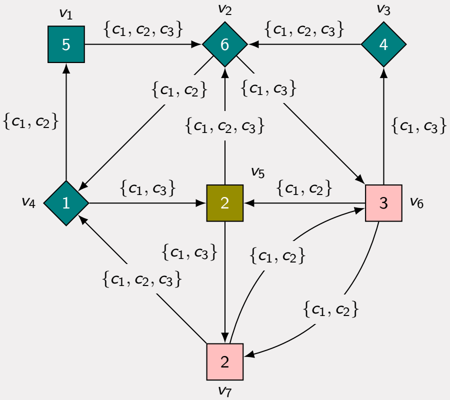
Verifying SPLs using parity games expressing variability



Sjef van Loo

6 November, 2019

Msc Thesis
Computer Science and Engineering

Supervised by T.A.C. Willemse

TU/e

Outline

- Verification & SPLs
- Problem statement
- Variability Parity Games & algorithms
- Experimental results
- Conclusions

Verification

- Creating correct software is difficult
- Even when testing is done rigorously errors can slip in

TU/e

Verification

- Creating correct software is difficult
- Even when testing is done rigorously errors can slip in
- Mathematically *model the behaviour* of software (LTS)
- Mathematically *specify a requirement* (modal μ-calculus)

Verification

- Creating correct software is difficult
- Even when testing is done rigorously errors can slip in
- Mathematically *model the behaviour* of software (LTS)
- Mathematically *specify a requirement* (modal μ-calculus)
- Check if the model satisfies the requirement

LTS M Modal μ-calculus formula φ

M satisfies φ?

Software product lines

- Software product lines are configurable systems
- Many variants of the same system, i.e. *software products*
- *e.g.* an elevator that can be configured to detect overload

Software product lines

- Software product lines are configurable systems
- Many variants of the same system, i.e. *software products*
- *e.g.* an elevator that can be configured to detect overload
- FTSs can be used to model the entire system using *features*

Software product lines

- Software product lines are configurable systems
- Many variants of the same system, i.e. *software products*
- *e.g.* an elevator that can be configured to detect overload
- FTSs can be used to model the entire system using *features*
- An FTS can be transformed to an LTS given a specific feature assignment

Problem statement

- Find all the products in an SPL that satisfy a requirement

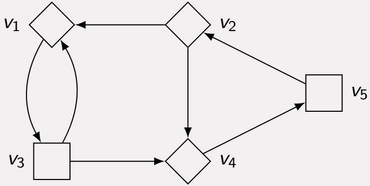
FTS M Modal μ-calculus formula φ

Which products in M satisfy φ?

- Do so more efficiently than verifying every product independently

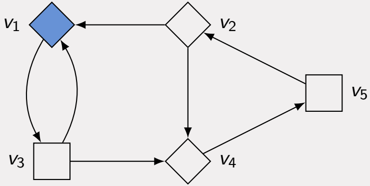
Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



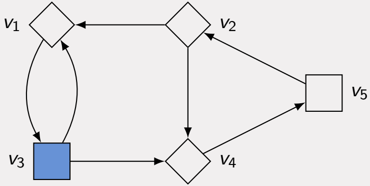
Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\diamond$) and 1 (odd,$\square$)



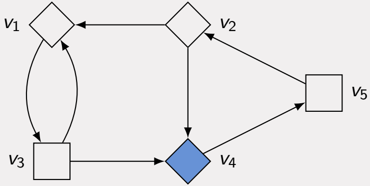
Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



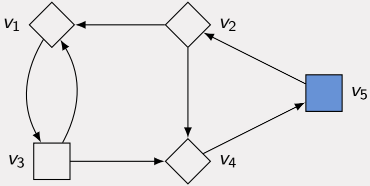
Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



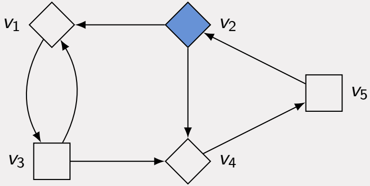
Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even, $\diamond$) and 1 (odd, $\square$)



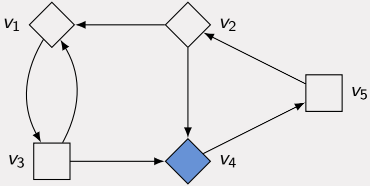
Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



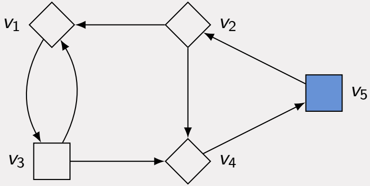
Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



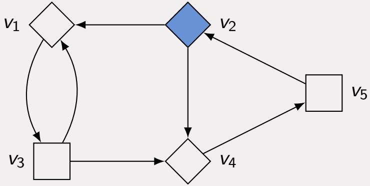
Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



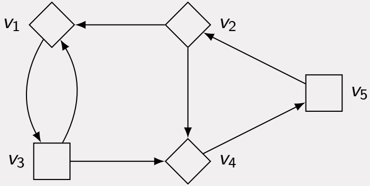
Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



Variability parity game

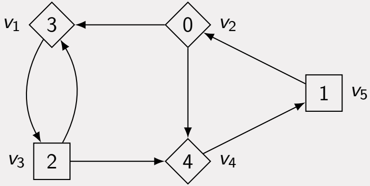
Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



- Infinite path starting at some vertex

Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)

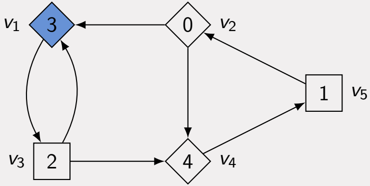


- Infinite path starting at some vertex
- The winner is determined by the parity of the priority occurring infinitely often

Variability parity game

Parity game: (V, V_0, V_1, E, Ω)

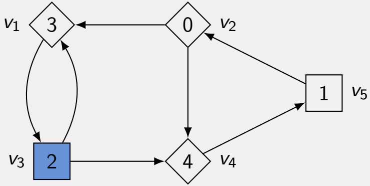
Players 0 (even,$\diamond$) and 1 (odd,$\square$)



- Infinite path starting at some vertex
- The winner is determined by the parity of the priority occurring infinitely often

Variability parity game

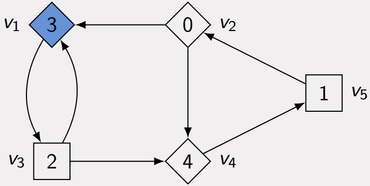
Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



- Infinite path starting at some vertex
- The winner is determined by the parity of the priority occurring infinitely often

Variability parity game

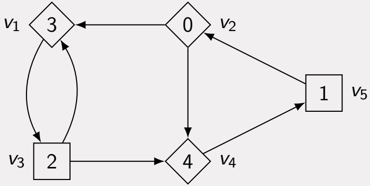
Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



- Infinite path starting at some vertex
- The winner is determined by the parity of the priority occurring infinitely often

Variability parity game

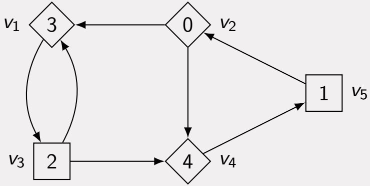
Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



- Infinite path starting at some vertex
- The winner is determined by the parity of the priority occurring infinitely often
- Player 1 wins $\{v_1, v_3\}$, using $v_3 \mapsto v_1$

Variability parity game

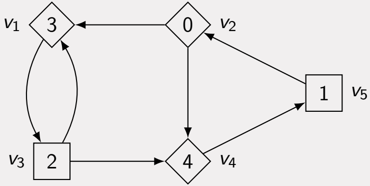
Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,◇) and 1 (odd,□)



- Infinite path starting at some vertex
- The winner is determined by the parity of the priority occurring infinitely often
- Player 1 wins $\{v_1, v_3\}$, using $v_3 \mapsto v_1$
- Player 0 wins $\{v_2, v_4, v_5\}$, using $v_2 \mapsto v_4$

Variability parity game

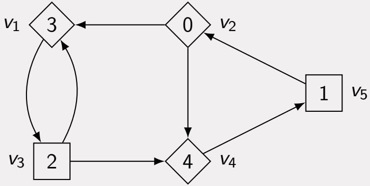
Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



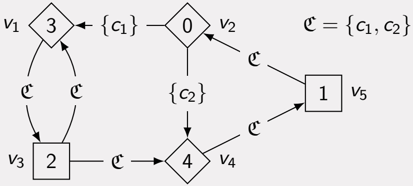
- Infinite path starting at some vertex
- The winner is determined by the parity of the priority occurring infinitely often
- Player 1 wins $\{v_1, v_3\}$, using $v_3 \mapsto v_1$
- Player 0 wins $\{v_2, v_4, v_5\}$, using $v_2 \mapsto v_4$
- *Solving*: Partition the vertices in W_0, W_1

Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



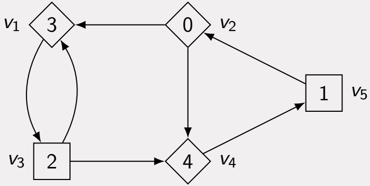
Variability parity game:
$(V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$



- Infinite path starting at some vertex
- The winner is determined by the parity of the priority occurring infinitely often
- Player 1 wins $\{v_1, v_3\}$, using $v_3 \mapsto v_1$
- Player 0 wins $\{v_2, v_4, v_5\}$, using $v_2 \mapsto v_4$
- *Solving*: Partition the vertices in W_0, W_1

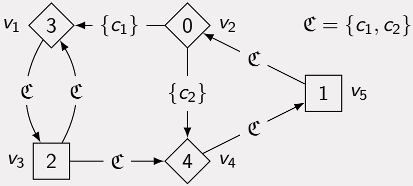
Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even, $\Diamond$) and 1 (odd, $\Box$)



- Infinite path starting at some vertex
- The winner is determined by the parity of the priority occurring infinitely often
- Player 1 wins $\{v_1, v_3\}$, using $v_3 \mapsto v_1$
- Player 0 wins $\{v_2, v_4, v_5\}$, using $v_2 \mapsto v_4$
- *Solving*: Partition the vertices in W_0, W_1

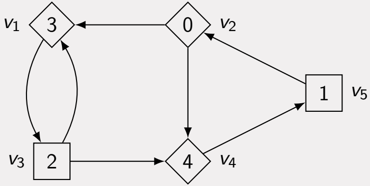
Variability parity game:
$(V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$



- Played for a specific configuration $c \in \mathfrak{C}$

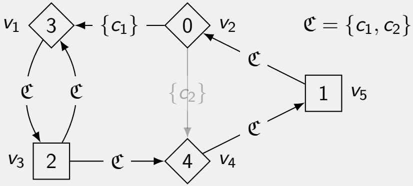
Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



- Infinite path starting at some vertex
- The winner is determined by the parity of the priority occurring infinitely often
- Player 1 wins $\{v_1, v_3\}$, using $v_3 \mapsto v_1$
- Player 0 wins $\{v_2, v_4, v_5\}$, using $v_2 \mapsto v_4$
- *Solving*: Partition the vertices in W_0, W_1

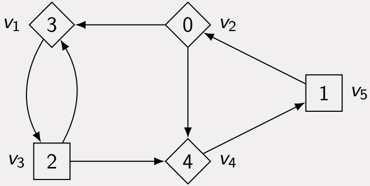
Variability parity game:
$(V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$



- Played for a specific configuration $c \in \mathfrak{C}$
- $W_0^{c_1} = \emptyset, W_1^{c_1} = \{v_1, \ldots, v_5\}$

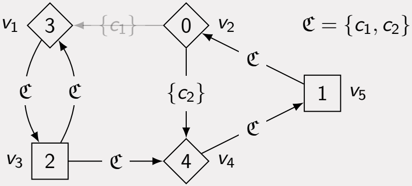
Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even,$\Diamond$) and 1 (odd,$\Box$)



- Infinite path starting at some vertex
- The winner is determined by the parity of the priority occurring infinitely often
- Player 1 wins $\{v_1, v_3\}$, using $v_3 \mapsto v_1$
- Player 0 wins $\{v_2, v_4, v_5\}$, using $v_2 \mapsto v_4$
- *Solving*: Partition the vertices in W_0, W_1

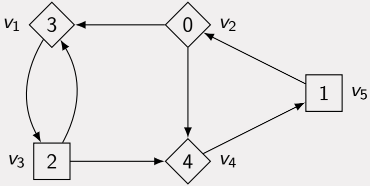
Variability parity game:
$(V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$



- Played for a specific configuration $c \in \mathfrak{C}$
- $W_0^{c_1} = \emptyset, W_1^{c_1} = \{v_1, \ldots, v_5\}$
- $W_0^{c_2} = \{v_1, v_3\}, W_1^{c_2} = \{v_2, v_4, v_5\}$

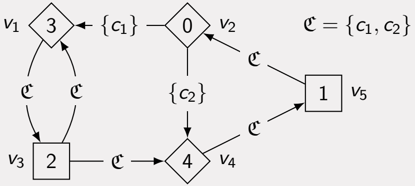
Variability parity game

Parity game: (V, V_0, V_1, E, Ω)
Players 0 (even, $\Diamond$) and 1 (odd, $\Box$)



- Infinite path starting at some vertex
- The winner is determined by the parity of the priority occurring infinitely often
- Player 1 wins $\{v_1, v_3\}$, using $v_3 \mapsto v_1$
- Player 0 wins $\{v_2, v_4, v_5\}$, using $v_2 \mapsto v_4$
- *Solving*: Partition the vertices in W_0, W_1

Variability parity game:
$(V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$



- Played for a specific configuration $c \in \mathfrak{C}$
- $W_0^{c_1} = \emptyset, W_1^{c_1} = \{v_1, \ldots, v_5\}$
- $W_0^{c_2} = \{v_1, v_3\}, W_1^{c_2} = \{v_2, v_4, v_5\}$
- *Solving*: Partition the vertices in W_0^c, W_1^c, for every $c \in \mathfrak{C}$

Variability parity game

LTS M

Modal μ-calculus formula φ

M satisfies φ?

Variability parity game

$$\begin{tikzcd}
\text{LTS } M \arrow[dr, dash] & & \text{Modal } \mu\text{-calculus formula } \varphi \arrow[dl, dash] \\
& \text{PG} \arrow[d] & \\
& M \text{ satisfies } \varphi? &
\end{tikzcd}$$

Theorem

A parity game can be constructed from an LTS and a modal μ-calculus formula φ such that M satisfies φ iff special vertex v_0 is won by player 0 in the resulting parity game.

Variability parity game

LTS M Modal μ-calculus formula φ

PG

$\downarrow$

M satisfies φ?

FTS M Modal μ-calculus formula φ

$\downarrow$

Which products in M satisfy φ?

Theorem

A parity game can be constructed from an LTS and a modal μ-calculus formula φ such that M satisfies φ iff special vertex v_0 is won by player 0 in the resulting parity game.

Variability parity game

$$\begin{array}{ccc} \text{LTS } M & & \text{Modal } \mu\text{-calculus formula } \varphi \\ & \searrow \quad \swarrow & \\ & \text{PG} & \\ & \downarrow & \\ & M \text{ satisfies } \varphi? & \end{array}$$

$$\begin{array}{ccc} \text{FTS } M & & \text{Modal } \mu\text{-calculus formula } \varphi \\ & \searrow \quad \swarrow & \\ & \text{VPG} & \\ & \downarrow & \\ & \text{Which products in } M \text{ satisfy } \varphi? & \end{array}$$

Theorem

A parity game can be constructed from an LTS and a modal μ-calculus formula φ such that M satisfies φ iff special vertex v_0 is won by player 0 in the resulting parity game.

Theorem

A VPG can be constructed from an FTS and a modal μ-calculus formula φ such that M satisfies φ for product p iff special vertex v_0 is won by player 0 in the resulting VPG played for p.

Variability parity game

Theorem

A parity game can be constructed from an LTS and a modal μ-calculus formula φ such that M satisfies φ iff special vertex v_0 is won by player 0 in the resulting parity game.

Theorem

A VPG can be constructed from an FTS and a modal μ-calculus formula φ such that M satisfies φ for product p iff special vertex v_0 is won by player 0 in the resulting VPG played for p.

$$
\begin{tikzcd}
\text{FTS} \arrow[d, dashed, "\Pi_p"'] & \\
\text{LTS} \arrow[r, "\varphi"] & \text{PG}
\end{tikzcd}
$$

Variability parity game

Theorem

A parity game can be constructed from an LTS and a modal μ-calculus formula φ such that M satisfies φ iff special vertex v_0 is won by player 0 in the resulting parity game.

Theorem

A VPG can be constructed from an FTS and a modal μ-calculus formula φ such that M satisfies φ for product p iff special vertex v_0 is won by player 0 in the resulting VPG played for p.

$$\begin{tikzcd}
\mathrm{FTS} \arrow[r, "\varphi"] \arrow[d, dashed, "\Pi_p"'] & \mathrm{VPG} \arrow[d, dashed, "\Pi_p"] \\
\mathrm{LTS} \arrow[r, "\varphi"] & \mathrm{PG}
\end{tikzcd}$$

VPG algorithms

- Solve VPGs *independently*; solve every parity game expressed by the VPG
- Solve VPGs *collectively*; solve the VPG as a whole

VPG algorithms

- Solve VPGs *independently*; solve every parity game expressed by the VPG
- Solve VPGs *collectively*; solve the VPG as a whole
- Introduced two collective algorithms
  - Recursive algorithm
  - Incremental pre-solve algorithm
- Evaluate performance of independent approach vs collective approach

VPG algorithms - Recursive algorithm

- Existing parity game algorithm
- Spends most time performing the *attractor* calculation
- Introduce an efficient attractor calculation for VPGs

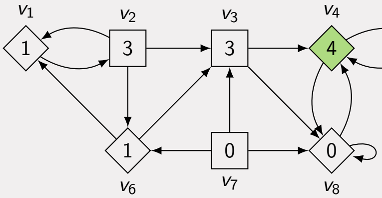
VPG algorithms - Recursive algorithm

Attractor calculation: Given player α and vertex set U: Find all vertices from where player α can force the play to U.

VPG algorithms - Recursive algorithm

Attractor calculation: Given player α and vertex set U: Find all vertices from where player α can force the play to U.

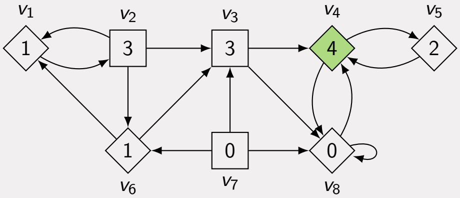
Example: $\alpha = 0, U = \{v_4\}$



VPG algorithms - Recursive algorithm

Attractor calculation: Given player α and vertex set U: Find all vertices from where player α can force the play to U.

Example: $\alpha = 0, U = \{v_4\}$



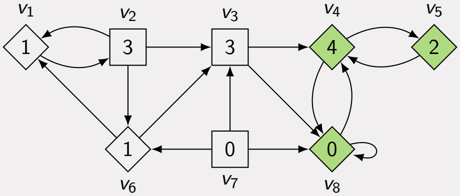
	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A				$\checkmark$				

TU/e

VPG algorithms - Recursive algorithm

Attractor calculation: Given player α and vertex set U: Find all vertices from where player α can force the play to U.

Example: $\alpha = 0, U = \{v_4\}$

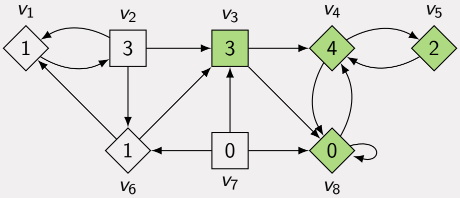


	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A				✓	✓			✓

VPG algorithms - Recursive algorithm

Attractor calculation: Given player α and vertex set U: Find all vertices from where player α can force the play to U.

Example: $\alpha = 0, U = \{v_4\}$

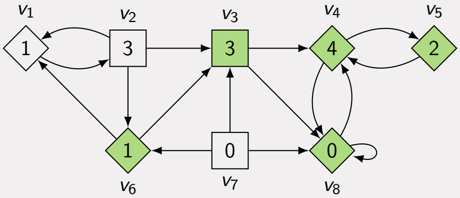


	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A			✓	✓	✓			✓

VPG algorithms - Recursive algorithm

Attractor calculation: Given player α and vertex set U: Find all vertices from where player α can force the play to U.

Example: $\alpha = 0, U = \{v_4\}$

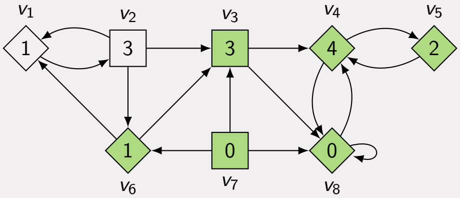


	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A			✓	✓	✓	✓		✓

VPG algorithms - Recursive algorithm

Attractor calculation: Given player α and vertex set U: Find all vertices from where player α can force the play to U.

Example: $\alpha = 0, U = \{v_4\}$

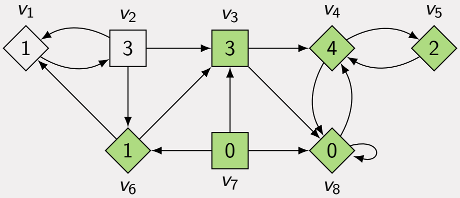


	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A			$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$

VPG algorithms - Recursive algorithm

Attractor calculation: Given player α and vertex set U: Find all vertices from where player α can force the play to U.

Example: $\alpha = 0, U = \{v_4\}$



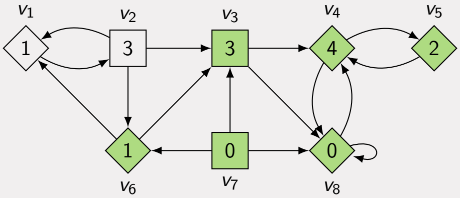
	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A			✓	✓	✓	✓	✓	✓

$$A_0 = U, A = \cup_{i \geq 0} A_i$$

$$A_{i+1} = A_i \cup \{v \in V_\alpha \mid \exists_w : (v, w) \in E \land w \in A_i\}$$
$$\cup \{v \notin V_\alpha \mid \forall_w : (v, w) \in E \implies w \in A_i\}$$

VPG algorithms - Recursive algorithm

Attractor calculation: Given player α and vertex set U: Find all vertices from where player α can force the play to U.
Example: $\alpha = 0, U = \{v_4\}$



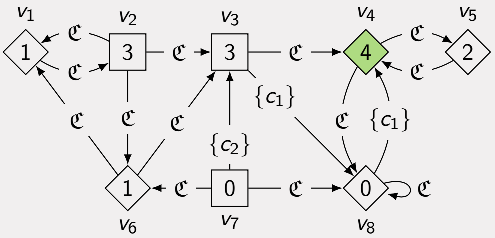
	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A_0				✓				
A_1				✓	✓			✓
A_2			✓	✓	✓			✓
A_3			✓	✓	✓	✓		✓
A_4			✓	✓	✓	✓	✓	✓
A_5			✓	✓	✓	✓	✓	✓

$$A_0 = U, A = \cup_{i \geq 0} A_i$$

$$A_{i+1} = A_i \cup \{v \in V_\alpha \mid \exists_w : (v,w) \in E \wedge w \in A_i\}$$
$$\cup \{v \notin V_\alpha \mid \forall_w : (v,w) \in E \implies w \in A_i\}$$

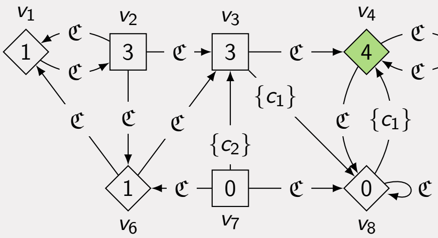
VPG algorithms - Recursive algorithm

For every vertex v, find a set of configurations C such that player α can force the play from v to U when the VPG is played for configuration $c \in C$. Example, $\mathfrak{C} = \{c_1, c_2\}$:



VPG algorithms - Recursive algorithm

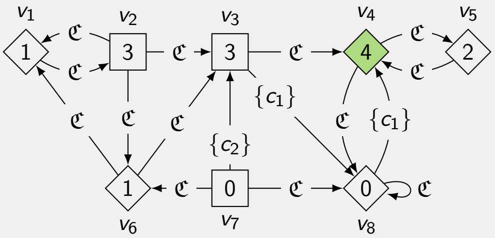
For every vertex v, find a set of configurations C such that player α can force the play from v to U when the VPG is played for configuration $c \in C$. Example, $\mathfrak{C} = \{c_1, c_2\}$:



$A : V \to 2^{\mathfrak{C}}$

VPG algorithms - Recursive algorithm

For every vertex v, find a set of configurations C such that player α can force the play from v to U when the VPG is played for configuration $c \in C$. Example, $\mathfrak{C} = \{c_1, c_2\}$:



$A : V \to 2^{\mathfrak{C}}$

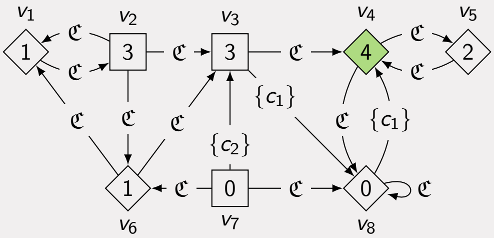
	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A_0				$\mathfrak{C}$				

$$A_{i+1}(v) = A_i(v) \cup \begin{cases} \bigcup_{(v,w)\in E} \left(\theta(v,w) \cap A_i(w)\right) & \text{if } v \in V_\alpha \\ \bigcap_{(v,w)\in E} \left((\mathfrak{C} \backslash \theta(v,w)) \cup A_i(w)\right) & \text{if } v \notin V_\alpha \end{cases}^{*}$$

*: Simplified version of the attractor definition presented in the report

VPG algorithms - Recursive algorithm

For every vertex v, find a set of configurations C such that player α can force the play from v to U when the VPG is played for configuration $c \in C$. Example, $\mathfrak{C} = \{c_1, c_2\}$:



$A : V \to 2^{\mathfrak{C}}$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A_0				$\mathfrak{C}$				

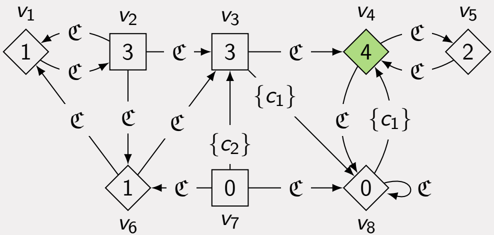
If $v \in V_\alpha$: Find for every outgoing edge which configurations play to A_i, add them all to $A_i(v)$

$$A_{i+1}(v) = A_i(v) \cup \begin{cases} \bigcup_{(v,w)\in E} \left(\theta(v,w) \cap A_i(w)\right) & \text{if } v \in V_\alpha \\ \bigcap_{(v,w)\in E} \left((\mathfrak{C}\backslash\theta(v,w)) \cup A_i(w)\right) & \text{if } v \notin V_\alpha \end{cases}^{*}$$

*: Simplified version of the attractor definition presented in the report

VPG algorithms - Recursive algorithm

For every vertex v, find a set of configurations C such that player α can force the play from v to U when the VPG is played for configuration $c \in C$. Example, $\mathfrak{C} = \{c_1, c_2\}$:



$A : V \to 2^{\mathfrak{C}}$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A_0				$\mathfrak{C}$				

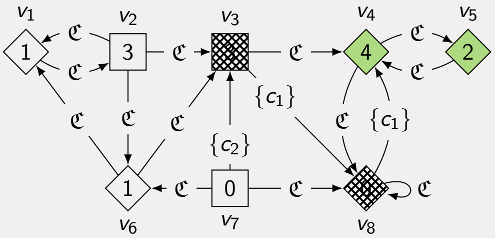
If $v \notin V_\alpha$: Find for every outgoing edge through which configurations player $1-\alpha$ cannot escape A_i, add the intersection to $A_i(v)$

$$A_{i+1}(v) = A_i(v) \cup \begin{cases} \bigcup_{(v,w)\in E} (\theta(v,w) \cap A_i(w)) & \text{if } v \in V_\alpha \\ \bigcap_{(v,w)\in E} ((\mathfrak{C}\backslash\theta(v,w)) \cup A_i(w)) & \text{if } v \notin V_\alpha \end{cases}{}^{*}$$

*: Simplified version of the attractor definition presented in the report

VPG algorithms - Recursive algorithm

For every vertex v, find a set of configurations C such that player α can force the play from v to U when the VPG is played for configuration $c \in C$. Example, $\mathfrak{C} = \{c_1, c_2\}$:



$A : V \to 2^{\mathfrak{C}}$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A_0				$\mathfrak{C}$				
A_1			$\{c_2\}$	$\mathfrak{C}$	$\mathfrak{C}$			$\{c_1\}$

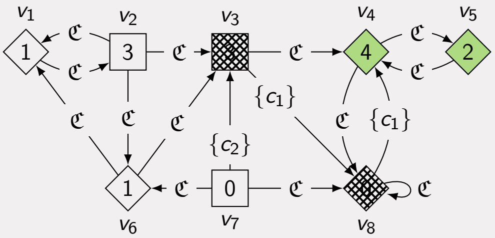
$$A_1(v_5) = A_0(v_5) \cup (\theta(v_5, v_4) \cap A_0(v_4))$$
$$= \emptyset \cup (\mathfrak{C} \cap \mathfrak{C}) = \mathfrak{C}$$

$$A_{i+1}(v) = A_i(v) \cup \begin{cases} \bigcup_{(v,w)\in E} (\theta(v,w) \cap A_i(w)) & \text{if } v \in V_\alpha \\ \bigcap_{(v,w)\in E} ((\mathfrak{C} \backslash \theta(v,w)) \cup A_i(w)) & \text{if } v \notin V_\alpha \end{cases}^{*}$$

*: Simplified version of the attractor definition presented in the report

VPG algorithms - Recursive algorithm

For every vertex v, find a set of configurations C such that player α can force the play from v to U when the VPG is played for configuration $c \in C$. Example, $\mathfrak{C} = \{c_1, c_2\}$:



$A : V \to 2^{\mathfrak{C}}$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A_0				$\mathfrak{C}$				
A_1			$\{c_2\}$	$\mathfrak{C}$	$\mathfrak{C}$			$\{c_1\}$

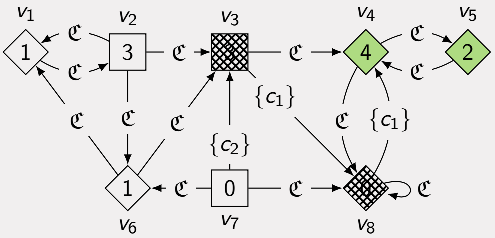
$$\begin{aligned} A_1(v_8) &= A_0(v_8) \cup (\theta(v_8, v_8) \cap A_0(v_8)) \\ &\cup (\theta(v_8, v_4) \cap A_0(v_4)) \\ &= \emptyset \cup (\mathfrak{C} \cap \emptyset) \cup (\{c_1\} \cap \mathfrak{C}) = \{c_1\} \end{aligned}$$

$$A_{i+1}(v) = A_i(v) \cup \begin{cases} \bigcup_{(v,w)\in E} (\theta(v,w) \cap A_i(w)) & \text{if } v \in V_\alpha \\ \bigcap_{(v,w)\in E} ((\mathfrak{C} \backslash \theta(v,w)) \cup A_i(w)) & \text{if } v \notin V_\alpha \end{cases}^{*}$$

*: Simplified version of the attractor definition presented in the report

VPG algorithms - Recursive algorithm

For every vertex v, find a set of configurations C such that player α can force the play from v to U when the VPG is played for configuration $c \in C$. Example, $\mathfrak{C} = \{c_1, c_2\}$:



$A : V \to 2^{\mathfrak{C}}$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A_0				$\mathfrak{C}$				
A_1			$\{c_2\}$	$\mathfrak{C}$	$\mathfrak{C}$			$\{c_1\}$

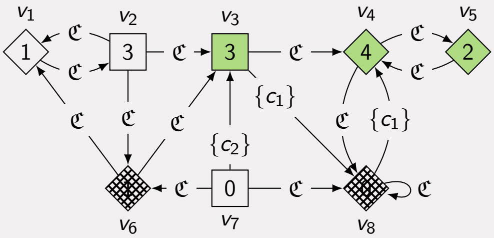
$$
\begin{aligned}
A_1(v_3) = A_0(v_3) \cup (\\
&((\mathfrak{C} \backslash \theta(v_3, v_4)) \cup A_0(v_4)) \cap \\
&((\mathfrak{C} \backslash \theta(v_3, v_8)) \cup A_0(v_8))) \\
&= \emptyset \cup ((\emptyset \cup \mathfrak{C}) \cap (\{c_2\} \cup \emptyset)) = \{c_2\}
\end{aligned}
$$

$$
A_{i+1}(v) = A_i(v) \cup \begin{cases} \bigcup_{(v,w) \in E} (\theta(v,w) \cap A_i(w)) & \text{if } v \in V_\alpha \\ \bigcap_{(v,w) \in E} ((\mathfrak{C} \backslash \theta(v,w)) \cup A_i(w)) & \text{if } v \notin V_\alpha \end{cases}^{*}
$$

*: Simplified version of the attractor definition presented in the report

VPG algorithms - Recursive algorithm

For every vertex v, find a set of configurations C such that player α can force the play from v to U when the VPG is played for configuration $c \in C$. Example, $\mathfrak{C} = \{c_1, c_2\}$:



$A : V \to 2^{\mathfrak{C}}$

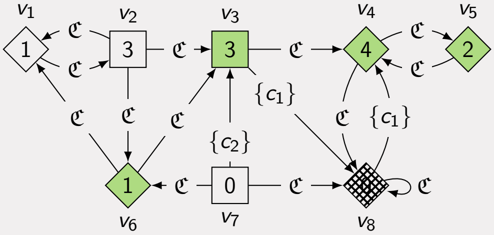
	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A_0				$\mathfrak{C}$				
A_1			$\{c_2\}$	$\mathfrak{C}$	$\mathfrak{C}$			$\{c_1\}$
A_2			$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	$\{c_2\}$		$\{c_1\}$

$$A_{i+1}(v) = A_i(v) \cup \begin{cases} \bigcup_{(v,w)\in E} \left(\theta(v,w) \cap A_i(w)\right) & \text{if } v \in V_\alpha \\ \bigcap_{(v,w)\in E} \left((\mathfrak{C} \backslash \theta(v,w)) \cup A_i(w)\right) & \text{if } v \notin V_\alpha \end{cases}^{*}$$

*: Simplified version of the attractor definition presented in the report

VPG algorithms - Recursive algorithm

For every vertex v, find a set of configurations C such that player α can force the play from v to U when the VPG is played for configuration $c \in C$. Example, $\mathfrak{C} = \{c_1, c_2\}$:



$A : V \to 2^{\mathfrak{C}}$

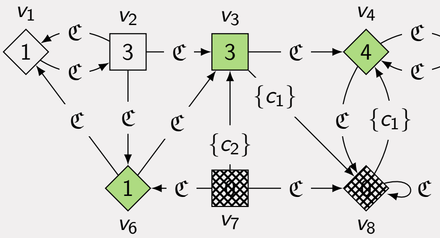
	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A_0				$\mathfrak{C}$				
A_1			$\{c_2\}$	$\mathfrak{C}$	$\mathfrak{C}$			$\{c_1\}$
A_2			$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	$\{c_2\}$		$\{c_1\}$
A_3			$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$		$\{c_1\}$

$$A_{i+1}(v) = A_i(v) \cup \begin{cases} \bigcup_{(v,w)\in E} \left(\theta(v,w) \cap A_i(w)\right) & \text{if } v \in V_\alpha \\ \bigcap_{(v,w)\in E} \left((\mathfrak{C} \backslash \theta(v,w)) \cup A_i(w)\right) & \text{if } v \notin V_\alpha \end{cases} {}^{*}$$

*: Simplified version of the attractor definition presented in the report

VPG algorithms - Recursive algorithm

For every vertex v, find a set of configurations C such that player α can force the play from v to U when the VPG is played for configuration $c \in C$. Example, $\mathfrak{C} = \{c_1, c_2\}$:



$A : V \to 2^{\mathfrak{C}}$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
A_0				$\mathfrak{C}$				
A_1			$\{c_2\}$	$\mathfrak{C}$	$\mathfrak{C}$			$\{c_1\}$
A_2			$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	$\{c_2\}$		$\{c_1\}$
A_3			$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$		$\{c_1\}$
A_4			$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	$\mathfrak{C}$	$\{c_1\}$	$\{c_1\}$

$$A_{i+1}(v) = A_i(v) \cup \begin{cases} \bigcup_{(v,w)\in E} (\theta(v,w) \cap A_i(w)) & \text{if } v \in V_\alpha \\ \bigcap_{(v,w)\in E} ((\mathfrak{C} \backslash \theta(v,w)) \cup A_i(w)) & \text{if } v \notin V_\alpha \end{cases}^{*}$$

*: Simplified version of the attractor definition presented in the report

VPG algorithms - Recursive algorithm

- Solve VPG by solving every parity game independently using the original recursive algorithm
- Solve VPG collectively using the modified attractor calculation

VPG algorithms - Recursive algorithm

- Solve VPG by solving every parity game independently using the original recursive algorithm
- Solve VPG collectively using the modified attractor calculation
- Modified attractor calculation relies on set operations betweens sets of configurations
- Can we perform these operations efficiently?

VPG algorithms - Recursive algorithm - Symbolically representing sets

- Represent sets simply as a collection of all its elements (*explicit*)
- Alternatively, represent sets as boolean formulas (*symbolic*)

VPG algorithms - Recursive algorithm - Symbolically representing sets

- Represent sets simply as a collection of all its elements (*explicit*)
- Alternatively, represent sets as boolean formulas (*symbolic*)
- Example: $S = \{s_0, \ldots, s_7\}$, $T = \{s_2, s_4, s_6, s_7\}$
- Boolean variables: x_2, x_1, x_0

VPG algorithms - Recursive algorithm - Symbolically representing sets

- Represent sets simply as a collection of all its elements (*explicit*)
- Alternatively, represent sets as boolean formulas (*symbolic*)
- Example: $S = \{s_0, \dots, s_7\}$, $T = \{s_2, s_4, s_6, s_7\}$
- Boolean variables: x_2, x_1, x_0

Boolean formula:

$$F(x_2, x_1, x_0) = (\neg x_2 \wedge x_1 \wedge \neg x_0) \vee (x_2 \wedge (x_1 \vee \neg x_0))$$

$\mathbf{x_2x_1x_0}$	$\mathbf{F(x_2, x_1, x_0)}$
000	0
001	0
010	1
011	0
100	1
101	0
110	1
111	1

VPG algorithms - Recursive algorithm - Symbolically representing sets

- Represent sets simply as a collection of all its elements (*explicit*)
- Alternatively, represent sets as boolean formulas (*symbolic*)
- Example: $S = \{s_0, \ldots, s_7\}$, $T = \{s_2, s_4, s_6, s_7\}$
- Boolean variables: x_2, x_1, x_0
- Boolean formulas can be very small
- *e.g.* $T = \{s_0, \ldots, s_3\}$ can be expressed as $\neg x_2$
- *e.g.* S can be expressed as **true**

Boolean formula:

$$F(x_2, x_1, x_0) = (\neg x_2 \wedge x_1 \wedge \neg x_0) \vee (x_2 \wedge (x_1 \vee \neg x_0))$$

$\mathbf{x_2 x_1 x_0}$	$\mathbf{F(x_2, x_1, x_0)}$
000	0
001	0
010	1
011	0
100	1
101	0
110	1
111	1

VPG algorithms - Recursive algorithm - Symbolically representing sets

- Represent sets simply as a collection of all its elements (*explicit*)
- Alternatively, represent sets as boolean formulas (*symbolic*)
- Example: $S = \{s_0, \ldots, s_7\}$, $T = \{s_2, s_4, s_6, s_7\}$
- Boolean variables: x_2, x_1, x_0
- Boolean formulas can be very small
- *e.g.* $T = \{s_0, \ldots, s_3\}$ can be expressed as $\neg x_2$
- *e.g.* S can be expressed as **true**
- Boolean operators $\vee, \wedge$ coincide with set operators $\cup, \cap$

Boolean formula:

$$F(x_2, x_1, x_0) = (\neg x_2 \wedge x_1 \wedge \neg x_0) \vee (x_2 \wedge (x_1 \vee \neg x_0))$$

$\mathbf{x_2 x_1 x_0}$	$\mathbf{F(x_2, x_1, x_0)}$
000	0
001	0
010	1
011	0
100	1
101	0
110	1
111	1

VPG algorithms - Recursive algorithm - Symbolically representing sets

- Boolean formulas can be expressed as BDDs
- Simple formulas $\implies$ small BDDs $\implies$ quick set operators

VPG algorithms - Recursive algorithm - Symbolically representing sets

- Boolean formulas can be expressed as BDDs
- Simple formulas $\implies$ small BDDs $\implies$ quick set operators
- FTSs use features
- FTSs use boolean formulas to enable/disable parts of the system

VPG algorithms - Recursive algorithm - Symbolically representing sets

- Boolean formulas can be expressed as BDDs
- Simple formulas $\implies$ small BDDs $\implies$ quick set operators
- FTSs use features
- FTSs use boolean formulas to enable/disable parts of the system
- VPGs are constructed such that every edge either:
  - admits all configurations, or
  - is guarded by a set that coincides with a formula from the FTS

VPG algorithms - Recursive algorithm - Time complexities

n: # vertices, e: # edges, d: # distinct priorities, c # configurations

Set operations	
Explicit sets	
Symbolic sets	

Algorithms	
Original recursive algorithm	
Independent approach	
Collective algorithm (explicit)	
Collective algorithm (symbolic)	

VPG algorithms - Recursive algorithm - Time complexities

n: # vertices, e: # edges, d: # distinct priorities, c # configurations

Set operations	
Explicit sets	$O(c)$
Symbolic sets	$O(c^2)$

Algorithms	
Original recursive algorithm	
Independent approach	
Collective algorithm (explicit)	
Collective algorithm (symbolic)	

VPG algorithms - Recursive algorithm - Time complexities

n: # vertices, e: # edges, d: # distinct priorities, c # configurations

Set operations	
Explicit sets	$O(c)$
Symbolic sets	$O(c^2)$

Algorithms	
Original recursive algorithm	$O(e * n^d)$
Independent approach	$O(c * e * n^d)$
Collective algorithm (explicit)	
Collective algorithm (symbolic)	

VPG algorithms - Recursive algorithm - Time complexities

n: # vertices, e: # edges, d: # distinct priorities, c # configurations

Set operations	
Explicit sets	$O(c)$
Symbolic sets	$O(c^2)$

Algorithms	
Original recursive algorithm	$O(e * n^d)$
Independent approach	$O(c * e * n^d)$
Collective algorithm (explicit)	$O(n * c^2 * e * n^d)$
Collective algorithm (symbolic)	

VPG algorithms - Recursive algorithm - Time complexities

n: # vertices, e: # edges, d: # distinct priorities, c # configurations

Set operations	
Explicit sets	$O(c)$
Symbolic sets	$O(c^2)$

Algorithms	
Original recursive algorithm	$O(e * n^d)$
Independent approach	$O(c * e * n^d)$
Collective algorithm (explicit)	$O(n * c^2 * e * n^d)$
Collective algorithm (symbolic)	$O(n * c^3 * e * n^d)$

Experimental results - SPL games

Minepump SPL

- Keep a mine shaft free from water
- 10 features that change the sensor/actor setup

Experimental results - SPL games

Minepump SPL

- Keep a mine shaft free from water
- 10 features that change the sensor/actor setup
- 128 valid feature assignments
- 600 states and 1400 transitions
- 9 requirements
- 9 VPGs ranging from 3000 to 9200 vertices and 2 to 4 distinct priorities

Experimental results - SPL games

Minepump SPL

- Keep a mine shaft free from water
- 10 features that change the sensor/actor setup
- 128 valid feature assignments
- 600 states and 1400 transitions
- 9 requirements
- 9 VPGs ranging from 3000 to 9200 vertices and 2 to 4 distinct priorities

Elevator SPL

- Elevator travelling between five floor
- 5 features, including overload detection and parking

Experimental results - SPL games

Minepump SPL

- Keep a mine shaft free from water
- 10 features that change the sensor/actor setup
- 128 valid feature assignments
- 600 states and 1400 transitions
- 9 requirements
- 9 VPGs ranging from 3000 to 9200 vertices and 2 to 4 distinct priorities

Elevator SPL

- Elevator travelling between five floor
- 5 features, including overload detection and parking
- 64 valid feature assignments
- 34k states and 200k
- 7 requirements
- 7 VPGs ranging from 440k and 1.85m vertices and 2 to 3 distinct priorities

Experimental results - SPL games

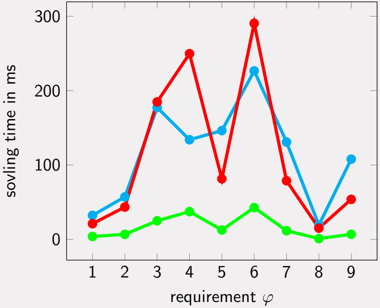


Figure: Running times, in ms, on the minepump games.

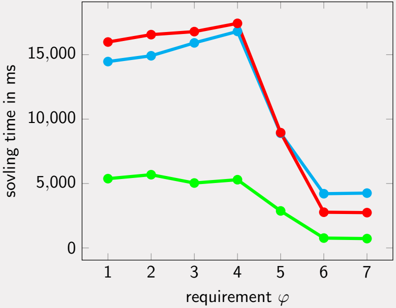


Figure: Running times, in ms, on the elevator games.

Independent recursive algorithm
Collective recursive algorithm with a symbolic representation of configurations
Collective recursive algorithm with an explicit representation of configurations

Discussion

Collective recursive algorithm:

- Symbolic variant increases performance 3-18 times (SPL games)
- For certain random games the symbolic performance drops rapidly, explicit performance is steady

Discussion

Collective recursive algorithm:

- Symbolic variant increases performance 3-18 times (SPL games)
- For certain random games the symbolic performance drops rapidly, explicit performance is steady

Incremental pre-solve algorithm:

- Same time complexity as independent approach ($O(c * e * n^d)$)
- Slightly increases performance of SPL games and random games
- Not consistent or very significant

Locally solving (V)PGs

- Parity games: Terminate when special vertex v_0 is solved
- VPGs: Terminate when special vertex v_0 is solved for all configurations

Locally solving (V)PGs

- Parity games: Terminate when special vertex v_0 is solved
- VPGs: Terminate when special vertex v_0 is solved for all configurations
- Introduced local variants of existing algorithms and of the novel VPG algorithms

Locally solving (V)PGs

- Parity games: Terminate when special vertex v_0 is solved
- VPGs: Terminate when special vertex v_0 is solved for all configurations
- Introduced local variants of existing algorithms and of the novel VPG algorithms
- Relative performance: How much quicker are the collective algorithms?
- Is the relative local performance greater than the relative global performance?

Locally solving (V)PGs

- Parity games: Terminate when special vertex v_0 is solved
- VPGs: Terminate when special vertex v_0 is solved for all configurations
- Introduced local variants of existing algorithms and of the novel VPG algorithms
- Relative performance: How much quicker are the collective algorithms?
- Is the relative local performance greater than the relative global performance?

Results:

- Recursive algorithms: no significant increase in relative performance
- Incremental pre-solve algorithm: increases relative performance for random games

Conclusions

- VPGs can be used to verify SPLs
- Collective approaches outperform independent approaches
- Locally solving VPGs can increase performance (more so than locally solving parity games does)

Extra - Incremental pre-solve

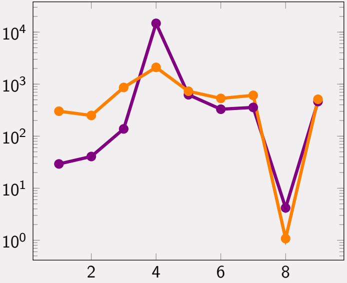


Figure: Running times, in ms, on the minepump games.

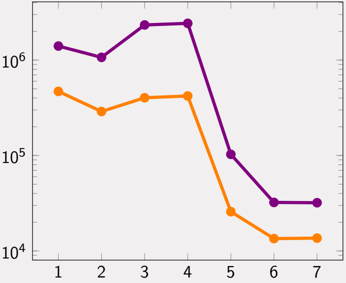


Figure: Running times, in ms, on the elevator games.