Alpaga

A Tool for Solving Parity Games with Imperfect Information

Dietmar Berwanger[1] Krishnendu Chatterjee[2]
Martin De Wulf[3] Laurent Doyen[3,4] Tom Henzinger[4]

[1] ENS Cachan [2] UC Santa Cruz
[3] ULB Brussels [4] EPFL Lausanne

TACAS 2009

Why imperfect information ?

Example

```
void main () {
     int got_lock = 0;
     do {
1:         if (*) {
2:              lock ();
3:              got_lock++;
           }
4:         if (got_lock != 0) {
5:              unlock ();
           }
6:         got_lock--;
     } while (*);
}
```

```
void lock () {
     assert(L == 0);
     L = 1;
}
```

```
void unlock () {
     assert(L == 1);
     L = 0;
}
```

Example

```
void main () {
    int got_lock = 0;
    do {
1:      if (*) {
2:          lock ();
3:          got_lock++;
        }
4:      if (got_lock != 0) {
5:          unlock ();
        }
6:      got_lock--;
    } while (*);
}
```

Wrong!

```
void lock () {
    assert(L == 0);
    L = 1;
}
```

```
void unlock () {
    assert(L == 1);
    L = 0;
}
```

Example

```
void main () {
     int got_lock = 0;
     do {
1:         if (*) {
2:              lock ();
3:              s0 | s1 | inc | dec;
           }
4:         if (got_lock != 0) {
5:              unlock ();
           }
6:         s0 | s1 | inc | dec;
     } while (*);
}
```

```
void lock () {
     assert(L == 0);
     L = 1;
}
```

```
void unlock () {
     assert(L == 1);
     L = 0;
}
```

s0 ≡ got_lock = 0
s1 ≡ got_lock = 1
Inc ≡ got_lock++
dec ≡ got_lock--

Example

```
void main () {
      int got_lock = 0;
      do {
1:          if (*) {
2:              lock ();
3:              s0 | s1 | inc | dec;
            }
4:          if (got_lock != 0) {
5:              unlock ();
            }
6:          s0 | s1 | inc | dec;
      } while (*);
}
```

```
void lock () {
      assert(L == 0);
      L = 1;
}
```

```
void unlock () {
      assert(L == 1);
      L = 0;
}
```

Repair/synthesis as a game:

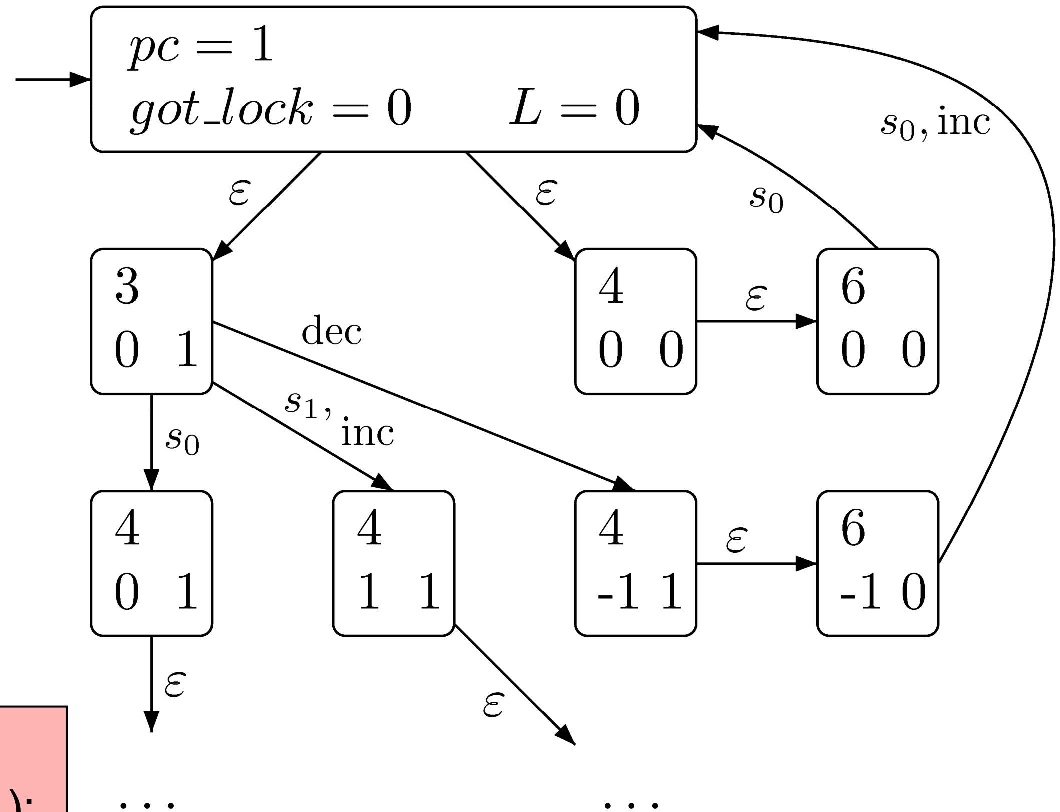
- System vs. Environment
- Turn-based game graph
- ω-regular objective

Example

```
    void main () {
        int got_lock = 0;
        do {
1:          if (*) {
2:              lock ();
3:              s0 | s1 | inc | dec;
            }
4:          if (got_lock != 0) {
5:              unlock ();
            }
6:          s0 | s1 | inc | dec;
        } while (*);
    }
```

```
void lock () {
    assert(L == 0);
    L = 1;
}
```

```
void unlock () {
    assert(L == 1);
    L = 0;
}
```



Example

```
     void main () {
         int got_lock = 0;
         do {
1:           if (*) {
2:               lock ();
3:               s0 | s1 | inc | dec;
             }
4:           if (got_lock != 0) {
5:               unlock ();
             }
6:           s0 | s1 | inc | dec;
         } while (*);
     }
```

```
void lock () {
    assert(L == 0);
    L = 1;
}
```

```
void unlock () {
    assert(L == 1);
    L = 0;
}
```

$pc = 1$

$got\_lock = 0 \qquad L = 0$

s_0, inc

A winning strategy may use the value of *L* to update *got_lock* accrodingly:

- if *L*==0 then play s0 (*got_lock* = 0)
- if *L*==1 then play s1 (*got_lock* = 1)

ε

$\cdots \qquad \cdots$

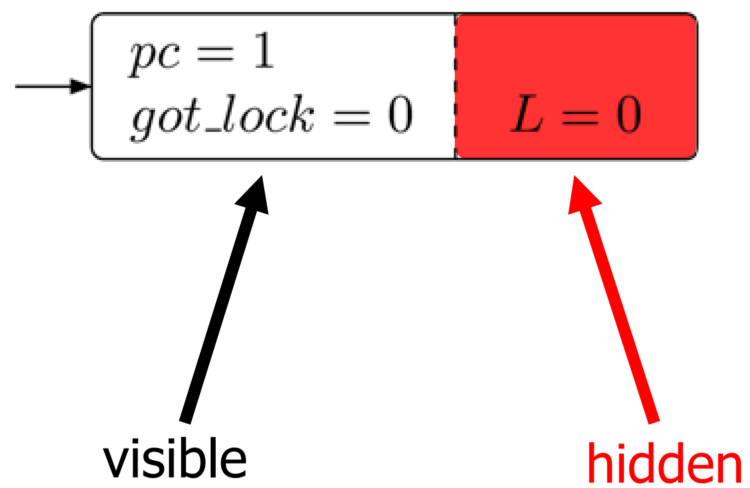
Example

```
void main () {
    int got_lock = 0;
    do {
1:      if (*) {
2:          lock ();
3:          s0 | s1 | inc | dec;
        }
4:      if (got_lock != 0) {
5:          unlock ();
        }
6:      s0 | s1 | inc | dec;
    } while (*);
}
```

```
void lock () {
    assert(L == 0);
    L = 1;
}
```

```
void unlock () {
    assert(L == 1);
    L = 0;
}
```

Repair/synthesis as a game of imperfect information:



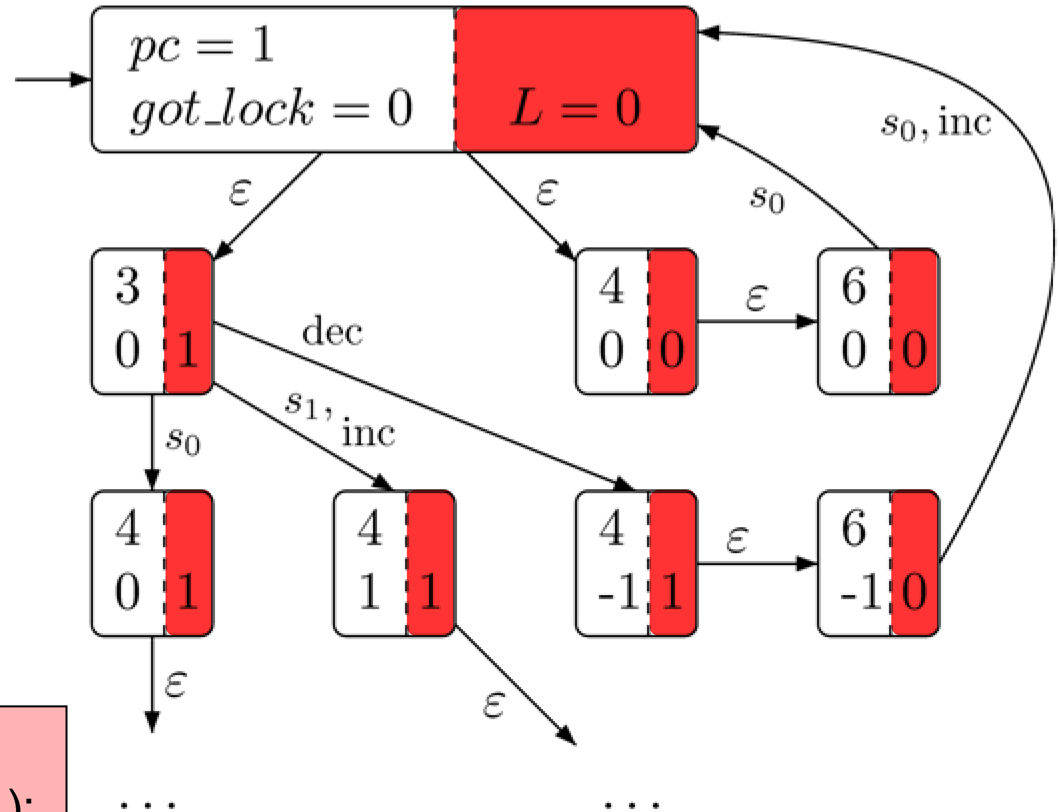
States that differ only by the value of *L* have the same observation

Example

```
void main () {
    int got_lock = 0;
    do {
1:        if (*) {
2:            lock ();
3:              s0 | s1 | inc | dec;
          }
4:        if (got_lock != 0) {
5:            unlock ();
          }
6:          s0 | s1 | inc | dec;
    } while (*);
}
```

```
void lock () {
    assert(L == 0);
    L = 1;
}
```

```
void unlock () {
    assert(L == 1);
    L = 0;
}
```

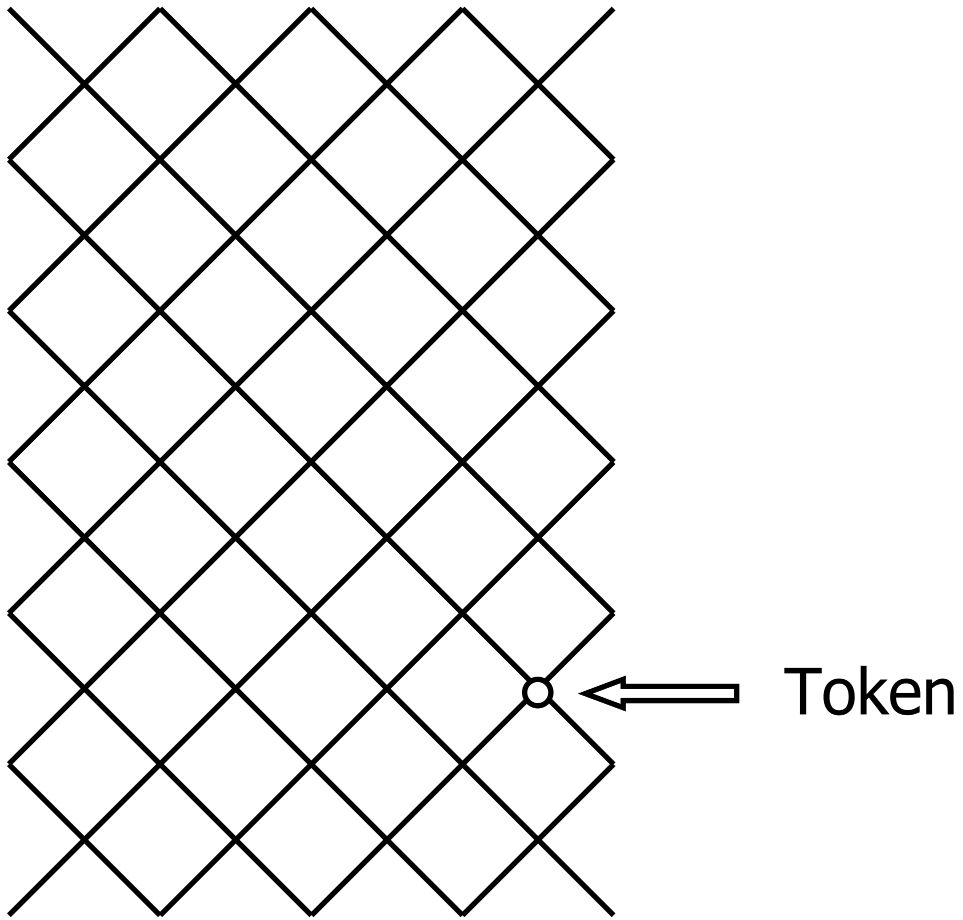


Why Imperfect Information ?

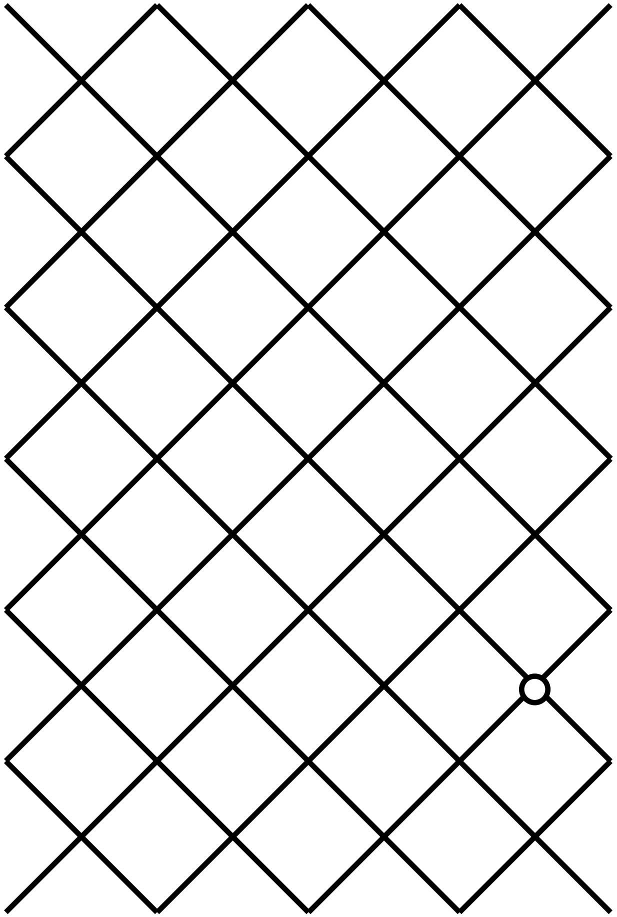
- Program repair/synthesis
- Distributed synthesis of processes with public/private variables
- Synthesis of robust controllers
- Synthesis of automata specifications
- Decision problems in automata theory
- Planning with partial observabillity, information flow secrecy, …

A model of imperfect information

Imperfect information

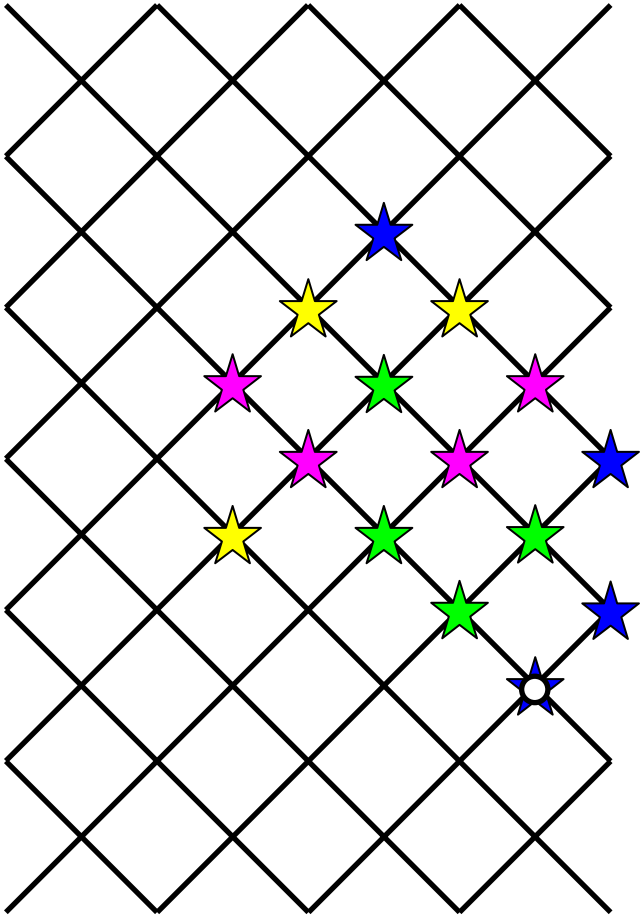


Imperfect information



System: actions ↓,↑

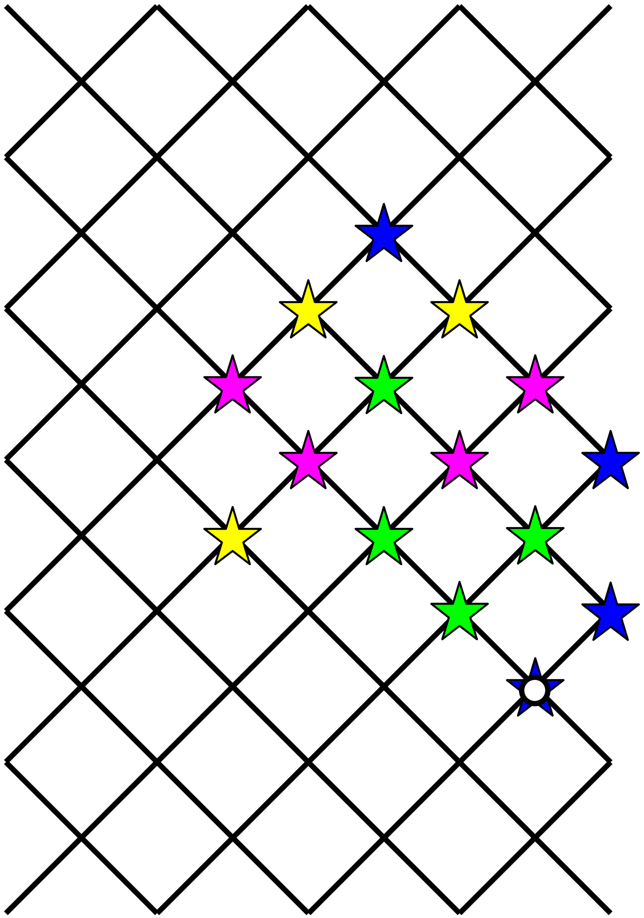
Imperfect information



System: actions ↓,↑

Environment: observations ★, ★, ★, ★

Imperfect information

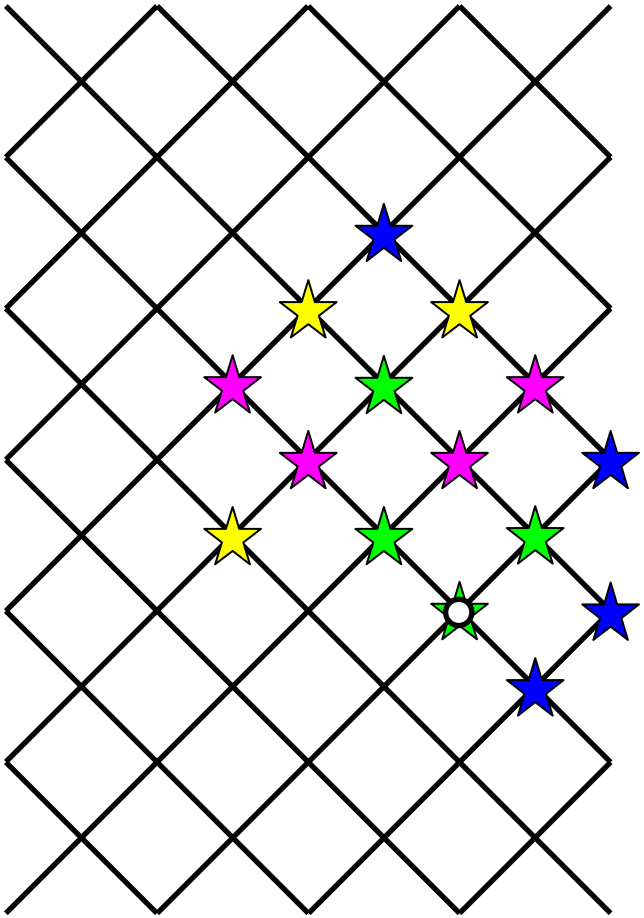


System: actions ↓,↑

Environment: observations ★, ★, ★, ★

A play: ★

Imperfect information



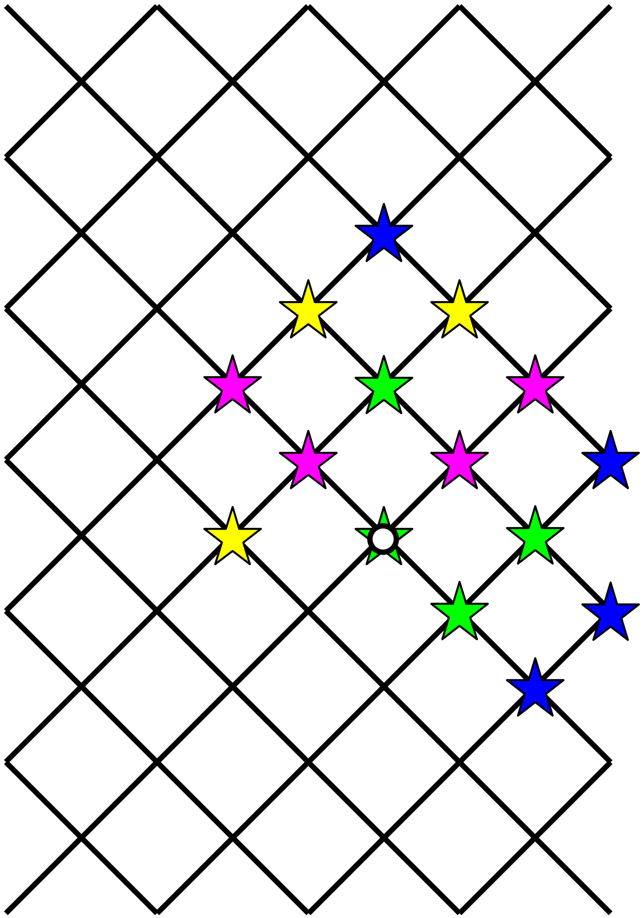
System: actions ↓,↑

Environment: observations

★, ★, ★, ★

A play: ★ ↑ ★

Imperfect information



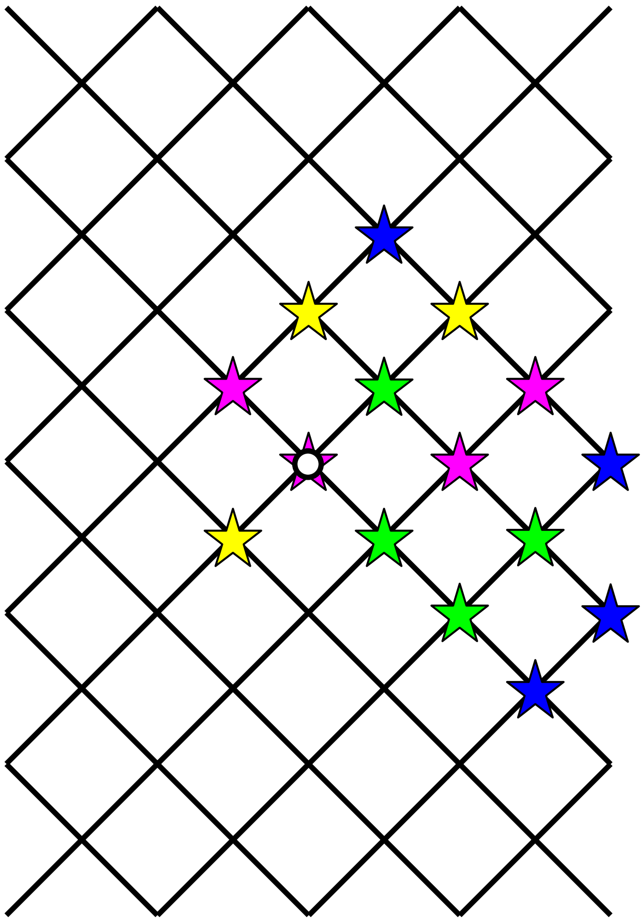
System: actions ↓,↑

Environment: observations

★, ★, ★, ★

A play: ★ ↑ ★ ↑ ★

Imperfect information



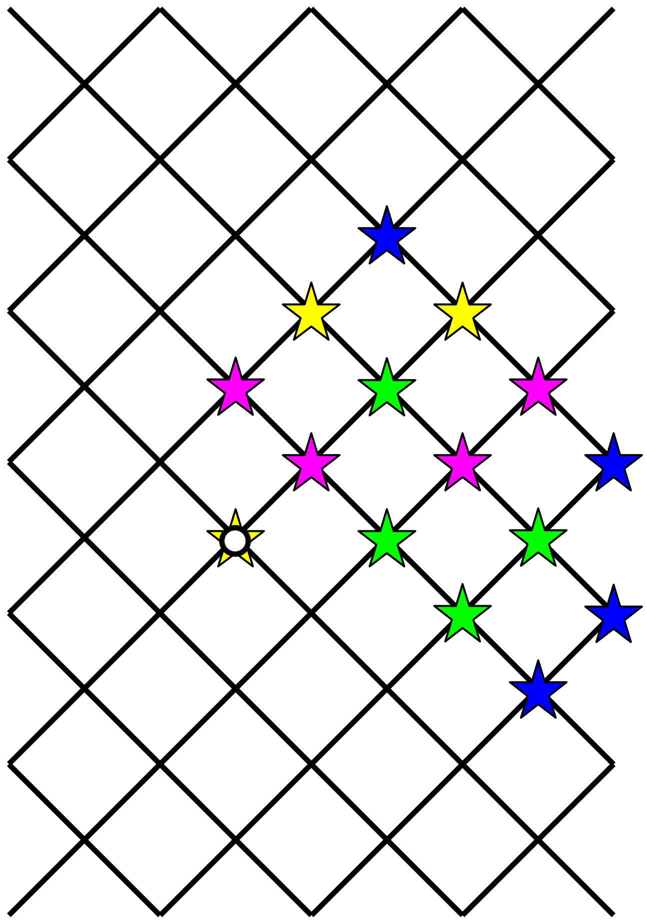
System: actions ↓,↑

Environment: observations

★, ★, ★, ★

A play: ★ ↑ ★ ↑ ★ ↑ ★

Imperfect information



System: actions ↓,↑

Environment: observations

★, ★, ★, ★

A play: ★ ↑ ★ ↑ ★ ↑ ★ ↓ ★

Objectives

- Reachability: eventually observe a good event
- Safety: never observe a bad event

- Parity: observation priorities – least one seen infinitely often is even
  - nested reachability & safety
  - generic for ω-regular specifications

Parity games with imperfect information

Questions:

- decide if System wins from the initial state
- construct a winning strategy

Assumptions:

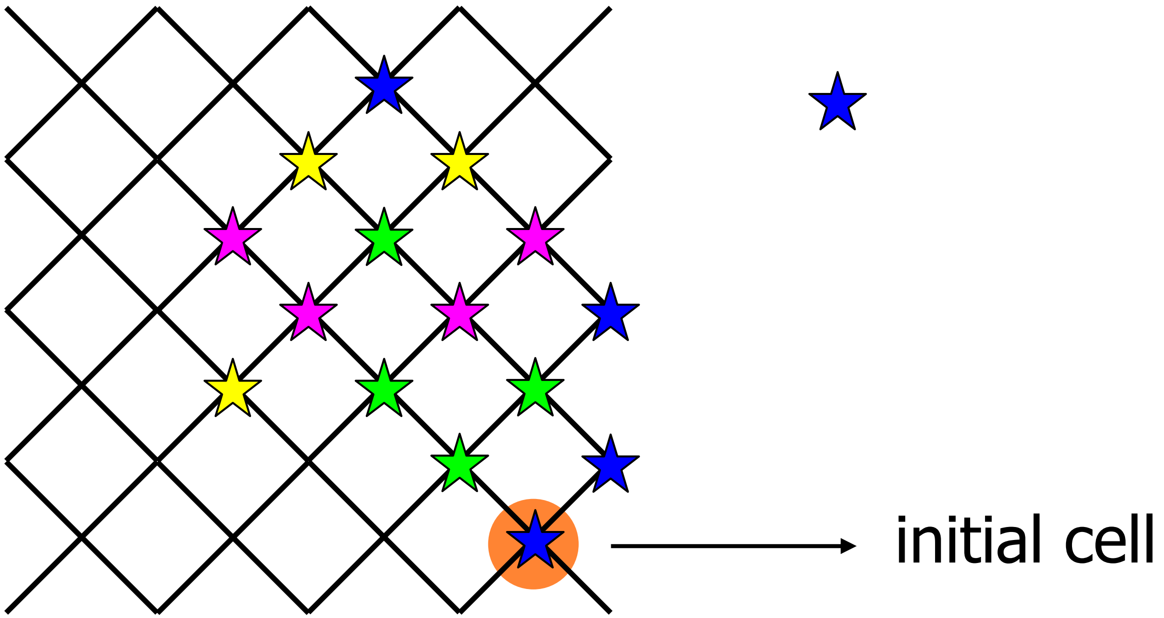
- Visible objective
- Sure-winning

Algorithms

Classical solution

Powerset construction [Reif84]:

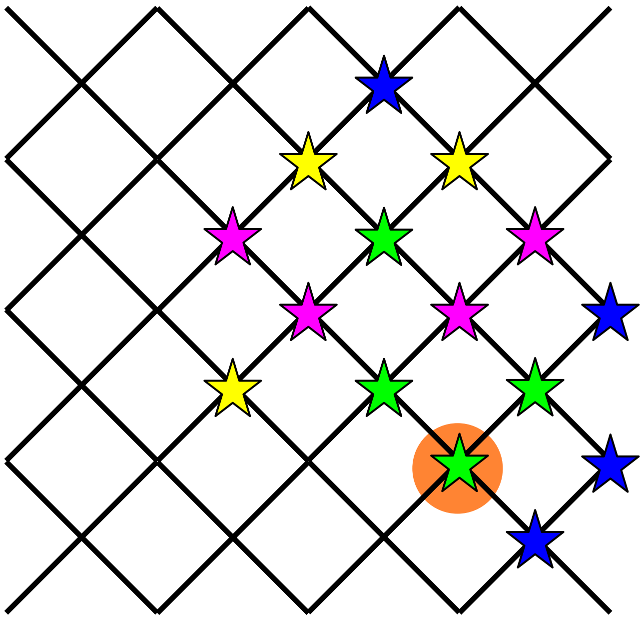
- keeps track of the knowledge of System
- yields equivalent game of perfect information



Classical solution

Powerset construction [Reif84]:

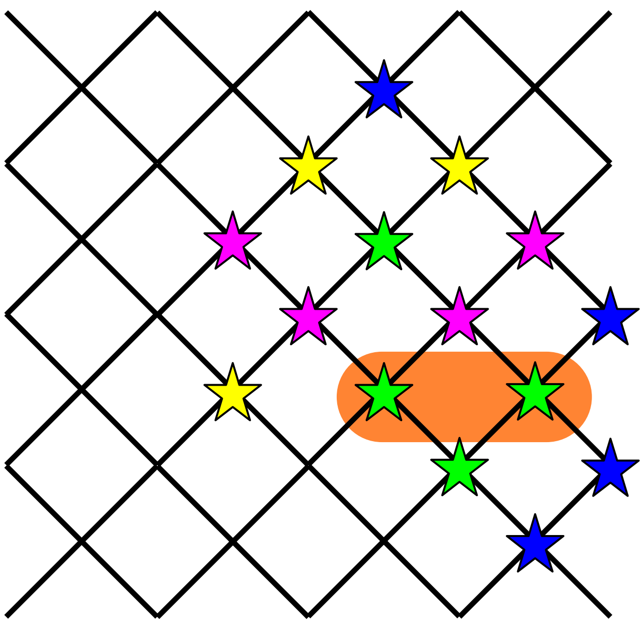
- keeps track of the knowledge of System
- yields equivalent game of perfect information



Classical solution

Powerset construction [Reif84]:

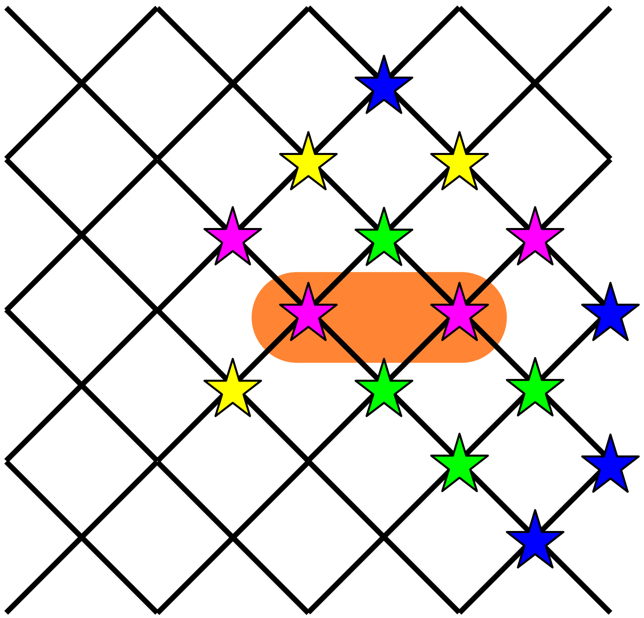
- keeps track of the knowledge of System
- yields equivalent game of perfect information

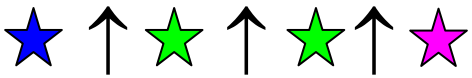


Classical solution

Powerset construction [Reif84]:

- keeps track of the knowledge of System
- yields equivalent game of perfect information

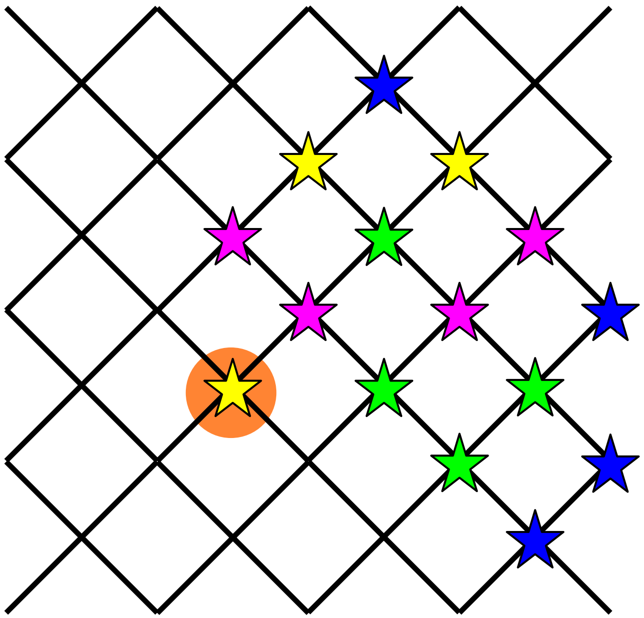


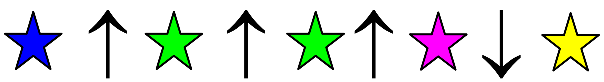


Classical solution

Powerset construction [Reif84]:

- keeps track of the knowledge of System
- yields equivalent game of perfect information





Classical solution

Powerset construction [Reif84]:

- keeps track of the knowledge of System
- yields equivalent game of perfect information

Memoryless strategies (in perfect-information) translate to finite-memory strategies

(memory automaton tracks set of possible positions)

Complexity

- Problem is EXPTIME-complete
  (even for safety and reachability)
- Exponential memory might be needed

The powerset solution [Reif84]

- is an exponential construction
- is not on-the-fly
- is independent of the objective

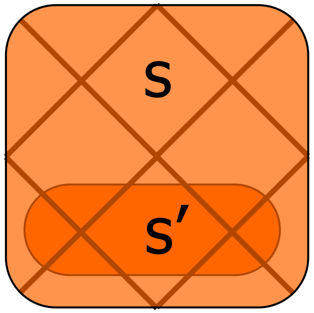
Can we do better ?

Antichains

Antichains

- Winning knowledge-sets are downward-closed:

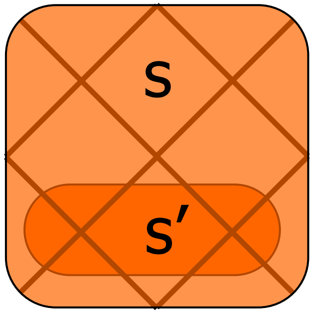
If System wins from s, then she also wins from s'



Antichains

- Winning knowledge-sets are downward-closed:

If System wins from s, then she also wins from s′



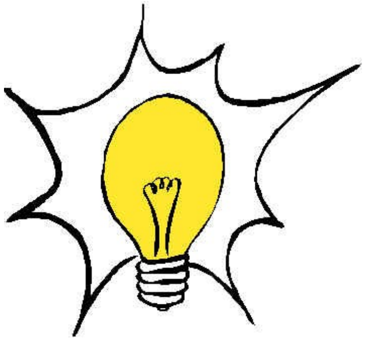
- Useful operations preserve downward-closedness

$\cap$, $\cup$, Cpre

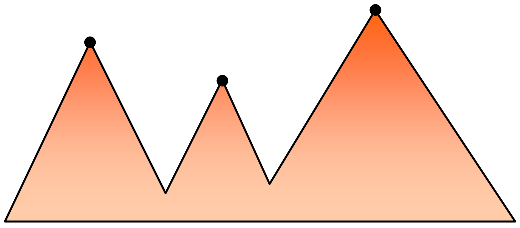
Cpre(X) = {Y from which System can force the play into X}

Antichains

- Winning knowledge-sets are downward-closed
- Useful operations preserve downward-closedness



Compact representation using maximal elements $\rightarrow$ Antichains



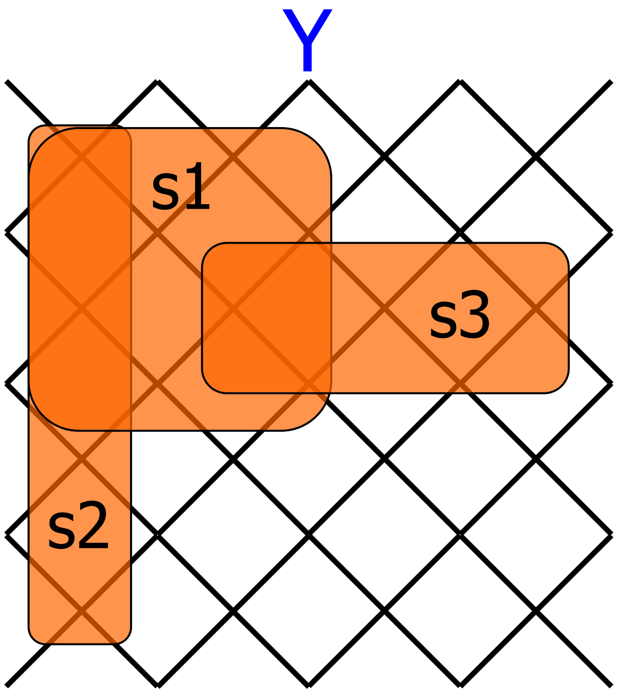
Antichain algorithm

- [CSL'06] computes winning sets of positions as a μ-calculus formula over the lattice of antichains

Antichain algorithm

- [CSL'06] computes winning sets of positions as a μ-calculus formula over the lattice of antichains

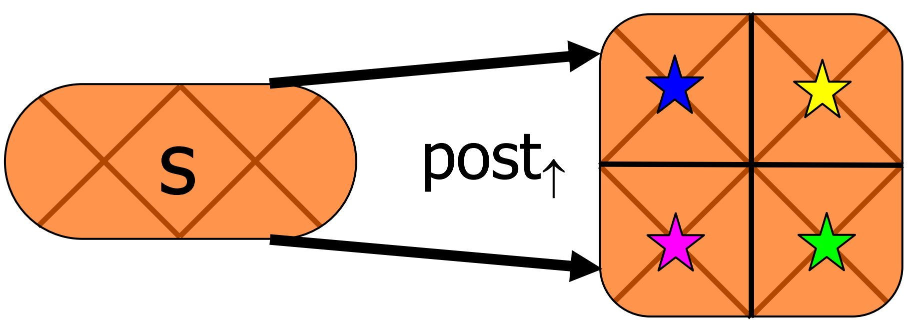
CPre(Y) = ?

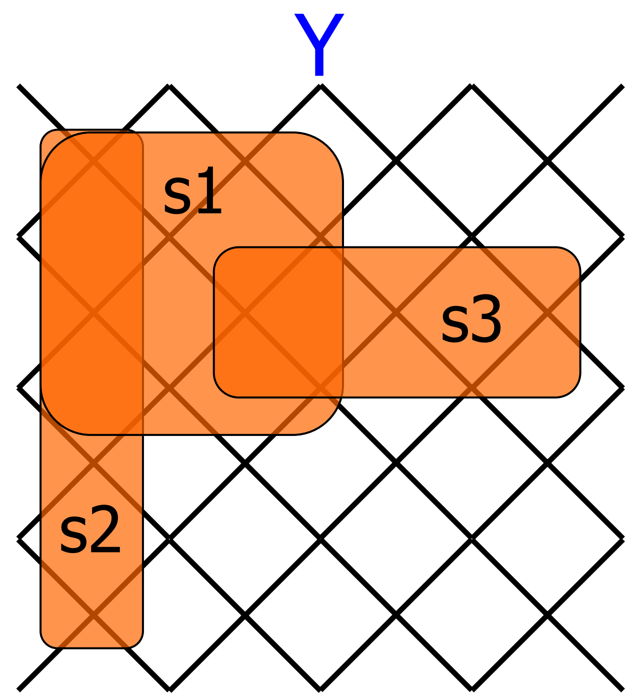


Y = {s1, s2, s3} set of winning positions so far

Antichain algorithm

- [CSL'06] computes winning sets of positions as a μ-calculus formula over the lattice of antichains



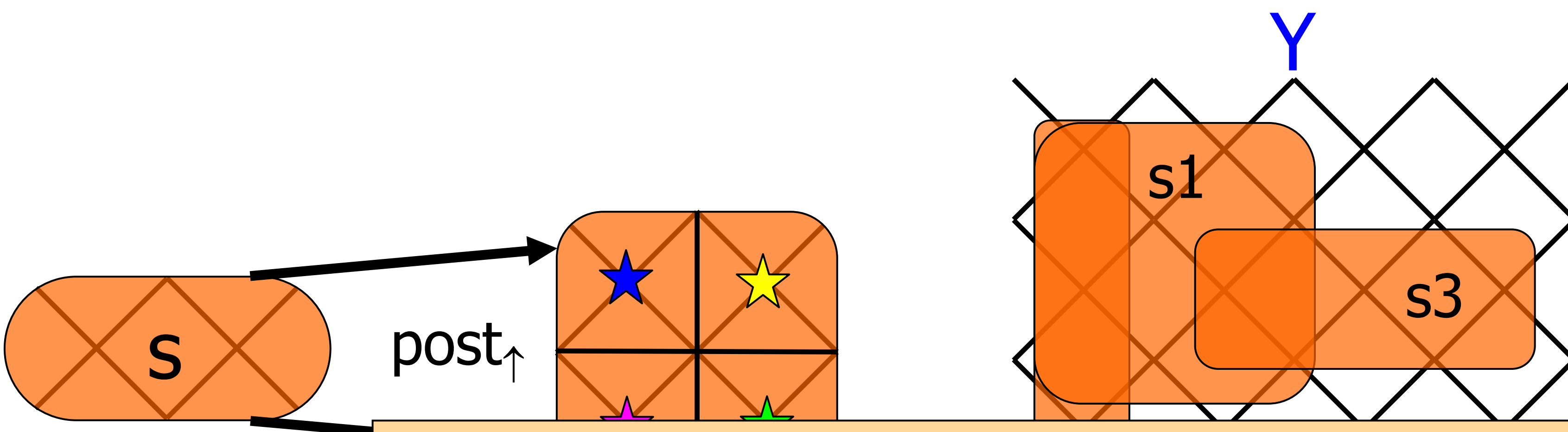


$s \in \mathrm{CPre}(Y)$

if for all ☆, there exists a set in Y that contains $\mathrm{post}_{\uparrow}(s) \cap$ ☆

Antichain algorithm

- [CSL'06] computes winning sets of positions as a μ-calculus formula over the lattice of antichains



- combinatorially hard to compute
- implemented using BDDs

s $\in$ CPre(Y)

if for all ☆, there exists a set in Y that contains post$_\uparrow$(s) $\cap$☆

Antichain algorithm

- [CSL’06] computes winning sets of positions as a μ-calculus formula over the lattice of antichains

- [Concur’08] computes winning strategy recursively for a combination of safety & reachability objectives

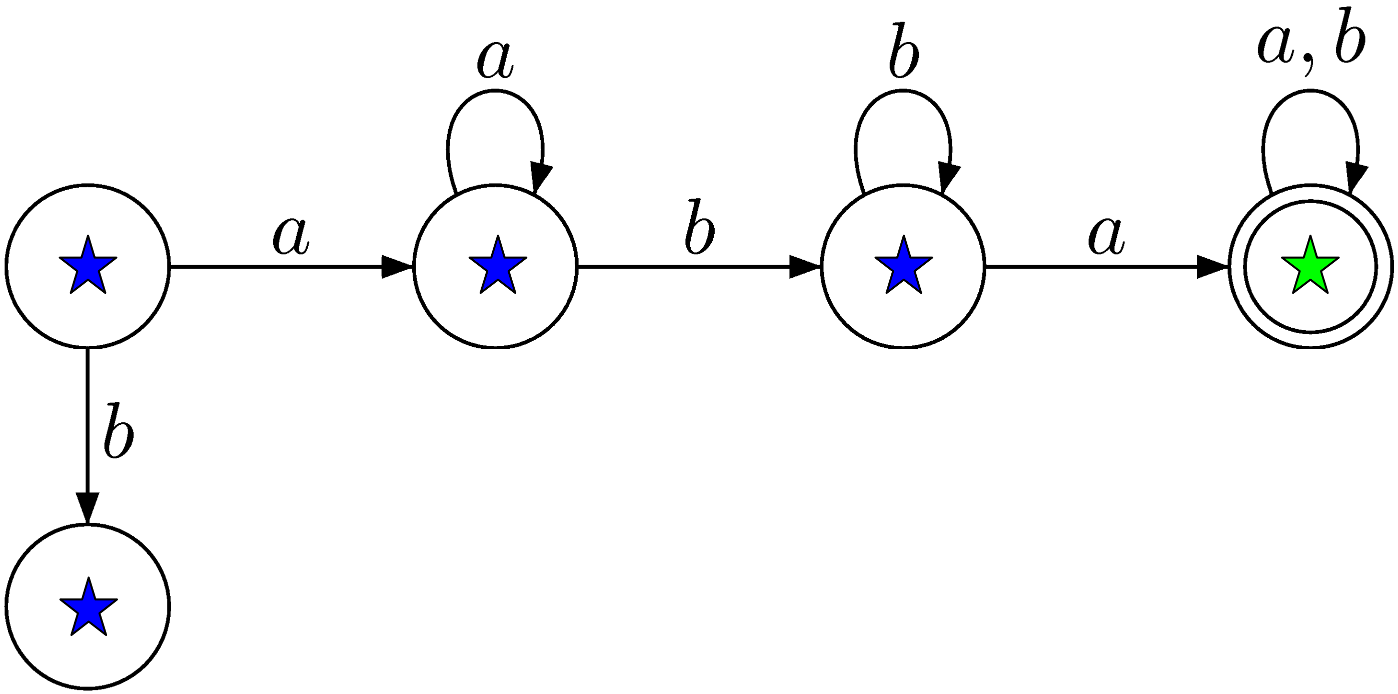
Antichain algorithm

- [CSL'06] computes winning sets of positions as a μ-calculus formula over the lattice of antichains

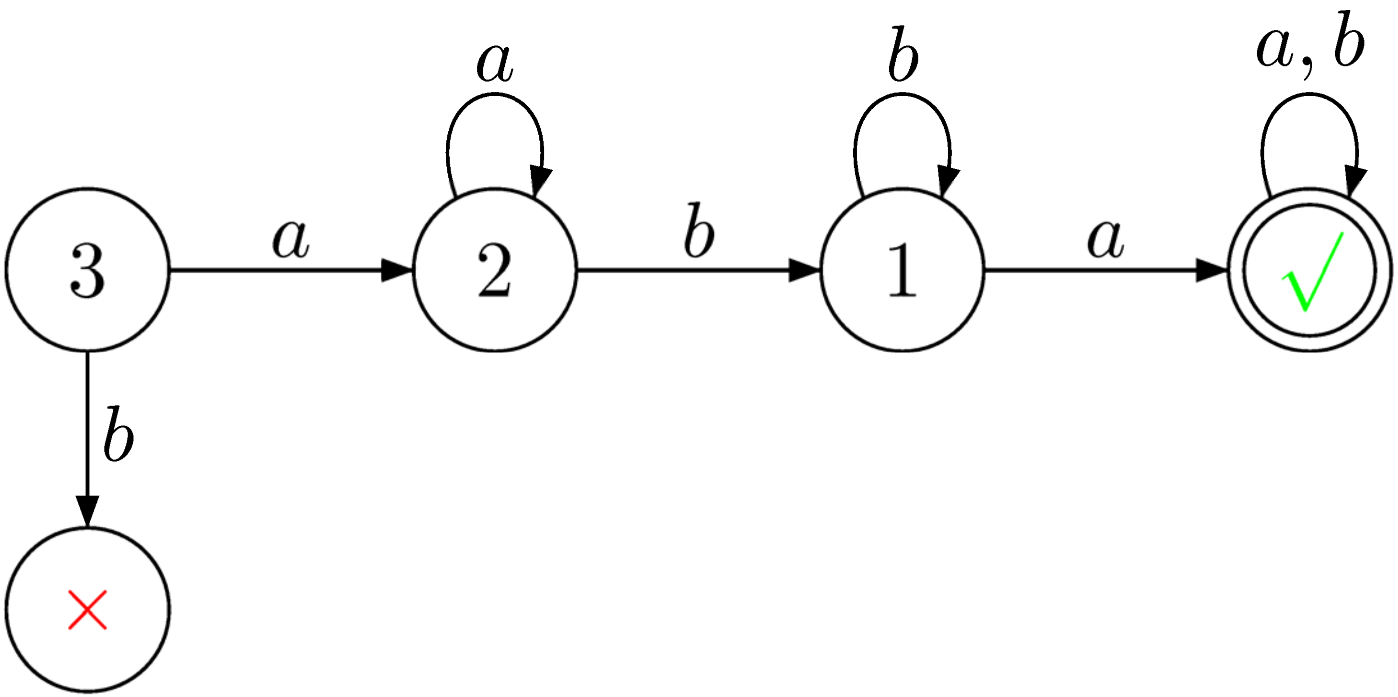
- [Concur'08] computes winning strategy recursively for a combination of safety & reachability objectives

    - Safety: extract strategy from fixpoint
    - Reachability: fixpoint is not sufficient

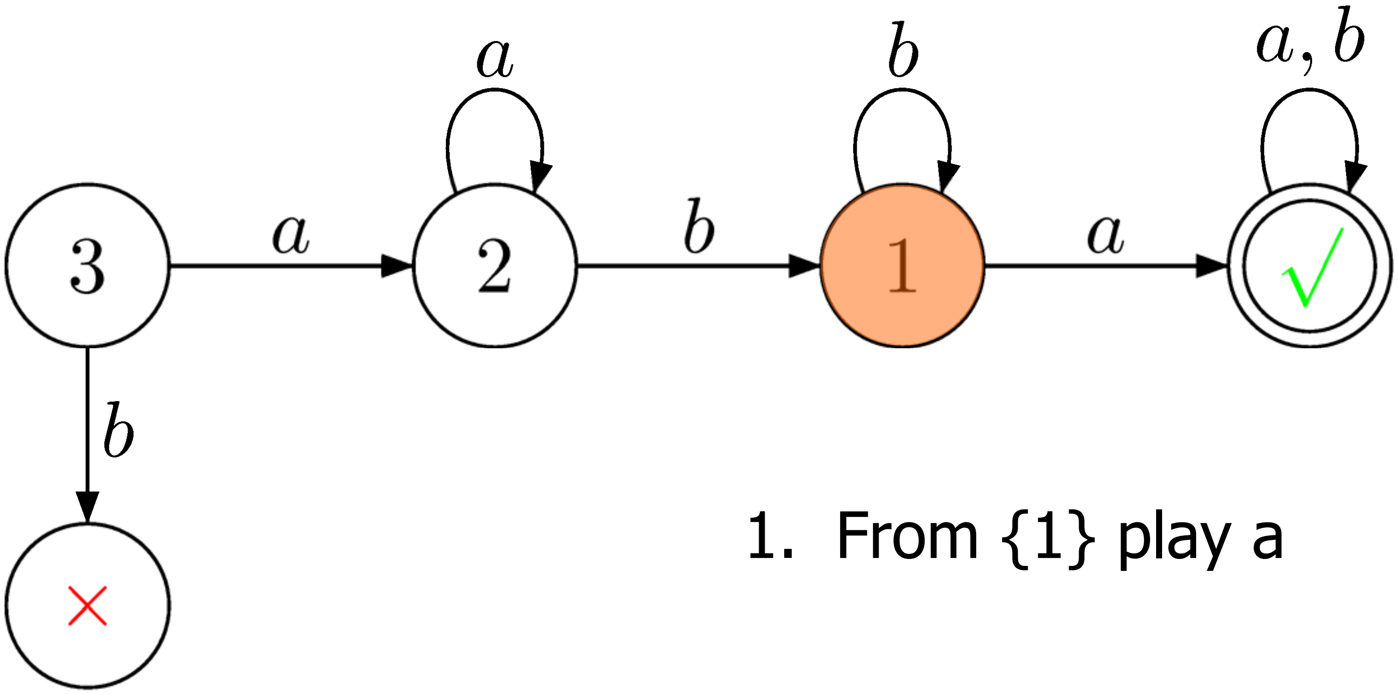
Reachability



Reachability

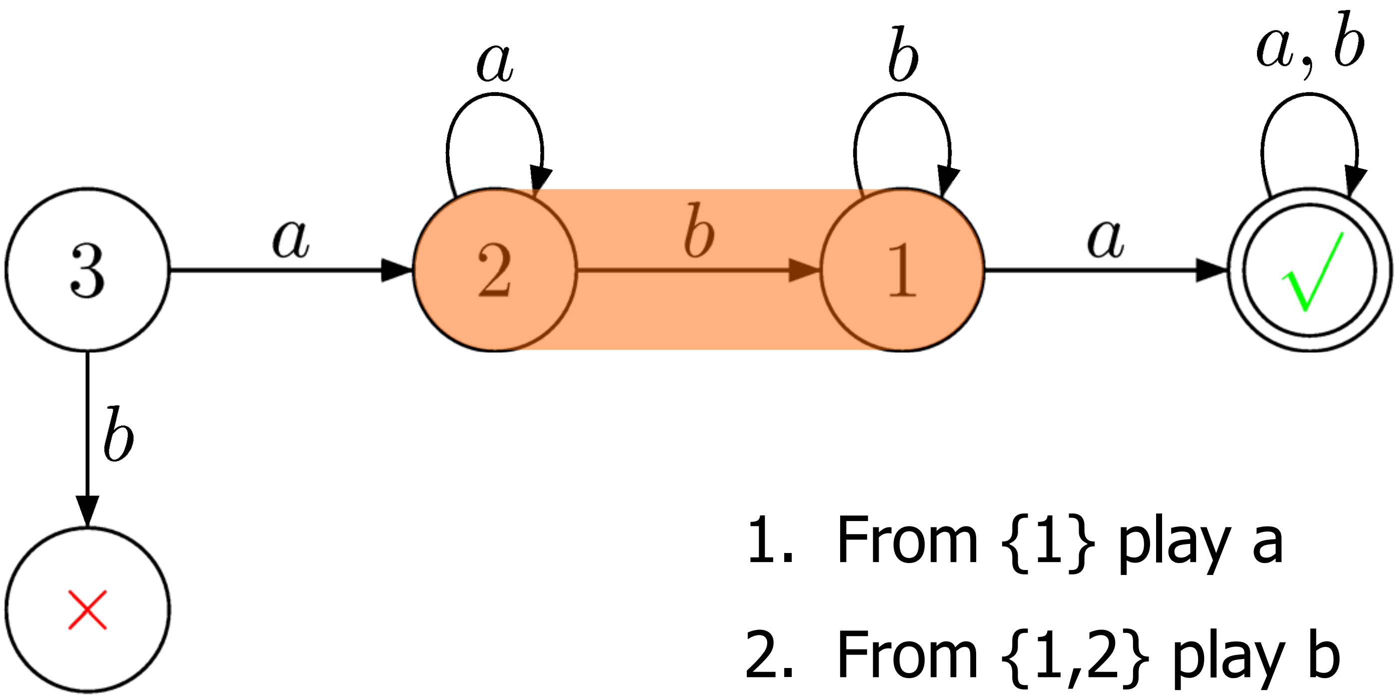


Reachability



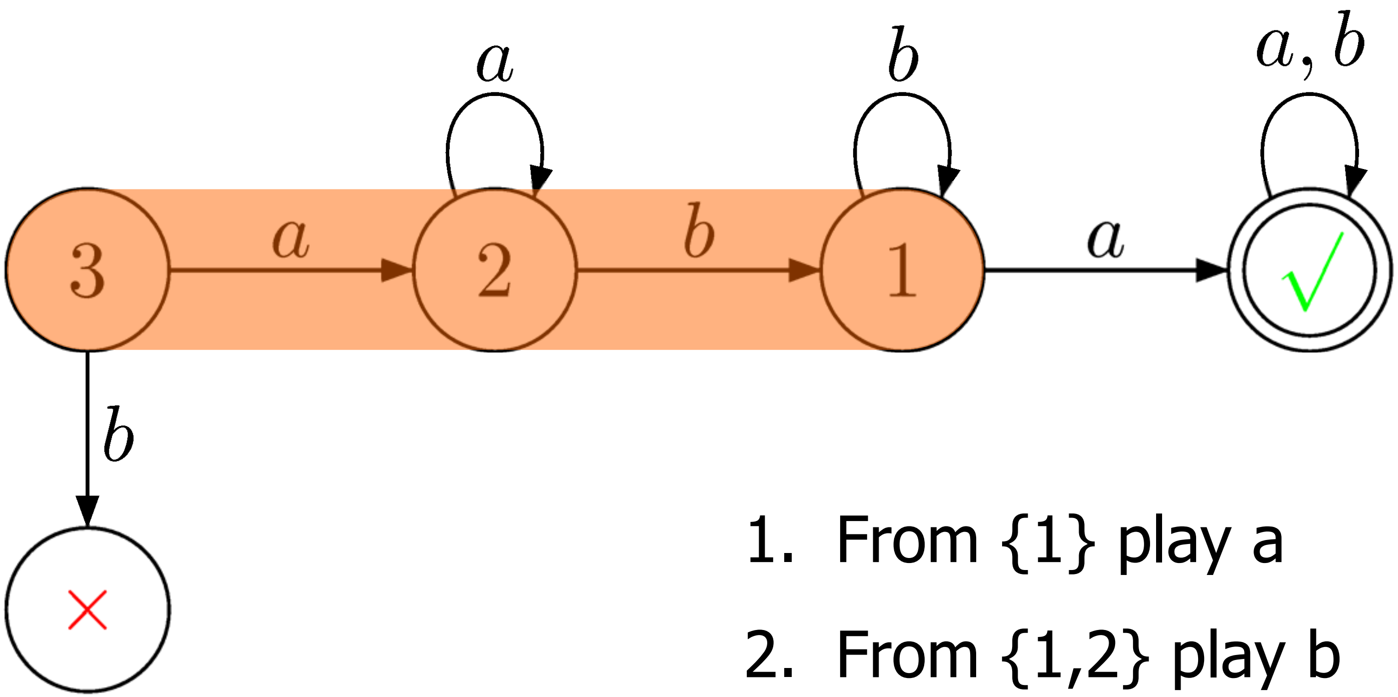
1. From {1} play a

Reachability



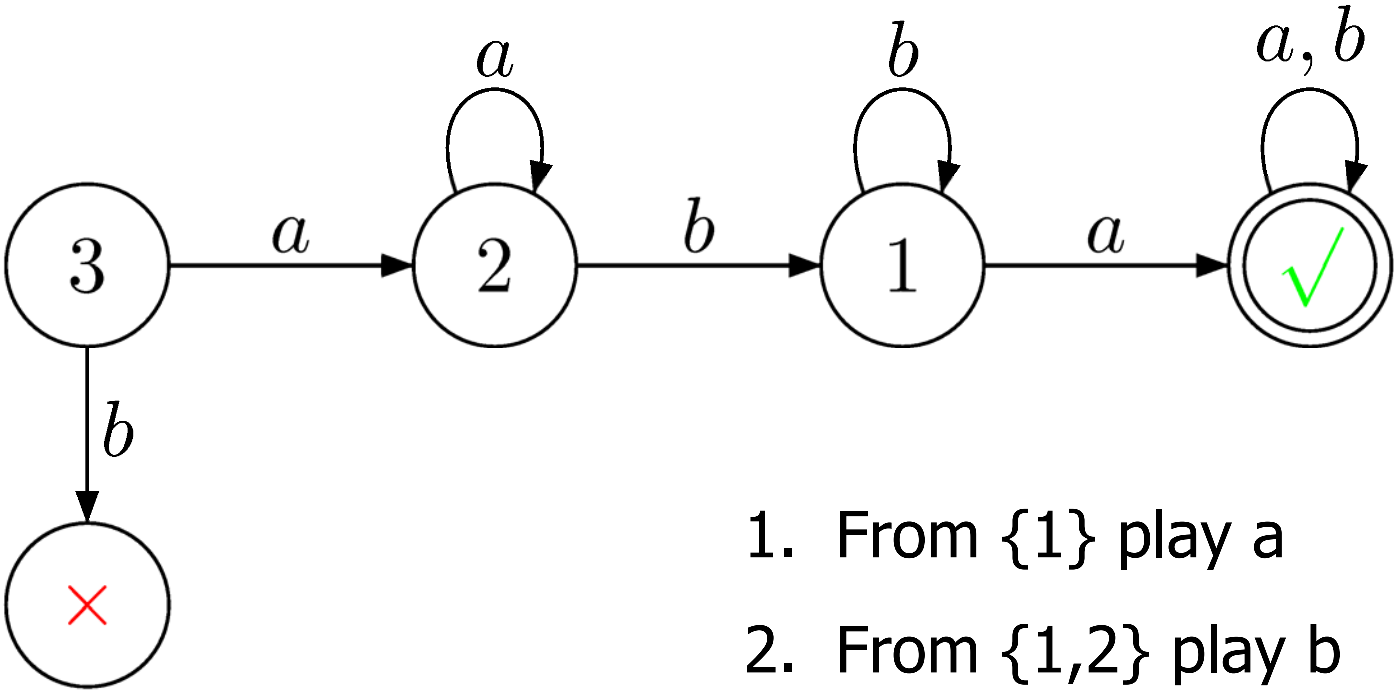
1. From {1} play a
2. From {1,2} play b

Reachability



1. From {1} play a
2. From {1,2} play b
3. From {1,2,3} play a

Reachability

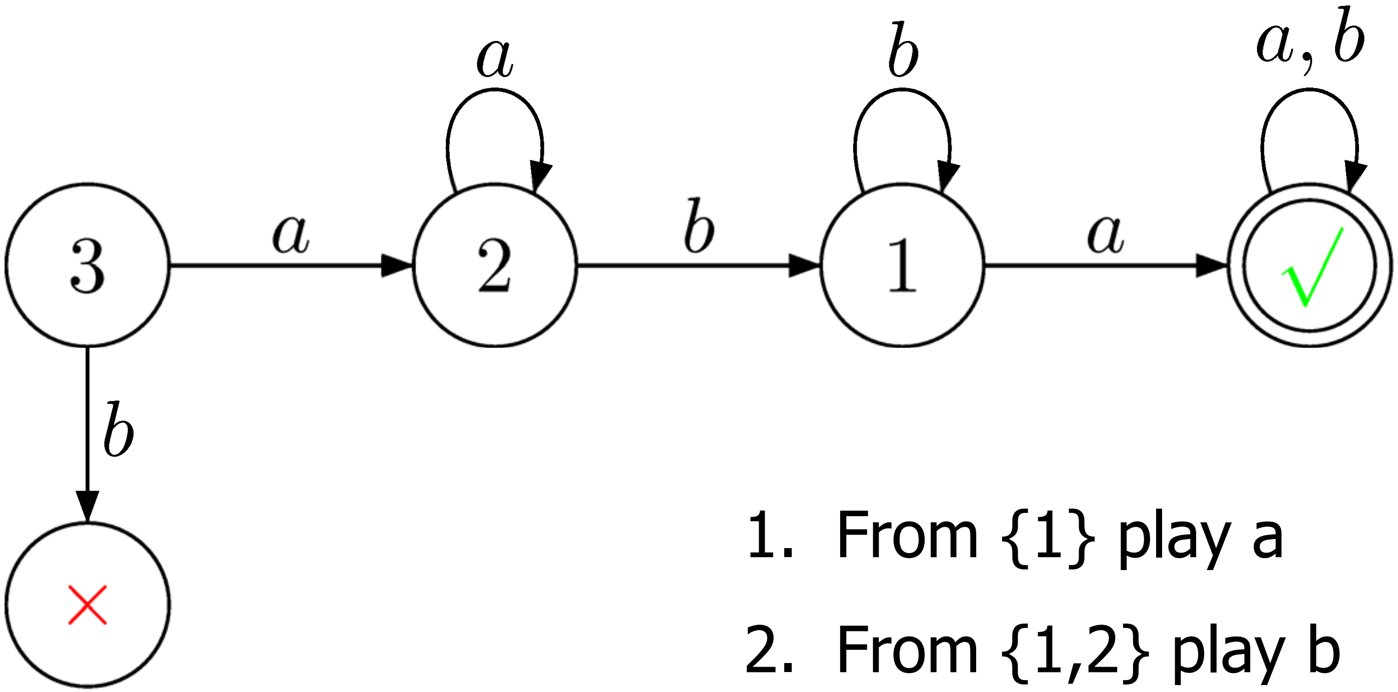


1. From {1} play a
2. From {1,2} play b
3. From {1,2,3} play a

Fixpoint of winning cells: {{1,2,3}}

Winning strategy ??

Reachability

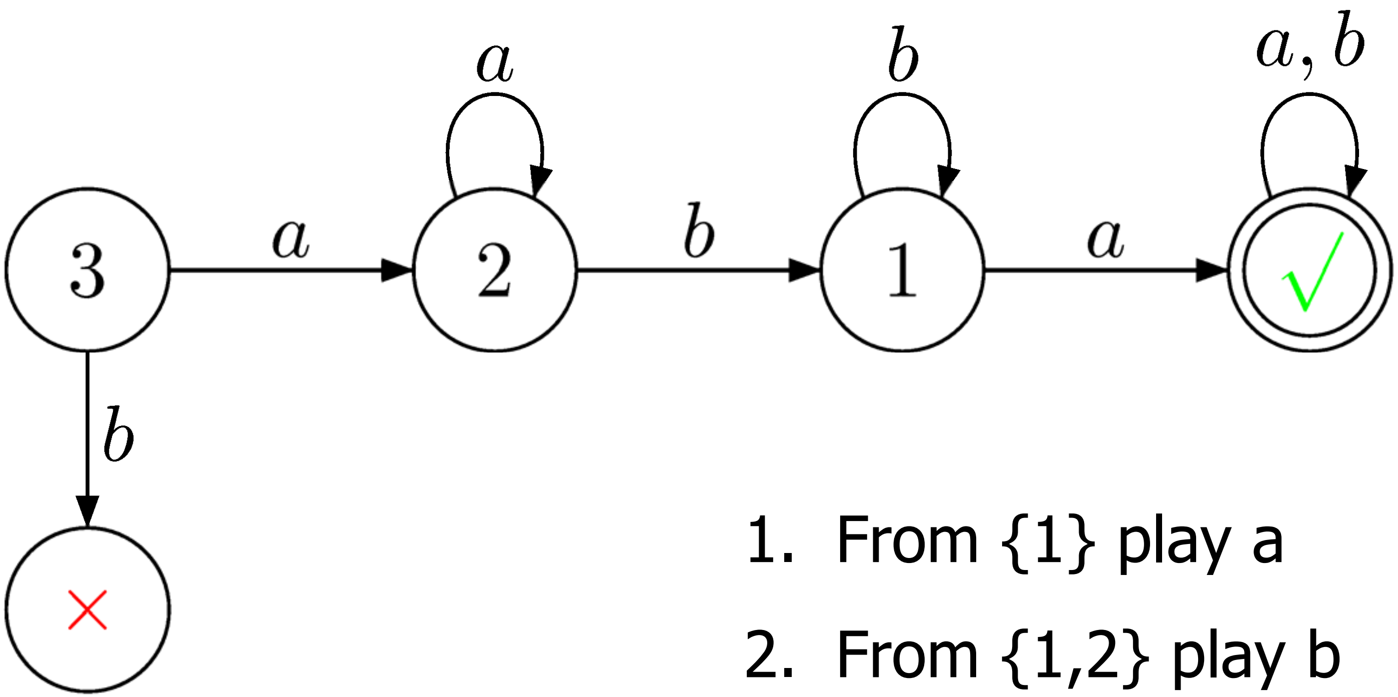


1. From {1} play a
2. From {1,2} play b
3. From {1,2,3} play a

Fixpoint of winning (cell, action): $\{\{1,2,3\}_a,\{1,2\}_b\}$

Winning strategy ??

Reachability



1. From {1} play a
2. From {1,2} play b
3. From {1,2,3} play a

Winning strategy

Current knowledge K: select earliest (cell,action)

such that K $\subseteq$ cell, play action

Strategy simplification #1

computed later

1. From {1,2,3} play a
2. From … play …
3. From … play …
4. From … play …
5. From {2} play a

Strategy simplification #1

computed later

1. From {1,2,3} play a
2. From … play …
3. From … play …
4. From … play …
5. ~~From {2} play a~~ Not necessary !

Rule 1: delete subsumed pairs computed later

Strategy simplification #2

computed later

1. From {1,2} play a
2. From {3,4} play …
3. From {1,3} play a
4. From {3,5} play …
5. From {1,2,3} play a

Strategy simplification #2

computed
later

1. ~~From {1,2} play a~~
2. From {3,4} play …
3. From {1,3} play a
4. From {3,5} play …
5. From {1,2,3} play a

Not necessary !

Rule 2: delete strongly-subsumed pairs

Alpaga

First prototype for solving parity games of imperfect information

- Use antichains as compact representation of winning sets of positions
- Compute Controllable Predecessor with BDDs
- Publish Reachability/Safety attractor moves to compose the strategy (earlier published move sticks)
- Strategy simplification

Alpaga

First prototype for solving parity games of imperfect information

- Implemented in Python + CUDD
- $\leq$1000 LoC
- Solves 50 states, 28 observations, 3 priorities (explicit game graph)

http://www.antichains.be/alpaga

Some experiments

	Size	Obs	Priorities	Time (s)
Game1	4	4	Reach.	.1
Game2	3	2	Reach.	.1
Game3	6	3	3	.1
Game4	8	5	5	1.4
Game5	8	5	7	9.4
Game6	11	9	10	50.7
Game7	11	8	10	579.0
Locking	22	14	Safety	.6
Mutex	50	28	3	57.7

http://www.antichains.be/alpaga

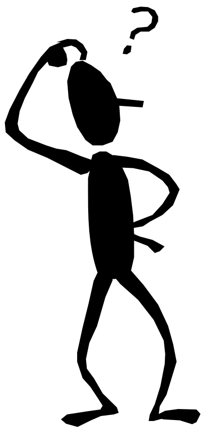
Alpaga

First prototype for solving parity games of imperfect information

Outlook

- Symbolic game graph
- Compact representation of strategies
- Almost-sure winning
- Relaxing visibility

Thank you !



Questions ?

http://www.antichains.be/alpaga

References

- [Reif84] J. H. Reif. The Complexity of Two-Player Games of Incomplete Information. J. Comput. Syst. Sci. 29(2): 274-301, 1984
- [CSL'06] K. Chatterjee, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Algorithms for Omega-regular Games of Incomplete Information. Proc. of CSL, LNCS 4207, Springer, 2006, pp. 287-302
- [Concur'08] D. Berwanger, K. Chatterjee, L. Doyen, T. A. Henzinger, and S. Raje. Strategy Construction for Parity Games with Imperfect Information. Proc. of Concur, LNCS 5201, Springer, 2008, pp. 325-339

Alpaga demo

- Login on mtcserever
- Cd research/2008/StrategyConstruction/CodeMartin/Alpaga-2008-Aug-20/alpaga
- Help: python src/alpaga.py –h

- Locking example: python src/alpaga.py -t -i examples/locking.gii
  - Interactive mode: >> go