Investigating the Utility of Answer Set Programming and Inductive Logic Techniques in Learning Two-Player Game Strategies

Susana Hahn, Atreya Shankar
Cognitive Systems: Language, Learning, and Reasoning (M.Sc.)

PM - Computational Intelligence
Knowledge Processing and Information Systems

June 03, 2020

Overview

Introduction

Introduction

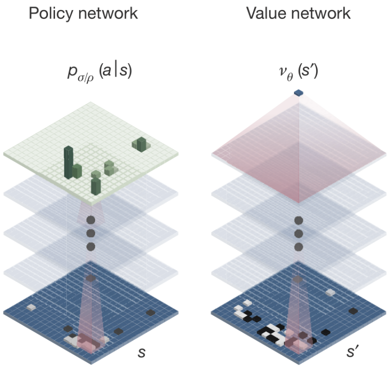


Figure 1: Schematic of policy and value neural networks in AlphaGo

- New advancements in computer agents using AI for two-players games *AlphaGo*
  - Deep reinforcement learning
  - GPU hardware-acceleration
  - Monte-Carlo tree search

✓ Make decisions for games with exponentially complex search spaces

× Lack of interpretability of black-box models

Introduction

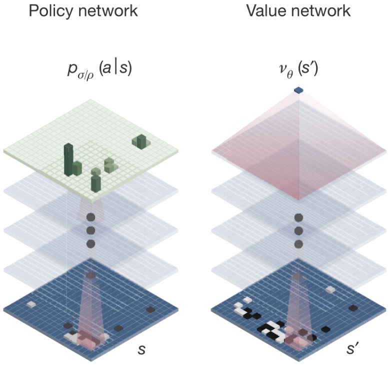


Figure 1: Schematic of policy and value neural networks in AlphaGo

- New advancements in computer agents using AI for two-players games *AlphaGo*
  - Deep reinforcement learning
  - GPU hardware-acceleration
  - Monte-Carlo tree search

✓ Make decisions for games with exponentially complex search spaces

× Lack of interpretability of black-box models

Introduction

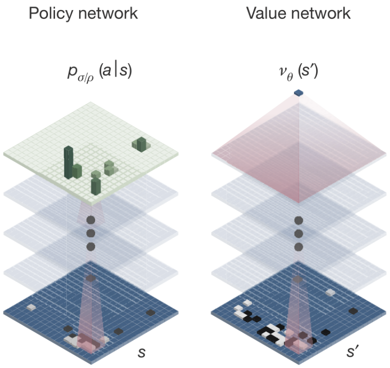


Figure 1: Schematic of policy and value neural networks in AlphaGo

- New advancements in computer agents using AI for two-players games *AlphaGo*
  - Deep reinforcement learning
  - GPU hardware-acceleration
  - Monte-Carlo tree search

✓ Make decisions for games with exponentially complex search spaces

× Lack of interpretability of black-box models

Introduction

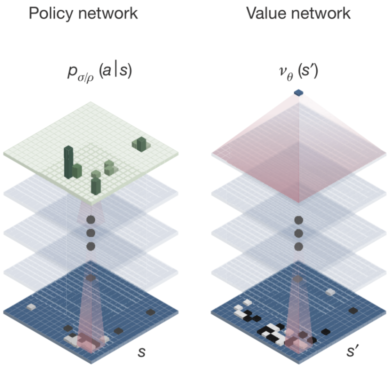


Figure 1: Schematic of policy and value neural networks in AlphaGo

- New advancements in computer agents using AI for two-players games *AlphaGo*
  - Deep reinforcement learning
  - GPU hardware-acceleration
  - Monte-Carlo tree search

✓ Make decisions for games with exponentially complex search spaces

× Lack of interpretability of black-box models

Introduction

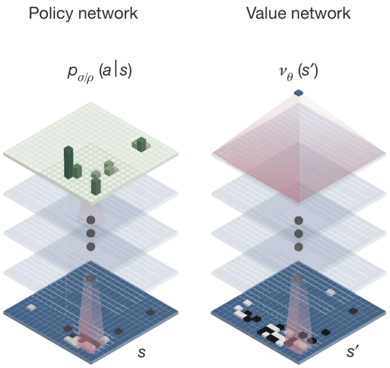


Figure 1: Schematic of policy and value neural networks in AlphaGo

- New advancements in computer agents using AI for two-players games *AlphaGo*
  - Deep reinforcement learning
  - GPU hardware-acceleration
  - Monte-Carlo tree search

✓ Make decisions for games with exponentially complex search spaces

× Lack of interpretability of black-box models

Introduction

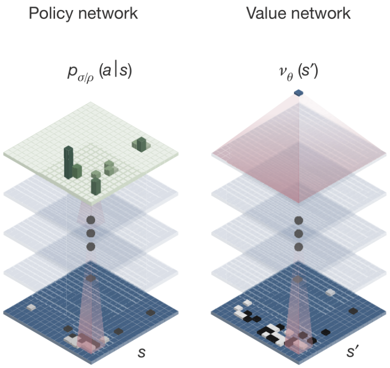


Figure 1: Schematic of policy and value neural networks in AlphaGo

- New advancements in computer agents using AI for two-players games *AlphaGo*
  - Deep reinforcement learning
  - GPU hardware-acceleration
  - Monte-Carlo tree search

✓ Make decisions for games with exponentially complex search spaces

× Lack of interpretability of black-box models

Focus

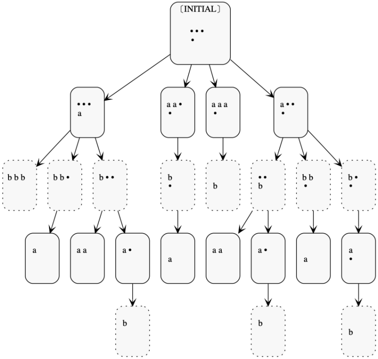


Figure 2: Schematic of game tree

- Logic-based methods
  - ✓ Interpretability of strategies
  - × Search large game spaces to learn strategies
- Small two-player games: *Nim* and *Tic-Tac-Toe*
- Use formalisms from Game Description Language
- Use Answer Set Programming and Inductive Logic Techniques to learn interpretable game-winning strategies

Focus

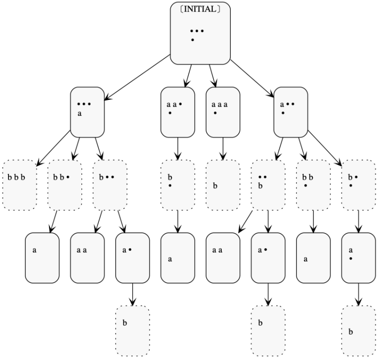


Figure 2: Schematic of game tree

- Logic-based methods
  - ✓ Interpretability of strategies
  - × Search large game spaces to learn strategies
- Small two-player games: *Nim* and *Tic-Tac-Toe*
- Use formalisms from Game Description Language
- Use Answer Set Programming and Inductive Logic Techniques to learn interpretable game-winning strategies

Focus

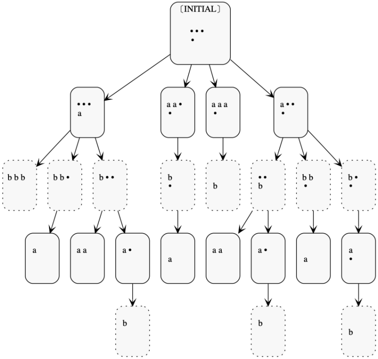


Figure 2: Schematic of game tree

- Logic-based methods
  - ✓ Interpretability of strategies
  - × Search large game spaces to learn strategies
- Small two-player games: *Nim* and *Tic-Tac-Toe*
- Use formalisms from Game Description Language
- Use Answer Set Programming and Inductive Logic Techniques to learn interpretable game-winning strategies

Focus

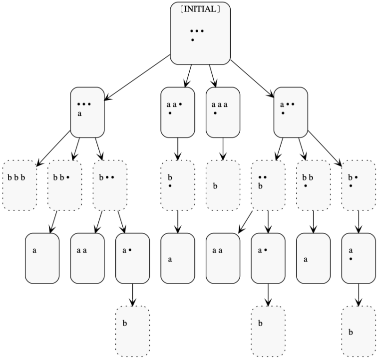


Figure 2: Schematic of game tree

- Logic-based methods
  - ✓ Interpretability of strategies
  - × Search large game spaces to learn strategies
- Small two-player games: *Nim* and *Tic-Tac-Toe*
- Use formalisms from Game Description Language
- Use Answer Set Programming and Inductive Logic Techniques to learn interpretable game-winning strategies

Focus

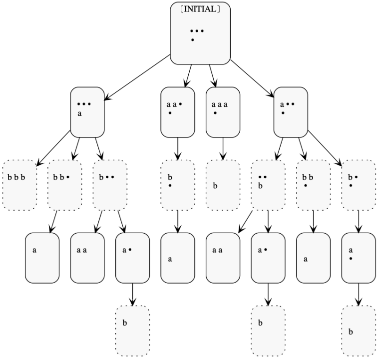


Figure 2: Schematic of game tree

- Logic-based methods
  - ✓ Interpretability of strategies
  - × Search large game spaces to learn strategies
- Small two-player games: *Nim* and *Tic-Tac-Toe*
- Use formalisms from Game Description Language
- Use Answer Set Programming and Inductive Logic Techniques to learn interpretable game-winning strategies

Focus

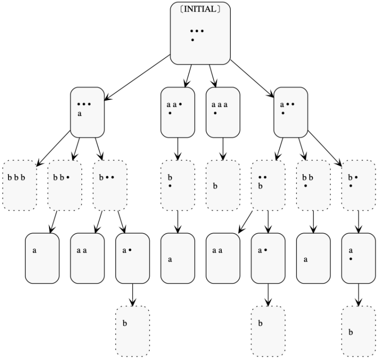


Figure 2: Schematic of game tree

- Logic-based methods
  - ✓ Interpretability of strategies
  - × Search large game spaces to learn strategies
- Small two-player games: *Nim* and *Tic-Tac-Toe*
- Use formalisms from Game Description Language
- Use Answer Set Programming and Inductive Logic Techniques to learn interpretable game-winning strategies

Background Concepts

Two-Player Game: Nim

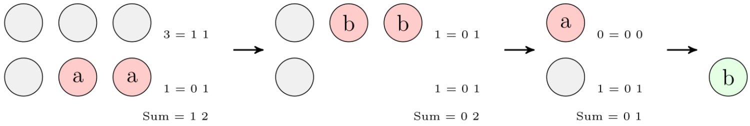


Figure 3: Schematic of Nim gameplay; red implies counters taken while green implies win

- *Nim* is an old game from Ancient China
- Players play alternately and must take at least one counter from each pile
- The player which takes the last counter wins the game
  $\implies$ there is always a winning strategy
- Mathematical strategy is to leave the next player with even *Nim Sums*, where a *Nim Sum* is the binary digital sum of counters in each pile

Two-Player Game: Nim

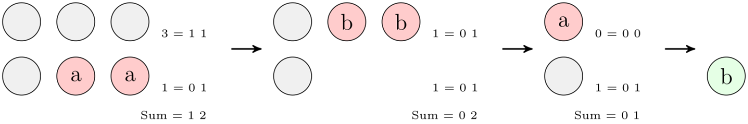


Figure 3: Schematic of Nim gameplay; red implies counters taken while green implies win

- *Nim* is an old game from Ancient China
- Players play alternately and must take at least one counter from each pile
- The player which takes the last counter wins the game
  $\implies$ there is always a winning strategy
- Mathematical strategy is to leave the next player with even *Nim Sums*, where a *Nim Sum* is the binary digital sum of counters in each pile

Two-Player Game: Nim

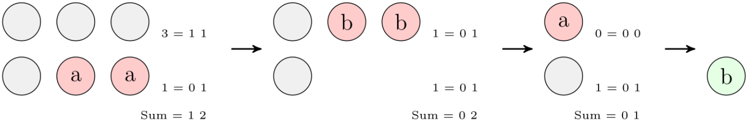


Figure 3: Schematic of Nim gameplay; red implies counters taken while green implies win

- *Nim* is an old game from Ancient China
- Players play alternately and must take at least one counter from each pile
- The player which takes the last counter wins the game
  $\implies$ there is always a winning strategy
- Mathematical strategy is to leave the next player with even *Nim Sums*, where a *Nim Sum* is the binary digital sum of counters in each pile

Two-Player Game: Nim

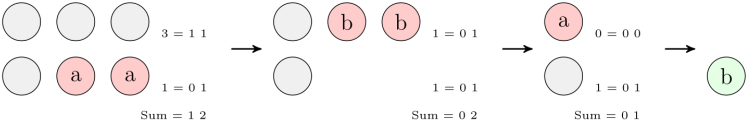


Figure 3: Schematic of Nim gameplay; red implies counters taken while green implies win

- *Nim* is an old game from Ancient China
- Players play alternately and must take at least one counter from each pile
- The player which takes the last counter wins the game
  $\Longrightarrow$ there is always a winning strategy
- Mathematical strategy is to leave the next player with even *Nim Sums*, where a *Nim Sum* is the binary digital sum of counters in each pile

Two-Player Game: Tic-Tac-Toe

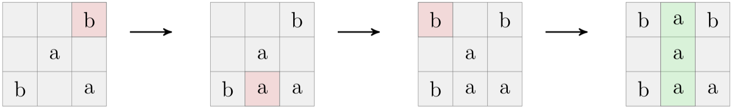


Figure 4: Schematic of Tic-Tac-Toe gameplay; red implies square taken while green implies winning streak

- *Tic-Tac-Toe* is a (m, n, k) type game which is played on a $m \times n$ size grid where each player alternately places a unique object on the grid
- The first player to place k of its items in a row, column or diagonal wins
- *Tic-Tac-Toe* will be treated here as a $(m, n, k) \equiv (3, 3, 3)$ game
- Unlike *Nim*, a winning strategy cannot be guaranteed (might end in draw)

Two-Player Game: Tic-Tac-Toe

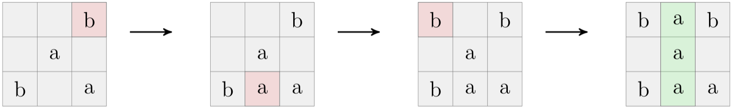


Figure 4: Schematic of Tic-Tac-Toe gameplay; red implies square taken while green implies winning streak

- *Tic-Tac-Toe* is a (m, n, k) type game which is played on a $m \times n$ size grid where each player alternately places a unique object on the grid
- The first player to place k of its items in a row, column or diagonal wins
- *Tic-Tac-Toe* will be treated here as a $(m, n, k) \equiv (3, 3, 3)$ game
- Unlike *Nim*, a winning strategy cannot be guaranteed (might end in draw)

Two-Player Game: Tic-Tac-Toe

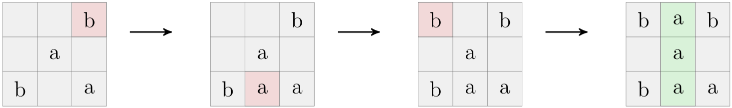


Figure 4: Schematic of Tic-Tac-Toe gameplay; red implies square taken while green implies winning streak

- *Tic-Tac-Toe* is a (m, n, k) type game which is played on a $m \times n$ size grid where each player alternately places a unique object on the grid
- The first player to place k of its items in a row, column or diagonal wins
- *Tic-Tac-Toe* will be treated here as a $(m, n, k) \equiv (3, 3, 3)$ game
- Unlike *Nim*, a winning strategy cannot be guaranteed (might end in draw)

Two-Player Game: Tic-Tac-Toe

Figure 4: Schematic of Tic-Tac-Toe gameplay; red implies square taken while green implies winning streak

- *Tic-Tac-Toe* is a (m, n, k) type game which is played on a $m \times n$ size grid where each player alternately places a unique object on the grid
- The first player to place k of its items in a row, column or diagonal wins
- *Tic-Tac-Toe* will be treated here as a $(m, n, k) \equiv (3, 3, 3)$ game
- Unlike *Nim*, a winning strategy cannot be guaranteed (might end in draw)

Game Description Language (GDL)

- GDL is declarative programming language
- Representation formalism for axiomatising the rules of any game

Relevant GDL relations

$\boldsymbol{role(r)}$: r is a valid role in the game

$\boldsymbol{true(p)}$: proposition p is true in the current state

$\boldsymbol{next(p)}$: proposition p is true in the next state

$\boldsymbol{legal(r,a)}$: player with role r can legally perform action a in current state

$\boldsymbol{does(r,a)}$: player with role r performs action a in the current state

$\boldsymbol{goal(r,n)}$: player with role r achieves a utility of n in the current state

$\boldsymbol{terminal}$: the current state is the final or terminal state

Game Description Language (GDL)

- GDL is declarative programming language
- Representation formalism for axiomatising the rules of any game

Relevant GDL relations

$\boldsymbol{role(r)}$: r is a valid role in the game

$\boldsymbol{true(p)}$: proposition p is true in the current state

$\boldsymbol{next(p)}$: proposition p is true in the next state

$\boldsymbol{legal(r, a)}$: player with role r can legally perform action a in current state

$\boldsymbol{does(r, a)}$: player with role r performs action a in the current state

$\boldsymbol{goal(r, n)}$: player with role r achieves a utility of n in the current state

$\boldsymbol{terminal}$: the current state is the final or terminal state

Game Description Language (GDL)

- GDL is declarative programming language
- Representation formalism for axiomatising the rules of any game

Relevant GDL relations

$role(r)$: r is a valid role in the game

$true(p)$: proposition p is true in the current state

$next(p)$: proposition p is true in the next state

$legal(r, a)$: player with role r can legally perform action a in current state

$does(r, a)$: player with role r performs action a in the current state

$goal(r, n)$: player with role r achieves a utility of n in the current state

$terminal$: the current state is the final or terminal state

Game Description Language (GDL)

- GDL is declarative programming language
- Representation formalism for axiomatising the rules of any game

Relevant GDL relations

$\boldsymbol{role(r)}$: r is a valid role in the game

$\boldsymbol{true(p)}$: proposition p is true in the current state

$\boldsymbol{next(p)}$: proposition p is true in the next state

$\boldsymbol{legal(r,a)}$: player with role r can legally perform action a in current state

$\boldsymbol{does(r,a)}$: player with role r performs action a in the current state

$\boldsymbol{goal(r,n)}$: player with role r achieves a utility of n in the current state

$\boldsymbol{terminal}$: the current state is the final or terminal state

Inductive Learning of Answer Set Programs (ILASP)

- Inductive Logic Programming is to find a hypothesis that explains a set of examples with some background knowledge
- ILASP is an inductive logic framework developed largely by Mark Law from the Imperial College London

Overview of ILASP framework

Given an ASP program B called the **background knowledge**, a set of ASP rules S called the **search space** and two sets of partial interpretations E^+ and E^- called the **positive and negative examples** respectively, the goal is to find another program H called a **hypothesis** such that:

1. H is composed of the rules in S ($H \subseteq S$)
2. Each positive example is extended by at least one answer set of $B \cup H$ (can be a different answer set for each positive example)
3. No negative example is extended by any answer set of $B \cup H$

Inductive Learning of Answer Set Programs (ILASP)

- Inductive Logic Programming is to find a hypothesis that explains a set of examples with some background knowledge
- ILASP is an inductive logic framework developed largely by Mark Law from the Imperial College London

Overview of ILASP framework

Given an ASP program B called the **background knowledge**, a set of ASP rules S called the **search space** and two sets of partial interpretations E^+ and E^- called the **positive and negative examples** respectively, the goal is to find another program H called a **hypothesis** such that:

1. H is composed of the rules in S ($H \subseteq S$)
2. Each positive example is extended by at least one answer set of $B \cup H$ (can be a different answer set for each positive example)
3. No negative example is extended by any answer set of $B \cup H$

Inductive Learning of Answer Set Programs (ILASP)

- Inductive Logic Programming is to find a hypothesis that explains a set of examples with some background knowledge
- ILASP is an inductive logic framework developed largely by Mark Law from the Imperial College London

Overview of ILASP framework

Given an ASP program B called the **background knowledge**, a set of ASP rules S called the **search space** and two sets of partial interpretations E^+ and E^- called the **positive and negative examples** respectively, the goal is to find another program H called a **hypothesis** such that:

1. H is composed of the rules in S ($H \subseteq S$)
2. Each positive example is extended by at least one answer set of $B \cup H$ (can be a different answer set for each positive example)
3. No negative example is extended by any answer set of $B \cup H$

Inductive Learning of Answer Set Programs (ILASP)

- Inductive Logic Programming is to find a hypothesis that explains a set of examples with some background knowledge
- ILASP is an inductive logic framework developed largely by Mark Law from the Imperial College London

Overview of ILASP framework

Given an ASP program B called the **background knowledge**, a set of ASP rules S called the **search space** and two sets of partial interpretations E^+ and E^- called the **positive and negative examples** respectively, the goal is to find another program H called a **hypothesis** such that:

1. H is composed of the rules in S ($H \subseteq S$)
2. Each positive example is extended by at least one answer set of $B \cup H$ (can be a different answer set for each positive example)
3. No negative example is extended by any answer set of $B \cup H$

Inductive Learning of Answer Set Programs (ILASP)

- Provides a method to extend the above workflow to learn weak constraints from ordered examples
- Particularly powerful for situations in two-player games with multiple optimum decisions with varying degrees of preference

Key Definitions from ILASP extension

Ordering example: An ordering example is a tuple $o = \langle e_1, e_2 \rangle$, where e_1 and e_2 are partial (positive-example) interpretations.

Brave ordering: An ASP program P bravely respects o iff $\exists A_1, A_2 \in AS(P)$, such that $AS(P)$ is the answer set of P, A_1 extends e_1, A_2 extends e_2 and $A_1 \succ_p A_2$.

Inductive Learning of Answer Set Programs (ILASP)

- Provides a method to extend the above workflow to learn weak constraints from ordered examples
- Particularly powerful for situations in two-player games with multiple optimum decisions with varying degrees of preference

Key Definitions from ILASP extension

Ordering example: An ordering example is a tuple $o = \langle e_1, e_2 \rangle$, where e_1 and e_2 are partial (positive-example) interpretations.

Brave ordering: An ASP program P bravely respects o iff $\exists A_1, A_2 \in AS(P)$, such that $AS(P)$ is the answer set of P, A_1 extends e_1, A_2 extends e_2 and $A_1 \succ_p A_2$.

Inductive Learning of Answer Set Programs (ILASP)

- Provides a method to extend the above workflow to learn weak constraints from ordered examples
- Particularly powerful for situations in two-player games with multiple optimum decisions with varying degrees of preference

Key Definitions from ILASP extension

Ordering example: An ordering example is a tuple $o = \langle e_1, e_2 \rangle$, where e_1 and e_2 are partial (positive-example) interpretations.

Brave ordering: An ASP program P bravely respects o iff $\exists A_1, A_2 \in AS(P)$, such that $AS(P)$ is the answer set of P, A_1 extends e_1, A_2 extends e_2 and $A_1 \succ_p A_2$.

Inductive Learning of Answer Set Programs (ILASP)

- Provides a method to extend the above workflow to learn weak constraints from ordered examples
- Particularly powerful for situations in two-player games with multiple optimum decisions with varying degrees of preference

Key Definitions from ILASP extension

Ordering example: An ordering example is a tuple $o = \langle e_1, e_2 \rangle$, where e_1 and e_2 are partial (positive-example) interpretations.

Brave ordering: An ASP program P bravely respects o iff $\exists A_1, A_2 \in AS(P)$, such that $AS(P)$ is the answer set of P, A_1 extends e_1, A_2 extends e_2 and $A_1 \succ_p A_2$.

Methodologies

Overview of Methodologies

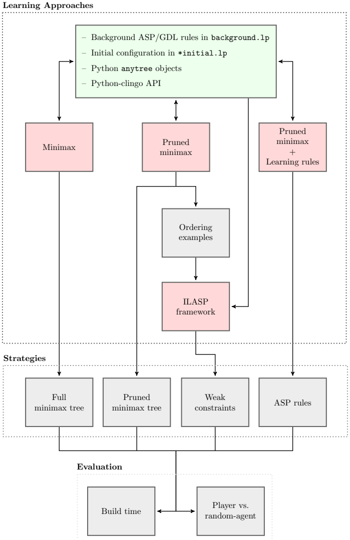


Figure 5: Overview and segmentation of methodologies

- Exploit ASP and python-clingo API to create a dynamic game system governed with GDL relations
- Python *anytree* module to represent the game trees which are annotated with scores and produce pretty visualizations
- Flavours of the *Minimax* algorithm in ASP to learn optimum game strategies
- Pipe optimum decisions as bravely ordered positive examples for ILASP
- Evaluate different methodologies by comparing build times and performance against a control (random) player

Overview of Methodologies

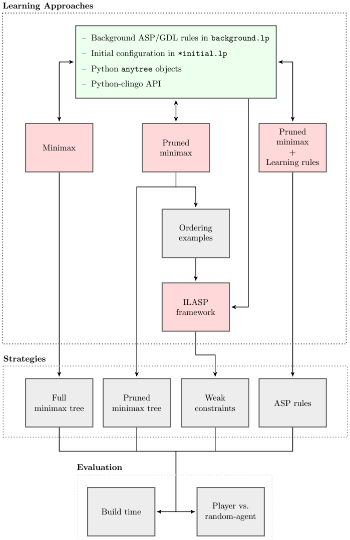


Figure 5: Overview and segmentation of methodologies

- Exploit ASP and python-clingo API to create a dynamic game system governed with GDL relations
- Python *anytree* module to represent the game trees which are annotated with scores and produce pretty visualizations
- Flavours of the *Minimax* algorithm in ASP to learn optimum game strategies
- Pipe optimum decisions as bravely ordered positive examples for ILASP
- Evaluate different methodologies by comparing build times and performance against a control (random) player

Overview of Methodologies

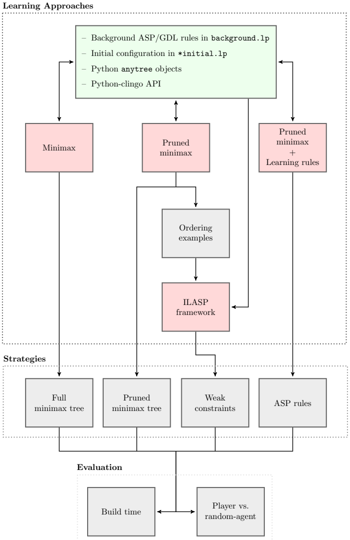


Figure 5: Overview and segmentation of methodologies

- Exploit ASP and python-clingo API to create a dynamic game system governed with GDL relations
- Python *anytree* module to represent the game trees which are annotated with scores and produce pretty visualizations
- Flavours of the *Minimax* algorithm in ASP to learn optimum game strategies
- Pipe optimum decisions as bravely ordered positive examples for ILASP
- Evaluate different methodologies by comparing build times and performance against a control (random) player

Overview of Methodologies

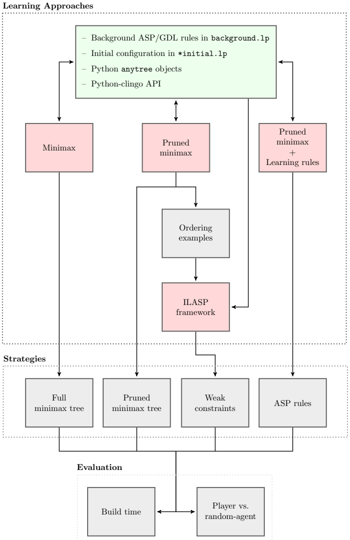


Figure 5: Overview and segmentation of methodologies

- Exploit ASP and python-clingo API to create a dynamic game system governed with GDL relations
- Python *anytree* module to represent the game trees which are annotated with scores and produce pretty visualizations
- Flavours of the *Minimax* algorithm in ASP to learn optimum game strategies
- Pipe optimum decisions as bravely ordered positive examples for ILASP
- Evaluate different methodologies by comparing build times and performance against a control (random) player

Overview of Methodologies

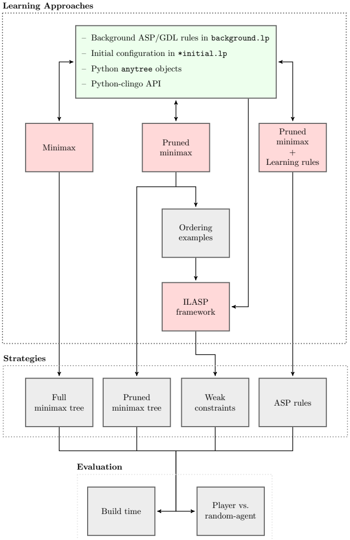


Figure 5: Overview and segmentation of methodologies

- Exploit ASP and python-clingo API to create a dynamic game system governed with GDL relations
- Python *anytree* module to represent the game trees which are annotated with scores and produce pretty visualizations
- Flavours of the *Minimax* algorithm in ASP to learn optimum game strategies
- Pipe optimum decisions as bravely ordered positive examples for ILASP
- Evaluate different methodologies by comparing build times and performance against a control (random) player

Game simulation

ASP program encoding the rules of the game for one step

Example (Rules TicTacToe)

```
1 {does(X,A):legal(X,A)} 1 :-
  true(control(X)),
  not terminal.

next(control(a)) :-
  true(control(b)),
  not terminal.

goal(P,1) :-
  true(has(P,C1)),
  true(has(P,C2)),
  true(has(P,C3)),
  in_line(C1,C2,C3).
...
```

Facts defining the state

Example (State TicTacToe)

```
true(control(a)).
true(free(1,3)).
true(free(3,1)).
true(free(2,3)).
...
```

↓ (Clingo's API)

One stable model per legal action

Example (Next state TicTacToe)

```
does(a,take(2,2))
next(control(b))
next(free(1,3))
next(free(2,3))
...
```

Game simulation

ASP program encoding the rules of the game for one step

Example (Rules TicTacToe)

```
1 {does(X,A):legal(X,A)} 1 :-
  true(control(X)),
  not terminal.

next(control(a)) :-
  true(control(b)),
  not terminal.

goal(P,1) :-
  true(has(P,C1)),
  true(has(P,C2)),
  true(has(P,C3)),
  in_line(C1,C2,C3).
...
```

Facts defining the state

Example (State TicTacToe)

```
true(control(a)).
true(free(1,3)).
true(free(3,1)).
true(free(2,3)).
...
```

↓ (Clingo's API)

One stable model per legal action

Example (Next state TicTacToe)

```
does(a,take(2,2))
next(control(b))
next(free(1,3))
next(free(2,3))
...
```

Game simulation

ASP program encoding the rules of the game for one step

Example (Rules TicTacToe)

```
1 {does(X,A):legal(X,A)} 1 :-
  true(control(X)),
  not terminal.

next(control(a)) :-
  true(control(b)),
  not terminal.

goal(P,1) :-
  true(has(P,C1)),
  true(has(P,C2)),
  true(has(P,C3)),
  in_line(C1,C2,C3).
...
```

Facts defining the state

Example (State TicTacToe)

```
true(control(a)).
true(free(1,3)).
true(free(3,1)).
true(free(2,3)).
...
```

↓ (Clingo's API)

One stable model per legal action

Example (Next state TicTacToe)

```
does(a,take(2,2))
next(control(b))
next(free(1,3))
next(free(2,3))
...
```

Minimax Algorithm

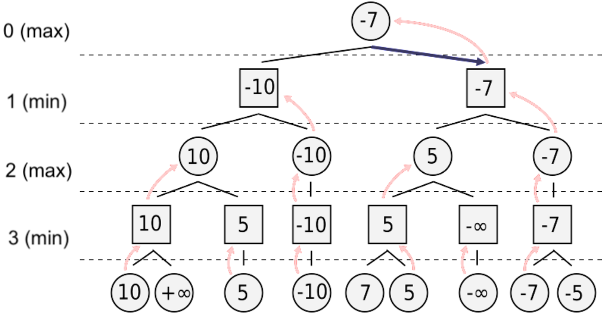


Figure 6: Minimax sample schematic for a two-player game

- One player maximizes utility while another minimizes utility
- Bottom-Up procedure assuming there is a complete game tree with utility-annotated leaf nodes
- Time complexity: $O(b^D)$; Space complexity: $O(bD)$; where b is the average branching factor and D is the maximum depth of the tree

Minimax Algorithm

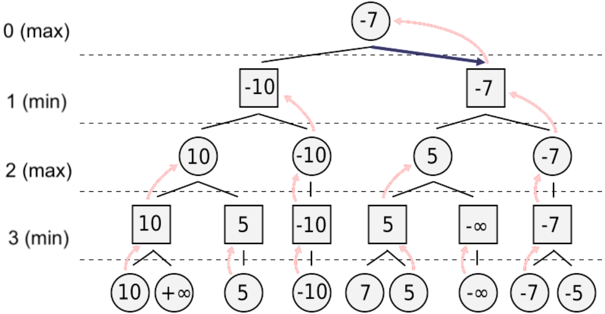


Figure 6: Minimax sample schematic for a two-player game

- One player maximizes utility while another minimizes utility
- Bottom-Up procedure assuming there is a complete game tree with utility-annotated leaf nodes
- Time complexity: $O(b^D)$; Space complexity: $O(bD)$; where b is the average branching factor and D is the maximum depth of the tree

Minimax Algorithm

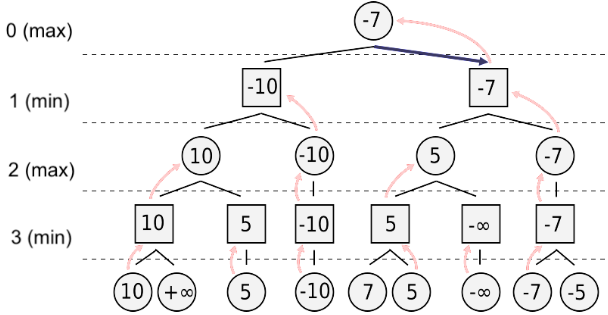


Figure 6: Minimax sample schematic for a two-player game

- One player maximizes utility while another minimizes utility
- Bottom-Up procedure assuming there is a complete game tree with utility-annotated leaf nodes
- Time complexity: $O(b^D)$; Space complexity: $O(bD)$; where b is the average branching factor and D is the maximum depth of the tree

Minimax Algorithm

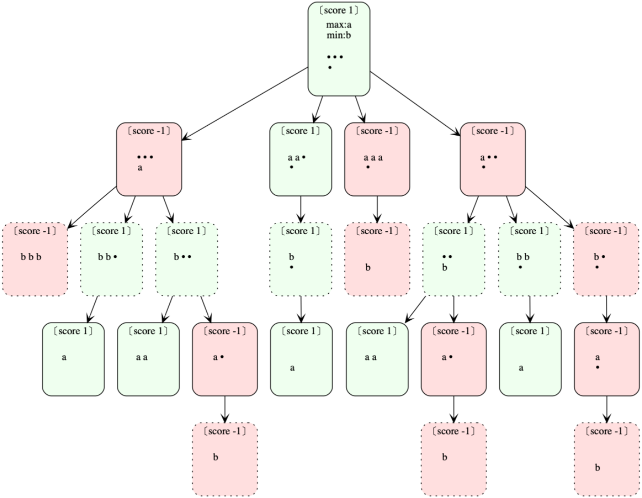


Figure 7: Minimax schematic for a small Nim game where player a maximizes utility while player b minimizes utility; using the Minimax algorithm we can deduce that player a starts off with a winning strategy

Pruned Minimax Algorithm

Motivation

- Construct only the necessary parts of the tree (Alphabeta pruning)
- Treat it as a planning problem
- Use optimization statements from *clingo*

Explicit time encoding

1. Replace `true(F)` by `holds(F,T)`
2. Replace `next(F)` by `holds(F,T+1)`
3. Replace `p(F1..Fn)` by `p(F1..Fn,T)` for $\{legal, does, goal, terminal\}$
4. Add the time in in the body with predicate `time(T)`
5. Add a new fact `time(0..N).` for horizon N

Optimization statements

- `#maximize{N,T:goal(a,N,T)}`

× Assumes player b will chose actions under the same optimization.

Pruned Minimax Algorithm

Motivation

- Construct only the necessary parts of the tree (Alphabeta pruning)
- Treat it as a planning problem
- Use optimization statements from *clingo*

Explicit time encoding

1. Replace `true(F)` by `holds(F,T)`
2. Replace `next(F)` by `holds(F,T+1)`
3. Replace `p(F1..Fn)` by `p(F1..Fn,T)` for $\{legal, does, goal, terminal\}$
4. Add the time in in the body with predicate `time(T)`
5. Add a new fact `time(0..N).` for horizon N

Optimization statements

- `#maximize{N,T:goal(a,N,T)}`
- × Assumes player b will chose actions under the same optimization.

Pruned Minimax Algorithm

Motivation

- Construct only the necessary parts of the tree (Alphabeta pruning)
- Treat it as a planning problem
- Use optimization statements from *clingo*

Explicit time encoding

1. Replace `true(F)` by `holds(F,T)`
2. Replace `next(F)` by `holds(F,T+1)`
3. Replace `p(F1..Fn)` by `p(F1..Fn,T)` for $\{legal, does, goal, terminal\}$
4. Add the time in in the body with predicate `time(T)`
5. Add a new fact `time(0..N).` for horizon N

Optimization statements

- `#maximize{N,T:goal(a,N,T)}`

× Assumes player b will chose actions under the same optimization.

Pruned Minimax Algorithm

Motivation

- Construct only the necessary parts of the tree (Alphabeta pruning)
- Treat it as a planning problem
- Use optimization statements from *clingo*

Explicit time encoding

1. Replace `true(F)` by `holds(F,T)`
2. Replace `next(F)` by `holds(F,T+1)`
3. Replace `p(F1..Fn)` by `p(F1..Fn,T)` for $\{legal, does, goal, terminal\}$
4. Add the time in in the body with predicate `time(T)`
5. Add a new fact `time(0..N).` for horizon N

Optimization statements

- `#maximize{N,T:goal(a,N,T)}`

× Assumes player b will chose actions under the same optimization.

Pruned Minimax Algorithm

Motivation

- Construct only the necessary parts of the tree (Alphabeta pruning)
- Treat it as a planning problem
- Use optimization statements from *clingo*

Explicit time encoding

1. Replace `true(F)` by `holds(F,T)`
2. Replace `next(F)` by `holds(F,T+1)`
3. Replace `p(F1..Fn)` by `p(F1..Fn,T)` for $\{legal, does, goal, terminal\}$
4. Add the time in in the body with predicate `time(T)`
5. Add a new fact `time(0..N).` for horizon N

Optimization statements

- `#maximize{N,T:goal(a,N,T)}`

× Assumes player b will chose actions under the same optimization.

Pruned Minimax Algorithm

Motivation

- Construct only the necessary parts of the tree (Alphabeta pruning)
- Treat it as a planning problem
- Use optimization statements from *clingo*

Explicit time encoding

1. Replace `true(F)` by `holds(F,T)`
2. Replace `next(F)` by `holds(F,T+1)`
3. Replace `p(F1..Fn)` by `p(F1..Fn,T)` for $\{legal, does, goal, terminal\}$
4. Add the time in in the body with predicate `time(T)`
5. Add a new fact `time(0..N).` for horizon N

Optimization statements

- `#maximize{N,T:goal(a,N,T)}`

× Assumes player b will chose actions under the same optimization.

Pruned Minimax Algorithm

Motivation

- Construct only the necessary parts of the tree (Alphabeta pruning)
- Treat it as a planning problem
- Use optimization statements from *clingo*

Explicit time encoding

1. Replace `true(F)` by `holds(F,T)`
2. Replace `next(F)` by `holds(F,T+1)`
3. Replace `p(F1..Fn)` by `p(F1..Fn,T)` for $\{legal, does, goal, terminal\}$
4. Add the time in in the body with predicate `time(T)`
5. Add a new fact `time(0..N).` for horizon N

Optimization statements

- `#maximize{N,T:goal(a,N,T)}`

× Assumes player b will chose actions under the same optimization.

Pruned Minimax Algorithm

Motivation

- Construct only the necessary parts of the tree (Alphabeta pruning)
- Treat it as a planning problem
- Use optimization statements from *clingo*

Explicit time encoding

1. Replace `true(F)` by `holds(F,T)`
2. Replace `next(F)` by `holds(F,T+1)`
3. Replace `p(F1..Fn)` by `p(F1..Fn,T)` for $\{legal, does, goal, terminal\}$
4. Add the time in in the body with predicate `time(T)`
5. Add a new fact `time(0..N).` for horizon N

Optimization statements

- `#maximize{N,T:goal(a,N,T)}`

× Assumes player b will chose actions under the same optimization.

Pruned Minimax Algorithm

Motivation

- Construct only the necessary parts of the tree (Alphabeta pruning)
- Treat it as a planning problem
- Use optimization statements from *clingo*

Explicit time encoding

1. Replace `true(F)` by `holds(F,T)`
2. Replace `next(F)` by `holds(F,T+1)`
3. Replace `p(F1..Fn)` by `p(F1..Fn,T)` for $\{legal, does, goal, terminal\}$
4. Add the time in in the body with predicate `time(T)`
5. Add a new fact `time(0..N).` for horizon N

Optimization statements

- `#maximize{N,T:goal(a,N,T)}`

× Assumes player b will chose actions under the same optimization.

Pruned Minimax Algorithm

Motivation

- Construct only the necessary parts of the tree (Alphabeta pruning)
- Treat it as a planning problem
- Use optimization statements from *clingo*

Explicit time encoding

1. Replace `true(F)` by `holds(F,T)`
2. Replace `next(F)` by `holds(F,T+1)`
3. Replace `p(F1..Fn)` by `p(F1..Fn,T)` for $\{legal, does, goal, terminal\}$
4. Add the time in in the body with predicate `time(T)`
5. Add a new fact `time(0..N).` for horizon N

Optimization statements

- `#maximize{N,T:goal(a,N,T)}`

× Assumes player b will chose actions under the same optimization.

Pruned Minimax Algorithm

Motivation

- Construct only the necessary parts of the tree (Alphabeta pruning)
- Treat it as a planning problem
- Use optimization statements from *clingo*

Explicit time encoding

1. Replace `true(F)` by `holds(F,T)`
2. Replace `next(F)` by `holds(F,T+1)`
3. Replace `p(F1..Fn)` by `p(F1..Fn,T)` for $\{legal, does, goal, terminal\}$
4. Add the time in in the body with predicate `time(T)`
5. Add a new fact `time(0..N).` for horizon N

Optimization statements

- `#maximize{N,T:goal(a,N,T)}`

× Assumes player b will chose actions under the same optimization.

Pruned Minimax Algorithm

Motivation

- Construct only the necessary parts of the tree (Alphabeta pruning)
- Treat it as a planning problem
- Use optimization statements from *clingo*

Explicit time encoding

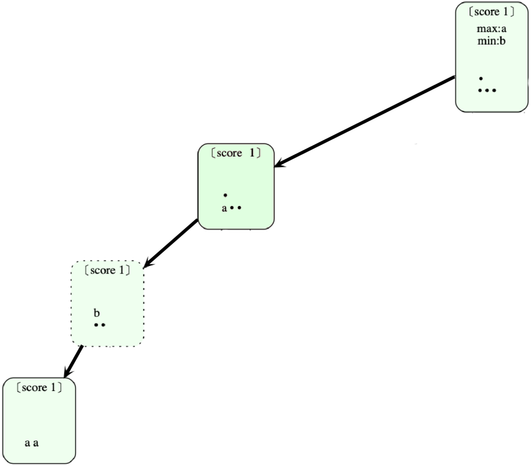
1. Replace `true(F)` by `holds(F,T)`
2. Replace `next(F)` by `holds(F,T+1)`
3. Replace `p(F1..Fn)` by `p(F1..Fn,T)` for $\{legal, does, goal, terminal\}$
4. Add the time in in the body with predicate `time(T)`
5. Add a new fact `time(0..N).` for horizon N

Optimization statements

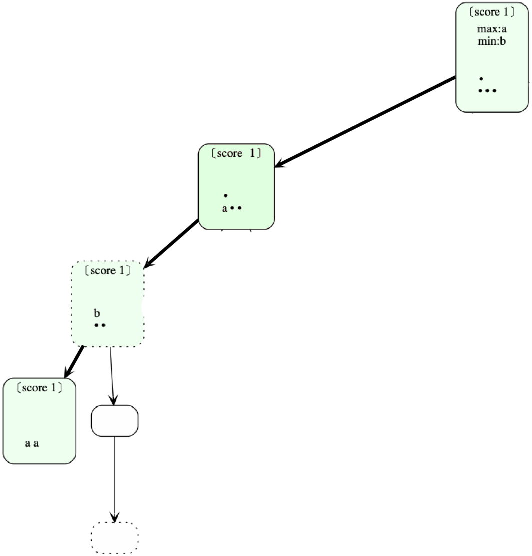
- `#maximize{N,T:goal(a,N,T)}`

× Assumes player b will chose actions under the same optimization.

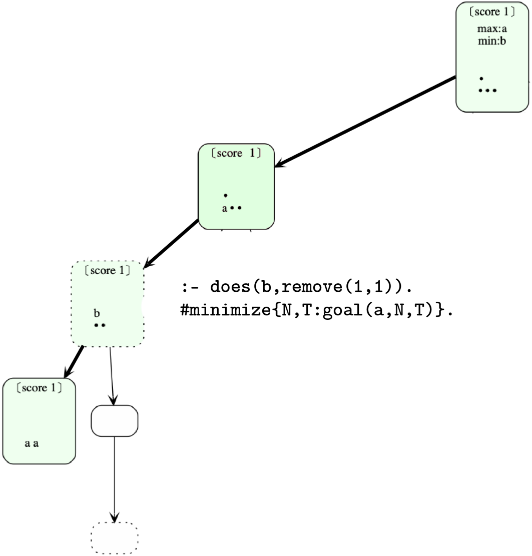
Pruned Minimax Algorithm



Pruned Minimax Algorithm

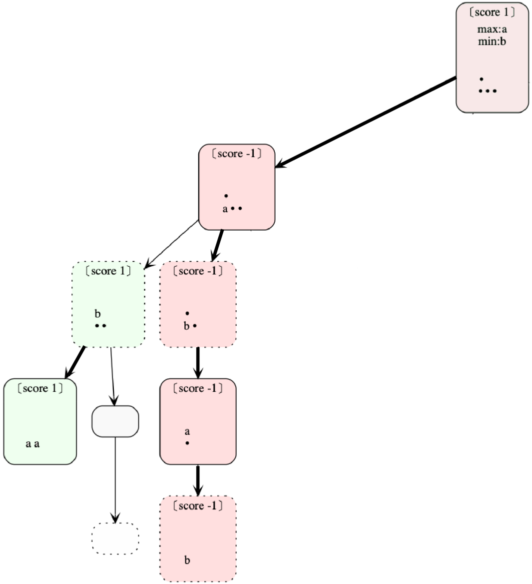


Pruned Minimax Algorithm

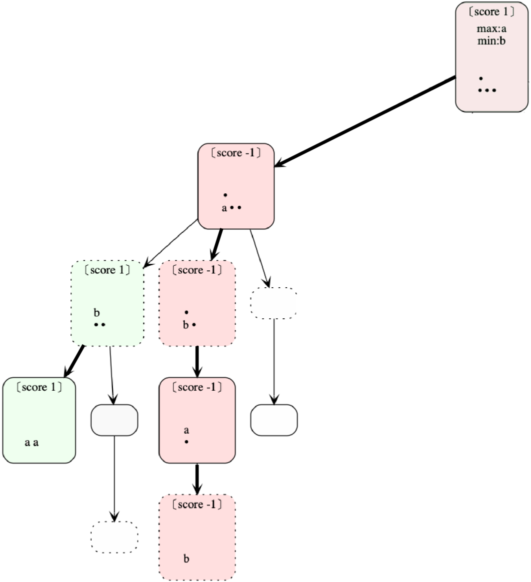



```
:- does(b,remove(1,1)).
#minimize{N,T:goal(a,N,T)}.
```

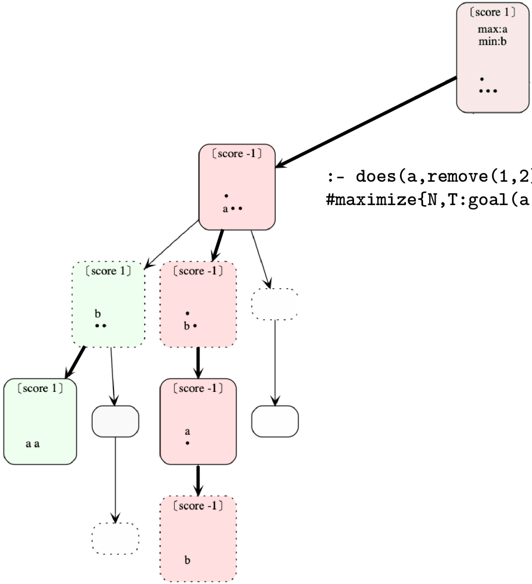
Pruned Minimax Algorithm



Pruned Minimax Algorithm

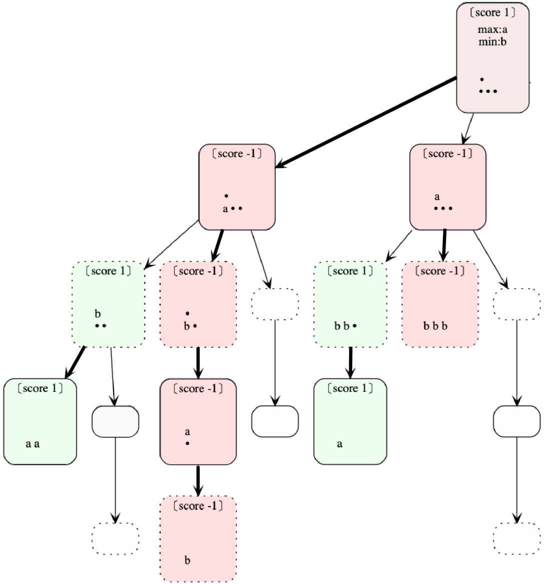


Pruned Minimax Algorithm

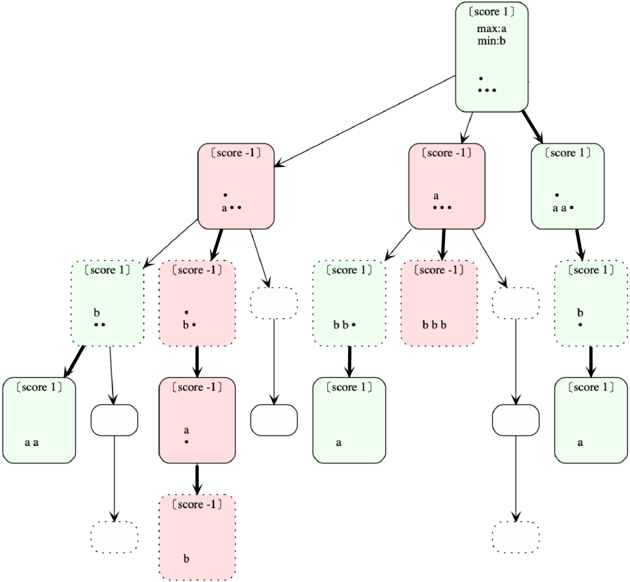



```
:- does(a,remove(1,2)).
#maximize{N,T:goal(a,N,T)}.
```

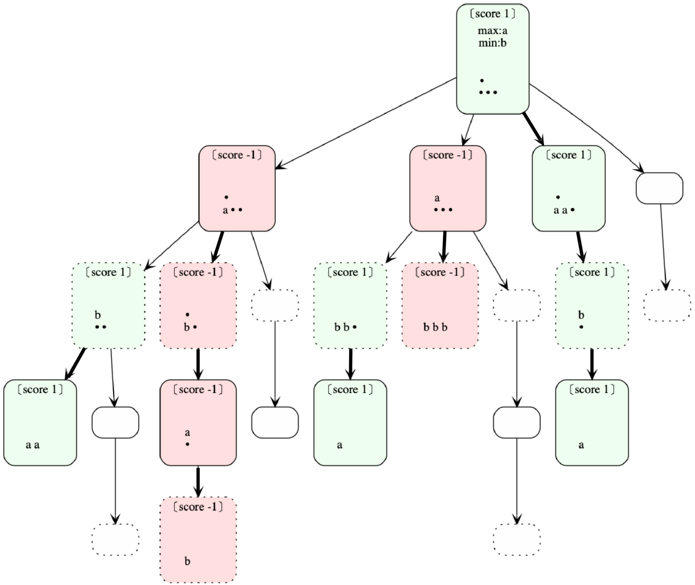
Pruned Minimax Algorithm



Pruned Minimax Algorithm



Pruned Minimax Algorithm



Pruned Minimax Algorithm + Learning Rules

Learn Rules: Create a representation of the best moves using ASP

Example

```
best_do(a,remove(2,2),T):- holds(control(a),T), holds(has(1,0),T),
                           holds(has(2,2),T), holds(has(3,0),T),
                           holds(has(4,0),T).
```

Enforce the use of these best actions

```
1{does(P,A,T):best_do(P,A,T)}1:- time(T), not goal(_,_,T),
                                 best_do(P,A,T)>0, true(control(P)).
```

Generalize using predefined options for substitution

Example

```
best_do(Va,remove(V2,1),T):- holds(control(Va),T),
                            holds(has(V1,0),T), holds(has(V2,2),T),
                            holds(has(V3,0),T), holds(has(V4,0),T),
                            V1!=V3,V1!=V2,V1!=Va,
                            V3!=V2,V3!=Va,V2!=Va.
```

Pruned Minimax Algorithm + Learning Rules

Learn Rules: Create a representation of the best moves using ASP

Example

```
best_do(a,remove(2,2),T):- holds(control(a),T), holds(has(1,0),T),
                           holds(has(2,2),T), holds(has(3,0),T),
                           holds(has(4,0),T).
```

Enforce the use of these best actions

```
1{does(P,A,T):best_do(P,A,T)}1:- time(T), not goal(_,_,T),
                                 best_do(P,A,T)>0, true(control(P)).
```

Generalize using predefined options for substitution

Example

```
best_do(Va,remove(V2,1),T):- holds(control(Va),T),
                        holds(has(V1,0),T), holds(has(V2,2),T),
                        holds(has(V3,0),T), holds(has(V4,0),T),
                        V1!=V3,V1!=V2,V1!=Va,
                        V3!=V2,V3!=Va,V2!=Va.
```

Pruned Minimax Algorithm + Learning Rules

Learn Rules: Create a representation of the best moves using ASP

Example

```
best_do(a,remove(2,2),T):- holds(control(a),T), holds(has(1,0),T),
                          holds(has(2,2),T), holds(has(3,0),T),
                          holds(has(4,0),T).
```

Enforce the use of these best actions

```
1{does(P,A,T):best_do(P,A,T)}1:- time(T), not goal(_,_,T),
                                best_do(P,A,T)>0, true(control(P)).
```

Generalize using predefined options for substitution

Example

```
best_do(Va,remove(V2,1),T):- holds(control(Va),T),
                        holds(has(V1,0),T), holds(has(V2,2),T),
                        holds(has(V3,0),T), holds(has(V4,0),T),
                        V1!=V3,V1!=V2,V1!=Va,
                        V3!=V2,V3!=Va,V2!=Va.
```

Pruned Minimax Algorithm + Learning Rules

Learn Rules: Create a representation of the best moves using ASP

Example

```
best_do(a,remove(2,2),T):- holds(control(a),T), holds(has(1,0),T),
                           holds(has(2,2),T), holds(has(3,0),T),
                           holds(has(4,0),T).
```

Enforce the use of these best actions

```
1{does(P,A,T):best_do(P,A,T)}1:- time(T), not goal(_,_,T),
                                 best_do(P,A,T)>0, true(control(P)).
```

Generalize using predefined options for substitution

Example

```
best_do(Va,remove(V2,1),T):- holds(control(Va),T),
                             holds(has(V1,0),T), holds(has(V2,2),T),
                             holds(has(V3,0),T), holds(has(V4,0),T),
                             V1!=V3,V1!=V2,V1!=Va,
                             V3!=V2,V3!=Va,V2!=Va.
```

Weak constraints with ILASP

- Background Knowledge: Encoding with rules
- Ordered positive examples: Generated in the decision points of Pruned Minimax

Example

```
#pos(e0,{}, {},{
  true(control(a)). true(has(1,0)). true(has(2,2)).
  true(has(3,0)). true(has(3,0)). does(a,remove(2,2)).}).
#pos(e1,{}, {}, {
  true(control(a)). true(has(1,0)). true(has(2,2)).
  true(has(3,0)). true(has(4,0)). does(a,remove(2,1)). }).
#brave_ordering(e0,e1).
```

- Language bias: Must be handcrafted to define the search space of the ILASP framework

$\implies$ Hypotheses: Weak constraints representing the strategy

Weak constraints with ILASP

- Background Knowledge: Encoding with rules
- Ordered positive examples: Generated in the decision points of Pruned Minimax

Example

```
#pos(e0,{}, {},{
  true(control(a)). true(has(1,0)). true(has(2,2)).
  true(has(3,0)). true(has(3,0)). does(a,remove(2,2)).}).
#pos(e1,{}, {}, {
  true(control(a)). true(has(1,0)). true(has(2,2)).
  true(has(3,0)). true(has(4,0)). does(a,remove(2,1)). }).
#brave_ordering(e0,e1).
```

- Language bias: Must be handcrafted to define the search space of the ILASP framework

$\Longrightarrow$ Hypotheses: Weak constraints representing the strategy

Weak constraints with ILASP

- Background Knowledge: Encoding with rules
- Ordered positive examples: Generated in the decision points of Pruned Minimax

Example

```
#pos(e0,{}, {},{
  true(control(a)). true(has(1,0)). true(has(2,2)).
  true(has(3,0)). true(has(3,0)). does(a,remove(2,2)).}).
#pos(e1,{}, {}, {
  true(control(a)). true(has(1,0)). true(has(2,2)).
  true(has(3,0)). true(has(4,0)). does(a,remove(2,1)). }).
#brave_ordering(e0,e1).
```

- Language bias: Must be handcrafted to define the search space of the ILASP framework

$\Longrightarrow$ Hypotheses: Weak constraints representing the strategy

Weak constraints with ILASP

- Background Knowledge: Encoding with rules
- Ordered positive examples: Generated in the decision points of Pruned Minimax

Example

```
#pos(e0,{}, {},{
  true(control(a)). true(has(1,0)). true(has(2,2)).
  true(has(3,0)). true(has(3,0)). does(a,remove(2,2)).}).
#pos(e1,{}, {}, {
  true(control(a)). true(has(1,0)). true(has(2,2)).
  true(has(3,0)). true(has(4,0)). does(a,remove(2,1)). }).
#brave_ordering(e0,e1).
```

- Language bias: Must be handcrafted to define the search space of the ILASP framework

$\Longrightarrow$ Hypotheses: Weak constraints representing the strategy

Weak constraints with ILASP

- Background Knowledge: Encoding with rules
- Ordered positive examples: Generated in the decision points of Pruned Minimax

Example

```
#pos(e0,{}, {},{
  true(control(a)). true(has(1,0)). true(has(2,2)).
  true(has(3,0)). true(has(3,0)). does(a,remove(2,2)).}).
#pos(e1,{}, {}, {
  true(control(a)). true(has(1,0)). true(has(2,2)).
  true(has(3,0)). true(has(4,0)). does(a,remove(2,1)). }).
#brave_ordering(e0,e1).
```

- Language bias: Must be handcrafted to define the search space of the ILASP framework

$\Longrightarrow$ Hypotheses: Weak constraints representing the strategy

Weak constraints with ILASP

- Background Knowledge: Encoding with rules
- Ordered positive examples: Generated in the decision points of Pruned Minimax

Example

```
#pos(e0,{}, {},{
  true(control(a)). true(has(1,0)). true(has(2,2)).
  true(has(3,0)). true(has(3,0)). does(a,remove(2,2)).}).
#pos(e1,{}, {}, {
  true(control(a)). true(has(1,0)). true(has(2,2)).
  true(has(3,0)). true(has(4,0)). does(a,remove(2,1)). }).
#brave_ordering(e0,e1).
```

- Language bias: Must be handcrafted to define the search space of the ILASP framework

$\Longrightarrow$ Hypotheses: Weak constraints representing the strategy

Results

Setup

- Time to learn strategy and its size
- Abstraction of strategy from a smaller instance

Setup

- Time to learn strategy and its size
- Abstraction of strategy from a smaller instance

Setup

- Time to learn strategy and its size
- Abstraction of strategy from a smaller instance

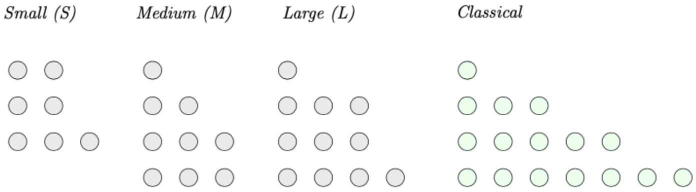


Fig. 14. Initial configurations for *Nim*.

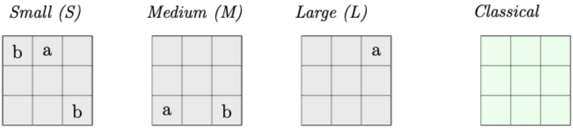


Fig. 15. Initial configurations for *TicTacToe*

Learning phase

Approach	*Nim*			*TicTacToe*		
	S	M	L	S	M	L
Minimax	390	4630	61023	896	7583	59704
Pruned minimax tree	83	347	2675	105	73	2835
Pruned minimax rules	35	85	283	109	61	2399

Table 1: Number of nodes in the computed trees

Learning phase ILASP strategies

Example (TicTacToe instance M)

```
:~ next(has(V0,V1)), next(has(V0,V2)), next(has(V0,V3)),
   in_line(V1,V2,V3).[-1@2, 2, V0, V1, V2, V3]
:~ in_line(V0,V1,V2), next(free(V2)).[-1@1, 1, V0, V1, V2]
```

Example (Language bias TicTacToe)

```
#modeo(3,next(has(var(player),var(cell))),(positive)).
#modeo(1,in_line(var(cell),var(cell),var(cell)),(positive)).
#modeo(1,next(free(var(cell))),(positive)).
#modeo(1,next(control(var(player))),(positive)).
#weight(-1). #weight(1).
#maxp(2). #maxv(4).
```

Learning phase ILASP strategies

Example (TicTacToe instance M)

```
:~ next(has(V0,V1)), next(has(V0,V2)), next(has(V0,V3)),
   in_line(V1,V2,V3).[-1@2, 2, V0, V1, V2, V3]
:~ in_line(V0,V1,V2), next(free(V2)).[-1@1, 1, V0, V1, V2]
```

Example (Language bias TicTacToe)

```
#modeo(3,next(has(var(player),var(cell))),(positive)).
#modeo(1,in_line(var(cell),var(cell),var(cell)),(positive)).
#modeo(1,next(free(var(cell))),(positive)).
#modeo(1,next(control(var(player))),(positive)).
#weight(-1). #weight(1).
#maxp(2). #maxv(4).
```

Learning phase ILASP strategies

Example (Nim instances M and L)

```
b_pile(V3,V1,V2) :- binary(V0,V1,V2), next(has(V3,V0)).
nim_sum(V1,0,0) :- b_pile(_,V1,_).
nim_sum(V1,V3+V2,V0) :- b_pile(V0,V1,V2), nim_sum(V1,V3,V0-1).
:~ nim_sum(V0,V1,4), V1\2 != 0.[1@1, 4, V0, V1]
```

Example (Language bias Nim)

```
#constant(pile,1..4).
#modeh(b(var(pile),var(d),var(bool)),(positive)).
#modeh(nim_sum(var(d),var(total)+var(bool),var(pile)),(positive)).
#modeh(nim_sum(var(d),0,0),(positive)).
#modeb(1,b(var(pile),var(d),var(bool)),(positive)).
#modeb(1,nim_sum(var(d),var(total),var(pile)-1),(positive)).
#modeb(1,binary(var(num),var(d),var(bool)),(positive)).
#modeb(1,next(has(var(pile),var(num))),(positive)).
#modeo(1,nim_sum(var(d),var(t),const(pile))).
#modeo(1,var(t)\2 != 0).
#weight(1). #weight(-1). #maxp(1). #maxv(4).
```

Learning phase ILASP strategies

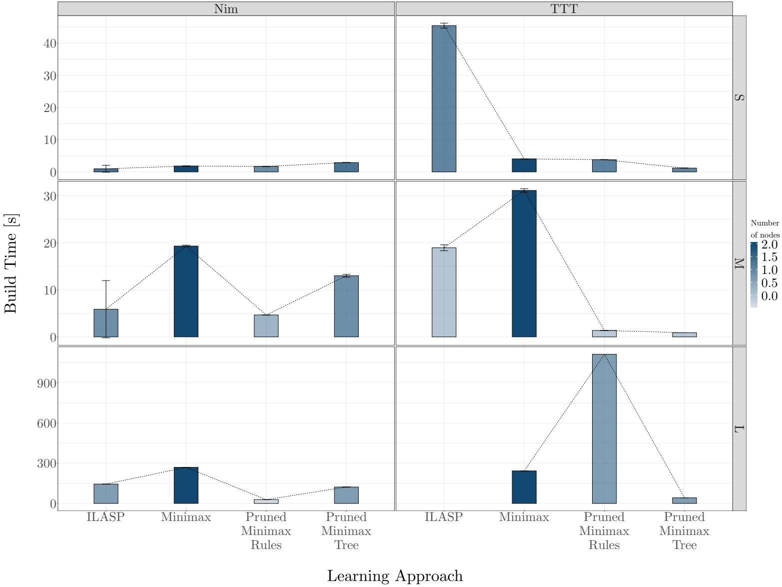
Example (Nim instances *M* and *L*)

```
b_pile(V3,V1,V2) :- binary(V0,V1,V2), next(has(V3,V0)).
nim_sum(V1,0,0) :- b_pile(_,V1,_).
nim_sum(V1,V3+V2,V0) :- b_pile(V0,V1,V2), nim_sum(V1,V3,V0-1).
:~ nim_sum(V0,V1,4), V1\2 != 0.[1@1, 4, V0, V1]
```

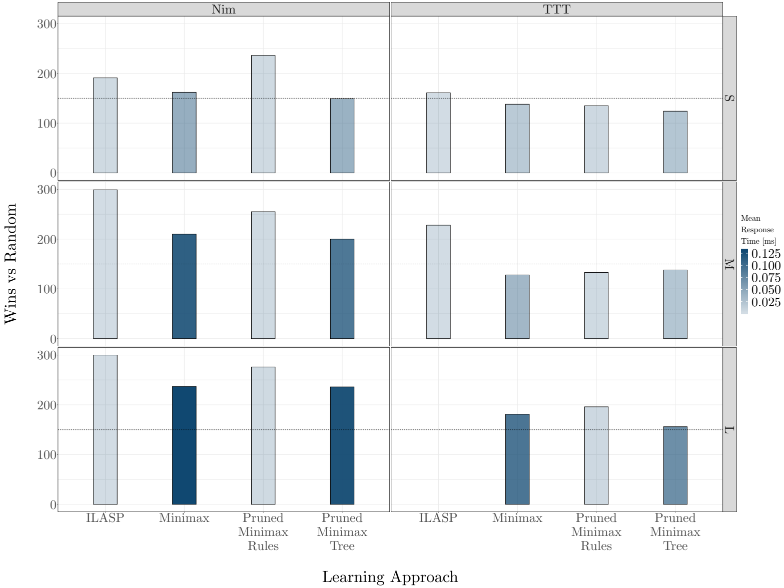
Example (Language bias Nim)

```
#constant(pile,1..4).
#modeh(b(var(pile),var(d),var(bool)),(positive)).
#modeh(nim_sum(var(d),var(total)+var(bool),var(pile)),(positive)).
#modeh(nim_sum(var(d),0,0),(positive)).
#modeb(1,b(var(pile),var(d),var(bool)),(positive)).
#modeb(1,nim_sum(var(d),var(total),var(pile)-1),(positive)).
#modeb(1,binary(var(num),var(d),var(bool)),(positive)).
#modeb(1,next(has(var(pile),var(num))),(positive)).
#modeo(1,nim_sum(var(d),var(t),const(pile))).
#modeo(1,var(t)\2 != 0).
#weight(1). #weight(-1). #maxp(1). #maxv(4).
```

Learning phase



Play against Random-Agent



Conclusion

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Conclusion

- Minimax
  - ✓ Good for small instances
  - × Slow learning and not scalable
- Pruned Minimax
  - ✓ Reduces tree search space
- Pruned Minimax Learning Rules
  - ✓ Reduces tree search space
  - ✓ Faster making decision
  - ✓ Allow generalization
  - × Requires symmetry specification
  - ✓ Novel approach
- ILASP
  - ✓ Explainable abstract strategies
  - × Dependent on language bias

Our framework

- Using ASP for the game dynamics
- Extendible with new learning approaches
- Customizable games
- Command line tools
- Game Tree visualizations

Bibliography

References

- Mohaned Abu Dalffa, Bassem S Abu-Nasser, and Samy S Abu-Naser.
  Tic-tac-toe learning using artificial neural networks.
  2019.
- Christian Anger, Kathrin Konczak, Thomas Linke, and Torsten Schaub.
  A glimpse of answer set programming.
  19:12–, 2005.
- Charles L Bouton.
  Nim, a game with a complete mathematical theory.
  The Annals of Mathematics, 3(1/4):35–39, 1901.
- Erik D. Demaine, Fermi Ma, and Erik Waingarten.
  Playing dominoes is hard, except by yourself.
  In Alfredo Ferro, Fabrizio Luccio, and Peter Widmayer, editors, Fun with Algorithms, pages 137–146, Cham, 2014. Springer International Publishing.
  ISBN 978-3-319-07890-8.
- Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub.
  Clingo= asp+ control: Preliminary report.
  arXiv preprint arXiv:1405.3694, 2014.
- Michael Gelfond and Vladimir Lifschitz.
  Classical negation in logic programs and disjunctive databases.
  New Generation Computing, 9(3):365–385, Aug 1991.
  ISSN 1882-7055.

An analysis of alpha-beta pruning.
Artificial intelligence, 6(4):293–326, 1975.

- Mark Law, Alessandra Russo, and Krysia Broda.
  The ILASP system for learning answer set programs.
  www.ilasp.com, 2015.
- Mark Law, Alessandra Russo, and Krysia Broda.
  Inductive learning of answer set programs v3. 1.0, 2017.
- Vladimir Lifschitz.
  Answer set programming.
  Springer International Publishing, 2019.
- Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth.
  General game playing: Game description language specification.
  Stanford University, 2008.
- Stuart J Russell and Peter Norvig.
  Artificial intelligence: a modern approach.
  Malaysia; Pearson Education Limited,, 2016.
- Murray Shanahan.
  The event calculus explained.
  In Artificial intelligence today, pages 409–430. Springer, 1999.
- David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan,