\{AD(g) | g \text{ is a } \mu\text{-subformula of } f\}
\end{align*}
\]

Examples:

\[
AD \left( (\mu X_1. \nu X_2. X_1 \lor X_2) \land (\mu X_3. \mu X_4. (X_3 \land \mu X_5.p \lor X_5)) \right) = 2
\]

\[
AD \left( (\mu X_1. \nu X_2. X_1 \lor X_2) \land (\mu X_3. \nu X_4. (X_3 \land \mu X_5.p \lor X_5)) \right) = 3
\]

\(X_5\) does not depend on \(X_3\) and \(X_4\)
Dependent Alternation Depth (dAD): number of alternating fixed points, such that the innermost fixed point depends on the outermost.

The definition of $dAD$ is identical to $AD$, except for

$$
\begin{align*}
\text{dAD}(\mu X.f) & := \max(\text{dAD}(f), \\
& \quad 1 + \max\{\text{dAD}(g) \mid \\
& \quad g \text{ is a } \nu\text{-subformula of } f \text{ and } X \text{ occurs in } g\}\}
\end{align*}
$$

$$
\begin{align*}
\text{dAD}(\nu X.f) & := \max(\text{dAD}(f), \\
& \quad 1 + \max\{\text{dAD}(g) \mid \\
& \quad g \text{ is a } \mu\text{-subformula of } f \text{ and } X \text{ occurs in } g\}\}
\end{align*}
$$

Examples:

$$
\text{dAD}\left((\mu X_1. \nu X_2. \ X_1 \lor X_2) \land (\mu X_3. \mu X_4. \ (X_3 \land \mu X_5.p \lor X_5))\right)
$$

$$
\text{dAD}\left((\mu X_1. \nu X_2. \ X_1 \lor X_2) \land (\mu X_3. \nu X_4. \ (X_3 \land \mu X_5.p \lor X_5))\right)
$$
Alteration Depth

**Dependent Alternation Depth (\(dAD\)):** number of alternating fixed points, such that the innermost fixed point depends on the outermost.

The definition of \(dAD\) is identical to \(AD\), except for

\[
\begin{align*}
\text{\(dAD(\mu X. f)\)} & := \max(dAD(f), 1 + \max \{ dAD(g) \mid g \text{ is a \(\nu\)-subformula of } f \text{ and } X \text{ occurs in } g \}) \\
\text{\(dAD(\nu X. f)\)} & := \max(dAD(f), 1 + \max \{ dAD(g) \mid g \text{ is a \(\mu\)-subformula of } f \text{ and } X \text{ occurs in } g \})
\end{align*}
\]

**Examples:**

\[
\begin{align*}
dAD\left( (\mu X_1. \nu X_2. \ X_1 \lor X_2) \land (\mu X_3. \mu X_4. \ (X_3 \land \mu X_5. p \lor X_5)) \right) & = 2 \\
\end{align*}
\]

\[
\begin{align*}
dAD\left( (\mu X_1. \nu X_2. \ X_1 \lor X_2) \land (\mu X_3. \nu X_4. \ (X_3 \land \mu X_5. p \lor X_5)) \right) & = 2
\end{align*}
\]
Naive Algorithm

```python
1  def eval(f):
2      if f = tt: return S
3      elif f = ff: return ∅
4      elif f = p: return {s ∈ S | p ∈ L(s)}
5      elif f = ¬p: return {s ∈ S | p ∉ L(s)}
6      elif f = g₁ ∧ g₂: return eval(g₁) ∩ eval(g₂)
7      elif f = g₁ ∨ g₂: return eval(g₁) ∪ eval(g₂)
8      elif f = [a]g: return {s ∈ S | ∀t ∈ S: s →⁢ t ⇒ t ∈ eval(g)}
9      elif f = ⟨a⟩g: return {s ∈ S | ∃t ∈ S: s →⁢ t ∧ (t ∈ eval(g))}
10     elif ...: ...
```
Naive Algorithm

1 def eval(f):
2     if ...: ...
3     elif $f = X_i$: return $A[i]$
4     elif $f = \nu X_i.g(X_i)$:
5         $A[i] := S$
6         while $A[i]$ changes:
7             $A[i] := \text{eval}(g)$
8         return $A[i]$
9     elif $f = \mu X_i.g(X_i)$:
10        $A[i] := \emptyset$
11        while $A[i]$ changes:
12           $A[i] := \text{eval}(g)$
13        return $A[i]$
Assume $\mathcal{A}ct = \{a\}$. There is a straightforward translation of CTL to the $\mu$-calculus:

- $Tr(p) = p$
- $Tr(\neg f) = \neg Tr(f)$
- $Tr(f \land g) = Tr(f) \land Tr(g)$
- $Tr(\mathbf{E} X f) = \langle a \rangle \ Tr(f)$
- $Tr(\mathbf{E} G f) = \nu Y. (Tr(f) \land \langle a \rangle Y)$
- $Tr(\mathbf{E} [f \mathbf{U} g]) = \mu Y. (Tr(g) \lor (Tr(f) \land \langle a \rangle Y))$
Outline

1. Temporal logics CTL and LTL
2. The modal $\mu$-calculus
3. Parity games
4. Attractor based algorithms
5. Fixed point based algorithms
Bird’s Eye View

- Area: formal verification of systems
  - Verify if a system implements the specification
  - Synthesize a controller for an incomplete system that implements the specification
- “Does X have property p” as a game (or compute X such that...)
  - player 0 wants to prove this (or synthesize a controller)
  - player 1 wants to refute this
  - players make choices
- Interesting systems are often “reactive” (run forever)
  - when a car arrives, eventually the traffic light turns green
  - the reset button always works
  - “X is true until Y is true”
  - “X may not happen before Y”

Hence: properties regarding infinite runs of a finite-state system
Why do we want to solve parity games?

- Capture the expressive power of nested least and greatest fixpoint operators
- Equivalent (in polynomial time) to:
  - modal $\mu$-calculus model-checking (CTL*, LTL...)
  - solving Boolean Equation Systems
- Backend for LTL model checking and LTL synthesis
  - important industrial applications (PSL, SVA)
Parity Games

Why do we want to solve parity games?

- Capture the expressive power of nested least and greatest fixpoint operators
- Equivalent (in polynomial time) to:
  - modal $\mu$-calculus model-checking (CTL*, LTL...)
  - solving Boolean Equation Systems
- Backend for LTL model checking and LTL synthesis
  - important industrial applications (PSL, SVA)

Open question: Is solving parity games in $P$?

- It is in $UP \cap co-UP$ and $NP \cap co-NP$
- It is believed a polynomial solution exists
- Hot topic! Recently: quasi-polynomial solution sparked great interest, several new algorithms that are all quasi-polynomial
(Incomplete list of) published algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>McNaughton/Zielonka</td>
<td>$O(e \cdot n^d)$, $O(2^n)$</td>
<td>1998</td>
</tr>
<tr>
<td>Small Progress Measures</td>
<td>$O(d \cdot e \cdot (n/d)^{d/2})$</td>
<td>1998</td>
</tr>
<tr>
<td>Strategy Improvement</td>
<td>$O(n \cdot e \cdot 2^e)$</td>
<td>2000</td>
</tr>
<tr>
<td>Dominion Decomposition</td>
<td>$O(n^{\sqrt{n}})$</td>
<td>2006</td>
</tr>
<tr>
<td>Big Step</td>
<td>$O(e \cdot n^{d/3})$</td>
<td>2007</td>
</tr>
<tr>
<td>APT</td>
<td>$O(n^d)$</td>
<td>2016</td>
</tr>
<tr>
<td>Priority Promotion</td>
<td>$\Omega(2^{\sqrt{n}})$</td>
<td>2016</td>
</tr>
<tr>
<td>Quasi-Polynomial (multiple)</td>
<td>$O(n^{6+\log d})$</td>
<td>2016 – 2018</td>
</tr>
<tr>
<td>Tangle Learning</td>
<td>$\Omega(2^{\sqrt{n}})$</td>
<td>2018</td>
</tr>
<tr>
<td>Recursive Tangle Learning</td>
<td>tbd</td>
<td>2018</td>
</tr>
</tbody>
</table>
A parity game is played on a directed graph.

Two players: Even ♦ and Odd □

The players move a token along the edges of the graph.

Each vertex is owned by one player who chooses a successor.
A parity game is played on a directed graph.

Two players: Even ♦ and Odd ♠

The players move a token along the edges of the graph.

Each vertex is owned by one player who chooses a successor.
A parity game is played on a directed graph.

Two players: Even $\heartsuit$ and Odd $\diamondsuit$.

The players move a token along the edges of the graph.

Each vertex is owned by one player who chooses a successor.

Highest priority seen infinitely often determines the winner.

Player Even wins if this number is even.
A parity game is played on a directed graph.

Two players: Even \(\diamondsuit\) and Odd \(\Box\).

The players move a token along the edges of the graph.

Each vertex is owned by one player who chooses a successor.
A parity game is played on a directed graph.

Two players: Even ♦ and Odd □.

The players move a token along the edges of the graph.

Each vertex is owned by one player who chooses a successor.

In the graph:
- Vertex 'a' is owned by Even.
- Vertex 'b' is owned by Odd.
- Vertex 'c' is owned by Even.
- Vertex 'd' is owned by Odd.
- Vertex 'e' is owned by Even.

Each vertex has a priority \{0, 1, 2, ..., d\}.

The highest priority seen infinitely often determines the winner.

Player Even wins if this number is even.
Parity Games

- A parity game is played on a directed graph
- Two players: Even ♦ and Odd ◊
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor

![Graph Diagram]
Parity Games

- A parity game is played on a directed graph.
- Two players: Even $\Diamond$ and Odd $\Box$
- The players move a token along the edges of the graph.
- Each vertex is owned by one player who chooses a successor.

![Diagram of a parity game graph](image-url)
• A parity game is played on a directed graph
• Two players: Even ◇ and Odd □
• The players move a token along the edges of the graph
• Each vertex is owned by one player who chooses a successor
A parity game is played on a directed graph.

Two players: Even ♦ and Odd □

The players move a token along the edges of the graph.

Each vertex is owned by one player who chooses a successor.
Parity Games

- A parity game is played on a directed graph.
- Two players: Even ♦ and Odd □.
- The players move a token along the edges of the graph.
- Each vertex is owned by one player who chooses a successor.
A parity game is played on a directed graph.

Two players: Even ♠ and Odd ♠

The players move a token along the edges of the graph.

Each vertex is owned by one player who chooses a successor.

How do we determine who wins a play?
A parity game is played on a directed graph.

Two players: Even ♦ and Odd □.

The players move a token along the edges of the graph.

Each vertex is owned by one player who chooses a successor.

Each vertex has a priority \( \{0, 1, 2, \ldots, d\} \).

Highest priority seen infinitely often determines winner.

Player Even wins if this number is even.
A parity game is played on a directed graph.

Two players: Even ✷ and Odd □.

The players move a token along the edges of the graph.

Each vertex is owned by one player who chooses a 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 □.
- The players move a token along the edges of the graph.
- Each vertex is owned by one player who chooses a successor.

A player wins a vertex if it has a strategy to win all plays from that vertex.
A parity game is played on a directed graph.

Two players: Even $\Diamond$ and Odd $\Box$

The players move a token along the edges of the graph.

Each vertex is owned by one player who chooses a successor.

Which vertices are won by which player?
Parity Games

- A parity game is played on a directed graph
- Two players: Even $\Diamond$ and Odd $\Box$
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor

Player Odd wins all vertices with strategy $\{d \rightarrow e\}$
Parity Games

Known facts of parity games

- Some vertices are won by Even, some vertices are won by Odd
- The winner has a memoryless strategy to win

Memoryless strategy

“If I always play from $v$ to $w$, then I win all plays from $v$”
Parity Games

Known facts of parity games

- Some vertices are won by Even, some vertices are won by Odd
- The winner has a memoryless strategy to win

Memoryless strategy

“If I always play from $v$ to $w$, then I win all plays from $v$”

Solving a parity game

- Determine the winner of each vertex
- Compute the strategy for each player
Games and automata for verification and synthesis

- **Verification with automata**
  1. Construct an automaton of the specification (typically negation)
  2. Cross-product with the Kripke Structure or LTS
  3. Solve resulting automaton (accept/reject)
     (typically produces a counterexample)

- **Verification with games**
  1. Construct a two-player game of the specification
     - one player tries to prove the specification (‘existential‘)
     - one player tries to violate the specification (‘universal‘)
  2. Cross-product with the Kripke Structure or LTS
  3. Solve resulting game
     (winner + strategy of winner.)

- **Synthesis with games**
  - strategy is an implementation of a controller (or a counterexample)
  - system + controller = guaranteed to implement the specification
  - (Actually used in practice to synthesize controllers for LTL properties!)
• Adam picks $t$ from $s \xrightarrow{a} t$ such that $t \not\models (p_1 \lor (p_2 \land p_3))$

• Eve replies by showing that either $t \models p_1$ or that $t \models p_2$ and $t \models p_3$. 
Model checking via parity games
Create node \((s, \psi)\) for every state \(s\) of \(M\) and every formula \(\psi\) in the closure of \(\phi\). Eve’s goal is to show that a formula holds.

\[(s, p)\]  Eve wins if \(p\) holds in \(s\), that is \(s \models p\)

Thus assign \((s, p)\) to Adam and we put no transitions from it

\[(s, \neg p)\]  Same as \((s, p)\) but reversing Adam and Eve’s roles

\[(s, (a)\beta)\]  Connect to \((t, \beta)\) for all \(t\) such that \(s \xrightarrow{a} t\) and

assign \((s, [a]\beta)\) to Adam and \((s, (a)\beta)\) to Eve

\[(s, \mu X.\beta(X))\]  Connect to \((s, \beta(\mu X.\beta(X)))\) and to \((s, \nu X.\beta(X)))\)

\[(s, \nu X.\beta(X))\]  This corresponds to the intuition that a fixed-point is equivalent to its unfolding.

\[
\begin{align*}
[\mu X.\alpha]^M_V &= [\alpha[\mu X.\alpha/X]]^M_V \\
[\nu X.\alpha]^M_V &= [\alpha[\nu X.\alpha/X]]^M_V
\end{align*}
\]

- Parity winning condition based on dependent alternation depth.
  - Priority \(2 \cdot \lfloor \text{dAD}(\phi)/2 \rfloor\) if \(\phi\) is of the form \(\nu X.\psi\)
  - Priority \(2 \cdot \lfloor \text{dAD}(\phi)/2 \rfloor + 1\) if \(\phi\) is of the form \(\mu X.\psi\)
  - Priority \(0\) otherwise
Oink

- Modern implementation of parity game algorithms
  - Zielonka’s Algorithm (with optimizations; parallel)
  - Small progress measures (with optimizations)
  - Priority Promotion (different versions)
  - Strategy Improvement (parallel)
  - QPT progress measures
  - Succinct progress measures
  - Tangle learning

- The usual preprocessing algorithms
  - Inflation and compression
  - Remove self-loops
  - Detect winner-controlled winning cycles
  - SCC decomposition

- https://www.github.com/trolando/oink

- Simple to use/extend library in C++
#include "oink.hpp"

pg::Game parity_game;
parity_game.parse_pgsolver(cin);

pg::Oink solver(parity_game);
solver.setSolver("zlk");
solver.run();

parity_game.write_sol(cout);
Parity Games

Notation for parity games

- A parity game $\mathcal{G}$ is a tuple $(V_\Diamond, V_\Box, E, \text{pr})$
- Vertices $V = V_\Diamond \cup V_\Box$ controlled by players Even and Odd
- Transitions $E: V \times V$ such that $E$ is left-total.
  - We write $u \rightarrow v$ for $(u, v) \in E$
  - $E(u)$ denotes the successors of $u$: $\{v \mid u \rightarrow v\}$
  - $E(U)$ denotes all successors of vertices in $\{v \mid u \rightarrow v \mid u \in U\}$
  - Each vertex has at least one successor.
- Priority function $\text{pr}: V \rightarrow \{0, 1, 2, \ldots, d\}$
- A path is a finite sequence $v_0v_1v_2\ldots$ consistent with $E$
- A play is an infinite sequence $v_0v_1v_2\ldots$ consistent with $E$
- A play $\pi$ is won by player Even iff $\max(\text{pr}(\inf \pi))$ is even
Notation for strategies

- A strategy for player $\alpha$ is a partial function $\sigma : V_\alpha \rightarrow V$ that assigns one successor to each vertex of player $\alpha$.
- A path or play is consistent with $\sigma$ if each $v_i$ for which $\sigma(v_i)$ is defined, $v_{i+1} = \sigma(v_i)$.
- $\text{Plays}(v, \sigma)$ is the set of plays consistent with $\sigma$ starting in $v$.
- $\sigma$ is a winning strategy from $v$ for player $\alpha$ if all plays in $\text{Plays}(v, \sigma)$ are winning for $\alpha$. 
Notation for closed sets and dominions

- A set $W$ is **closed w.r.t. a strategy** $\sigma$ if for all $v \in W$:
  - if $v$ is owned by $\alpha$, 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 $\alpha$ if $\alpha$ has a strategy $\sigma$ that is winning for all $v \in D$ and $D$ is closed w.r.t. $\sigma$.

- The winning regions of either player are dominions.
Outline

1 Temporal logics CTL and LTL
2 The modal $\mu$-calculus
3 Parity games
4 Attractor based algorithms
5 Fixed point based algorithms
Attractor computation

Compute all vertices from which player $\alpha \in \{\Diamond, \Box\}$ can ensure arrival in a given target set

Start with the target set $A$, then iteratively add vertices to $A$:

- All vertices of $\alpha$ with an edge to $A$
- All vertices of $\overline{\alpha}$ with only edges to $A$
Parity Games

Example of attractor computation

Computing the $\square$-attractor to $a$

Initial set: $\{a\}$
Can attract: $d$ but not $b$
Example of attractor computation

Computing the $\square$-attractor to $a$

Current set: $\{a, d\}$
Can attract: $b$ but not $e$
Example of attractor computation

Computing the $\square$-attractor to $a$

Current set: \{a, b, d\}
Can attract: neither c nor e
Roughly two types

- **Local value iteration**
  Based on locally improving the value of individual vertices by looking at their successors.

- **Attractor-based**
  Based on properties over sets of vertices computed with attractors.
Attractor-based algorithms

- Partition the game into regions using attractors.
- Start with the highest priority (top-down).
- Each region is tentatively won by one player.
- Refine winning regions until dominion found.

Example: Zielonka’s Recursive Algorithm (1998)

Attract higher regions downward after computing lower regions. If your opponent attracts from your region, recompute your part.

Example: Priority Promotion (2016)

Merge regions upwards when the region is closed (in the subgame). Then recompute lower regions.
Zielonka’s recursive algorithm

1. `def zielonka(∅):
2.     if ∅ = ∅:
3.         return ∅, ∅    // empty game
4.     α ← pr(∅) mod 2  // winner of highest priority
5.     Z ← pr^{-1}(pr(∅)) // vertices of highest priority
6.     A ← Attr (∅)(Z) // attracted to highest priority
7.     W_0, W_0 ← zielonka(∅ \ A) // recursive solution
8.     B ← Attr (∅)(W_0) // check if opponent attracts
9.     if B = W_0:
10.        W_0 ← W_0 ∪ A    // A is won by α
11.    else:
12.        W_0, W_0 ← zielonka(∅ \ B) // recompute remainder
13.        W_0 ← W_0 ∪ B    // B is won by \bar{α}
14.    return W_0, W_0`
Computing strategy

- Strategy is computed by attractor
  - Every attracted $\alpha$-vertex $u$ to some $v$ in the set: strategy is $u \rightarrow v$
- Special case: $\alpha$-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 highest priority vertices before attracting
Zielonka’s recursive algorithm

```python
1 def zielonka(\emptyset):
2     if \emptyset = \emptyset:
3         return \emptyset, \emptyset  # empty game
4     \alpha \leftarrow \text{pr}(\emptyset) \mod 2  # winner of highest priority
5     Z \leftarrow \text{pr}^{-1}(\text{pr}(\emptyset))  # vertices of highest priority
6     A, \sigma_A \leftarrow \text{Attr}_\alpha(Z)  # attracted to highest priority
7     W_\Diamond, W_\Box, \sigma_\Diamond, \sigma_\Box \leftarrow zielonka(\emptyset \setminus A)  # recursive solution
8     B, \sigma_B \leftarrow \text{Attr}_{\overline{\alpha}}(W_{\overline{\alpha}})  # check if opponent attracts
9     \sigma_B \leftarrow \sigma_B \cup \sigma_\alpha  # add strategy of W_{\overline{\alpha}}
10    if B = W_{\overline{\alpha}}:
11        W_\alpha \leftarrow W_\alpha \cup A  # A is won by \alpha
12        \sigma_\alpha \leftarrow \sigma_\alpha \cup \sigma_A \cup ((z \in Z) \mapsto \text{pick}(E(z) \cap W_\alpha))
13    else:
14        W_\Diamond, W_\Box, \sigma_\Diamond, \sigma_\Box \leftarrow zielonka(\emptyset \setminus B)  # recompute remainder
15        W_{\overline{\alpha}} \leftarrow W_{\overline{\alpha}} \cup B  # B is won by \overline{\alpha}
16        \sigma_{\overline{\alpha}} \leftarrow \sigma_{\overline{\alpha}} \cup \sigma_B
17    return W_\Diamond, W_\Box, \sigma_\Diamond, \sigma_\Box
```
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.
• After regions 8 (player Even) and 7 (player Odd).
• Continue (recursively) with region 6.
- After regions 8, 7 and 6.
- Continue (recursively) with region 5.
• After regions 8, 7, 6 and 5.
• Continue (recursively) with region 3.
• After regions 8, 7, 6, 5 and 3.
• Now remains just region 2.
• 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).
• 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.
Zielonka’s Algorithm

- Vertex \( g \) is attracted to the Odd region.
- So now recompute the (remainder of the) lower regions of Even.
  - Actually, nothing changes in the recursion.
- Up in region 7: does the lower Even region attract from 7?
Zielonka’s Algorithm

- Vertex **g** is attracted to the Odd region.
- So now **recompute the (remainder of the) lower regions of Even**.
  - Actually, nothing changes in the recursion.
- Up in region **7**: does the lower Even region attract from **7**?
- Yes, the lower Even region attracts vertex **c**.
• **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!
• Region 6: now the Odd region attracts vertex g again.
• Vertex \(g\) is attracted to the Odd region.
• Recursive game of 6 is empty.
• Up in region 8, does Odd now attract 8?
• Vertex $g$ is attracted to the Odd region.
• Recursive game of $6$ is empty.
• Up in region $8$, does Odd now attract $8$?
• But vertex $f$ is attracted to the Odd region.
• Attracting at priority $8$ attracts all vertices to player Odd.
Zielonka’s Algorithm

- Final result, entire game won by player Odd.
The main idea of priority promotion...

Region invariant

- In any region, the opponent either plays to a higher region of the player, or via the highest priority vertices.
- (Invariant holds for the regions of the “$\alpha$-maximal partition”)

Closed region

- A region of player $\alpha$ that is globally closed is a dominion of player $\alpha$.
- A region of player $\alpha$ is locally closed iff the opponent can only escape to a higher region of player $\alpha$.
- So: the opponent must escape to the lowest higher region.
  $\Rightarrow$ Promote the region, i.e., merge the regions.
Priority promotion

1 def prioprom(∅):
2     r ← V ↦ ⊥ // all vertices to ⊥
3     p ← pr(∅) // highest priority
4     while True :
5         α ← p mod 2 // current player
6         Z ← {v | r(v) ≤ p} // current subgame
7         A ← Attr_{α∩Z} ({v ∈ Z | r(v) = p ∨ pr(v) = p}) // attract
8         C ← {v ∈ A_α | E(v) ∩ A = ∅} // open α-vertices
9         X ← E(A_α) \ A // escapes
10        if C ≠ ∅ ∨ (X ∩ Z) ≠ ∅ :
11            r ← r[A ↦ p] // set region
12            p ← pr(Z \ A) // continue with next highest
13        elif X ≠ ∅ :
14            p ← min{r(v) | v ∈ X} // set p to lowest escape
15            r ← r[A ↦ p][{v | r(v) < p} ↦ ⊥] // merge and reset
16        else:
17            return α, A // dominion!
Priority promotion

Notes

- The lowest region is always locally closed.
- Region resets only if at least 1 vertex promotes.
- This is sufficient to prove termination.
- Each call to prioprom computes a dominion of a player $\alpha$.
- Attract for player $\alpha$ to the computed dominion, repeat until game solved.

Computing strategy

- Strategy is computed by attractor
  - Every attracted $\alpha$-vertex $u$ to some $v$ in the set: strategy is $u \rightarrow v$
- Special case: $\alpha$-vertices of the original target set
  - Pick any successor in result as strategy
- Implementation: use only a single strategy array, reset the strategy of highest priority vertices before attracting
• We start by attracting to 8 for player Even, 7 for player Odd, etc.
• After regions 8 (player Even) and 7 (player Odd).
After regions 6 and 5.
• After region 3, region 3 is now closed!

• **Note:** region 2 would also be closed.
• After region 3, region 3 is now closed!

• **Note:** region 2 would also be closed.

• The loser must escape to a higher region of the winner.

• So promote 3 to 5.

• Meaning *the set* \{h, i\} is attracted *as a whole* to region 5.
• Region 3 is promoted to region 5.
• Now region 5 is closed. (region 2 would also be closed)
• So promote 5 to 7.
• After promotion, vertex g is attracted to region 7.
• Continue the partition with region 2...
• Region 2 is locally closed.
• So promote 2 to (the lowest escape) 8.
• Region 2 is promoted to region 8.
• Meaning the set \{a, b\} is attracted to region 8.
• Region 8 now also attracts vertex c!!
• Recompute the subgame...
• Now \{h, i\} can be attracted to region 5 again.
Region 5 is closed in the entire game.
Meaning that it is a dominion won by player Odd.
Priority promotion

Variations

- **PP⁺**: only reset regions of $\overline{\alpha}$.
- **RP**: only reset a region when the strategy of player $\alpha$ of the remaining vertices of the stored region leaves the region.
- **DP**: “delayed promotion” strategy.
Tangle Learning

Tangle

A tangle is:

- a (strongly connected) subgraph of a parity game,
- such that one player $\alpha$ has a strategy $\sigma$,
- such that the tangle restricted by $\sigma$ is still strongly connected,
- and player $\alpha$ wins all plays (cycles) in the tangle.

Definition

A $p$-tangle is a nonempty set of vertices $U \subseteq V$ with $p = \text{pr}(U)$, for which player $\alpha \equiv_2 p$ has a strategy $\sigma : U_\alpha \to U$, such that the graph $(U, E')$, with $E' := E \cap (\sigma \cup (U_\alpha \times U))$, is strongly connected and player $\alpha$ wins all cycles in $(U, E')$. 
Tangle Learning

Tangle

A tangle is a strongly connected subgraph for which one player has a strategy to win all cycles in the subgraph.

Properties

- Player $\alpha$ has a single strategy for every $\alpha$-vertex.
- Player $\overline{\alpha}$ must escape (or lose).
- Player $\overline{\alpha}$ can reach all vertices of the tangle.
- Tangles have subtangles when player $\overline{\alpha}$ can avoid vertices.
- Every dominion is naturally composed of subtangles.
A tangle is a strongly connected subgraph for which one player has a strategy to win all cycles in the subgraph.

A 5-dominion with a 5-tangle and a 3-tangle
Tangle learning

Tangle attractor

Because player $\bar{\alpha}$ must escape the tangle, we can use tangles to attract the vertices of a tangles together, if player $\bar{\alpha}$ can only escape to the attracting set.

- Add all $v \in V_\alpha \setminus A$ for which $E(v) \cap A \neq \emptyset$.
- Add all $v \in V_{\bar{\alpha}} \setminus A$ for which $E(v) \subseteq A$.
- Add all $\{v \in V(T(t)) \setminus A \mid t \in T_\alpha\}$ for which $E_T(t) \subseteq A$.

Tangle learning

- Partition game into $\alpha$-maximal regions with tangle attractor.
- Add bottom SCCs of closed regions to the set of tangles.
- Repeat until a dominion is found, i.e., $E_T(t) = \emptyset$. 
• search returns new tangles of $\emptyset$, given known tangles $T$.
• Note: store for each tangle its player $\alpha$ strategy.

```
1 def search(∅, T):
2     if ∅ = ∅: return ∅
3     p ← pr(∅), α ← pr(∅) mod 2
4     Z, σ ← TAttr_α,T(pr^{-1}(p))
5     O ← \{v ∈ Z_α | E(v) \cap Z = ∅\} ∪ \{v ∈ Z_\overline{α} | E(v) \not\subseteq Z\}
6     if O = ∅:
7         return search(∅ \ Z, T) ∪ bottom-sccs(Z, σ)
8     else:
9         return search(∅ \ Z, T)
```
• search returns new tangles of $\mathcal{D}$, given known tangles $T$.
• Note: store for each tangle its player $\alpha$ strategy.

```python
1 def search($\mathcal{D}$, $T$):
2     if $\mathcal{D} = \emptyset$ : return $\emptyset$
3     $p \leftarrow \text{pr}(\mathcal{D})$, $\alpha \leftarrow \text{pr}(\mathcal{D}) \mod 2$
4     $Z, \sigma \leftarrow \text{Attr}_{\alpha}^{\mathcal{D},T}(\text{pr}^{-1}(p))$
5     $O \leftarrow \{v \in Z_{\alpha} | E(v) \cap Z = \emptyset\} \cup \{v \in Z_{\bar{\alpha}} | E(v) \not\subseteq Z\}$
6     if $O = \emptyset$ :
7         return search($\mathcal{D} \setminus Z, T$) $\cup$ bottom-sccs($Z, \sigma$)
8     else:
9         return search($\mathcal{D} \setminus Z, T$) $\cup$ search($\mathcal{D} \cap (Z \setminus \text{Attr}_{\alpha}^{\mathcal{D} \cap Z, T}(O)), T$)
```

The “recursive” variant of tangle learning
def tanglelearning(∅):
    W₀ ← ∅, σ₀ ← ∅, W₀ ← ∅, σ₀ ← ∅, T ← ∅
    while ∅ ≠ ∅:
        Y ← search(∅, T)
        T ← T ∪ {t ∈ Y | ET(t) ≠ ∅}
        D ← {t ∈ Y | ET(t) = ∅}
        if D ≠ ∅:
            D⁺₀, σ ← TAttr₀, T (∪ D₀)
            W₀ ← W₀ ∪ D⁺₀, σ₀ ← σ₀ ∪ σ
            D⁺₀, σ ← TAttr₀, T (∪ D₀)
            W₀ ← W₀ ∪ D⁺₀, σ₀ ← σ₀ ∪ σ
            ∅ ← ∅ \ (D⁺₀ ∪ D⁺₀)
            T ← T ∩ ∅
    return W₀, W₀, σ₀, σ₀
Tangle learning

Computing strategy

- Similar to priority promotion: compute strategy with the attractor, select any successor in the region for the highest priority vertices of $\alpha$
- Store the $\sigma$ of every tangle and use the stored $\sigma$ as the strategy for $\alpha$ when attracting a tangle
Tangle learning

- After first partition into $\alpha$-maximal regions.
- Regions 2 and 3 are closed (in their subgame).
- Tangle \{a, b\} attracted to 8.
- Tangle \{h, i\} attracted to 5.
• Tangles: \{a, b\} (2) and \{h, i\} (3).

• After tangle attractor to 8...
• Tangles: \( \{a, b\} \) (2) and \( \{h, i\} \) (3).

• After tangle attractor to 6...
• Tangles: \{a, b\} (2) and \{h, i\} (3).
• After tangle attractor to 5...
• Only closed region: 5.
• One tangle, which is also a dominion.
• Vertex $b$ is a distraction for player Even.
• Learn opponent tangles to attract the distractions.
• Tangle $\{c\}$ is attracted to region 5.
• Now vertex $a$ is not distracted by vertex $b$. 
• First round: tangle \{c\} (attracts distraction b).
• Second round: tangle \{a, e\} (attracts distraction h).
• Third round: tangle \{g\} (dominion).
Outline

1 Temporal logics CTL and LTL
2 The modal $\mu$-calculus
3 Parity games
4 Attractor based algorithms
5 Fixed point based algorithms
Core idea of value iteration (1/2)

- **Measure** $\rho: V \rightarrow \mathbb{M}$ assign a value to every vertex from some domain $\mathbb{M}$, containing a special symbol $\top$.
- The measure represents *how good is the “best” continuation?*
  - “best” for one of the players, e.g., Even
  - symbol $\top$ means “winning for the player“ (Even)
  - Even wants high values, Odd wants low values
- A **monotone** function $\text{Prog}(m, \rho)$ that computes the value of playing from a vertex with priority $p$ to a vertex with measure $m$
- $\rho$ is the **least** parity game progress measure, if smallest $\rho$ such that:

\[
\forall v \in V: \rho(v) = \begin{cases} 
\max_{\sqsubseteq} \{ \text{Prog}(\rho(w), \text{pr}(v)) \mid w \in E(v) \} & v \in V_{\Diamond} \\
\min_{\sqsubseteq} \{ \text{Prog}(\rho(w), \text{pr}(v)) \mid w \in E(v) \} & v \in V_{\Box}
\end{cases}
\]

*with respect to a special ordering (see later)*
Core idea of value iteration (2/2)

- If $\rho$ is the least parity game progress measure, then:
  - $W_\Diamond = \{ v \mid \rho(v) = \top \}$, $W_\Box = \{ v \mid \rho(v) \neq \top \}$
  - if $v \in W_\Box$, then $\rho(v) = \Prog(\rho(\sigma(w)), \pr(v))$
    meaning: the winning strategy for Odd is the best continuation
  - no* winning strategy for Even

- It is a least fixed point: starting with $\bot$, update $\rho$ until fixed point

- This is called lifting the measures

- Idea: this is like playing an “optimal” game backwards
  - Player Even finds better paths
  - Player Odd then selects the least bad option

*except with some extra effort: Gazda and Willemse, 2014
Small progress measures

Even measures

- Measures are tuples $\langle e_6, e_4, e_2, e_0 \rangle$ (with highest even priority 6)
- Each $e_p = [0..n_p]$ with $n_p$ the number of vertices with priority $p$
- Example: $\mathbb{M} = (\{0\} \times \{0,1,2\} \times \{0\} \times \{0,1\}) \cup \{\top\}$
- A total order $\sqsubseteq$ which is lexicographic: $m_1 \sqsubseteq m_2$ iff there is a highest unequal priority $z$ and $m_1(z) < m_2(z)$ (and $\top = \top$)
  \[
  \langle 1,0,0,0 \rangle \sqsubseteq \langle 1,0,0,1 \rangle \\
  \langle 4,2,10,5 \rangle \sqsubseteq \langle 4,3,0,0 \rangle
  \]

Odd measures

- Same, but with the odd priorities
Small progress measures

Even measures

- Measures are tuples \( \langle e_6, e_4, e_2, e_0 \rangle \) (with highest even priority 6)
- A \( p \)-truncation keeps only elements \( \geq p \):
  \[
  \langle 1, 2, 3, 2 \rangle|_1 = \langle 1, 2, 3 \rangle \\
  \langle 1, 2, 3, 2 \rangle|_4 = \langle 1, 2 \rangle \\
  \langle 1, 2, 3, 2 \rangle|_7 = \varepsilon
  \]
- Notation: \( m_1 \trianglerighteq_p m_2 \equiv m_1|_p \trianglerighteq m_2|_p \)
- An edge \( v \to u \) is progressive if \( \rho(v) \trianglerighteq_{pr(v)} \rho(u) \) if \( v \) is odd and \( \rho(v) \trianglerighteq_{pr(v)} \rho(u) \) if \( v \) is even
- \( \rho \) is a progress measure if:
  - for every vertex of Even, some outgoing edge is progressive in \( \rho \)
  - for every vertex of Odd, every outgoing edge is progressive in \( \rho \)
  (remember: we are interested in the least progress measure)
Small progress measures

The \textit{Prog} function

Playing from a vertex with priority \(p\) to a vertex with measure \(m\) yields:

\[
\text{Prog}(m, p) := \begin{cases} 
\min \left\{ m' \in \mathbb{M} \mid m' \sqsupseteq_p m \right\} & \text{if } p \text{ is even} \\
\min \left\{ m' \in \mathbb{M} \mid m' \sqsubseteq_p m \right\} & \text{if } p \text{ is odd}
\end{cases}
\]

Example (with highest value 3 for all elements):

\[
\begin{align*}
\text{Prog}(\langle 3, 2, 3, 2 \rangle, 0) & = \langle 3, 2, 3, 3 \rangle \\
\text{Prog}(\langle 3, 2, 3, 2 \rangle, 1) & = \langle 3, 2, 3, 0 \rangle \\
\text{Prog}(\langle 3, 2, 3, 2 \rangle, 2) & = \langle 3, 3, 0, 0 \rangle \\
\text{Prog}(\langle 3, 2, 3, 2 \rangle, 3) & = \langle 3, 2, 0, 0 \rangle \\
\text{Prog}(\langle 3, 2, 3, 2 \rangle, 4) & = \langle 3, 3, 0, 0 \rangle \\
\text{Prog}(\langle 3, 2, 3, 2 \rangle, 5) & = \langle 3, 0, 0, 0 \rangle \\
\text{Prog}(\langle 3, 2, 3, 2 \rangle, 6) & = \top \\
\text{Prog}(\langle 3, 2, 3, 2 \rangle, 7) & = \langle 0, 0, 0, 0 \rangle
\end{align*}
\]
Small progress measures

Operational interpretation [Gazda, Willemse, 2015]

- $p$-dominated stretches: how often priority $p$ is encountered before a higher priority
- Example: play $00102120232142656201$ corresponds to $\langle 2,1,3,2 \rangle$
  - priority 0 is seen $2 \times$ before a higher priority
  - priority 2 is seen $3 \times$ before a higher priority
  - priority 4 is seen $1 \times$ before a higher priority
  - priority 6 is seen $2 \times$ before a higher priority
- If priority $p$ is seen more than $n_p$ times, there must be a cycle!

Operational interpretation [Van Dijk, 2018]

- *Notice the “overflow mechanism”!*  
  If priority $p$ overflows, our optimal path contains a cycle of priority $p$.  
  Keep increasing the measure until the opponent “escapes”  
  (Compare to priority promotion / tangles!)
Small progress measures

\[
\begin{align*}
\text{a} & : \langle 0,0,0,0,0 \rangle \quad \text{to} \quad \langle 0,0,0,0,1 \rangle \quad 0 \\
\text{b} & : \langle 0,0,0,0,- \rangle \quad \text{to} \quad \langle 0,0,0,1,- \rangle \quad 2 \\
\text{c} & : \langle 0,-,-,-,-,- \rangle \quad \text{to} \quad \langle 0,-,-,-,-,- \rangle \quad 7 \\
\text{d} & : \langle 0,0,0,0,- \rangle \quad \text{to} \quad \langle 0,0,0,0,- \rangle \quad 1 \\
\text{e} & : \langle 0,0,-,-,-,- \rangle \quad \text{to} \quad \langle 0,0,-,-,-,- \rangle \quad 5 \\
\text{f} & : \langle 0,-,-,-,-,- \rangle \quad \text{to} \quad \langle 1,-,-,-,-,- \rangle \quad 8 \\
\text{g} & : \langle 0,0,-,-,-,- \rangle \quad \text{to} \quad \langle 0,1,-,-,-,- \rangle \quad 6 \\
\text{h} & : \langle 0,0,0,0,- \rangle \quad \text{to} \quad \langle 0,0,0,1,- \rangle \quad 2 \\
\text{i} & : \langle 0,0,0,-,-,- \rangle \quad \text{to} \quad \langle 0,0,0,-,-,- \rangle \quad 3
\end{align*}
\]
Small progress measures

\[ a \langle 0, 0, 0, 0, 1 \rangle \quad \text{to} \quad \langle 0, 0, 0, 0, 1, 1 \rangle \quad 02 \]
\[ b \langle 0, 0, 0, 1, - \rangle \quad \text{to} \quad \langle 0, 0, 0, 1, - \rangle \quad 2 \]
\[ c \langle 0, -, -, -, -, - \rangle \quad \text{to} \quad \langle 0, -, -, -, -, - \rangle \quad 7 \]
\[ d \langle 0, 0, 0, 0, - \rangle \quad \text{to} \quad \langle 0, 0, 0, 0, - \rangle \quad 1 \]
\[ e \langle 0, 0, -, -, -, - \rangle \quad \text{to} \quad \langle 0, 0, -, -, -, - \rangle \quad 5 \]
\[ f \langle 1, -, -, -, -, - \rangle \quad \text{to} \quad \langle 1, -, -, -, -, - \rangle \quad 8 \]
\[ g \langle 0, 1, -, -, -, - \rangle \quad \text{to} \quad \langle 0, 1, -, -, -, - \rangle \quad 6 \]
\[ h \langle 0, 0, 0, 1, - \rangle \quad \text{to} \quad \langle 0, 0, 0, 1, - \rangle \quad 2 \]
\[ i \langle 0, 0, 0, -, - \rangle \quad \text{to} \quad \langle 0, 0, 0, -, - \rangle \quad 3 \]
Small progress measures

\[ \langle 0,0,0,1,1 \rangle \text{ to } \langle 0,0,0,1,1 \rangle 02 \]
\[ \langle 0,0,0,1,1 \rangle \text{ to } \langle 0,0,0,2,1 \rangle 202 \]
\[ \langle 0,0,0,1,1 \rangle \text{ to } \langle 0,0,0,1,1 \rangle 02 \]
\[ \langle 0,0,0,1,1 \rangle \text{ to } \langle 0,0,0,2,1 \rangle 202 \]
\[ \langle 0,0,0,1,1 \rangle \text{ to } \langle 0,0,0,2,1 \rangle 202 \]
\[ \langle 0,0,0,1,1 \rangle \text{ to } \langle 0,0,0,2,1 \rangle 202 \]
\[ \langle 0,0,0,1,1 \rangle \text{ to } \langle 0,0,0,2,1 \rangle 202 \]
\[ \langle 0,0,0,1,1 \rangle \text{ to } \langle 0,0,0,2,1 \rangle 202 \]
\[ \langle 0,0,0,1,1 \rangle \text{ to } \langle 0,0,0,2,1 \rangle 202 \]
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 \quad \text{to} \quad \langle 0, 0, 0, 2, 1 \rangle \quad 0202

b \langle 0, 0, 0, 2, - \rangle \quad \text{to} \quad \langle 0, 1, 0, 0, - \rangle \quad 20202

c \langle 0, -, -, -, - \rangle \quad \text{to} \quad \langle 0, -, -, -, - \rangle \quad 7

d \langle 0, 0, 0, 0, - \rangle \quad \text{to} \quad \langle 0, 0, 0, 0, - \rangle \quad 1

e \langle 0, 0, -, -, - \rangle \quad \text{to} \quad \langle 0, 0, -, -, - \rangle \quad 5

f \langle 1, -, -, -, - \rangle \quad \text{to} \quad \langle 1, -, -, -, - \rangle \quad 8

g \langle 0, 1, -, -, - \rangle \quad \text{to} \quad \langle 0, 1, -, -, - \rangle \quad 6

h \langle 0, 0, 0, 1, - \rangle \quad \text{to} \quad \langle 0, 0, 0, 1, - \rangle \quad 2

i \langle 0, 0, 0, -, - \rangle \quad \text{to} \quad \langle 0, 0, 0, -, - \rangle \quad 3
Small progress measures

\[
\begin{align*}
\text{a} & \quad \langle 0, 0, 0, 2, 1 \rangle \quad \text{to} \quad \langle 0, 1, 0, 0, 1 \rangle & \quad 020202 \\
\text{b} & \quad \langle 0, 1, 0, 0, - \rangle \quad \text{to} \quad \langle 0, 1, 0, 0, - \rangle & \quad 20202 \\
\text{c} & \quad \langle 0, -, -, -, - \rangle \quad \text{to} \quad \langle 0, -, -, -, - \rangle & \quad 7 \\
\text{d} & \quad \langle 0, 0, 0, 0, - \rangle \quad \text{to} \quad \langle 0, 0, 0, 0, - \rangle & \quad 1 \\
\text{e} & \quad \langle 0, 0, -, -, - \rangle \quad \text{to} \quad \langle 0, 0, -, -, - \rangle & \quad 5 \\
\text{f} & \quad \langle 1, -, -, -, - \rangle \quad \text{to} \quad \langle 1, -, -, -, - \rangle & \quad 8 \\
\text{g} & \quad \langle 0, 1, -, -, - \rangle \quad \text{to} \quad \langle 0, 1, -, -, - \rangle & \quad 6 \\
\text{h} & \quad \langle 0, 0, 0, 1, - \rangle \quad \text{to} \quad \langle 0, 0, 0, 1, - \rangle & \quad 2 \\
\text{i} & \quad \langle 0, 0, 0, - , - \rangle \quad \text{to} \quad \langle 0, 0, 0, - , - \rangle & \quad 3
\end{align*}
\]
Small progress measures

```
\begin{align*}
\text{a} & : \langle 0,1,0,0,1 \rangle \quad \text{to} \quad \langle 0,1,0,0,1 \rangle & 020202 \\
\text{b} & : \langle 0,1,0,0,- \rangle \quad \text{to} \quad \langle 0,1,0,1,- \rangle & 2020202 \\
\text{c} & : \langle 0,-,-,-,- \rangle \quad \text{to} \quad \langle 0,-,-,-,- \rangle & 7 \\
\text{d} & : \langle 0,0,0,0,- \rangle \quad \text{to} \quad \langle 0,0,0,0,- \rangle & 1 \\
\text{e} & : \langle 0,0,-,-,- \rangle \quad \text{to} \quad \langle 0,0,-,-,- \rangle & 5 \\
\text{f} & : \langle 1,-,-,-,- \rangle \quad \text{to} \quad \langle 1,-,-,-,- \rangle & 8 \\
\text{g} & : \langle 0,1,-,-,- \rangle \quad \text{to} \quad \langle 0,1,-,-,- \rangle & 6 \\
\text{h} & : \langle 0,0,0,1,- \rangle \quad \text{to} \quad \langle 0,0,0,1,- \rangle & 2 \\
\text{i} & : \langle 0,0,0,-,- \rangle \quad \text{to} \quad \langle 0,0,0,-,- \rangle & 3
\end{align*}
```
Small progress measures

\[
\begin{align*}
\langle 0, 1, 0, 0, 1 \rangle & \quad \text{to} \quad \langle 0, 1, 0, 1, 1 \rangle & \quad 02020202 \\
\langle 0, 1, 0, 1, - \rangle & \quad \text{to} \quad \langle 0, 1, 0, 1, - \rangle & \quad 2020202 \\
\langle 0, - , - , - , - \rangle & \quad \text{to} \quad \langle 0, - , - , - , - \rangle & \quad 7 \\
\langle 0, 0, 0, 0, - \rangle & \quad \text{to} \quad \langle 0, 0, 0, 0, - \rangle & \quad 1 \\
\langle 0, 0, - , - , - \rangle & \quad \text{to} \quad \langle 0, 0, - , - , - \rangle & \quad 5 \\
\langle 1, - , - , - , - \rangle & \quad \text{to} \quad \langle 1, - , - , - , - \rangle & \quad 8 \\
\langle 0, 1, - , - , - \rangle & \quad \text{to} \quad \langle 0, 1, - , - , - \rangle & \quad 6 \\
\langle 0, 0, 0, 1, - \rangle & \quad \text{to} \quad \langle 0, 0, 0, 1, - \rangle & \quad 2 \\
\langle 0, 0, 0, - , - \rangle & \quad \text{to} \quad \langle 0, 0, 0, - , - \rangle & \quad 3
\end{align*}
\]
Small progress measures

\[
\begin{array}{c}
\text{a} & \text{b} & \text{c} & \text{d} & \text{e} \\
0 & 2 & 7 & 1 & 5 \\
8 & 6 & 2 & 3 & \\
\text{f} & \text{g} & \text{h} & \text{i} & \\
\end{array}
\]

\[
\begin{align*}
\text{a} & : & \langle 0,1,0,1,1 \rangle & \rightarrow & \langle 0,1,0,1,1 \rangle & & 0 \quad 02020202 \\
\text{b} & : & \langle 0,1,0,1,- \rangle & \rightarrow & \langle 0,1,0,2,- \rangle & & 2 \quad 202020202 \\
\text{c} & : & \langle 0,-,-,-,- \rangle & \rightarrow & \langle 0,-,-,-,- \rangle & & 7 \\
\text{d} & : & \langle 0,0,0,0,- \rangle & \rightarrow & \langle 0,0,0,0,- \rangle & & 1 \\
\text{e} & : & \langle 0,0,-,-,- \rangle & \rightarrow & \langle 0,0,-,-,- \rangle & & 5 \\
\text{f} & : & \langle 1,-,-,-,- \rangle & \rightarrow & \langle 1,-,-,-,- \rangle & & 8 \\
\text{g} & : & \langle 0,1,-,-,- \rangle & \rightarrow & \langle 0,1,-,-,- \rangle & & 6 \\
\text{h} & : & \langle 0,0,0,1,- \rangle & \rightarrow & \langle 0,0,0,1,- \rangle & & 2 \\
\text{i} & : & \langle 0,0,0,-,- \rangle & \rightarrow & \langle 0,0,0,-,- \rangle & & 3 \\
\end{align*}
\]
Small progress measures

\[
\begin{align*}
a & : \langle 0,1,0,1,1 \rangle \quad \text{to} \quad \langle 0,1,0,2,1 \rangle & \quad 0202020202 \\
b & : \langle 0,1,0,2,- \rangle \quad \text{to} \quad \langle 0,1,0,2,- \rangle & \quad 202020202 \\
c & : \langle 0,-,-,-,- \rangle \quad \text{to} \quad \langle 0,-,-,-,- \rangle & \quad 7 \\
d & : \langle 0,0,0,0,- \rangle \quad \text{to} \quad \langle 0,0,0,0,- \rangle & \quad 1 \\
e & : \langle 0,0,-,-,- \rangle \quad \text{to} \quad \langle 0,0,-,-,- \rangle & \quad 5 \\
f & : \langle 1,-,-,-,- \rangle \quad \text{to} \quad \langle 1,-,-,-,- \rangle & \quad 8 \\
g & : \langle 0,1,-,-,- \rangle \quad \text{to} \quad \langle 0,1,-,-,- \rangle & \quad 6 \\
h & : \langle 0,0,0,1,- \rangle \quad \text{to} \quad \langle 0,0,0,1,- \rangle & \quad 2 \\
i & : \langle 0,0,0,-,- \rangle \quad \text{to} \quad \langle 0,0,0,-,- \rangle & \quad 3
\end{align*}
\]
Small progress measures

\begin{align*}
\text{a} & \quad \langle 0, 1, 0, 2, 1 \rangle \quad \text{to} \quad \langle 0, 1, 0, 2, 1 \rangle \quad 0202020202 \\
\text{b} & \quad \langle 0, 1, 0, 2, - \rangle \quad \text{to} \quad \langle 1, 0, 0, 0, - \rangle \quad 20202020202 \\
\text{c} & \quad \langle 0, -, -, -, - \rangle \quad \text{to} \quad \langle 0, -, -, -, - \rangle \quad 7 \\
\text{d} & \quad \langle 0, 0, 0, 0, - \rangle \quad \text{to} \quad \langle 0, 0, 0, 0, - \rangle \quad 1 \\
\text{e} & \quad \langle 0, 0, -, -, - \rangle \quad \text{to} \quad \langle 0, 0, -, -, - \rangle \quad 5 \\
\text{f} & \quad \langle 1, -, -, -, - \rangle \quad \text{to} \quad \langle 1, -, -, -, - \rangle \quad 8 \\
\text{g} & \quad \langle 0, 1, -, -, - \rangle \quad \text{to} \quad \langle 0, 1, -, -, - \rangle \quad 6 \\
\text{h} & \quad \langle 0, 0, 0, 1, - \rangle \quad \text{to} \quad \langle 0, 0, 0, 1, - \rangle \quad 2 \\
\text{i} & \quad \langle 0, 0, 0, -, - \rangle \quad \text{to} \quad \langle 0, 0, 0, -, - \rangle \quad 3
\end{align*}
Small progress measures

\[ \langle 0,1,0,2,1 \rangle \quad \text{to} \quad \langle 1,0,0,0,1 \rangle \quad 020202020202 \]

\[ \langle 1,0,0,0,0,\rangle \quad \text{to} \quad \langle 1,0,0,0,0,\rangle \quad 202020202020 \]

\[ \langle 0,-,-,-,-,-\rangle \quad \text{to} \quad \langle 1,-,-,-,-,-\rangle \quad 720202020202 \]

\[ \langle 0,0,0,0,0,-\rangle \quad \text{to} \quad \langle 0,0,0,0,-\rangle \quad 1 \]

\[ \langle 0,0,-,-,-,-\rangle \quad \text{to} \quad \langle 0,0,-,-,-,-\rangle \quad 5 \]

\[ \langle 1,-,-,-,-,-\rangle \quad \text{to} \quad \langle 1,-,-,-,-,-\rangle \quad 8 \]

\[ \langle 0,1,-,-,-,-\rangle \quad \text{to} \quad \langle 0,1,-,-,-,-\rangle \quad 6 \]

\[ \langle 0,0,0,1,-\rangle \quad \text{to} \quad \langle 0,0,0,1,-\rangle \quad 2 \]

\[ \langle 0,0,0,-,-,-\rangle \quad \text{to} \quad \langle 0,0,0,-,-,-\rangle \quad 3 \]
Small progress measures

\[
\begin{align*}
\text{a} & \quad \langle 1, 0, 0, 0, 1 \rangle \quad \text{to} \quad \langle 1, 0, 0, 0, 1 \rangle & 020202020202 \\
\text{b} & \quad \langle 1, 0, 0, 0, - \rangle \quad \text{to} \quad \langle 1, 0, 0, 1, - \rangle & 28 \\
\text{c} & \quad \langle 1, -, -, -, - \rangle \quad \text{to} \quad \langle 1, -, -, -, - \rangle & 728 \\
\text{d} & \quad \langle 0, 0, 0, 0, - \rangle \quad \text{to} \quad \langle 0, 0, 0, 0, - \rangle & 1 \\
\text{e} & \quad \langle 0, 0, - , -, - \rangle \quad \text{to} \quad \langle 0, 0, - , -, - \rangle & 5 \\
\text{f} & \quad \langle 1, -, -, -, - \rangle \quad \text{to} \quad \langle 1, -, -, -, - \rangle & 8 \\
\text{g} & \quad \langle 0, 1, -, -, - \rangle \quad \text{to} \quad \langle 0, 1, -, -, - \rangle & 6 \\
\text{h} & \quad \langle 0, 0, 0, 1, - \rangle \quad \text{to} \quad \langle 0, 0, 0, 1, - \rangle & 2 \\
\text{i} & \quad \langle 0, 0, 0, - , - \rangle \quad \text{to} \quad \langle 0, 0, 0, - , - \rangle & 3
\end{align*}
\]
Small progress measures

\[
\begin{align*}
  a & \quad \langle 1, 0, 0, 0, 1 \rangle \quad \text{to} \quad \langle 1, 0, 0, 1, 1 \rangle \quad 028 \\
  b & \quad \langle 1, 0, 0, 1, - \rangle \quad \text{to} \quad \langle 1, 0, 0, 1, - \rangle \quad 28 \\
  c & \quad \langle 1, - , - , - , - \rangle \quad \text{to} \quad \langle 1, - , - , - , - \rangle \quad 728 \\
  d & \quad \langle 0, 0, 0, 0, - \rangle \quad \text{to} \quad \langle 0, 0, 0, 0, - \rangle \quad 1 \\
  e & \quad \langle 0, 0, - , - , - \rangle \quad \text{to} \quad \langle 0, 0, - , - , - \rangle \quad 5 \\
  f & \quad \langle 1, - , - , - , - \rangle \quad \text{to} \quad \langle 1, - , - , - , - \rangle \quad 8 \\
  g & \quad \langle 0, 1, - , - , - \rangle \quad \text{to} \quad \langle 0, 1, - , - , - \rangle \quad 6 \\
  h & \quad \langle 0, 0, 0, 1, - \rangle \quad \text{to} \quad \langle 0, 0, 0, 1, - \rangle \quad 2 \\
  i & \quad \langle 0, 0, 0, - , - \rangle \quad \text{to} \quad \langle 0, 0, 0, - , - \rangle \quad 3
\end{align*}
\]
Small progress measures

• All vertices are won by Odd
  No vertices are lifted to \( \top \)

• Strategy for Odd
  • from \( b \) to \( f \)
  • from \( d \) to \( e \)

\[
\begin{align*}
a & \langle 1,0,0,1,1 \rangle \\
b & \langle 1,0,0,1,- \rangle \\
c & \langle 1,-,-,-,-,- \rangle \\
d & \langle 0,0,0,0,- \rangle \\
e & \langle 0,0,-,-,-,- \rangle \\
f & \langle 1,-,-,-,-,- \rangle \\
g & \langle 0,1,-,-,-,- \rangle \\
h & \langle 0,0,0,1,- \rangle \\
i & \langle 0,0,0,-,-,- \rangle \\
\end{align*}
\]
Small progress measures

1 def spm(□):
2    ρ ← V ⇄ ⟨0, ..., 0⟩
3 while ρ(v) □ Lift(ρ, v) for some v : ρ ← ρ[ v ← Lift(ρ, v) ]
4    W□ ← \{ v | ρ(v) = ⊤ \}
5    W◇ ← \{ v | ρ(v) ≠ ⊤ \}
6    σ□ ← ( v ∈ W□ ∩ V□ ) ⇄ pick( \{ u ∈ E(v) | ρ(v) = Prog(ρ(w), pr(v)) \} )
7 return W□, W◇

\[
\text{Lift}(\rho, v) := \begin{cases} 
\max_{\square} \{ \text{Prog}(\rho(w), \text{pr}(v)) | w ∈ E(v) \} & v ∈ V□ \\
\min_{\square} \{ \text{Prog}(\rho(w), \text{pr}(v)) | w ∈ E(v) \} & v ∈ V◇ 
\end{cases}
\]

\[
\text{Prog}(m, p) := \begin{cases} 
\min_{\square} \{ m' ∈ M | m' ≡_p m \} & p \text{ is even} \\
\min_{\square} \{ m' ∈ M | m' ⊆_p m \} & p \text{ is odd}
\end{cases}
\]
Small progress measures

```python
1 def spm(\varnothing):
2     \rho \leftarrow V \mapsto \langle 0, \ldots, 0 \rangle
3     Z \leftarrow V
4     // use a queue or a stack
5     while Z \neq \emptyset :
6         v \leftarrow \text{pick}(Z)
7         Z \leftarrow Z \setminus \{v\}
8         if \rho(v) \sqsubseteq \text{Lift}(\rho, v) :
9             \rho \leftarrow \rho[v \mapsto \text{Lift}(\rho, v)]
10            Z \leftarrow Z \cup E^{-1}(v)
11     W_{\triangledown} \leftarrow \{ v \mid \rho(v) = T \}
12     W_{\square} \leftarrow \{ v \mid \rho(v) \neq T \}
13     \sigma_{\square} \leftarrow (v \in W_{\square} \cap V_{\square}) \mapsto \text{pick}(\{ u \in E(v) \mid \rho(v) = \text{Prog}(\rho(w), \text{pr}(v)) \})
14     return W_{\triangledown}, W_{\square}
```
Implementation notes

- Use a queue or stack to store “to do” vertices
- After lifting a vertex, add its predecessors to the queue (only once!)
- When lifting an **even priority vertex** to $\top$, decrease $n_p$ by 1
- Also compute odd measures (strategy for Even)
- **Advanced technique**: occasionally, compute the attractor to vertices in $Z$, any vertex not attracted and not $\top$ is won by the other player!
- Preprocessing: use **compression** and **SCC-decomposition** and **self-loop solving**.
Measures as tree navigation paths

Core idea

- A tuple \(\langle 4, 2, 3 \rangle\) can be a *navigation path* of a tree
- Follow branch 4, then branch 2, then branch 3
- Then:
  - the set of measures form a tree with \(n\) leaves and \(\lceil d/2 \rceil\) height
  - the measures **essentially** encode the current order between vertices
  - the exact numbers \(4, 2, 3\) are not important!
  - what matters is the order
- Example: \((1,2), (0,2), (1,1), (1,2), (2,1), (0,1), (1,0)\)
  - Draw tree corresponding to this set of navigation paths
  - Notice how the labels of the tree are irrelevant, only the order matters
Universal trees (see explanation by Fijalkow 2018)

A \((n,h)\)-universal tree is a tree that can embed all trees of height \(h\) and with \(n\) leaves.

The naive \((5,2)\)-universal tree of size 25

A \((5,2)\)-universal tree of size 11
Universal trees

A \((n,h)\)-universal tree is a tree that can embed all trees with height \(h\) and \(n\) leaves.

The tree on the left is embedded into the universal tree.
Measures as tree navigation paths

Universal trees

Simple algorithm:

- Split tree in three parts: Left, Middle, Right
- Such that $|Left| < n/2$ and $|Right| < n/2$
- Repeat left/right to obtain all branches, and repeat recursively...
Measures as tree navigation paths

Universal trees

Tree encoding:
- Instead of $\langle 4, 2, 3 \rangle$, encode as a tuple of bitstrings
- For example $\langle 100, 010, 011 \rangle$
Universal trees

Tree encoding:

- Instead of \(\langle 4, 2, 3 \rangle\), encode as a tuple of bitstrings
- For example \(\langle 100, 10, 11 \rangle\)
Universal trees

Succinct tree encoding:

- Encode as a tuple of bitstrings (empty allowed)
- Order on bits: $0 \preceq \varepsilon \preceq 1$
- Order on bitstrings: $0s \preceq s \preceq 1s$
  
  Example: $00 \preceq 0, 0 \preceq 01, 1 \preceq 10$

- Order on tuples: lexicographic, and shorter prefix is lower
  
  Example: $\langle 01, \varepsilon \rangle \preceq \langle 01, \varepsilon, 00 \rangle$, but $\langle 01, \varepsilon, 000 \rangle \npreceq \langle 1000, \varepsilon \rangle$
Measures as tree navigation paths

Universal trees

Succinct tree encoding:

- Prefix Left with 0, Right with 1, Middle with $\varepsilon$.
- For example $\langle 100, 10, 11 \rangle$

- Maximum bitstring length: 2 bits
Measures as tree navigation paths

Universal trees

Lifting in the succinct tree encoding
(notice: slightly different notation here)

Example of lifting $v_5$: it is pushed to the left in order to satisfy $v_5 \prec_3 v_7$ and $v_5 \prec_2 v_1$
Implementation notes

- Implementation is complicated.
- Core idea is the same: keep lifting vertices to the smallest higher measure, either the maximum (player Even) or the minimum (player Odd)
“Ordered” progress measures

Core idea

- Domain: \( _< 7 < 5 < 3 < 1 < 0 < 2 < 4 < 6 \)
- Tuples \( \langle i_{32}, i_{16}, i_8, i_4, i_2, i_1 \rangle \) encode so-called \( i \)-witnesses
- An \( i_k \)-witness encodes the existence of a path where Even (or Odd) dominates \( k \) times
- Example: 1213142321563212
- \( _ \) means “no such witness”
- 7 means a witness, but starting with odd 7
- 6 means a witness, starting with even 6
“Ordered” progress measures

Update rules

- \langle 7, _, _, _ \rangle \text{ and we see a 6: } \langle 7, _, _, 6 \rangle
- \langle 7, _, _, 6 \rangle \text{ and we see a 2: } \langle 7, _, 2, _ \rangle
- \langle 7, _, 2, _ \rangle \text{ and we see a 1: } \langle 7, _, 2, 1 \rangle
- \langle 7, _, 2, 1 \rangle \text{ and we see a 0: } \langle 7, _, 2, 0 \rangle
- \langle 7, _, 2, 0 \rangle \text{ and we see a 6: } \langle 7, 6, _, _ \rangle
- \langle 7, 6, _, _ \rangle \text{ and we see an 8: } \langle 8, _, _, _ \rangle

Problem

- Not quite monotone.
- Solution: “antagonistic update”. Given measure \( m \) and priority \( p \), compute \( \min\{\text{Prog}(m', p) \mid m' \sqsupseteq m\} \)
“Ordered” progress measures

Implementation notes

• See paper by Fearnley et al on arXiv
• See qpt.cpp in Oink
• It’s complicated...
Winner-controller winning cycles

Simple algorithm to find trivial winning regions

Algorithm

- For every vertex $v$ that is controlled by player $\alpha := \text{pr}(v) \mod 2$
- $Z, \sigma := \text{attract vertices in } \{u \in V_\alpha \mid \text{pr}(u) \leq \text{pr}(v)\}$ to $v$
  - Just backward DFS from $v$ via $\alpha$-vertices with $\leq$ priority
- If $Z$ is closed ($v$ is reached), then $Z$ is an $\alpha$-dominion with strategy $\sigma$; maximize $Z$ by attracting from the entire game to $Z$ and remove from the game

There are more optimal algorithms, employing SCC reductions, etc. See also Maks Verver’s MSc Thesis “Practical Improvements to Parity Game Solving” and fatal attractors of [Huth, Kwo, Piterman, 2014]
Strategy iteration

Strategy improvement/iteration overview

- Originates from policy iteration algorithms for Markov decision processes and similar algorithms for stochastic games.
- First parity game specific algorithm by Vöge and Jurdzinski in 2000
- Later numerous modified versions
  - better “best response” computation
  - smarter strategy selection heuristics (hoping to find one requiring polynomially many changes)
  - learning snares (kind of tangles): Fearnley 2011
- Suitable for parallel computation (e.g. van de Pol and Weber; Kandziora (2009) and Van de Berg (2010) on the Playstation 3; various GPU and multi-core implementations)
Strategy iteration

Core idea of strategy iteration

• Both players have a total strategy
  • strategy $\sigma$ for all $v \in V_{\Diamond}$
  • strategy $\tau$ for all $v \in V_{\Box}$

• These induce a single play $\pi$ for every $v \in V$

• Every play $\pi$ ends in a cycle

• **Play profile** $\rho: V \rightarrow \mathbb{M}$ assigns a value to $v$ based on $\pi$

• The value represents **how optimal are current strategies $\sigma$ and $\tau$?**

• Keep improving strategies until fixed point
  • Odd computes the best response to $\sigma$
  • Even uses $\rho$ to improve the strategy $\sigma$ **once**
  • Repeat

• Why improve against the best response? Because then each time you improve $\sigma$, you know that Odd could not find a better response
Strategy iteration

Algorithm

1. Start with some $\sigma$ for player 0
2. Compute the best response $\tau$ for player 1
   - Traditional approach: Bellman-Ford shortest path algorithm
   - [Fearnley 2017] proposes: use strategy iteration to compute $\tau$:
     1. Start with some $\tau$ for player 1 (e.g. previous $\tau$)
     2. Compute play profiles and switchable edges
     3. Select switchable edges for the next $\tau$
     4. Repeat until no more switchable edges
3. Compute the play profiles and the switchable edges (that would locally improve the valuation) for player 0
4. Select switchable edges for the next $\sigma$
   - Different proposed switching rules (can we do it in $P$ iterations)
5. Repeat from step 2 until no more switchable edges
Strategy iteration

Play profiles

- **Relevance order** $\prec$ (value is priority):
  - $u < v \iff \text{pr}(u) < \text{pr}(v)$
  - $\max_{\prec}(V) =$ highest priority vertex

- **Reward order** $\prec$ (value as seen from player 0):
  - $V_+ = \{ v \mid \text{pr}(v) \text{ is even} \}$ \hspace{1em} $V_- = \{ v \mid \text{pr}(v) \text{ is odd} \}$
  - $u < v \iff (u < v \land v \in V_+) \lor (v < u \land u \in V_-)$
  - $P < Q \iff P \neq Q \land \max_{\prec}(P \triangle Q) \in (Q \triangle V_-)$
    - highest vertex in symmetric difference is in $Q$ and even
    - highest vertex in symmetric difference is in $P$ and odd

- **Reward order** $\prec$ (alternative formulation)
  - $\text{rew}(v) := \text{pr}(v) \times (-1)^{\text{pr}(v)}$ (that is: negate if $\text{pr}(v)$ is odd)
  - $u < v \iff \text{rew}(u) < \text{rew}(v)$
Strategy iteration

Play profiles [VJ00]

- **Relevance** order $\prec$ and **reward** order $\preceq$
- Original play profile of [Vöge, Jurdzinski 2000]: tuple $\langle u, P, e \rangle$
  - $u_\pi$ is most relevant vertex in the loop of $\pi$: $u_\pi = \max <(\inf(\pi))$
  - $P_\pi$ is the set of vertices more relevant than $u_\pi$ in $\pi$
    (seen once in the prefix of $u_\pi$
  - $e_\pi$ is the number of vertices in $\pi$ before $u_\pi$

- $\langle u, P, e \rangle \prec \langle v, Q, f \rangle \iff$

$$
\begin{cases}
  u \prec v \lor \\
  (u = v \land P \prec Q) \lor \\
  (u = v \land P = Q \land v \in V_- \land e < f) \lor \\
  (u = v \land P = Q \land v \in V_+ \land e > f)
\end{cases}
$$

- A strategy is optimal in vertex $v$ if it selects the $\prec$-maximal successor in $E(v)$ for player 0 (or $\prec$-minimal for player 1)
- A strategy is optimal if it is optimal for all vertices
Strategy iteration

Play profiles [F17]

- Modify $\sigma$: now player Even is also allowed to *halt* the play (if the continuation is not favorable)
- Initially $\sigma$ is $\bot$ (halt) for all Even’s vertices
- Result: now every infinite play (cycle) is won by Even!
  - because otherwise Even would halt to avoid the losing cycle
  - except if Odd can win a cycle without any vertices of Even
- Requires preprocessing: remove winner-controlled winning cycles of Odd
  - or maybe: let Even force Odd vertices to halt instead...
- Play profile: $\top$ if $\pi$ is infinite; otherwise $\langle e_d, e_{d-1}, \ldots, e_1, e_0 \rangle$ with $e_p = |\{v \in \pi \mid \text{pr}(v) = p\}|$, i.e., count how often each priority $p$ is encountered in the finite path $\pi$
- Profile $X \prec Y$ if the highest different priority $p$ is either even and $X(p) < Y(p)$ or odd and $X(p) > Y(p)$; also $X \prec \top$ for all $X \neq \top$
Strategy iteration

Compute using a backward search from vertices where Even halts

1. **def** compute-valuations(∅, σ, τ):

2. \( \theta \leftarrow \sigma \cup \tau \) // for easier notation

3. \( Z \leftarrow \theta^{-1}(\bot) \) // where Even halts

4. \( \rho \leftarrow (V \mapsto \top) \) // initialize

5. **while** \( Z \neq \emptyset \):

6. \( v \leftarrow \text{pop}(Z) \) // pop any \( v \) from \( Z \)

7. \( m \leftarrow \begin{cases} \langle 0, \ldots, 0 \rangle & \theta(v) = \bot \\ \rho(\theta(v)) & \text{otherwise} \end{cases} \) // get successor profile

8. \( m(\text{pr}(v)) \leftarrow m(\text{pr}(v)) + 1 \) // update profile

9. \( \rho(v) \leftarrow m \) // set profile of \( v \)

10. \( Z \leftarrow Z \cup \theta^{-1}(v) \) // add predecessors to \( Z \)

11. **return** \( \rho \)

Implementation note

Two stages: first compute \( \theta^{-1} \), then do the backward search
Switching rule Greedy All Switches

Extend $\rho$ with a valuation of $\perp$; define $\rho$ over sets; define $\text{Best}_\alpha$ as the set of successors of $v$ with the optimal profile for player $\alpha$; define $\text{GreedyAll}_\alpha$ to update the strategy with a switchable edge (if current strategy is not optimal)

$$
\begin{align*}
\rho_\perp & := \rho \cup \{ \perp \mapsto \langle 0, \ldots, 0 \rangle \} \\
\rho_\perp(X) & := \{ \rho_\perp(x) \mid x \in X \} \\
\text{Best}_\square(\varnothing, \rho, v) & := \{ u \in E(v) \mid \rho(u) = \min_\prec \rho(E(v)) \} \\
\text{Best}_\Diamond(\varnothing, \rho, v) & := \{ u \in E(v) \cup \{ \perp \} \mid \rho(u) = \max_\prec \rho(E(v) \cup \{ \perp \}) \} \\
\text{GreedyAll}_\alpha(\varnothing, \sigma, \rho) & := V_\alpha \mapsto \\
& \begin{cases} 
\sigma(v) & \sigma(v) \in \text{Best}_\alpha(\varnothing, \rho, v) \\
\text{pick}(\text{Best}_\alpha(\varnothing, \rho, v)) & \text{otherwise}
\end{cases}
\end{align*}
$$
def si(O):
    σ ← (V⊙ ↦ ⊥), τ ← random strategy for Odd
    repeat
        repeat
            ρ ← compute-valuations(O, σ, τ)
            τ ← GreedyAll⋄(O, τ, ρ)
        until τ is unchanged
        σ ← GreedyAll⋄(O, σ, ρ)
        until σ is unchanged
    return W⋄, W□, σ, τ where W⋄ ← {v | ρ(v) = ⊤}, W□ ← V \ W⋄

Implementation note

After line 7, any vertex v with ρ(v) = ⊤, can be added to W⋄ already and does not need to be improved anymore; any vertex remaining in the end is then won by Odd
Strategy iteration

\[ \sigma: a \rightarrow \bot, \ c \rightarrow \bot, \ e \rightarrow \bot, \ f \rightarrow \bot, \ g \rightarrow \bot, \ h \rightarrow \bot, \ i \rightarrow \bot \]
Strategy iteration

σ: \( a \rightarrow \bot, \ c \rightarrow \bot, \ e \rightarrow \bot, \ f \rightarrow \bot, \ g \rightarrow \bot, \ h \rightarrow \bot, \ i \rightarrow \bot \)

best response \( \tau: \ b \rightarrow a \) and \( d \rightarrow c \)
Strategy iteration

\begin{align*}
\sigma: & \ a \rightarrow \bot, \ c \rightarrow \bot, \ e \rightarrow \bot, \ f \rightarrow \bot, \ g \rightarrow \bot, \ h \rightarrow \bot, \ i \rightarrow \bot \\
\text{best response } \tau: & \ b \rightarrow a \text{ and } d \rightarrow c \\
\sigma: & \ a \rightarrow b, \ c \rightarrow g, \ e \rightarrow \bot, \ f \rightarrow g, \ g \rightarrow h, \ h \rightarrow \bot, \ i \rightarrow \bot
\end{align*}
Strategy iteration

\[ a \rightarrow \bot, \ c \rightarrow \bot, \ e \rightarrow \bot, \ f \rightarrow \bot, \ g \rightarrow \bot, \ h \rightarrow \bot, \ i \rightarrow \bot \]

best response \( \tau \): \( b \rightarrow a \) and \( d \rightarrow c \)

\[ a \rightarrow b, \ c \rightarrow g, \ e \rightarrow \bot, \ f \rightarrow g, \ g \rightarrow h, \ h \rightarrow \bot, \ i \rightarrow \bot \]

best response \( \tau \): \( b \rightarrow f \) and \( d \rightarrow c \)
σ: \( a \rightarrow \perp, \ c \rightarrow \perp, \ e \rightarrow \perp, \ f \rightarrow \perp, \ g \rightarrow \perp, \ h \rightarrow \perp, \ i \rightarrow \perp \)

best response \( \tau: \ b \rightarrow a \) and \( d \rightarrow c \)

σ: \( a \rightarrow b, \ c \rightarrow g, \ e \rightarrow \perp, \ f \rightarrow g, \ g \rightarrow h, \ h \rightarrow \perp, \ i \rightarrow \perp \)

best response \( \tau: \ b \rightarrow f \) and \( d \rightarrow c \)

σ: \( a \rightarrow b, \ c \rightarrow b, \ e \rightarrow \perp, \ f \rightarrow g, \ g \rightarrow h, \ h \rightarrow \perp, \ i \rightarrow \perp \)
Strategy iteration

σ: a → ⊥, c → ⊥, e → ⊥, f → ⊥, g → ⊥, h → ⊥, i → ⊥
best response τ: b → a and d → c

σ: a → b, c → g, e → ⊥, f → g, g → h, h → ⊥, i → ⊥
best response τ: b → f and d → c

σ: a → b, c → b, e → ⊥, f → g, g → h, h → ⊥, i → ⊥
best response τ: b → f and d → e
Strategy iteration

$$\sigma: \ a \to \bot, \ c \to \bot, \ e \to \bot, \ f \to \bot, \ g \to \bot, \ h \to \bot, \ i \to \bot$$

best response $$\tau: \ b \to a$$ and $$d \to c$$

$$\sigma: \ a \to b, \ c \to g, \ e \to \bot, \ f \to g, \ g \to h, \ h \to \bot, \ i \to \bot$$

best response $$\tau: \ b \to f$$ and $$d \to c$$

$$\sigma: \ a \to b, \ c \to b, \ e \to \bot, \ f \to g, \ g \to h, \ h \to \bot, \ i \to \bot$$

best response $$\tau: \ b \to f$$ and $$d \to e$$

Odd wins entire game with strategy $$\tau$$
Fixed point iteration

Core idea

- We can solve $\mu$-calculus model checking by solving the fixed points explicitly
- We can solve $\mu$-calculus model checking by solving a parity game
- Here: we solve parity games by via a fixed point iteration
- Via weak alternating automata [Kupfermann, Vardi, 1998]
- APT implementation [Di Stasio, Murano, Perelli, Vardi, 2016]
- Via $\mu$-calculus: [Bruse, Falk, Lange, 2014]
- Quite fast for games with low number of priorities
Fixed point iteration

Core idea

• “Using fixed points, update winning regions using a 1-step attractor”
• Record “distraction sets” $Z_p \subseteq V_p$ \hspace{1cm} ($V_p = \{v \mid \text{pr}(v) = p\}$)
• A vertex is a distraction if:
  • it has even priority and is won by Odd
  • it has odd priority and is won by Even
• Monotonically update $Z_0$, then $Z_1$, etc.
• When adding vertices to $Z_p$, reset $Z_{<p}$ to $\emptyset$
Fixed point iteration

Given some set of distracted vertices $Z = Z_0 \cup Z_1 \cup \cdots \cup Z_d$,

$$\text{winner}(v, Z) := \begin{cases} 
\text{pr}(v) \mod 2 & v \notin Z \\
1 - (\text{pr}(v) \mod 2) & v \in Z 
\end{cases}$$

$$\text{next}(v, Z) := \begin{cases} 
0 & v \in V_\Diamond \land \exists u \in E(v) : \text{winner}(u, Z) = 0 \\
1 & v \in V_\Diamond \land \forall u \in E(v) : \text{winner}(u, Z) = 1 \\
1 & v \in V_\Box \land \exists u \in E(v) : \text{winner}(u, Z) = 1 \\
0 & v \in V_\Box \land \forall u \in E(v) : \text{winner}(u, Z) = 0 
\end{cases}$$
def fpi(∅):
    p ← 0  // start with lowest priority
    Z ← ∅  // start with no distractions
    while p ≤ d:
        Y ← {v ∈ V_p \ Z | next(v, Z) ≠ pr(v) mod 2}  // distractions
        if Y ≠ ∅:
            Z ← Z ∪ Y  // update current fixed point Z_p
            Z ← Z \ {v | pr(v) < p}  // reset all lower fixed points
            p ← 0  // continue with lowest priority
        else:
            p ← p + 1  // fixed point, continue higher
    return W_◇, W_◇ where W_◇ ← {v | winner(v, Z) = 0}, W_◇ ← V \ W_◇

Note: algorithm does not give a strategy (see [BFL14] for a method)!
Fixed point iteration

1 def fpi(Ω):
   /* assume vertices are sorted by priority, V(i) for ith vertex */
2  Z ← V ↦→ 0  // start with no distractions
3  i ← 0  // start with lowest vertex
4  p ← pr(V(i))  // start with lowest priority
5  Chg ← False  // whether Z_p is updated
6  while True:
7      if i = n ∨ pr(V(i)) ≠ p :
8          if Chg :
9              Z ← Z[{v | pr(v) < p} ↦→ 0]  // reset all lower vertices
10             goto 3  // restart with lowest vertex
11          elif i = n :
12              return {v | winner(v, Z) = 0}, {v | winner(v, Z) = 1}
13          else:
14              p ← pr(V(i))  // Z_p not updated; continue
15      else:
16          if ¬Z[i] ∧ next(V(i), Z) ≠ pr(V(i)) mod 2 :
17              Z[i] ← 1  // ith vertex is distraction
18              Chg ← True  // mark that Z_p is updated
19              i ← i + 1
Some notes...

- That was my own version of the fixed point algorithm
- **To prove:** that it is correct
- **To show:** that it is equivalent to [BFL14] and [KV98] and [dSMPV16]
- **To study:** whether [BFL14] also leads to a method of finding strategies
- Implementation can be a tight loop with $Z$ implemented as a bit vector, and all vertices sorted by priority... going from low to high, and resetting all lower priority vertices plus restarting the loop whenever some $Z[v]$ is set
h, g (reset h)
h, c (reset h, g)
h, g (reset h)
h, f (reset h, g, c)
b, h, a, g (reset b, h, a)
b, h, a