The simplex algorithm and the Hirsch conjecture: **Lecture 3**

Thomas Dueholm Hansen

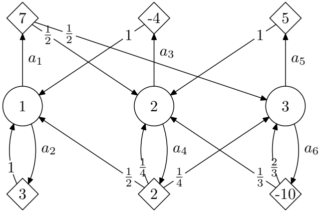
MADALGO & CTIC Summer School

August 11, 2011

Overview

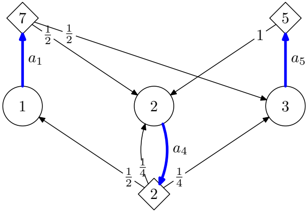
- **Lecture 1:**
  - Introduction to linear programming and the simplex algorithm.
  - Pivoting rules.
  - The RandomFacet pivoting rule.
- **Lecture 2:**
  - The Hirsch conjecture.
  - Introduction to Markov decision processes (MDPs).
  - Upper bound for the LargestCoefficient pivoting rule for MDPs.
- **Lecture 3:**
  - Lower bounds for pivoting rules utilizing MDPs. Example: Bland's rule.
  - Lower bound for the RandomEdge pivoting rule.
  - Abstractions and related problems.

Discounted Markov decision processes



$$J = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \\ 0 & 1 & 0 \\ 0 & \frac{1}{3} & \frac{2}{3} \end{bmatrix} \qquad c = \begin{bmatrix} 7 \\ 3 \\ -4 \\ 2 \\ 5 \\ -10 \end{bmatrix}$$

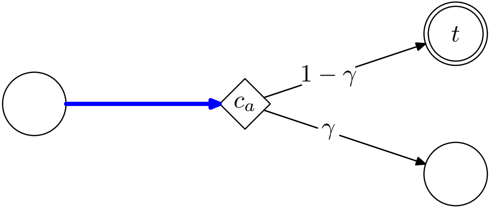
- A discounted MDP with n states and a total of m actions can be represented by:
    - A discount factor $\gamma < 1$.
    - A zero-one matrix $J \in \{0,1\}^{m \times n}$, with $J_{a,i} = 1$ iff $a \in A_i$.
    - A stochastic matrix $P \in \mathbb{R}^{m \times n}$.
    - A reward vector $c \in \mathbb{R}^m$.



$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad P_\pi = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \\ 0 & 1 & 0 \end{bmatrix} \qquad c_\pi = \begin{bmatrix} 7 \\ 2 \\ 5 \end{bmatrix}$$

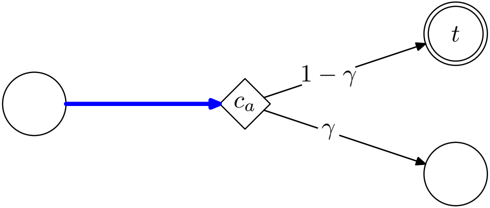
- A discounted MDP with n states and a total of m actions can be represented by:
  - A discount factor $\gamma < 1$.
  - A zero-one matrix $J \in \{0,1\}^{m \times n}$, with $J_{a,i} = 1$ iff $a \in A_i$.
  - A stochastic matrix $P \in \mathbb{R}^{m \times n}$.
  - A reward vector $c \in \mathbb{R}^m$.
- A policy π is a choice of an action from each state. π defines a Markov chain with rewards (P_π, c_π).

The stopping condition



- The discount factor $\gamma < 1$ was introduced because the expected total reward $\sum_{k=0}^{\infty} b^T P_\pi^k c_\pi$, where b is some initial distribution, may not converge.
- For every action a, $(1-\gamma)$ may be interpreted as the probability of moving to a **terminal state** t.

The stopping condition



- The discount factor $\gamma < 1$ was introduced because the expected total reward $\sum_{k=0}^{\infty} b^T P_\pi^k c_\pi$, where b is some initial distribution, may not converge.
- For every action a, $(1-\gamma)$ may be interpreted as the probability of moving to a **terminal state** t.
- To ensure convergence it is enough to satisfy the following condition:
  - **Stopping condition:** The terminal state is eventually reached with probability 1 from all states.

The stopping condition

- Let $P_\pi \in \mathbb{R}^{n \times n}$ be a matrix with non-negative entries such that each row sums to at most 1.
  - The difference between 1 and the sum of the a'th row is the probability of moving to the terminal state when using action a.
  - Note that γP, where P is an $n \times n$ stochastic matrix, is a special case.

The stopping condition

- Let $P_\pi \in \mathbb{R}^{n \times n}$ be a matrix with non-negative entries such that each row sums to at most 1.
  - The difference between 1 and the sum of the a'th row is the probability of moving to the terminal state when using action a.
  - Note that γP, where P is an $n \times n$ stochastic matrix, is a special case.
- If the stopping condition is satisfied, each row of P_π^n sums to less than 1, and $P_\pi^k \to 0$ for $k \to \infty$.
- It is again not difficult to show that:

$$(I - P_\pi)^{-1} = \sum_{k=0}^{\infty} P_\pi^k$$

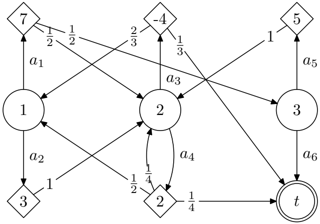
The stopping condition

- Let $P_\pi \in \mathbb{R}^{n \times n}$ be a matrix with non-negative entries such that each row sums to at most 1.
  - The difference between 1 and the sum of the a'th row is the probability of moving to the terminal state when using action a.
  - Note that γP, where P is an $n \times n$ stochastic matrix, is a special case.
- If the stopping condition is satisfied, each row of P_π^n sums to less than 1, and $P_\pi^k \to 0$ for $k \to \infty$.
- It is again not difficult to show that:

$$(I - P_\pi)^{-1} = \sum_{k=0}^{\infty} P_\pi^k$$

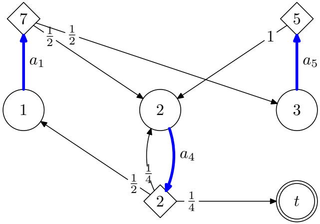
- Everything that was said in lecture 2 about discounted Markov chains with rewards remain true if γP is replaced by P, where P satisfies the stopping condition.

Markov decision processes



$$J = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 1 & 0 \\ \frac{2}{3} & 0 & 0 \\ \frac{1}{2} & \frac{1}{4} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad c = \begin{bmatrix} 7 \\ 3 \\ -4 \\ 2 \\ 5 \\ 0 \end{bmatrix}$$

- An MDP with n states and a total of m actions can be represented by:
  - A zero-one matrix $J \in \{0,1\}^{m \times n}$, with $J_{a,i} = 1$ iff $a \in A_i$.
  - A matrix $P \in \mathbb{R}^{m \times n}$ with non-negative entries and each row summing to at most 1.
  - A reward vector $c \in \mathbb{R}^m$.



$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad P_{\pi} = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{4} & 0 \\ 0 & 1 & 0 \end{bmatrix} \qquad c_{\pi} = \begin{bmatrix} 7 \\ 2 \\ 5 \end{bmatrix}$$

- An MDP with n states and a total of m actions can be represented by:
  - A zero-one matrix $J \in \{0,1\}^{m \times n}$, with $J_{a,i} = 1$ iff $a \in A_i$.
  - A matrix $P \in \mathbb{R}^{m \times n}$ with non-negative entries and each row summing to at most 1.
  - A reward vector $c \in \mathbb{R}^m$.
- An MDP satisfies the stopping condition if all policies π satisfy the stopping condition. For simplicity, we will generally assume that MDPs satisfy the stopping condition.

- Every policy π defines value and flux vectors:

$$v_\pi = (I - P_\pi)^{-1} c_\pi \qquad x_\pi^T = e^T (I - P_\pi)^{-1}$$

where $(I - P_\pi)^{-1} = \sum_{k=0}^{\infty} P_\pi^k$.

- The value of state i, $(v_\pi)_i$, is the expected total reward accumulated when starting there.

Markov decision processes

- Every policy π defines value and flux vectors:

$$v_\pi = (I - P_\pi)^{-1} c_\pi \qquad x_\pi^T = e^T (I - P_\pi)^{-1}$$

  where $(I - P_\pi)^{-1} = \sum_{k=0}^{\infty} P_\pi^k$.

- The value of state i, $(v_\pi)_i$, is the expected total reward accumulated when starting there.
- A policy π^* is **optimal** if it maximizes the values of all states: $v_{\pi^*} \geq v_\pi$ for all π.

Markov decision processes

- An optimal policy can be found by solving a linear program:

$$(P)\quad \begin{array}{lrcl} \max & c^T x & & \\ s.t. & (J-P)^T x & = & e \\ & x & \geq & 0 \end{array} \qquad (D)\quad \begin{array}{lrcl} \min & e^T y & & \\ s.t. & (J-P)y & \geq & c \end{array}$$

Markov decision processes

- An optimal policy can be found by solving a linear program:

$$(P)\quad \begin{array}{lrcl} \max & c^T x & & \\ s.t. & (J-P)^T x & = & e \\ & x & \geq & 0 \end{array} \qquad (D)\quad \begin{array}{lrcl} \min & e^T y & & \\ s.t. & (J-P) y & \geq & c \end{array}$$

- There is a one-to-one correspondence between policies and basic feasible solutions of the primal LP (P).

- An optimal policy can be found by solving a linear program:

$$(P)\quad \begin{array}{lrcl} \max & c^T x & & \\ s.t. & (J-P)^T x & = & e \\ & x & \geq & 0 \end{array} \qquad (D)\quad \begin{array}{lrcl} \min & e^T y & & \\ s.t. & (J-P) y & \geq & c \end{array}$$

- There is a one-to-one correspondence between policies and basic feasible solutions of the primal LP (P).
- The reduced cost vector, i.e. the coefficients of a tableau, $\bar{c}^{\pi} \in \mathbb{R}^m$ for a policy π is defined by:

$$\forall i \in S, \forall a \in A_i : \quad \bar{c}_a^{\pi} = (c_a + P_a v_{\pi}) - (v_{\pi})_i$$

- $\bar{c}_a^{\pi}$ is the improvement over the current value by using a for one step w.r.t. v_{π}.

- If $\bar{c}_a^\pi > 0$ we say that a is an **improving switch** w.r.t. π. I.e., $a \in A_i$ is an improving switch iff:

$$(v_\pi)_i < c_a + P_a v_\pi$$

Lemma (Howard (1960))

Let π' be obtained from π by jointly performing any non-empty set of improving switches. Then $v_{\pi'} \geq v_\pi$ and $v_{\pi'} \neq v_\pi$.

Lemma (Howard (1960))

A policy π is optimal iff there are no improving switches.

Function $\textsc{PolicyIteration}(\pi)$

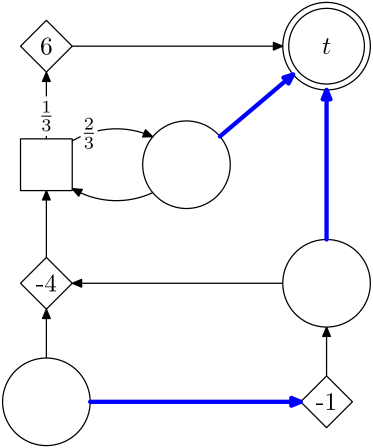
while $\exists$ *improving switch w.r.t.* π **do**
    Update π by performing improving switches

return π

- The simplex algorithm applied to the primal LP (P) is a special case of $\textsc{PolicyIteration}$.

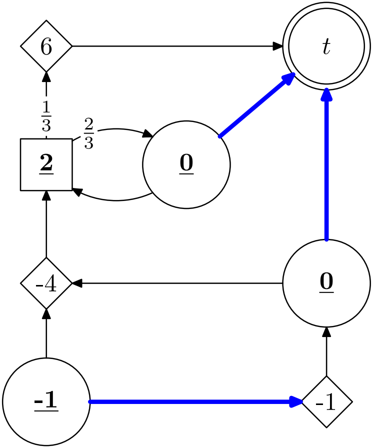
Example: A simple MDP

- Notation for graphical representation:
  - Circles are states.
  - Diamond-shaped vertices are rewards.
  - Squares are randomization vertices.
- A **policy** π is shown as bold blue arrows.



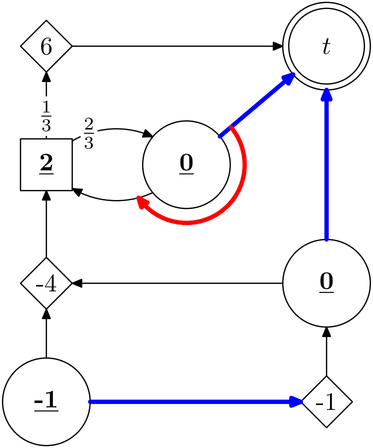
Example: A simple MDP

- Notation for graphical representation:
  - Circles are states.
  - Diamond-shaped vertices are rewards.
  - Squares are randomization vertices.
- A **policy** π is shown as bold blue arrows.
- States and randomization vertices are labelled by corresponding values of π.



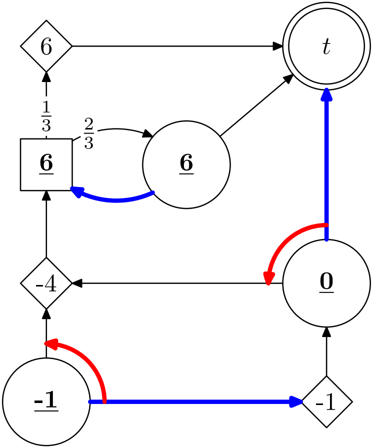
Example: A simple MDP

- Notation for graphical representation:
  - Circles are states.
  - Diamond-shaped vertices are rewards.
  - Squares are randomization vertices.
- A **policy** π is shown as bold blue arrows.
- States and randomization vertices are labelled by corresponding values of π.
- **Improving switches** are indicated by red arrows.



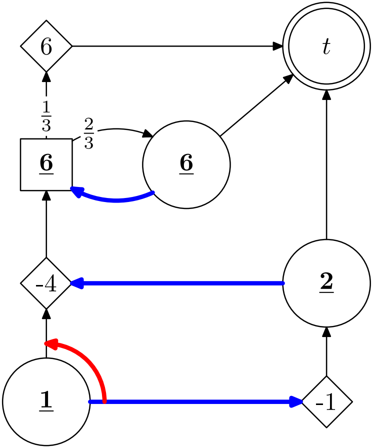
Example: A simple MDP

- Notation for graphical representation:
  - Circles are states.
  - Diamond-shaped vertices are rewards.
  - Squares are randomization vertices.
- A **policy** π is shown as bold blue arrows.
- States and randomization vertices are labelled by corresponding values of π.
- **Improving switches** are indicated by red arrows.



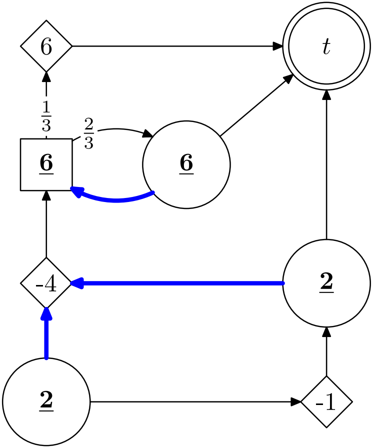
Example: A simple MDP

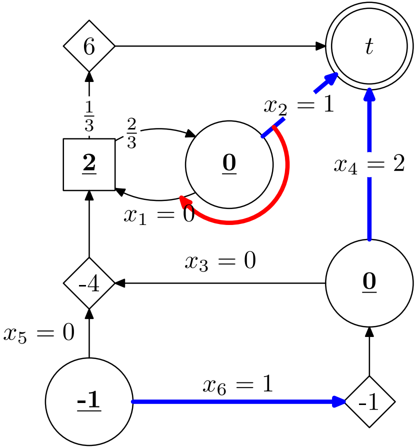
- Notation for graphical representation:
  - Circles are states.
  - Diamond-shaped vertices are rewards.
  - Squares are randomization vertices.
- A **policy** π is shown as bold blue arrows.
- States and randomization vertices are labelled by corresponding values of π.
- **Improving switches** are indicated by red arrows.



Example: A simple MDP

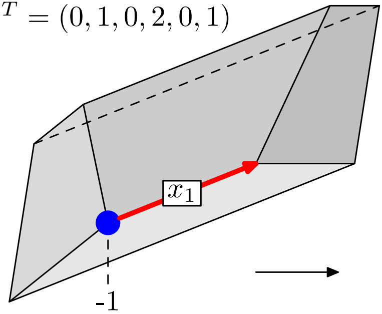
- Notation for graphical representation:
  - Circles are states.
  - Diamond-shaped vertices are rewards.
  - Squares are randomization vertices.
- A **policy** π is shown as bold blue arrows.
- States and randomization vertices are labelled by corresponding values of π.
- **Improving switches** are indicated by red arrows.



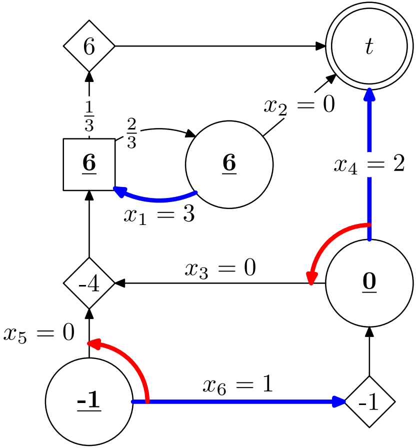


$$\begin{aligned} \max \quad & -1 + 2x_1 - 2x_3 - x_5 \\ \text{s.t.} \quad & x_2 = 1 - \tfrac{1}{3}x_1 + \tfrac{2}{3}x_3 + \tfrac{2}{3}x_5 \\ & x_4 = 2 - x_3 - x_5 \\ & x_6 = 1 - x_5 \\ & x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \end{aligned}$$

$$x^T = (0, 1, 0, 2, 0, 1)$$

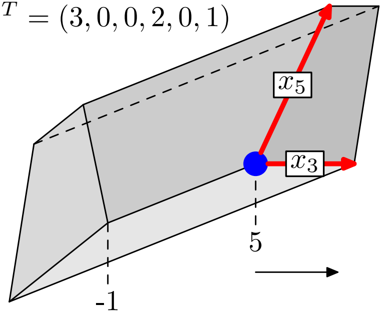


From MDP to LP

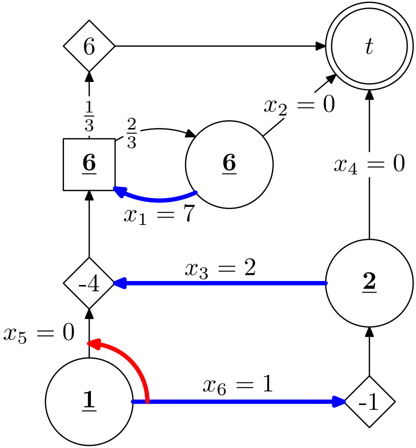


$$
\begin{aligned}
\max \quad & 5 - 6x_2 + 2x_3 + 3x_5 \\
\text{s.t.} \quad & x_1 = 3 - 3x_2 + 2x_3 + 2x_5 \\
& x_4 = 2 - x_3 - x_5 \\
& x_6 = 1 - x_5 \\
& x_1, x_2, x_3, x_4, x_5, x_6 \geq 0
\end{aligned}
$$

$x^T = (3, 0, 0, 2, 0, 1)$

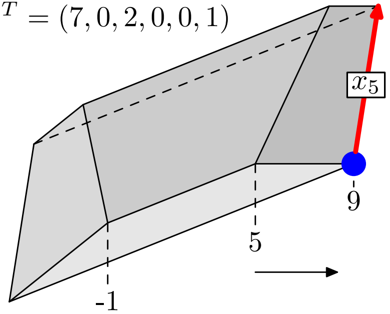


From MDP to LP

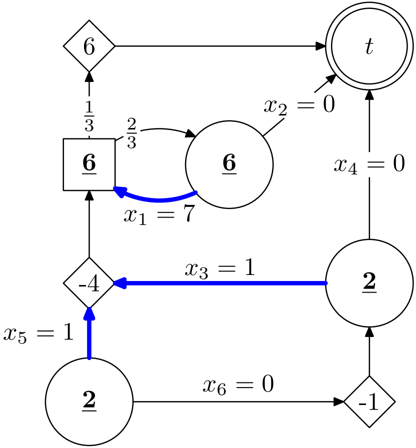


$$\begin{aligned} \max \quad & 9 - 6x_2 - 2x_4 + x_5 \\ \text{s.t.} \quad & x_1 = 7 - 3x_2 - 2x_4 \\ & x_3 = 2 - x_4 - x_5 \\ & x_6 = 1 - x_5 \\ & x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \end{aligned}$$

$$x^T = (7, 0, 2, 0, 0, 1)$$

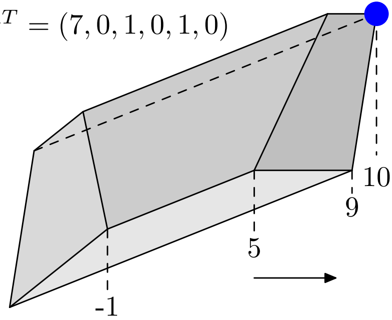


From MDP to LP



$$\begin{array}{llcl} \max & 10 - 6x_2 - 2x_4 - x_6 & & \\ \text{s.t.} & x_1 & = & 7 - 3x_2 - 2x_4 \\ & x_3 & = & 1 - x_4 + x_6 \\ & x_5 & = & 1 - x_6 \\ & x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 & & \end{array}$$

$x^T = (7,0,1,0,1,0)$



Lower bounds for pivoting rules utilizing MDPs

- To prove lower bounds for a pivoting rule for the simplex algorithm, we can prove lower bounds for the corresponding PolicyIteration algorithm for MDPs:
  - There is a one-to-one correspondence between policies and basic feasible solutions of the primal LP (P) for MDPs.
  - The simplex algorithm for the primal LP (P) is the special case of PolicyIteration, where only single improving switches are performed.

Lower bounds for pivoting rules utilizing MDPs

- To prove lower bounds for a pivoting rule for the simplex algorithm, we can prove lower bounds for the corresponding PolicyIteration algorithm for MDPs:
  - There is a one-to-one correspondence between policies and basic feasible solutions of the primal LP (P) for MDPs.
  - The simplex algorithm for the primal LP (P) is the special case of PolicyIteration, where only single improving switches are performed.

- We next construct an exponential lower bound for Bland's rule as a warmup before sketching the $2^{\Omega(n^{1/4})}$ lower bound for RandomEdge by Friedmann, Hansen and Zwick (2011).

Bland's rule

- Bland's rule, Bland (1977)
  - Always pick the available variable with the smallest index, both for entering and leaving the basis.

Bland's rule

- Bland's rule, Bland (1977)
  - Always pick the available variable with the smallest index, both for entering and leaving the basis.
- Bland's rule for MDPs
  - Perform the improving switch a with the smallest index.
  - There is always only one action that can be exchanged with a, namely the current action that originates from the same state as a.

- BLAND'S RULE, Bland (1977)
  - Always pick the available variable with the smallest index, both for entering and leaving the basis.
- BLAND'S RULE for MDPs
  - Perform the improving switch a with the smallest index.
  - There is always only one action that can be exchanged with a, namely the current action that originates from the same state as a.

- When constructing a lower bound, we may pick a worst-case ordering of the indices.

Lower bound construction

- We define a family of lower bound MDPs G_n such that BLAND'S RULE, and later RANDOMEDGE, simulates an n-bit binary counter.

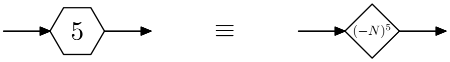
Lower bound construction

- We define a family of lower bound MDPs G_n such that BLAND'S RULE, and later RANDOMEDGE, simulates an n-bit binary counter.
- We make use of exponentially growing rewards (and penalties): To get a higher reward the MDP is willing to sacrifice everything that has been built up so far.

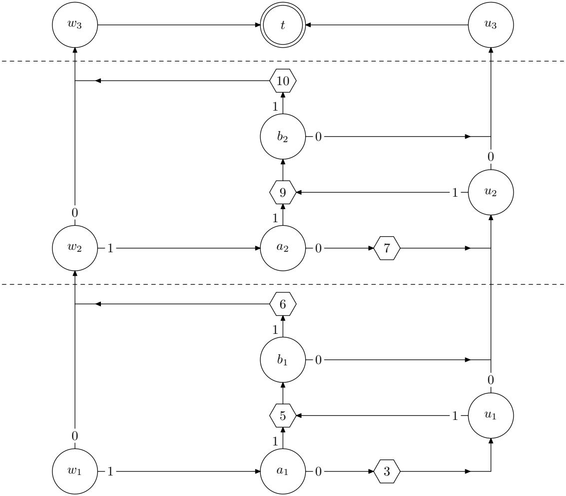
Lower bound construction

- We define a family of lower bound MDPs G_n such that BLAND'S RULE, and later RANDOMEDGE, simulates an n-bit binary counter.
- We make use of exponentially growing rewards (and penalties): To get a higher reward the MDP is willing to sacrifice everything that has been built up so far.
- Notation: Integer priority p corresponds to reward $(-N)^p$, where $N = 3n + 1$.

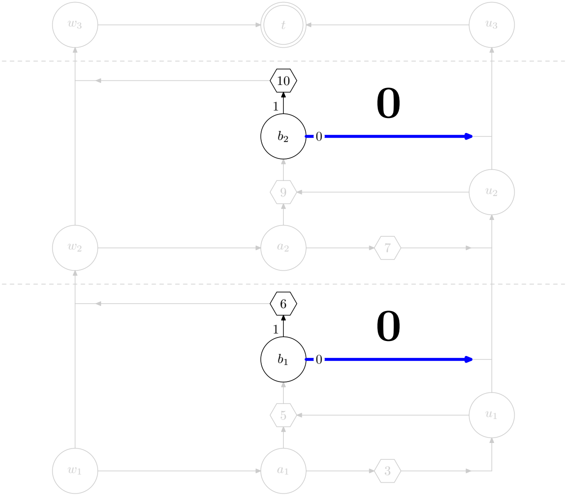
$$\dots < 5 < 3 < 1 < 2 < 4 < 6 < \dots$$



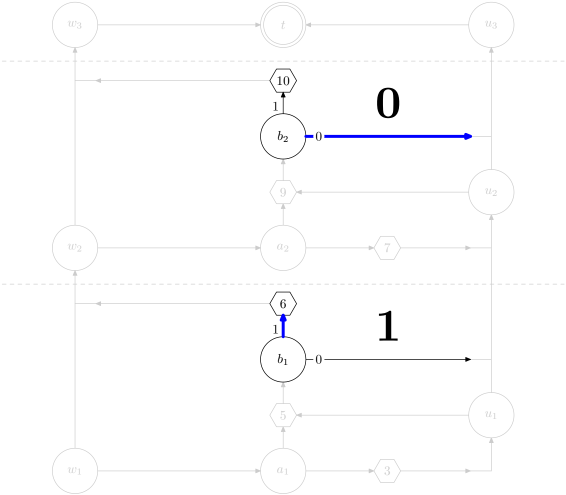
Lower bound for Bland's rule



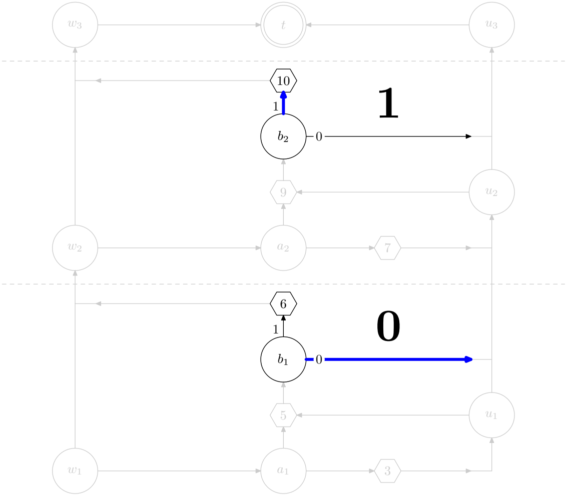
Lower bound for BLAND'S RULE



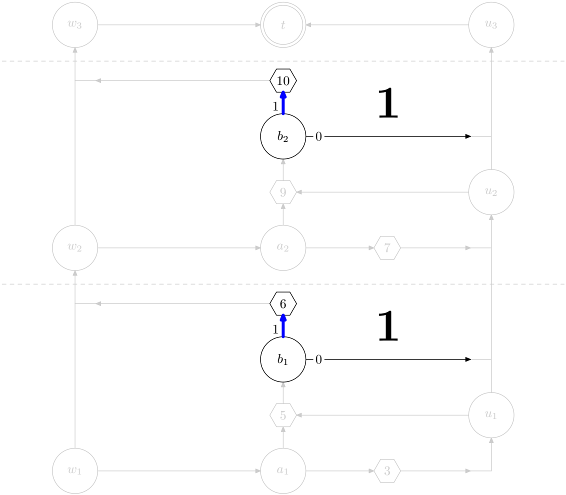
Lower bound for BLAND'S RULE

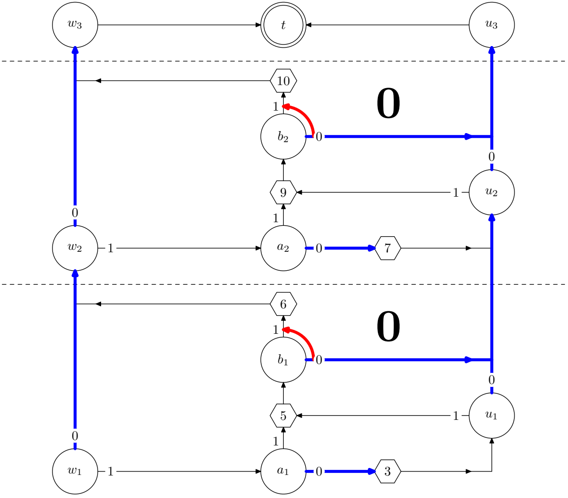


Lower bound for BLAND'S RULE



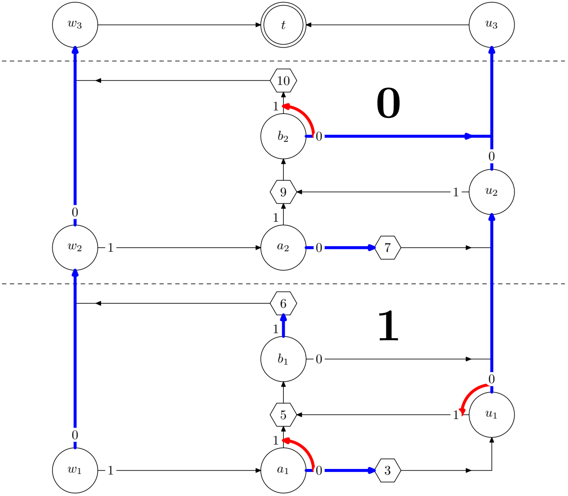
Lower bound for BLAND'S RULE





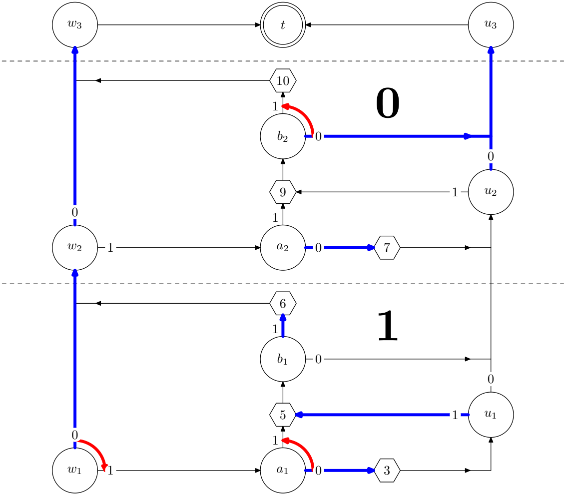
Order: $u_1^1, u_1^2, u_2^1, u_2^2, w_1^1, w_1^2, w_2^1, w_2^2, b_1^0, b_2^0, a_1^0, a_2^0, \underline{a_1^1, a_2^1, b_1^1, b_2^1}$

Lower bound for Bland's rule



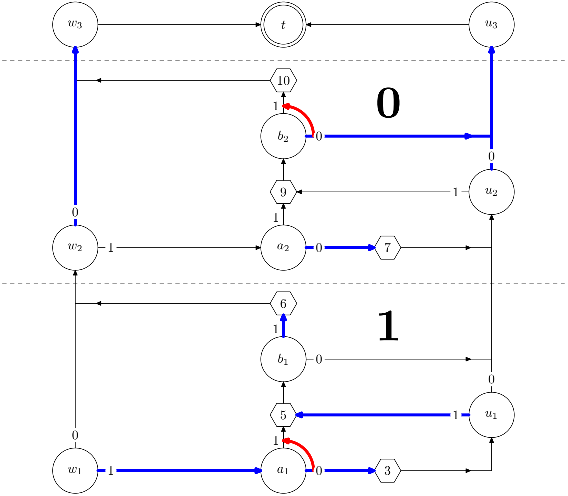
Order: $u_1^1, u_1^2, u_2^1, u_2^2, w_1^1, w_1^2, w_2^1, w_2^2, b_1^0, b_2^0, a_1^0, a_2^0, \underline{a_1^1, a_2^1, b_1^1, b_2^1}$

Lower bound for Bland's rule



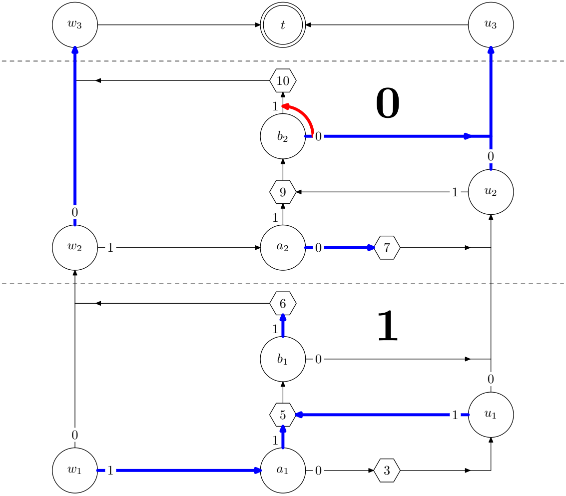
Order: $u_1^1, u_1^2, u_2^1, u_2^2, w_1^1, w_1^2, w_2^1, w_2^2, b_1^0, b_2^0, a_1^0, a_2^0, \underline{a_1^1, a_2^1, b_1^1, b_2^1}$

Lower bound for BLAND'S RULE



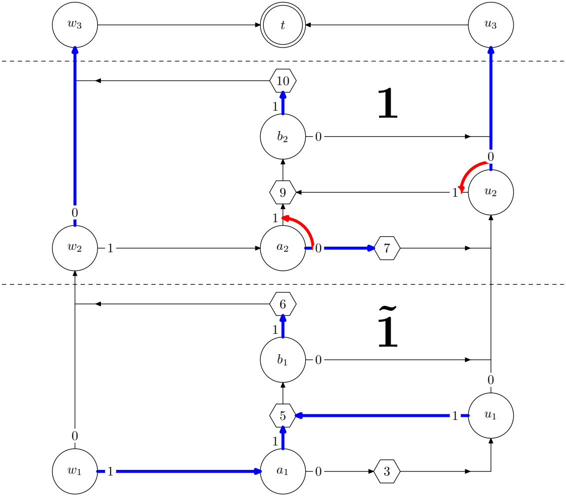
Order: $u_1^1, u_1^2, u_2^1, u_2^2, w_1^1, w_1^2, w_2^1, w_2^2, b_1^0, b_2^0, a_1^0, a_2^0, \underline{a_1^1, a_2^1, b_1^1, b_2^1}$

Lower bound for Bland's rule



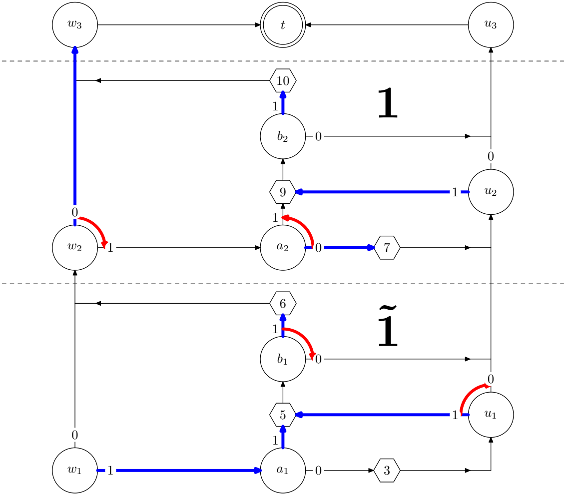
Order: $u_1^1, u_1^2, u_2^1, u_2^2, w_1^1, w_1^2, w_2^1, w_2^2, b_1^0, b_2^0, a_1^0, a_2^0, \underline{a_1^1, a_2^1, b_1^1, b_2^1}$

Lower bound for Bland's rule



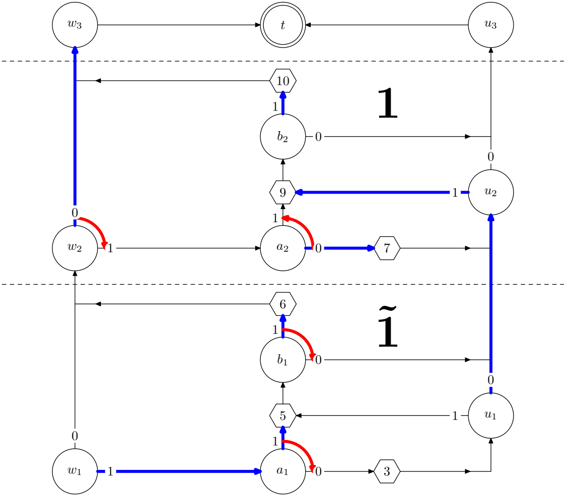
Order: $u_1^1, u_1^2, u_2^1, u_2^2, w_1^1, w_1^2, w_2^1, w_2^2, b_1^0, b_2^0, a_1^0, a_2^0, \underline{a_1^1, a_2^1, b_1^1, b_2^1}$

Lower bound for BLAND'S RULE



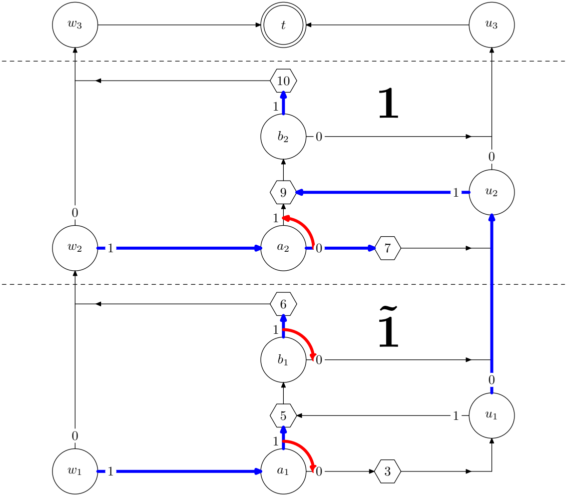
Order: $u_1^1, u_1^2, u_2^1, u_2^2, w_1^1, w_1^2, w_2^1, w_2^2, b_1^0, b_2^0, a_1^0, a_2^0, \underline{a_1^1, a_2^1, b_1^1, b_2^1}$

Lower bound for BLAND'S RULE



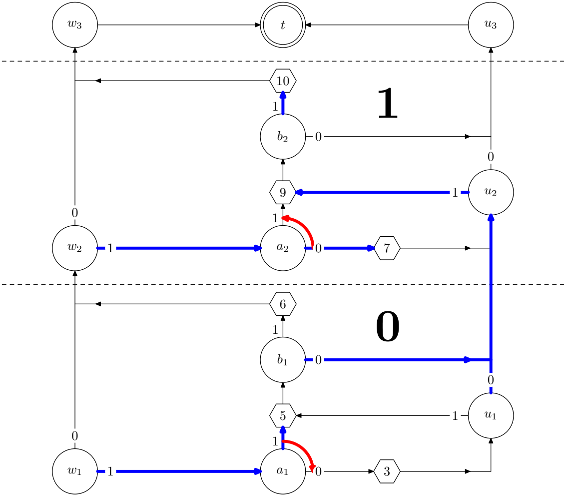
Order: $u_1^1, u_1^2, u_2^1, u_2^2, w_1^1, w_1^2, w_2^1, w_2^2, b_1^0, b_2^0, a_1^0, a_2^0, \underline{a_1^1, a_2^1, b_1^1, b_2^1}$

Lower bound for Bland's rule



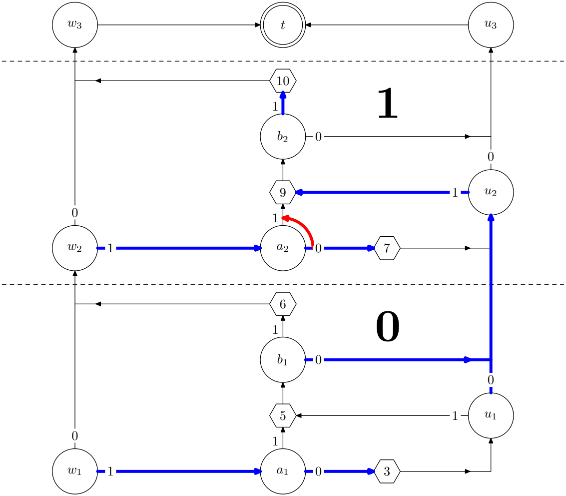
Order: $u_1^1, u_1^2, u_2^1, u_2^2, w_1^1, w_1^2, w_2^1, w_2^2, b_1^0, b_2^0, a_1^0, a_2^0, \underline{a_1^1, a_2^1, b_1^1, b_2^1}$

Lower bound for Bland's rule



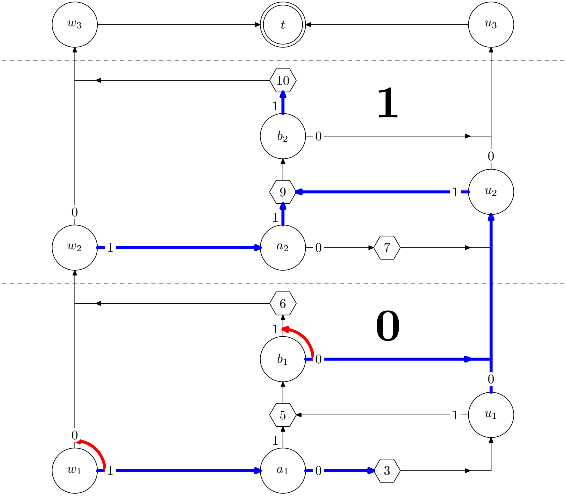
Order: $u_1^1, u_1^2, u_2^1, u_2^2, w_1^1, w_1^2, w_2^1, w_2^2, b_1^0, b_2^0, a_1^0, a_2^0, \underline{a_1^1, a_2^1, b_1^1, b_2^1}$

Lower bound for BLAND'S RULE



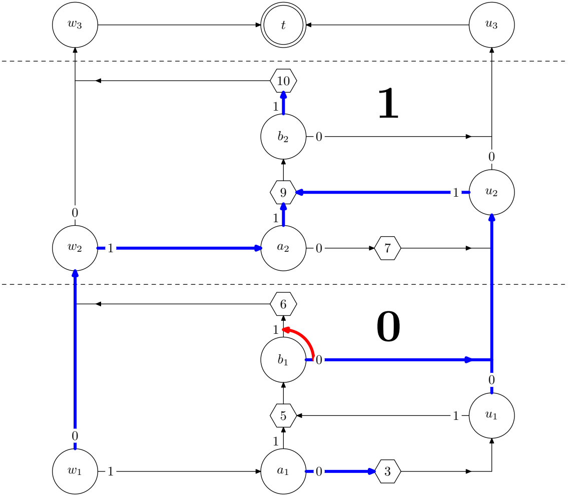
Order: $u_1^1, u_1^2, u_2^1, u_2^2, w_1^1, w_1^2, w_2^1, w_2^2, b_1^0, b_2^0, a_1^0, a_2^0, \underline{a_1^1, a_2^1, b_1^1, b_2^1}$

Lower bound for Bland's rule



Order: $u_1^1, u_1^2, u_2^1, u_2^2, w_1^1, w_1^2, w_2^1, w_2^2, b_1^0, b_2^0, a_1^0, a_2^0, \underline{a_1^1, a_2^1, b_1^1, b_2^1}$

Lower bound for BLAND'S RULE

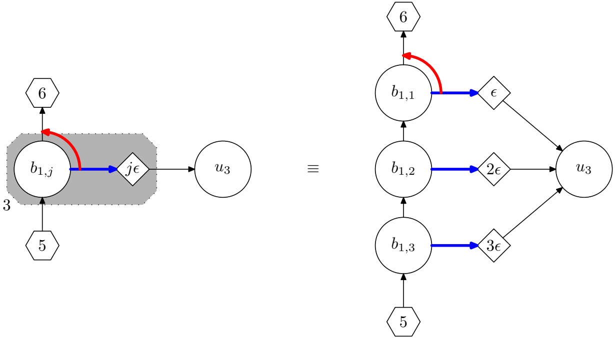


Order: $u_1^1, u_1^2, u_2^1, u_2^2, w_1^1, w_1^2, w_2^1, w_2^2, b_1^0, b_2^0, a_1^0, a_2^0, \underline{a_1^1, a_2^1, b_1^1, b_2^1}$

Lower bound for BLAND'S RULE

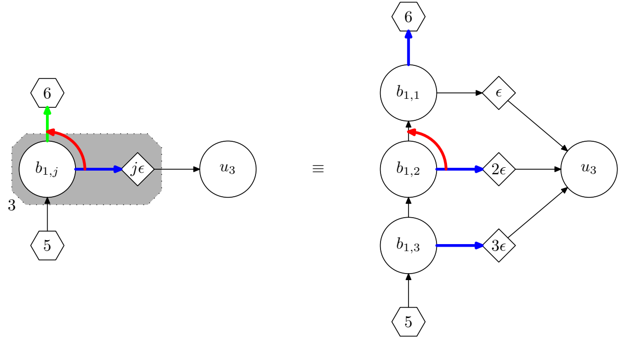
- Let k be the lowest unset bit. Incrementing the counter happened roughly through five phases:
  1. Make $b_k = 1$.
  2. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
  3. Make $b_i = 0$ and $a_i = 0$, for $i < k$.
  4. Make $a_k = 1$.
  5. Make $w_k = 1$ and $w_i = 0$, for $i < k$.
- Only the last part of the ordering, involving a_i^1 and b_i^1 edges, was important.
- To implement a lower bound for RANDOMEDGE we start out with the same construction. We need a gadget to delay improving switches like a_i^1 and b_i^1, however.

Delaying events



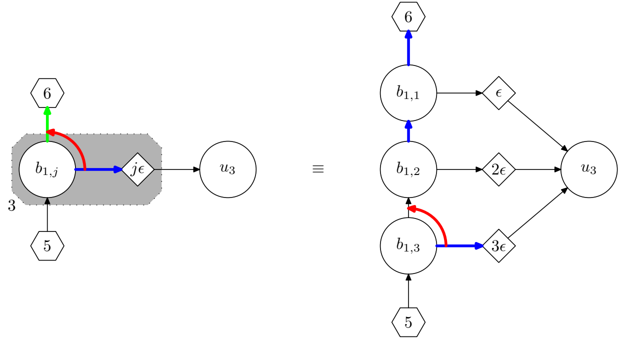
- By replacing a vertex by a chain of vertices, a specific sequence of improving switches has to be performed to get the same effect as performing one improving switch originally.
- RANDOMEDGE performs uniformly random improving switches, and a longer sequence therefore gives a longer delay.

Delaying events



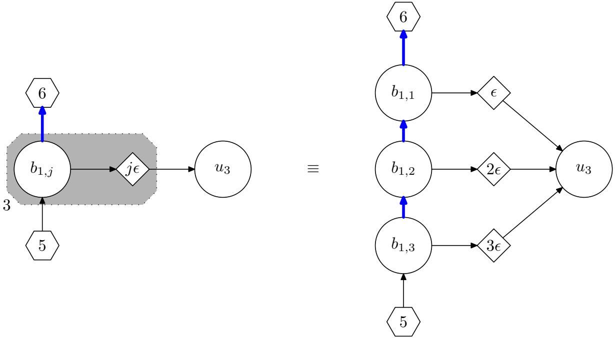
- By replacing a vertex by a chain of vertices, a specific sequence of improving switches has to be performed to get the same effect as performing one improving switch originally.
- RandomEdge performs uniformly random improving switches, and a longer sequence therefore gives a longer delay.

Delaying events



- By replacing a vertex by a chain of vertices, a specific sequence of improving switches has to be performed to get the same effect as performing one improving switch originally.
- RandomEdge performs uniformly random improving switches, and a longer sequence therefore gives a longer delay.

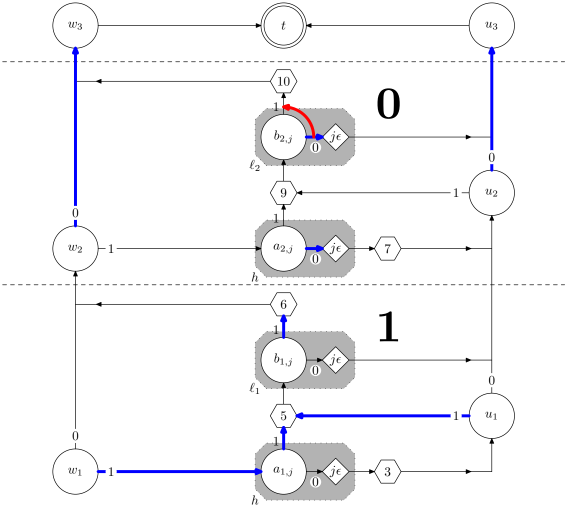
Delaying events



- By replacing a vertex by a chain of vertices, a specific sequence of improving switches has to be performed to get the same effect as performing one improving switch originally.
- RandomEdge performs uniformly random improving switches, and a longer sequence therefore gives a longer delay.

- Let k be the lowest unset bit. Incrementing the counter happens through five phases:

⇒ 1. Make $b_k = 1$.
2. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
3. Make $b_i = 0$ and $a_i = 0$, for $i < k$.
4. Make $a_k = 1$.
5. Make $w_k = 1$ and $w_i = 0$, for $i < k$.

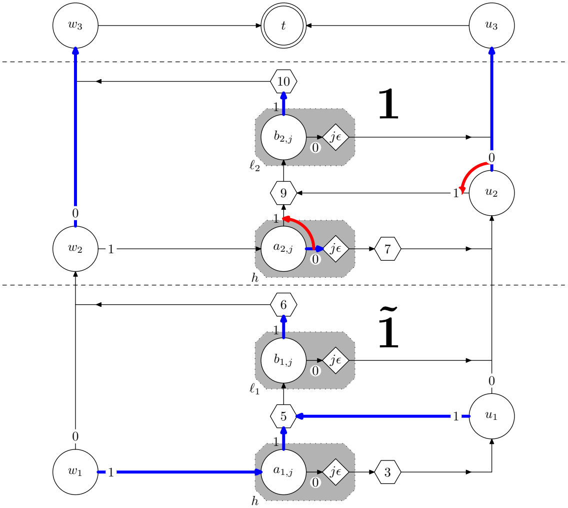


RandomEdge lower bound, first step

- Let k be the lowest unset bit. Incrementing the counter happens through five phases:

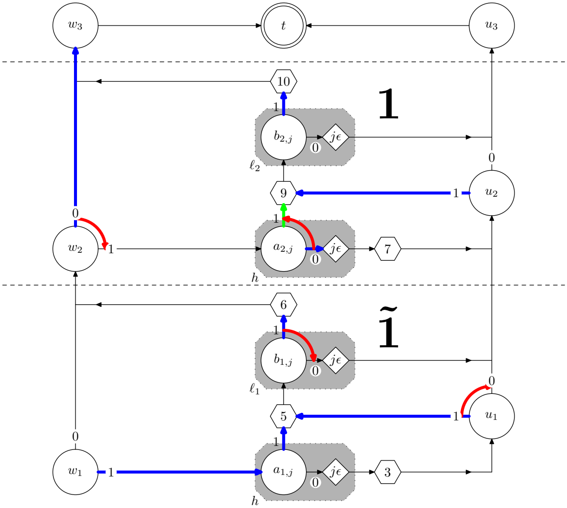
  1. Make $b_k = 1$.

  ⇒ 2. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
  3. Make $b_i = 0$ and $a_i = 0$, for $i < k$.
  4. Make $a_k = 1$.
  5. Make $w_k = 1$ and $w_i = 0$, for $i < k$.



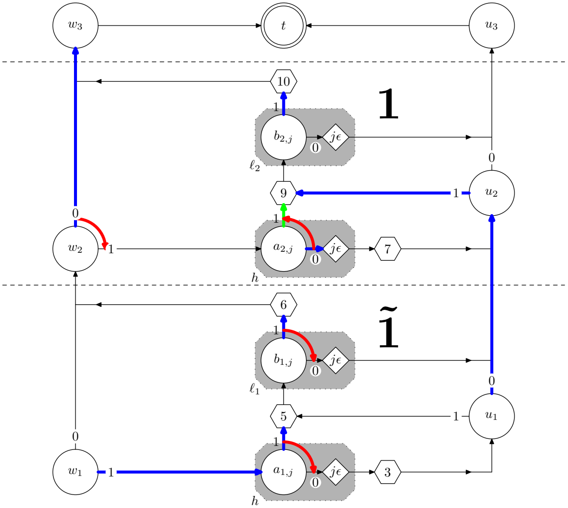
- Let k be the lowest unset bit. Incrementing the counter happens through five phases:

1. Make $b_k = 1$.
2. ⇒ Make $u_k = 1$ and $u_i = 0$, for $i < k$.
3. Make $b_i = 0$ and $a_i = 0$, for $i < k$.
4. Make $a_k = 1$.
5. Make $w_k = 1$ and $w_i = 0$, for $i < k$.

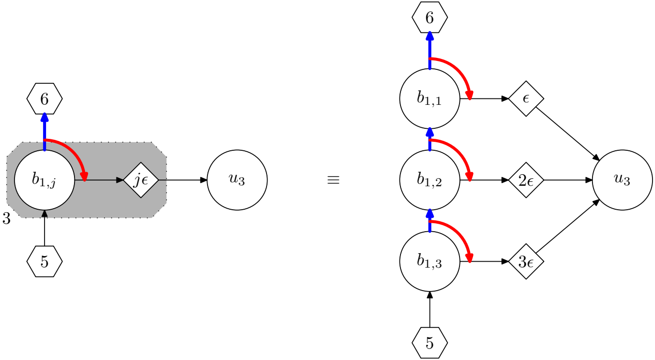


- Let k be the lowest unset bit. Incrementing the counter happens through five phases:

  1. Make $b_k = 1$.
  2. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
  3. ⇒ Make $b_i = 0$ and $a_i = 0$, for $i < k$.
  4. Make $a_k = 1$.
  5. Make $w_k = 1$ and $w_i = 0$, for $i < k$.



Fast resetting

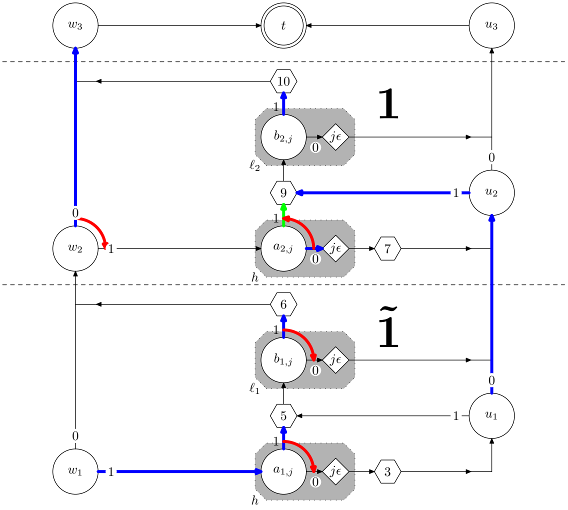


- Moving in the other directions happens much faster since all actions are improving switches simultaneously.

RandomEdge lower bound, first step

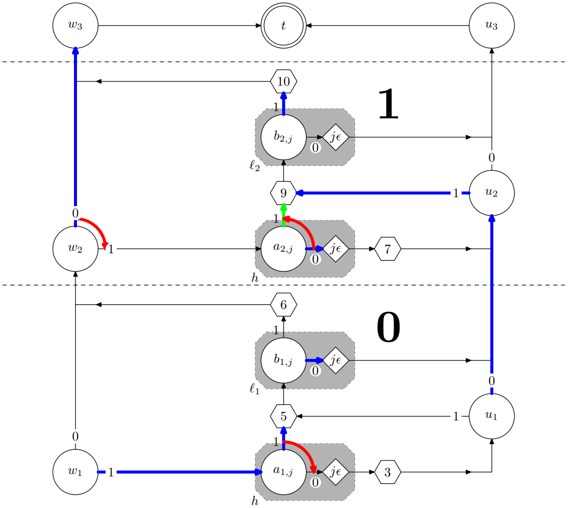
- Let k be the lowest unset bit. Incrementing the counter happens through five phases:

  1. Make $b_k = 1$.
  2. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
  3. ⇒ Make $b_i = 0$ and $a_i = 0$, for $i < k$.
  4. Make $a_k = 1$.
  5. Make $w_k = 1$ and $w_i = 0$, for $i < k$.



RandomEdge lower bound, first step

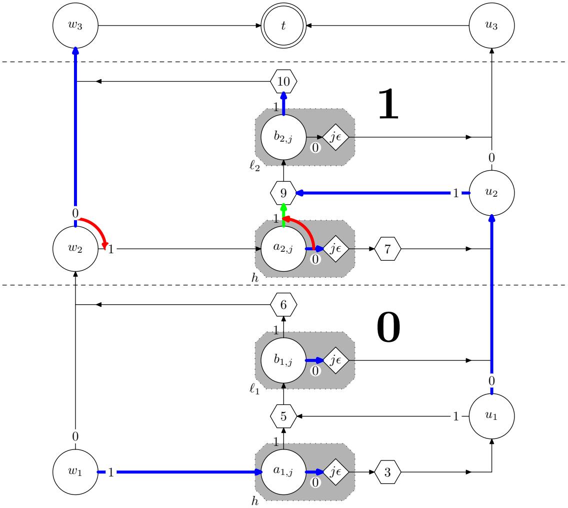
- Let k be the lowest unset bit. Incrementing the counter happens through five phases:
  1. Make $b_k = 1$.
  2. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
  3. ⇒ Make $b_i = 0$ and $a_i = 0$, for $i < k$.
  4. Make $a_k = 1$.
  5. Make $w_k = 1$ and $w_i = 0$, for $i < k$.



RandomEdge lower bound, first step

- Let k be the lowest unset bit. Incrementing the counter happens through five phases:

  1. Make $b_k = 1$.
  2. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
  3. Make $b_i = 0$ and $a_i = 0$, for $i < k$.
  4. ⇒ Make $a_k = 1$.
  5. Make $w_k = 1$ and $w_i = 0$, for $i < k$.

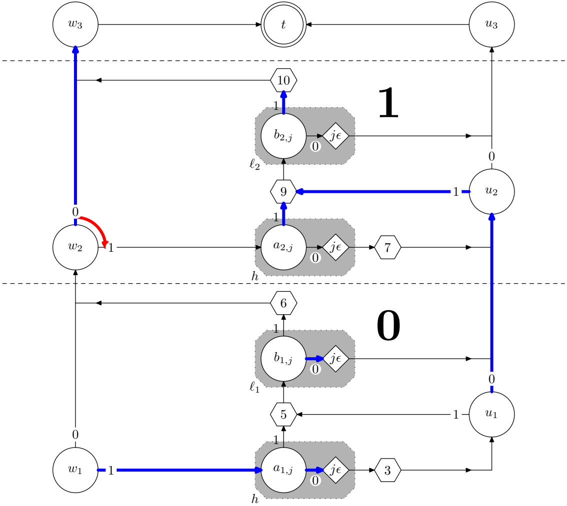


RandomEdge lower bound, first step

- Let k be the lowest unset bit. Incrementing the counter happens through five phases:

  1. Make $b_k = 1$.
  2. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
  3. Make $b_i = 0$ and $a_i = 0$, for $i < k$.
  4. Make $a_k = 1$.

  ⇒ 5. Make $w_k = 1$ and $w_i = 0$, for $i < k$.

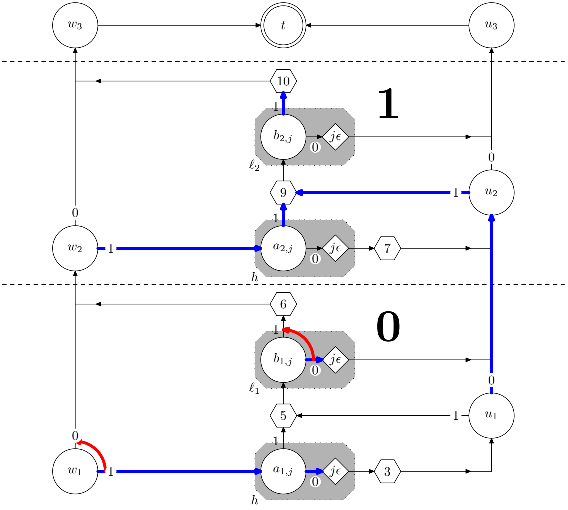


RandomEdge lower bound, first step

- Let k be the lowest unset bit. Incrementing the counter happens through five phases:

  1. Make $b_k = 1$.
  2. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
  3. Make $b_i = 0$ and $a_i = 0$, for $i < k$.
  4. Make $a_k = 1$.

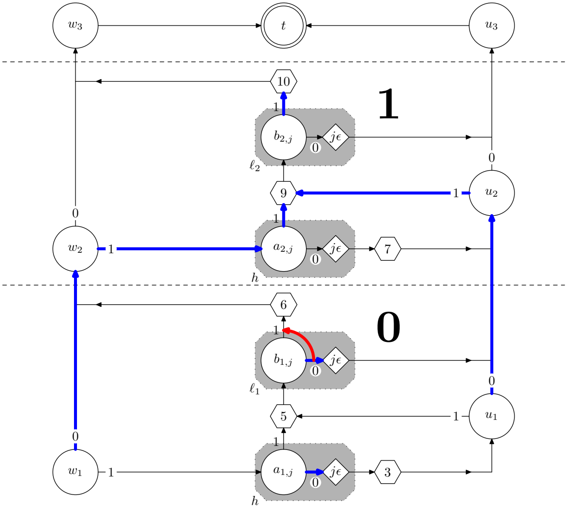
  ⇒ 5. Make $w_k = 1$ and $w_i = 0$, for $i < k$.



RandomEdge lower bound, first step

- Let k be the lowest unset bit. Incrementing the counter happens through five phases:

$\Rightarrow$ 1. Make $b_k = 1$.
2. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
3. Make $b_i = 0$ and $a_i = 0$, for $i < k$.
4. Make $a_k = 1$.
5. Make $w_k = 1$ and $w_i = 0$, for $i < k$.



Resetting partially set higher bits

- Problem: Higher bits get a head start in later counting steps. When resetting lower bits we must also reset higher bits that are partially set.

Resetting partially set higher bits

- Problem: Higher bits get a head start in later counting steps. When resetting lower bits we must also reset higher bits that are partially set.
- Higher bits must have access to lower bits: Introduce actions moving down to u_1.

Resetting partially set higher bits

- Problem: Higher bits get a head start in later counting steps. When resetting lower bits we must also reset higher bits that are partially set.
- Higher bits must have access to lower bits: Introduce actions moving down to u_1.
  - **Note:** The resulting MDP does not actually satisfy the stopping condition, but this just means the LP us unbounded towards $-\infty$. Alternatively, we can introduce randomization and always move up with an insignificant probability.

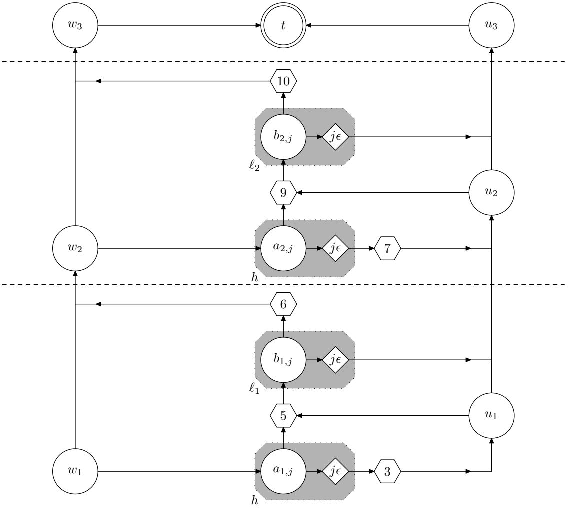
Resetting partially set higher bits

- Problem: Higher bits get a head start in later counting steps. When resetting lower bits we must also reset higher bits that are partially set.
- Higher bits must have access to lower bits: Introduce actions moving down to u_1.
  - **Note:** The resulting MDP does not actually satisfy the stopping condition, but this just means the LP us unbounded towards $-\infty$. Alternatively, we can introduce randomization and always move up with an insignificant probability.
- No state can have direct access to a large reward: Introduce a stochastic action such that this happens only with an insignificant probability $\epsilon = N^{-(4n+8)}$.

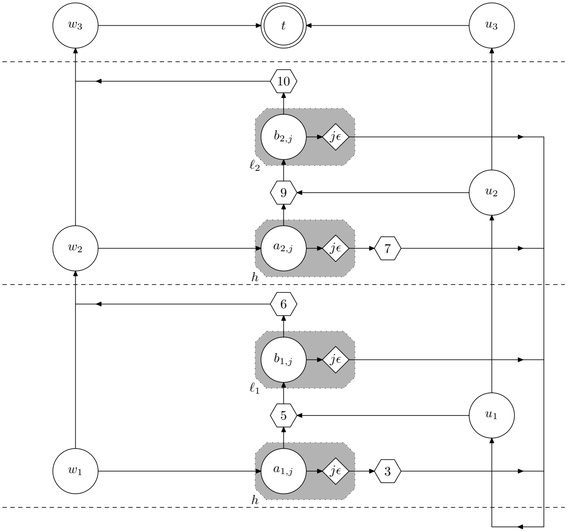
Resetting partially set higher bits

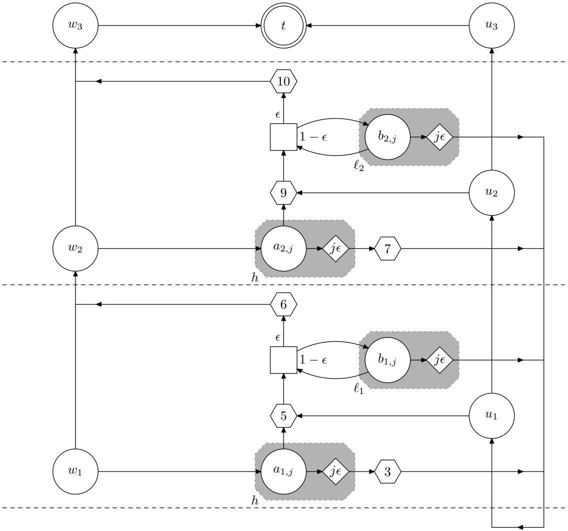
- Problem: Higher bits get a head start in later counting steps. When resetting lower bits we must also reset higher bits that are partially set.
- Higher bits must have access to lower bits: Introduce actions moving down to u_1.
  - **Note:** The resulting MDP does not actually satisfy the stopping condition, but this just means the LP us unbounded towards $-\infty$. Alternatively, we can introduce randomization and always move up with an insignificant probability.
- No state can have direct access to a large reward: Introduce a stochastic action such that this happens only with an insignificant probability $\epsilon = N^{-(4n+8)}$.
- Resetting higher bits requires alternating behaviour: Introduce an additional chain of c_i vertices.

RandomEdge lower bound, full construction



RandomEdge lower bound, full construction

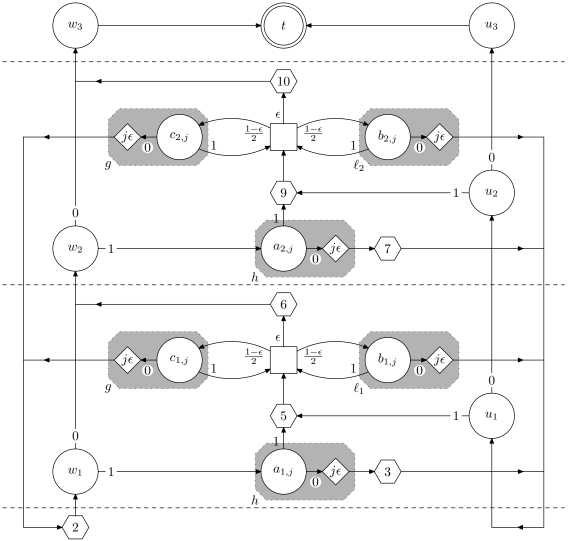



w_3
t
u_3
10
ϵ
$1-\epsilon$
$b_{2,j}$
$j\epsilon$
ℓ_2
9
u_2
w_2
$a_{2,j}$
$j\epsilon$
7
h
6
ϵ
$1-\epsilon$
$b_{1,j}$
$j\epsilon$
ℓ_1
5
u_1
w_1
$a_{1,j}$
$j\epsilon$
3
h

RandomEdge lower bound, full construction

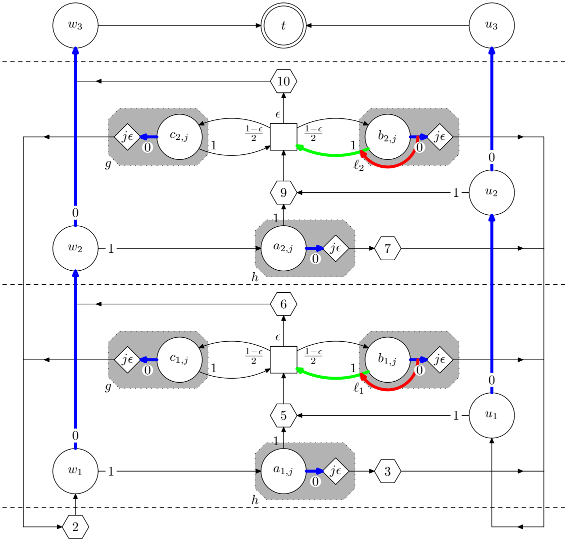
- Incrementing the counter happens through seven phases:

  1. Make $b_k = 1$.
  2. Make $c_k = 1$.
  3. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
  4. Make $b_i = 0$ and $a_i = 0$, for $i < k$. Reset b_i for unset bits $i > k$.
  5. Make $a_k = 1$.
  6. Make $w_k = 1$ and $w_i = 0$, for $i < k$.
  7. Reset c_i for all unset bits.



- Incrementing the counter happens through seven phases:

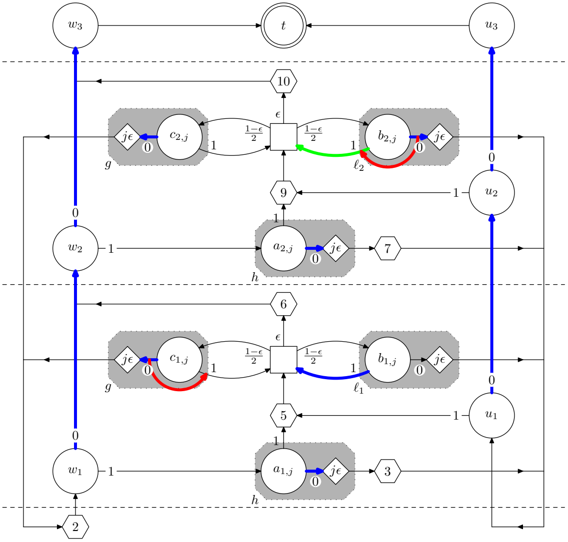
⇒ 1. Make $b_k = 1$.
2. Make $c_k = 1$.
3. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
4. Make $b_i = 0$ and $a_i = 0$, for $i < k$. Reset b_i for unset bits $i > k$.
5. Make $a_k = 1$.
6. Make $w_k = 1$ and $w_i = 0$, for $i < k$.
7. Reset c_i for all unset bits.



RandomEdge lower bound, full construction

- Incrementing the counter happens through seven phases:

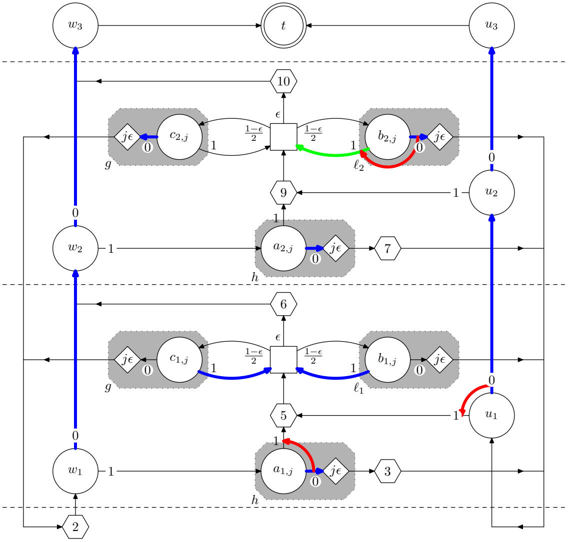
1. Make $b_k = 1$.
2. ⇒ Make $c_k = 1$.
3. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
4. Make $b_i = 0$ and $a_i = 0$, for $i < k$. Reset b_i for unset bits $i > k$.
5. Make $a_k = 1$.
6. Make $w_k = 1$ and $w_i = 0$, for $i < k$.
7. Reset c_i for all unset bits.



RandomEdge lower bound, full construction

- Incrementing the counter happens through seven phases:

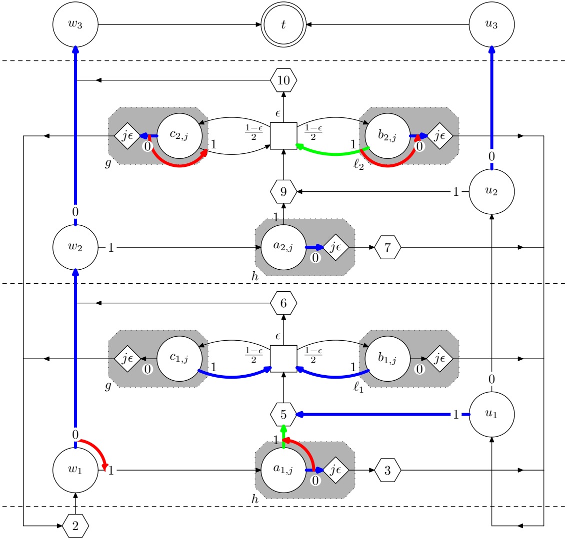
  1. Make $b_k = 1$.
  2. Make $c_k = 1$.
  3. ⇒ Make $u_k = 1$ and $u_i = 0$, for $i < k$.
  4. Make $b_i = 0$ and $a_i = 0$, for $i < k$. Reset b_i for unset bits $i > k$.
  5. Make $a_k = 1$.
  6. Make $w_k = 1$ and $w_i = 0$, for $i < k$.
  7. Reset c_i for all unset bits.



RandomEdge lower bound, full construction

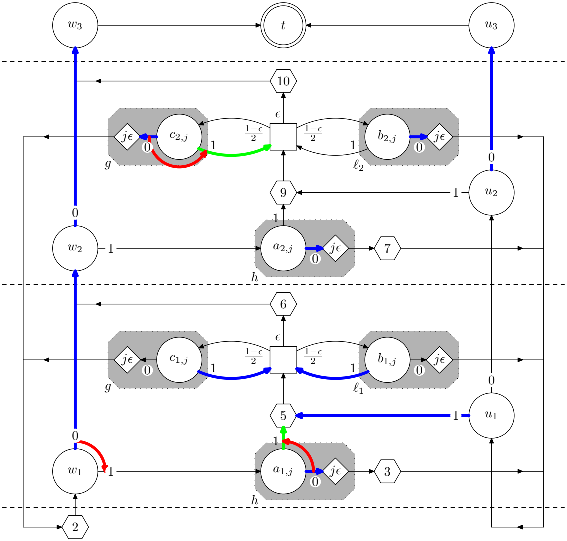
- Incrementing the counter happens through seven phases:

1. Make $b_k = 1$.
2. Make $c_k = 1$.
3. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
4. ⇒ Make $b_i = 0$ and $a_i = 0$, for $i < k$. Reset b_i for unset bits $i > k$.
5. Make $a_k = 1$.
6. Make $w_k = 1$ and $w_i = 0$, for $i < k$.
7. Reset c_i for all unset bits.



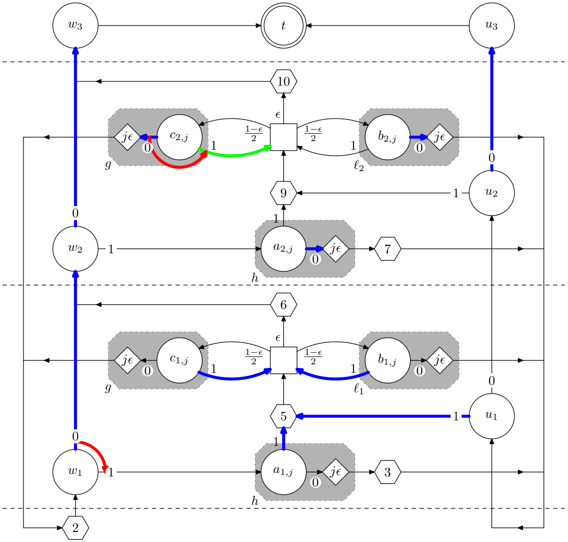
- Incrementing the counter happens through seven phases:

  1. Make $b_k = 1$.
  2. Make $c_k = 1$.
  3. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
  4. Make $b_i = 0$ and $a_i = 0$, for $i < k$. Reset b_i for unset bits $i > k$.
  5. ⇒ Make $a_k = 1$.
  6. Make $w_k = 1$ and $w_i = 0$, for $i < k$.
  7. Reset c_i for all unset bits.



- Incrementing the counter happens through seven phases:

  1. Make $b_k = 1$.
  2. Make $c_k = 1$.
  3. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
  4. Make $b_i = 0$ and $a_i = 0$, for $i < k$. Reset b_i for unset bits $i > k$.
  5. Make $a_k = 1$.
  6. ⇒ Make $w_k = 1$ and $w_i = 0$, for $i < k$.
  7. Reset c_i for all unset bits.

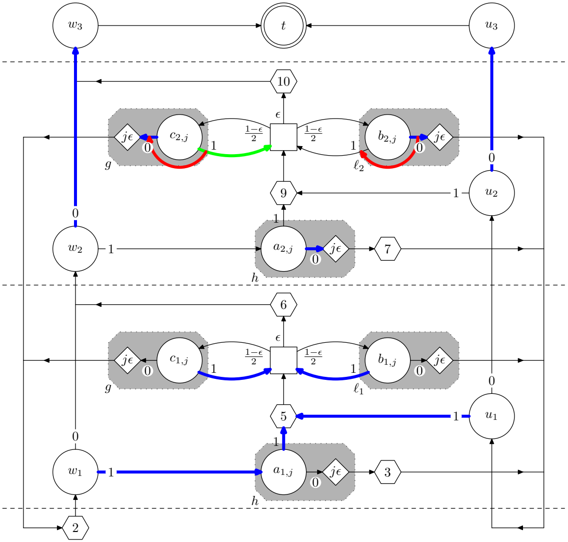


RandomEdge lower bound, full construction

- Incrementing the counter happens through seven phases:

  1. Make $b_k = 1$.
  2. Make $c_k = 1$.
  3. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
  4. Make $b_i = 0$ and $a_i = 0$, for $i < k$. Reset b_i for unset bits $i > k$.
  5. Make $a_k = 1$.
  6. Make $w_k = 1$ and $w_i = 0$, for $i < k$.

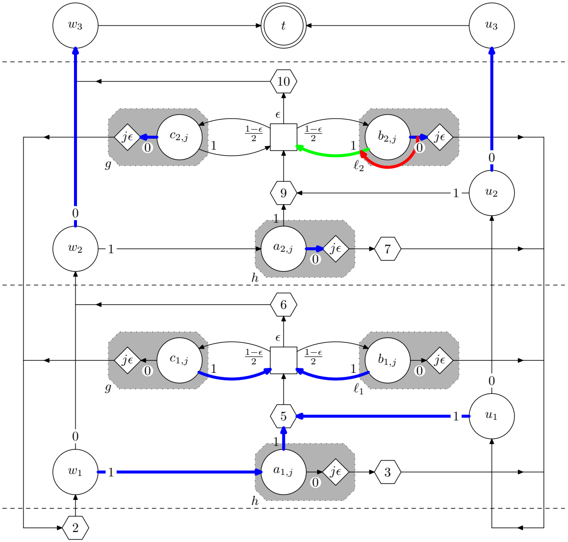
  ⇒ 7. Reset c_i for all unset bits.



RandomEdge lower bound, full construction

- Incrementing the counter happens through seven phases:

⇒ 1. Make $b_k = 1$.
2. Make $c_k = 1$.
3. Make $u_k = 1$ and $u_i = 0$, for $i < k$.
4. Make $b_i = 0$ and $a_i = 0$, for $i < k$. Reset b_i for unset bits $i > k$.
5. Make $a_k = 1$.
6. Make $w_k = 1$ and $w_i = 0$, for $i < k$.
7. Reset c_i for all unset bits.



Competing chains

- The greatest challenge when setting the parameters is to make sure that the lowest unset bit is incremented next with high probability.
- The situation occurs when two chains b_i and b_{i+1} of lengths ℓ_i and ℓ_{i+1} are competing to change from 0 to 1.
- In both chains there is always exactly one improving switch, which means that the RandomEdge pivoting rule will pick either of them with equal probability.
- We bound the probability of failure with a Chernoff bound, and show that it suffices to set $\ell_i = \Theta(i^2 n)$.

RandomEdge lower bound

Theorem (Friedmann, Hansen and Zwick (2011))

The worst-case expected number of pivoting steps performed by RandomEdge *on linear programs with n equalities and $2n$ non-negative variables is* $2^{\Omega(n^{1/4})}$.

$$\begin{array}{ll} \text{maximize} & \sum_{i=1}^{n} \sum_{j=1}^{h} ((-N)^{4i-1} + j\epsilon) a^0_{i,j} + \\ & \sum_{i=1}^{n} ((-N)^{4i+1} + \epsilon(-N)^{4i+2})(a^1_{i,1} + u^1_i) + \\ & \sum_{i=1}^{n} (\epsilon(-N)^{4i+2})(b^1_{i,1} + c^1_{i,1}) + \\ & \sum_{i=1}^{n} \sum_{j=1}^{\ell_i} j\epsilon b^0_{i,j} + \sum_{i=1}^{n} \sum_{j=1}^{g} j\epsilon c^0_{i,j} \end{array}$$

subject to

$$\begin{array}{lrcl}
\forall 1 \le i \le n : & a^0_{i,h} + a^1_{i,h} & = & 1 + w^1_i \\
\forall 1 \le i \le n, \forall 1 \le j < h : & a^0_{i,j} + a^1_{i,j} & = & 1 + a^1_{i,j+1} \\
\forall 1 \le i \le n : & b^0_{i,\ell_i} + b^1_{i,\ell_i} & = & 1 + \frac{1-\epsilon}{2}\left(a^1_{i,1} + b^1_{i,1} + c^1_{i,1} + u^1_i\right) \\
\forall 1 \le i \le n, \forall 1 \le j < \ell_i : & b^0_{i,j} + b^1_{i,j} & = & 1 + b^1_{i,j+1} \\
\forall 1 \le i \le n : & c^0_{i,g} + c^1_{i,g} & = & 1 + \frac{1-\epsilon}{2}\left(a^1_{i,1} + b^1_{i,1} + c^1_{i,1} + u^1_i\right) \\
\forall 1 \le i \le n, \forall 1 \le j < g : & c^0_{i,j} + c^1_{i,j} & = & 1 + c^1_{i,j+1} \\
 & u^0_1 + u^1_1 & = & 1 + \sum_{i=1}^{n} \left(\sum_{j=1}^{h} a^0_{i,j} + \sum_{j=1}^{\ell_i} b^0_{i,j} \right) \\
\forall 2 \le i \le n : & u^0_i + u^1_i & = & 1 + u^0_{i-1} \\
 & w^0_1 + w^1_1 & = & 1 + \sum_{i=1}^{n} \sum_{j=1}^{g} c^0_{i,j} \\
\forall 2 \le i \le n : & w^0_i + w^1_i & = & 1 + w^0_{i-1} + \epsilon\left(a^1_{i,1} + b^1_{i,1} + c^1_{i,1} + u^1_i\right)
\end{array}$$

$$a^0_{i,j}, a^1_{i,j}, b^0_{i,j}, b^1_{i,j}, c^0_{i,j}, c^1_{i,j}, u^0_i, u^1_i, w^0_i, w^1_i \ge 0$$

Function $\textsc{RandomFacet}(G, \pi)$

```
if π contains all actions then
    return π
else
    Choose unused action a uniformly at random
    π' ← RandomFacet(G \ {a}, π)
    if a is improving switch w.r.t. π' then
        π'' ← π'[a]
        return RandomFacet(G, π'')
    else
        return π'
```

Function $\textsc{RandomFacet}(G, \pi, \varphi)$

```
if π contains all actions then
    return π
else
    a ← argmin_{a ∈ A \ π} φ(a)
    π' ← RandomFacet(G \ {a}, π, φ)
    if a is improving switch w.r.t. π' then
        π'' ← π'[a]
        return RandomFacet(G, π'', φ)
    else
        return π'
```

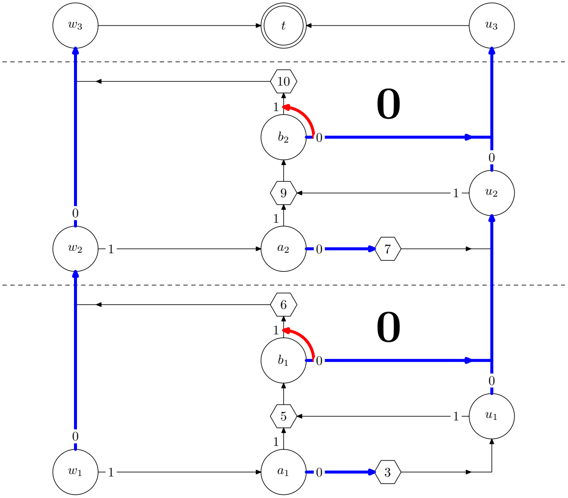
Lower bound for RandomFacet dropped to $2^{\tilde{\Omega}(n^{1/3})}$

- In Friedmann, Hansen and Zwick (SODA, 2011) we proved a $2^{\tilde{\Omega}(n^{1/2})}$ lower bound, for *parity games*, for the "modified RandomFacet algorithm" starting with a uniformly random permutation.
- We *incorrectly* believed, until less than three weeks ago, that by linearity of expectation the RandomFacet algorithm required the same expected number of steps. We now know that this is not true.
- Fortunately, using the same construction with different parameters we have been able to prove a $2^{\tilde{\Omega}(n^{1/3})}$ lower bound, which will be made public as soon as all the details have been written and verified.

Lower bound for RandomFacet dropped to $2^{\tilde{\Omega}(n^{1/3})}$

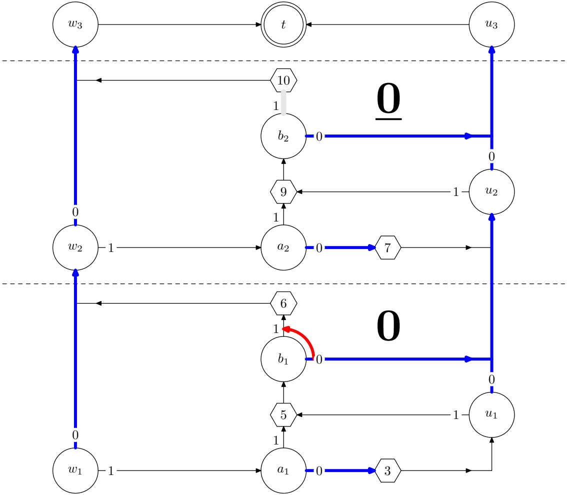
- Looking closer at the "modified RandomFacet algorithm", it turns out to be a dual, recursive variant of the Randomized Bland's rule.
- In Friedmann, Hansen and Zwick (STOC, 2011) we showed a simple transformation of our lower bound parity games to MDPs, thereby getting lower bounds for the simplex algorithm. This transformation remains the same.
  - The main result of this paper was the $2^{\Omega(n^{1/4})}$ lower bound for RandomEdge which is unaffected.

Simplified RandomFacet lower bound



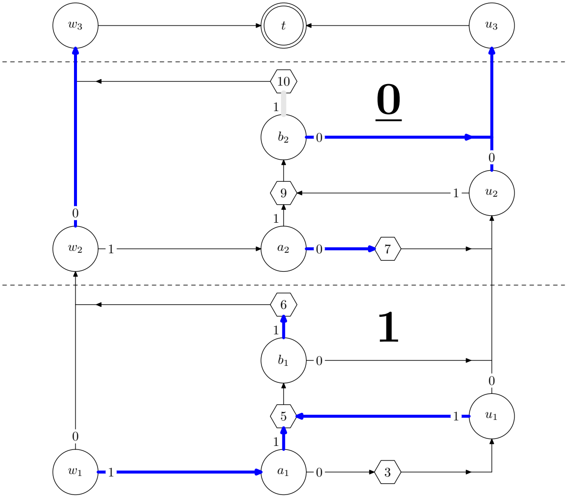
Simplified RandomFacet lower bound

- The RandomFacet algorithm picks a random edge, here b_2^1, and removes it, thereby disabling the bit.



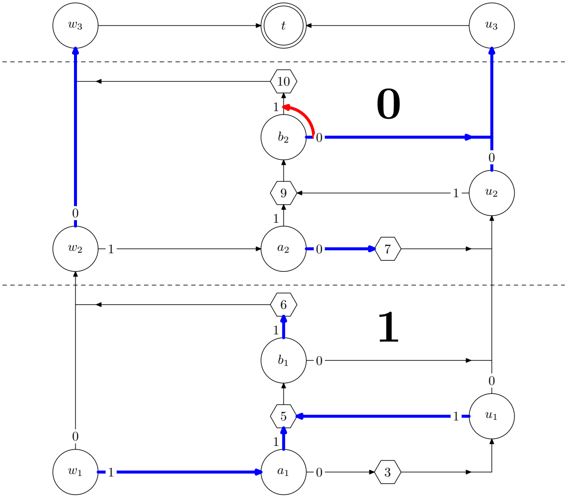
Simplified RandomFacet lower bound

- The RandomFacet algorithm picks a random edge, here b_2^1, and removes it, thereby disabling the bit.
- The MDP is then solved recursively.



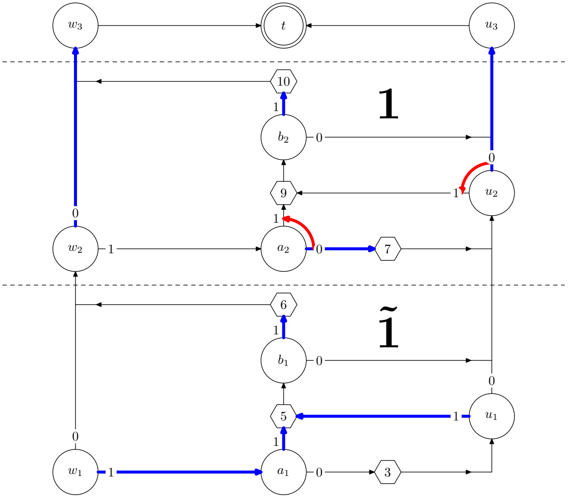
Simplified RandomFacet lower bound

- The RandomFacet algorithm picks a random edge, here b_2^1, and removes it, thereby disabling the bit.
- The MDP is then solved recursively.
- b_2^1 is reintroduced and a switch is made.



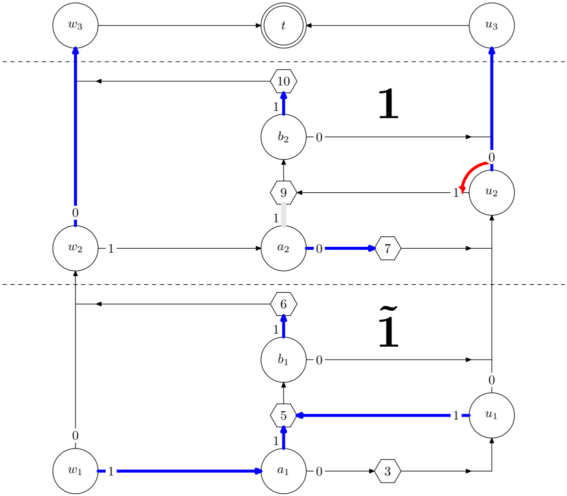
Simplified RandomFacet lower bound

- The RandomFacet algorithm picks a random edge, here b_2^1, and removes it, thereby disabling the bit.
- The MDP is then solved recursively.
- b_2^1 is reintroduced and a switch is made.



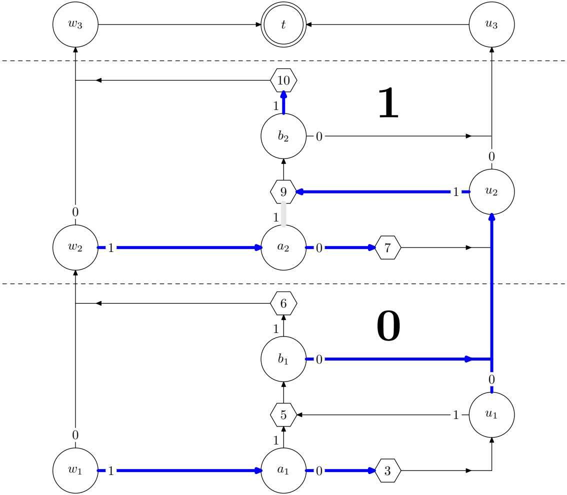
Simplified RandomFacet lower bound

- A new random edge, a_2^1, is removed.



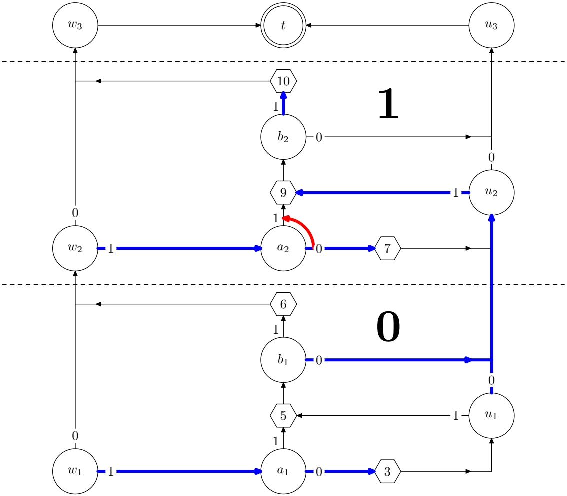
Simplified RandomFacet lower bound

- A new random edge, a_2^1, is removed.
- The MDP is again solved recursively.

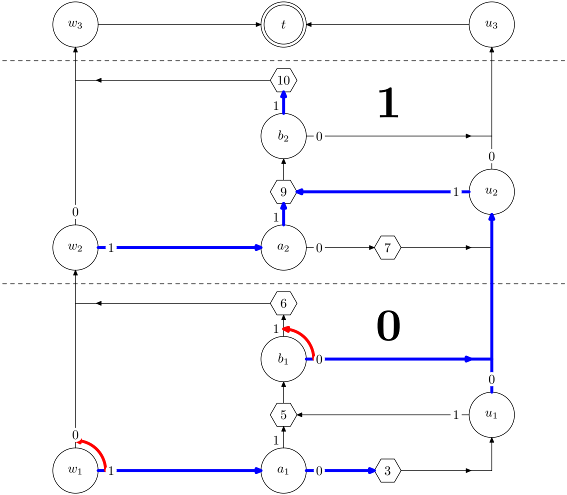


Simplified RandomFacet lower bound

- A new random edge, a_2^1, is removed.
- The MDP is again solved recursively.
- a_2^1 is reintroduced and a switch is made.



- A new random edge, a_2^1, is removed.
- The MDP is again solved recursively.
- a_2^1 is reintroduced and a switch is made.



We simulate a “randomized bitcounter”

Start with n bits with value 0: 00000

We simulate a "randomized bitcounter"

Start with n bits with value 0:	00000
Pick a random bit i and fix it:	00$\underline{0}$00

We simulate a "randomized bitcounter"

Start with n bits with value 0:	00000
Pick a random bit i and fix it:	00<u>0</u>00
Count recursively with the remaining $n-1$ bits:	11<u>0</u>11

We simulate a "randomized bitcounter"

Start with n bits with value 0:	00000
Pick a random bit i and fix it:	00$\underline{0}$00
Count recursively with the remaining $n-1$ bits:	11$\underline{0}$11
Increment the i'th bit:	11$\underline{1}$11

We simulate a "randomized bitcounter"

Start with n bits with value 0:	00000
Pick a random bit i and fix it:	$00\underline{0}00$
Count recursively with the remaining $n-1$ bits:	$11\underline{0}11$
Increment the i'th bit:	$11\underline{1}11$
Reset the $i-1$ lower bits:	$11\underline{1}00$

We simulate a "randomized bitcounter"

Start with n bits with value 0:	00000
Pick a random bit i and fix it:	00$\underline{0}$00
Count recursively with the remaining $n-1$ bits:	11$\underline{0}$11
Increment the i'th bit:	11$\underline{1}$11
Reset the $i-1$ lower bits:	11$\underline{1}$00
Count recursively with the $i-1$ lower bits:	11100

We simulate a "randomized bitcounter"

Start with n bits with value 0:	00000
Pick a random bit i and fix it:	00$\underline{0}$00
Count recursively with the remaining $n-1$ bits:	11$\underline{0}$11
Increment the i'th bit:	11$\underline{1}$11
Reset the $i-1$ lower bits:	11$\underline{1}$00
Count recursively with the $i-1$ lower bits:	11100

- Expected number of increments:

$$\begin{aligned} f(0) &= 0 \\ f(n) &= f(n-1) + 1 + \frac{1}{n}\sum_{i=0}^{n-1} f(i) \qquad \text{for} \qquad n > 0 \end{aligned}$$

We simulate a "randomized bitcounter"

Start with n bits with value 0:	00000
Pick a random bit i and fix it:	00$\underline{0}$00
Count recursively with the remaining $n-1$ bits:	11$\underline{0}$11
Increment the i'th bit:	11$\underline{1}$11
Reset the $i-1$ lower bits:	11$\underline{1}$00
Count recursively with the $i-1$ lower bits:	11100

- Expected number of increments:

$$
\begin{aligned}
f(0) &= 0 \\
f(n) &= f(n-1) + 1 + \frac{1}{n}\sum_{i=0}^{n-1} f(i) \qquad \text{for} \qquad n > 0
\end{aligned}
$$

- Solving the recurrence gives: $f(n) = 2^{\Theta(\sqrt{n})}$

Different challenges

- When constructing lower bounds for the RandomFacet pivoting rule, instead of delaying improving switches, the challenge is to make sure that certain actions are not removed before certain other actions.

Different challenges

- When constructing lower bounds for the RandomFacet pivoting rule, instead of delaying improving switches, the challenge is to make sure that certain actions are not removed before certain other actions.
- Suppose an action a must not be removed before another action b.

Different challenges

- When constructing lower bounds for the RandomFacet pivoting rule, instead of delaying improving switches, the challenge is to make sure that certain actions are not removed before certain other actions.
- Suppose an action a must not be removed before another action b.
- To achieve this with high probability we make use of redundancy: Let a and b be copied k times, in such a way that we only require that at least one copy of b is removed before all copies of a are removed.

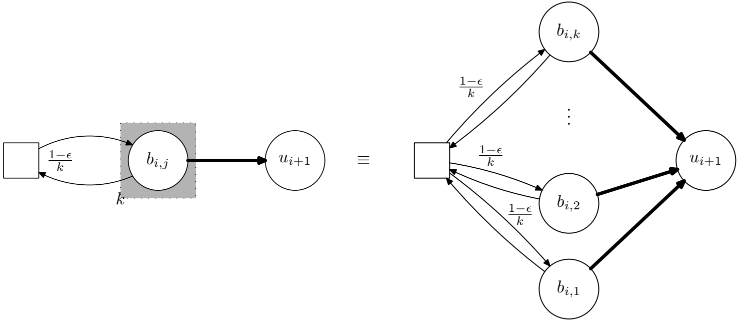
Different challenges

- When constructing lower bounds for the RandomFacet pivoting rule, instead of delaying improving switches, the challenge is to make sure that certain actions are not removed before certain other actions.
- Suppose an action a must not be removed before another action b.
- To achieve this with high probability we make use of redundancy: Let a and b be copied k times, in such a way that we only require that at least one copy of b is removed before all copies of a are removed.
- The only *bad* permutation for the "modified RandomFacet pivoting rule" is then: $aa \dots abb \dots b$

Different challenges

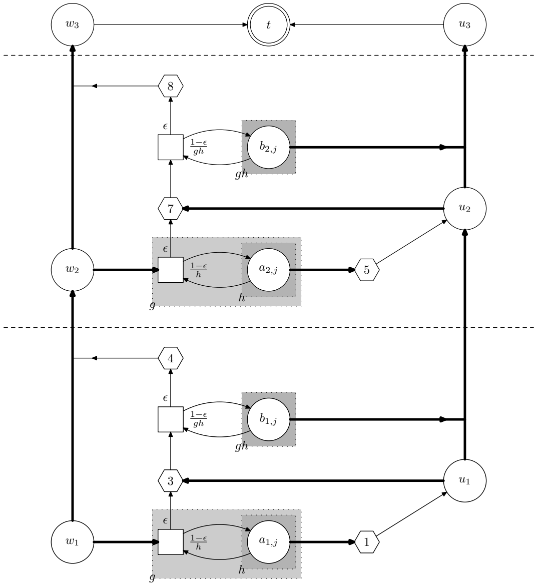
- When constructing lower bounds for the RandomFacet pivoting rule, instead of delaying improving switches, the challenge is to make sure that certain actions are not removed before certain other actions.
- Suppose an action a must not be removed before another action b.
- To achieve this with high probability we make use of redundancy: Let a and b be copied k times, in such a way that we only require that at least one copy of b is removed before all copies of a are removed.
- The only *bad* permutation for the "modified RandomFacet pivoting rule" is then: $aa \ldots abb \ldots b$
- The probability of choosing a bad permutation is $\frac{(k!)^2}{(2k)!} \leq \frac{1}{2^k}$.

Different gadgets



- We use different gadgets to ensure that we get the correct behaviour with high probability.
- For the "modified RandomFacet pivoting rule" this only increases the number of states and actions by a polylogarithmic factor.
- For the real RandomFacet pivoting rule the increase needs to be a factor $\tilde{O}(\sqrt{n})$.

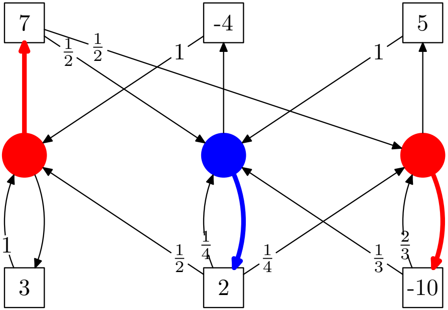
RandomFacet lower bound construction



Overview

- **Lecture 1:**
  - Introduction to linear programming and the simplex algorithm.
  - Pivoting rules.
  - The RandomFacet pivoting rule.
- **Lecture 2:**
  - The Hirsch conjecture.
  - Introduction to Markov decision processes (MDPs).
  - Upper bound for the LargestCoefficient pivoting rule for MDPs.
- **Lecture 3:**
  - Lower bounds for pivoting rules utilizing MDPs. Example: Bland's rule.
  - Lower bound for the RandomEdge pivoting rule.
  - Abstractions and related problems.

2-player turn-based stochastic games (2TBSGs)



- A 2TBSG is an MDP where the set of states is partitioned into two sets: $S_1 \cup S_2 = S$.
  - S_1 is controlled by player 1, the maximizer.
  - S_2 is controlled by player 2, the minimizer.
- A strategy π_1 (or π_2) for player 1 (or player 2) is a choice of an action for each state $i \in S_1$ (or $i \in S_2$).

2-player turn-based stochastic games (2TBSGs)

- A strategy profile $\pi = (\pi_1, \pi_2)$ is a pair of strategies, defining a Markov chain with rewards.
- The value vector for discounted 2TBSGs is again defined as:

$$v_\pi = (I - \gamma P_\pi)^{-1} c_\pi$$

2-player turn-based stochastic games (2TBSGs)

- A strategy profile $\pi = (\pi_1, \pi_2)$ is a pair of strategies, defining a Markov chain with rewards.
- The value vector for discounted 2TBSGs is again defined as:

$$v_\pi = (I - \gamma P_\pi)^{-1} c_\pi$$

- For a fixed strategy π_1 for player 1, a **best response** from player 2 is a strategy:

$$\pi_2(\pi_1) \in \operatorname*{argmin}_{\pi_2} \; v_{\pi_1,\pi_2}$$

2-player turn-based stochastic games (2TBSGs)

- A strategy profile $\pi = (\pi_1, \pi_2)$ is a pair of strategies, defining a Markov chain with rewards.
- The value vector for discounted 2TBSGs is again defined as:

$$v_\pi = (I - \gamma P_\pi)^{-1} c_\pi$$

- For a fixed strategy π_1 for player 1, a **best response** from player 2 is a strategy:

$$\pi_2(\pi_1) \in \operatorname*{argmin}_{\pi_2} \; v_{\pi_1,\pi_2}$$

- Note that $\pi_2(\pi_1)$ can be computed by solving an MDP.

2-player turn-based stochastic games (2TBSGs)

- A strategy profile $\pi = (\pi_1, \pi_2)$ is a pair of strategies, defining a Markov chain with rewards.
- The value vector for discounted 2TBSGs is again defined as:

$$v_\pi = (I - \gamma P_\pi)^{-1} c_\pi$$

- For a fixed strategy π_1 for player 1, a **best response** from player 2 is a strategy:

$$\pi_2(\pi_1) \in \operatorname*{argmin}_{\pi_2} \; v_{\pi_1, \pi_2}$$

- Note that $\pi_2(\pi_1)$ can be computed by solving an MDP.
- A best response from player 1, $\pi_1(\pi_2)$, is defined analogously.

2-player turn-based stochastic games (2TBSGs)

- π_1^* and π_2^* are optimal if:

$$\forall \pi_1 : \quad v_{\pi_1^*, \pi_2(\pi_1^*)} \geq v_{\pi_1, \pi_2(\pi_1)}$$
$$\forall \pi_2 : \quad v_{\pi_1(\pi_2^*), \pi_2^*} \leq v_{\pi_1(\pi_2), \pi_2}$$

2-player turn-based stochastic games (2TBSGs)

- π_1^* and π_2^* are optimal if:

$$\forall \pi_1 : \quad v_{\pi_1^*, \pi_2(\pi_1^*)} \geq v_{\pi_1, \pi_2(\pi_1)}$$
$$\forall \pi_2 : \quad v_{\pi_1(\pi_2^*), \pi_2^*} \leq v_{\pi_1(\pi_2), \pi_2}$$

- Alternatively, π_1^* and π_2^* are optimal if π_1^* is a best response to π_2^*, and π_2^* is a best response to π_1^*. We then say that (π_1^*, π_2^*) is a **Nash equilibrium**.

2-player turn-based stochastic games (2TBSGs)

- π_1^* and π_2^* are optimal if:

$$\forall \pi_1 : \quad v_{\pi_1^*, \pi_2(\pi_1^*)} \geq v_{\pi_1, \pi_2(\pi_1)}$$
$$\forall \pi_2 : \quad v_{\pi_1(\pi_2^*), \pi_2^*} \leq v_{\pi_1(\pi_2), \pi_2}$$

- Alternatively, π_1^* and π_2^* are optimal if π_1^* is a best response to π_2^*, and π_2^* is a best response to π_1^*. We then say that (π_1^*, π_2^*) is a **Nash equilibrium**.
- Shapley (1953): Optimal strategies always exist.

2-player turn-based stochastic games (2TBSGs)

- π_1^* and π_2^* are optimal if:

$$\forall \pi_1 : \quad v_{\pi_1^*, \pi_2(\pi_1^*)} \geq v_{\pi_1, \pi_2(\pi_1)}$$
$$\forall \pi_2 : \quad v_{\pi_1(\pi_2^*), \pi_2^*} \leq v_{\pi_1(\pi_2), \pi_2}$$

- Alternatively, π_1^* and π_2^* are optimal if π_1^* is a best response to π_2^*, and π_2^* is a best response to π_1^*. We then say that (π_1^*, π_2^*) is a **Nash equilibrium**.
- Shapley (1953): Optimal strategies always exist.
- Solving a 2TBSG means finding an optimal strategy profile.

2-player turn-based stochastic games (2TBSGs)

- π_1^* and π_2^* are optimal if:

$$\forall \pi_1 : \quad v_{\pi_1^*, \pi_2(\pi_1^*)} \geq v_{\pi_1, \pi_2(\pi_1)}$$
$$\forall \pi_2 : \quad v_{\pi_1(\pi_2^*), \pi_2^*} \leq v_{\pi_1(\pi_2), \pi_2}$$

- Alternatively, π_1^* and π_2^* are optimal if π_1^* is a best response to π_2^*, and π_2^* is a best response to π_1^*. We then say that (π_1^*, π_2^*) is a **Nash equilibrium**.
- Shapley (1953): Optimal strategies always exist.
- Solving a 2TBSG means finding an optimal strategy profile.
- Note that the decision problem corresponding to solving 2TBSGs is in **NP** $\cap$ *co***NP**, since an optimal strategy profile is a witness for both *yes* and *no* answers. The problem is not known to be in **P**.

2-player turn-based stochastic games (2TBSGs)

- We again say that $a \in A_i$, for $i \in S_1$, is an **improving switch** for player 1 w.r.t. π iff:

$$(v_\pi)_i < c_a + \gamma P_a v_\pi$$

- Similarly, $a \in A_i$, for $i \in S_2$, is an improving switch for player 2 w.r.t. π iff:

$$(v_\pi)_i > c_a + \gamma P_a v_\pi$$

2-player turn-based stochastic games (2TBSGs)

- We again say that $a \in A_i$, for $i \in S_1$, is an **improving switch** for player 1 w.r.t. π iff:

$$(v_\pi)_i < c_a + \gamma P_a v_\pi$$

- Similarly, $a \in A_i$, for $i \in S_2$, is an improving switch for player 2 w.r.t. π iff:

$$(v_\pi)_i > c_a + \gamma P_a v_\pi$$

- The vector of reduced costs for a strategy profile π is again defined as:

$$\bar{c}^\pi = c - (J - \gamma P) v_\pi$$

2-player turn-based stochastic games (2TBSGs)

- We again say that $a \in A_i$, for $i \in S_1$, is an **improving switch** for player 1 w.r.t. π iff:

$$(v_\pi)_i < c_a + \gamma P_a v_\pi$$

- Similarly, $a \in A_i$, for $i \in S_2$, is an improving switch for player 2 w.r.t. π iff:

$$(v_\pi)_i > c_a + \gamma P_a v_\pi$$

- The vector of reduced costs for a strategy profile π is again defined as:

$$\bar{c}^\pi = c - (J - \gamma P) v_\pi$$

- Note that (π_1^*, π_2^*) is a Nash equilibrium iff there are no improving switches.

Function StrategyIteration(π_1)

while $\exists$ *improving switch w.r.t.* $(\pi_1, \pi_2(\pi_1))$ **do**

  Update π_1 by performing improving switches

return $(\pi_1, \pi_2(\pi_1))$

Function StrategyIteration(π_1)

while $\exists$ *improving switch w.r.t.* $(\pi_1, \pi_2(\pi_1))$ **do**

  Update π_1 by performing improving switches

return $(\pi_1, \pi_2(\pi_1))$

- Howard's algorithm can be naturally extended to 2TBSGs by choosing:

$$\forall i \in S_1 : \quad \pi_1(i) \leftarrow \operatorname*{argmax}_{a \in A_i} \bar{c}_a^{\pi_1, \pi_2(\pi_1)}$$

Non-discounted MDPs and 2TBSGs

- We have already seen that discounted MDPs are a special case of MDPs satisfying the stopping condition, and the same is true for 2TBSGs.
- Liggett and Lippman (1969) showed that for any 2TBSG G there exists a discount factor $\gamma_G < 1$, such that the same strategies are optimal for all discount factors $\gamma' \in [\gamma_G, 1)$.

Non-discounted MDPs and 2TBSGs

- We have already seen that discounted MDPs are a special case of MDPs satisfying the stopping condition, and the same is true for 2TBSGs.
- Liggett and Lippman (1969) showed that for any 2TBSG G there exists a discount factor $\gamma_G < 1$, such that the same strategies are optimal for all discount factors $\gamma' \in [\gamma_G, 1)$.
  - Andersson and Miltersen (2009) showed that γ_G can be described with a number of bits that is polynomial in the bit complexity of G.

Non-discounted MDPs and 2TBSGs

- We have already seen that discounted MDPs are a special case of MDPs satisfying the stopping condition, and the same is true for 2TBSGs.
- Liggett and Lippman (1969) showed that for any 2TBSG G there exists a discount factor $\gamma_G < 1$, such that the same strategies are optimal for all discount factors $\gamma' \in [\gamma_G, 1)$.
  - Andersson and Miltersen (2009) showed that γ_G can be described with a number of bits that is polynomial in the bit complexity of G.
- A 2TBSG G is called **non-discounted** if it is implicitly using discount factor γ_G.

Non-discounted MDPs and 2TBSGs

- We have already seen that discounted MDPs are a special case of MDPs satisfying the stopping condition, and the same is true for 2TBSGs.
- Liggett and Lippman (1969) showed that for any 2TBSG G there exists a discount factor $\gamma_G < 1$, such that the same strategies are optimal for all discount factors $\gamma' \in [\gamma_G, 1)$.
  - Andersson and Miltersen (2009) showed that γ_G can be described with a number of bits that is polynomial in the bit complexity of G.
- A 2TBSG G is called **non-discounted** if it is implicitly using discount factor γ_G.
- MDPs (and 2TBSGs) satisfying the stopping condition are a special case of non-discounted MDPs (and 2TBSGs). See, e.g., Puterman (1994).

Special cases of 2TBSGs

- A non-discounted 2TBSG whose actions are all deterministic is called a **mean payoff game**.

Special cases of 2TBSGs

- A non-discounted 2TBSG whose actions are all deterministic is called a **mean payoff game**.
- An n-state mean payoff game where the reward of every action a is described by an integer priority p_a, such that $c_a = (-n)^{p_a}$, and where all actions leaving the same state have the same priority, is called a **parity game**.
  - Note that the mean of the rewards of a cycle is positive iff the parity of the largest priority is even.

Special cases of 2TBSGs

- A non-discounted 2TBSG whose actions are all deterministic is called a **mean payoff game**.
- An n-state mean payoff game where the reward of every action a is described by an integer priority p_a, such that $c_a = (-n)^{p_a}$, and where all actions leaving the same state have the same priority, is called a **parity game**.
  - Note that the mean of the rewards of a cycle is positive iff the parity of the largest priority is even.
- There is no known polynomial time algorithm for solving parity games.

A few results about StrategyIteration

- Friedmann (2009) showed that Howard's algorithm requires exponentially many iterations to solve parity games.
- Fearnley (2010) transformed Friedmann's construction to MDPs.

A few results about StrategyIteration

- Friedmann (2009) showed that Howard's algorithm requires exponentially many iterations to solve parity games.
- Fearnley (2010) transformed Friedmann's construction to MDPs.
  - These lower bounds are precursors for the lower bounds for RandomEdge, RandomFacet and Randomized Bland's rule by Friedmann, Hansen and Zwick (2011), and LeastEntered by Friedmann (2011). All of which were also first obtained for parity games.

A few results about StrategyIteration

- Friedmann (2009) showed that Howard's algorithm requires exponentially many iterations to solve parity games.
- Fearnley (2010) transformed Friedmann's construction to MDPs.
  - These lower bounds are precursors for the lower bounds for RandomEdge, RandomFacet and Randomized Bland's rule by Friedmann, Hansen and Zwick (2011), and LeastEntered by Friedmann (2011). All of which were also first obtained for parity games.
- Ye (2010): $O(\frac{mn}{1-\gamma} \log \frac{n}{1-\gamma})$ iterations for discounted MDPs with n states and m actions.

A few results about StrategyIteration

- Friedmann (2009) showed that Howard's algorithm requires exponentially many iterations to solve parity games.
- Fearnley (2010) transformed Friedmann's construction to MDPs.
    - These lower bounds are precursors for the lower bounds for RandomEdge, RandomFacet and Randomized Bland's rule by Friedmann, Hansen and Zwick (2011), and LeastEntered by Friedmann (2011). All of which were also first obtained for parity games.
- Ye (2010): $O(\frac{mn}{1-\gamma} \log \frac{n}{1-\gamma})$ iterations for discounted MDPs with n states and m actions.
- Hansen, Miltersen and Zwick (2011): $O(\frac{m}{1-\gamma} \log \frac{n}{1-\gamma})$ iterations for discounted 2TBSGs with n states and m actions.

2TBSGs are LP-type problems

- Ludwig (1995), Halman (2007): 2TBSGs are LP-type problems.
  - Let H be the set of actions for player 1, and let ω map a subgame to the sum of its optimal values. Bases are strategies for player 1.
  - *Monotonicity*: More available actions only increases the value.
  - *Locality*: If $F \subseteq G \subseteq H$ and $-\infty < \omega(F) = \omega(G)$, then F and G share optimal strategies, and if an added action $h \in H$ is an improving switch for one then it also is for the other.

2TBSGs are LP-type problems

- Ludwig (1995), Halman (2007): 2TBSGs are LP-type problems.
  - Let H be the set of actions for player 1, and let ω map a subgame to the sum of its optimal values. Bases are strategies for player 1.
  - *Monotonicity*: More available actions only increases the value.
  - *Locality*: If $F \subseteq G \subseteq H$ and $-\infty < \omega(F) = \omega(G)$, then F and G share optimal strategies, and if an added action $h \in H$ is an improving switch for one then it also is for the other.

- Hence, the dual RANDOMFACET algorithm can be used to solve 2TBSGs, and, in fact, it gives the best known bound for solving the non-discounted problem.

Unique sink orientations of cubes

- MDPs and 2TBSGs with two actions per state can be described abstractly by acyclic unique sink orientations (AUSOs) of hypercubes:
  - Strategies for player 1 map to vertices of the cube, and improving switches define an orientation of the edges such that in each face there is a unique sink.
  - An algorithm can evaluate vertices of the cube to learn the orientation of the adjacent edges, and the goal is to find the unique sink of the entire cube.

Unique sink orientations of cubes

- MDPs and 2TBSGs with two actions per state can be described abstractly by acyclic unique sink orientations (AUSOs) of hypercubes:
    - Strategies for player 1 map to vertices of the cube, and improving switches define an orientation of the edges such that in each face there is a unique sink.
    - An algorithm can evaluate vertices of the cube to learn the orientation of the adjacent edges, and the goal is to find the unique sink of the entire cube.
- Szabó and Welzl (2001) introduced the FibonacciSeesaw algorithm for solving n dimensional unique sink orientations with F_{n+2} vertex evaluations.

More extensions of 2TBSGs

- There is no known way to formulate 2TBSGs as linear programs.
- Gärtner and Rüst (2005), and Jurdziński and Ravani (2008) showed how to formulate 2TBSGs with two actions per state as P-matrix linear complementarity problems.

More extensions of 2TBSGs

- There is no known way to formulate 2TBSGs as linear programs.
- Gärtner and Rüst (2005), and Jurdziński and Ravani (2008) showed how to formulate 2TBSGs with two actions per state as P-matrix linear complementarity problems.
- P-matrix linear complementarity problems are also generalized by USOs, but not by AUSOs.

More extensions of 2TBSGs

- There is no known way to formulate 2TBSGs as linear programs.
- Gärtner and Rüst (2005), and Jurdziński and Ravani (2008) showed how to formulate 2TBSGs with two actions per state as P-matrix linear complementarity problems.
- P-matrix linear complementarity problems are also generalized by USOs, but not by AUSOs.
- Solving P-matrix linear complementarity problems, as well as 2TBSGs, is known to be in $\mathbf{PPAD} \cap \mathbf{PLS}$. Daskalakis and Papadimitriou (2011) suggested a new complexity class **CLS** (continuous local search) for capturing these and other problems.