UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

david

données et algorithmes pour une ville intelligente et durable

David Auger, **Pierre Coucheney**, Yann Strozecki

Solving Simple Stochastic Games with few Random Nodes faster using Bland's Rule

November 2019,
Journée COSMOS

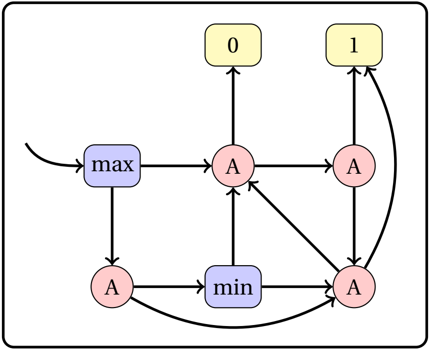
What’s an SSG ?

Playing with SSGs in real life

Simple stochastic game (SSG)

A Simple Stochastic Game (Shapley, Condon) is defined by a directed graph with :

- three sets of vertices V_{MAX}, V_{MIN}, V_{AVE} of outdegree 2
- two (or more) 'sink' vertices with values 0 and 1

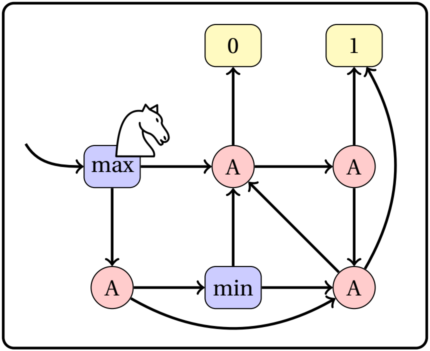


Two players : MAX and MIN, and *randomness*.

Rules of an SSG

A play consists in moving a *pebble* on the graph :

- player MAX wants to maximize the value of the sink reached.
- player MIN wants to minimize the value. If no sink is reached, the value is 0.

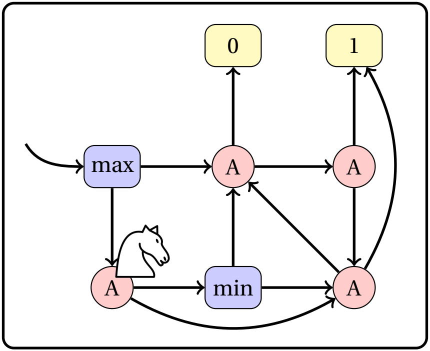


On a MAX node player MAX decides where to go next.

Rules of an SSG

A play consists in moving a *pebble* on the graph :

- player MAX wants to maximize the value of the sink reached.
- player MIN wants to minimize the value. If no sink is reached, the value is 0.

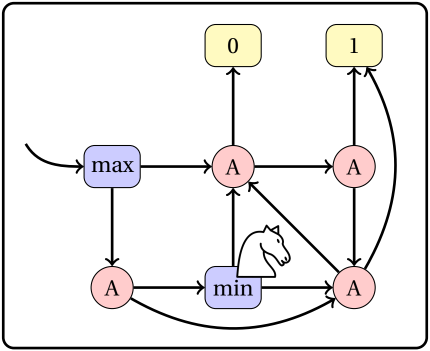


On a AVE node the next vertex is randomly determined.

Rules of an SSG

A play consists in moving a *pebble* on the graph :

- player MAX wants to maximize the value of the sink reached.
- player MIN wants to minimize the value. If no sink is reached, the value is 0.

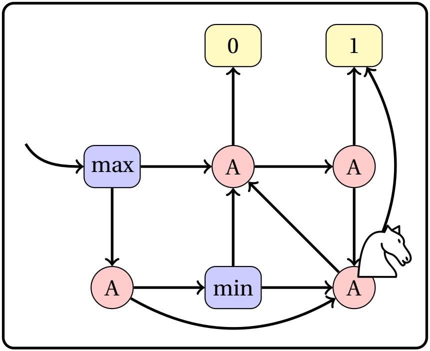


On a MIN node player MIN decides where to go next.

Rules of an SSG

A play consists in moving a *pebble* on the graph :

- player MAX wants to maximize the value of the sink reached.
- player MIN wants to minimize the value. If no sink is reached, the value is 0.

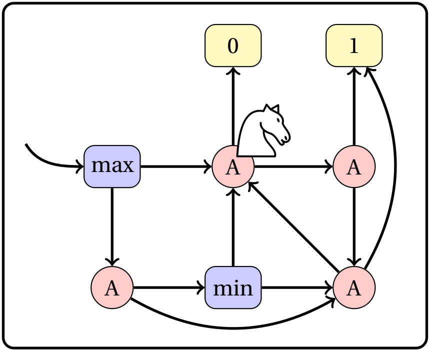


Etc.

Rules of an SSG

A play consists in moving a *pebble* on the graph :

- player MAX wants to maximize the value of the sink reached.
- player MIN wants to minimize the value. If no sink is reached, the value is 0.

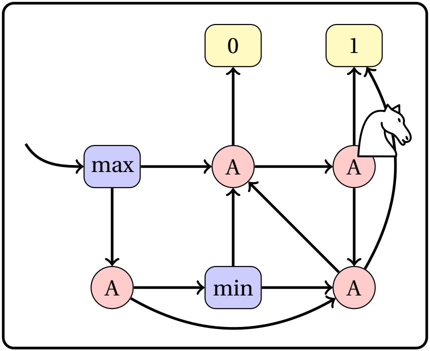


Etc.

Rules of an SSG

A play consists in moving a *pebble* on the graph :

- player MAX wants to maximize the value of the sink reached.
- player MIN wants to minimize the value. If no sink is reached, the value is 0.

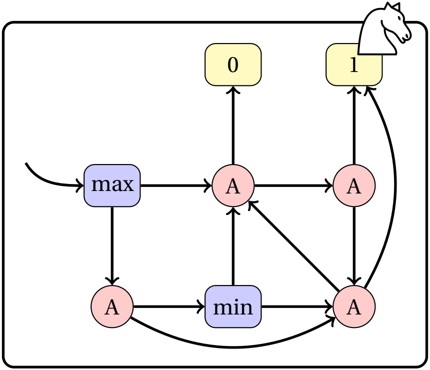


Etc.

Rules of an SSG

A play consists in moving a *pebble* on the graph :

- player MAX wants to maximize the value of the sink reached.
- player MIN wants to minimize the value. If no sink is reached, the value is 0.

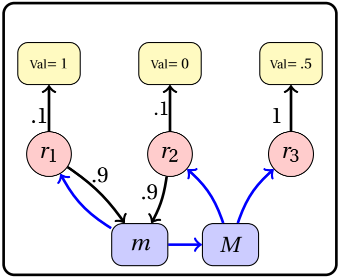


Etc.

Generalized SSGs

Generalize *binary* SSG :

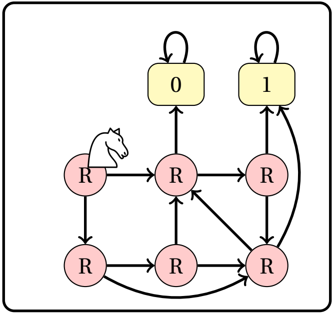
- arbitrary outdegree on the MAX and MIN nodes
- arbitrary values on sinks
- arbitrary probability distribution on the outneighbours of each AVE node



What’s the value of an SSG?

Markov Chain

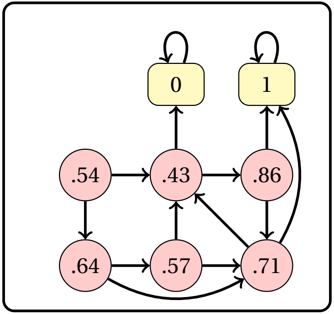
- a finite, stationnary markov chain as a collection of *random nodes* with a token moving
- suppose : proba 1 of reaching a *sink node*, each with a given *value*



***value* of node v = *average value* of the sink that is reached**

Values of nodes

Here : binary case (outdegree 2, uniform probability)

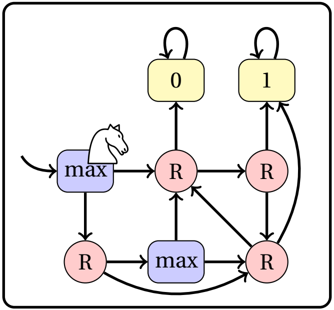


Easily computed by linear system :

$$\forall \text{ non sink node } v, \quad val[v] = \sum_{w} p(v,w) \cdot val[w]$$

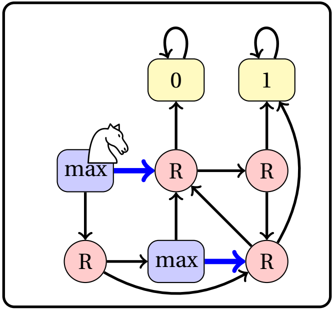
Markov Decision Process

- Add some *decision nodes* and 1 player
- on a decision node, the player chooses the next node among neighbours



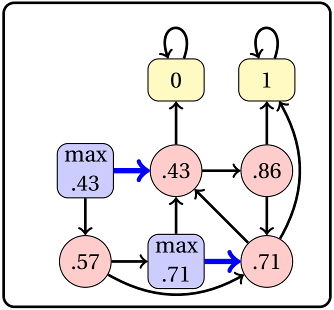
goal : maximize the *value* of a node / all nodes

Markovian property in MDP



- There is an optimal solution which is stationnary and pure (deterministic)
- *strategy* := choice of an outneighbour for every max node

Values of a strategy



- There is an optimal solution which is stationnary and pure (deterministic)
- *strategy* := choice of an outneighbour for every max node

Solving a MDP

Bellman equations for optimal values val_* (under mild conditions)

- $\forall v$ random node

$$val_*[v] = \sum_w p(v,w) \cdot val_*[w]$$

- $\forall$ max node

$$val_*[v] = \max_{(v,w)\in A} val_*[w]$$

- max / linear (average) system
- solved by LP

Optimal values in an SSG

We consider only *positional strategies* :

$$\sigma : V_{\text{MAX}} \longrightarrow V, \quad \tau : V_{\text{MIN}} \longrightarrow V$$

The *value* of a vertex x is the best expected value of a sink that MAX can guarantee starting from x :

$$val_*(x) = \max_{\substack{\sigma \text{ strategy} \\ \text{for MAX}}} \min_{\substack{\tau \text{ strategy} \\ \text{for MIN}}} \underbrace{\mathbb{E}_{\sigma,\tau}\left(\text{value of the sink reached} \mid \text{game starts in } x\right)}_{val_{\sigma,\tau}(x)}$$

Optimal values in an SSG

We consider only *positional strategies* :

$$\sigma : V_{\text{MAX}} \longrightarrow V, \quad \tau : V_{\text{MIN}} \longrightarrow V$$

The *value* of a vertex x is the best expected value of a sink that MAX can guarantee starting from x :

$$val_*(x) = \max_{\substack{\sigma \text{ strategy} \\ \text{for MAX}}} \min_{\substack{\tau \text{ strategy} \\ \text{for MIN}}} \underbrace{\mathbb{E}_{\sigma,\tau}\left(\text{value of the sink reached} \mid \text{game starts in } x\right)}_{val_{\sigma,\tau}(x)}$$

Problem : given a game and a vertex, compute the value of the vertex.

Decision problem : $val_*(x) > 0.5$?

Solving an SSG

Bellman equations for optimal values val_* (under mild conditions)

- $\forall v$ random node
$$val_*[v] = \sum_w p(v,w) \cdot val_*[w]$$
- $\forall v$ MAX node
$$val_*[v] = \max_{(v,w)\in A} val_*[w]$$
- $\forall v$ MIN node
$$val_*[v] = \min_{(v,w)\in A} val_*[w]$$

Solving an SSG

Bellman equations for optimal values val_* (under mild conditions)

- $\forall v$ random node
$$val_*[v] = \sum_w p(v,w) \cdot val_*[w]$$
- $\forall v$ MAX node
$$val_*[v] = \max_{(v,w)\in A} val_*[w]$$
- $\forall v$ MIN node
$$val_*[v] = \min_{(v,w)\in A} val_*[w]$$

- max / min / linear (average) system
- Complexity in $NP \cap co-NP$; is it in P?
- Harder than *Parity Game, Mean payoff Game, Discounted payoff Game* but equivalent to their stochastic versions.

The big picture

1. Simple Stochastic Games are a class of two-players, turn-based, zero-sum games played on graphs.

They are hard to solve.

2. Ludwig's randomized algorithm has expected complexity of $p(n) \cdot 2^{O(\sqrt{n})}$

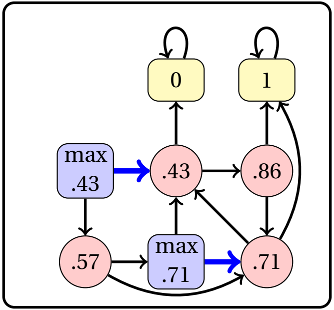
3. Gimbert and Horn deterministic algorithm has parametrized complexity of $p(n) \cdot 2^{O(k \log k)}$
k = number of random nodes.

4. We give a randomized algorithm using both techniques, with expected parametrized complexity of $p(n) \cdot 2^{O(k)}$

Beside the $\log k$ factor, interesting structures and properties arise that need further examination.

Algorithms to solve SSGs

The switch operation



- The strategy at the upper max node is *switchable*: the Bellman equation is not satisfied.
- If we switch, we obtain a better strategy

Switch = pivot operation that stricly improves the current strategy

Strategy iteration algorithms

Switch Operation for SSGs :

- The *values* of MAX -strategy σ are the values of σ against a best response to σ from the MIN player.

Lemma

Switching a switchable node increases the value of a strategy.

Strategy iteration algorithms

Switch Operation for SSGs :

- The *values* of MAX -strategy σ are the values of σ against a best response to σ from the MIN player.

Lemma

Switching a switchable node increases the value of a strategy.

Strategy iteration Algorithm :

input **:** SSG

· start with an initial MAX strategy σ

while σ *is not optimal (check Bellman eq.)* **do**

  · choose S a subset of the switchables nodes

  · switch the nodes of S in σ

  · update the values of σ (against a best response)

return σ

Complexity of strategy improvement

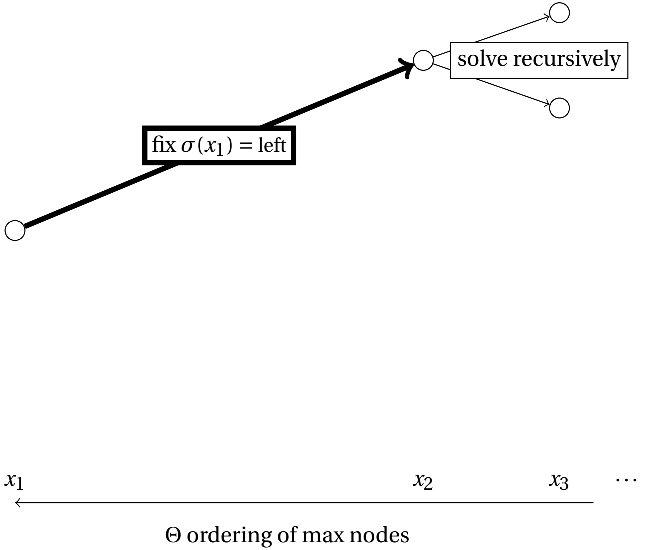
Similar to the simplex algorithm. One degree of freedom : choice of the vertices which are switched at each step.

- Switch all switchable nodes : at most $2^n/n$ steps [Kumar, Valkanova,Tripathi].
- Switch a random subset : $2^{0,78n}$ steps in average.

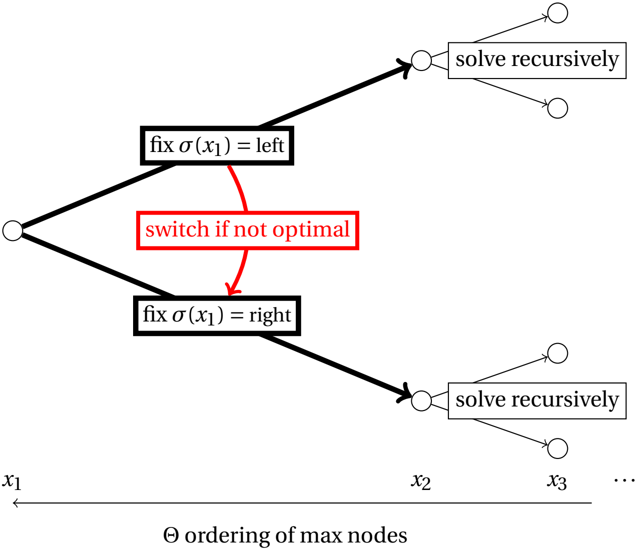
Lower bounds for these methods $2^{\sqrt{n}}$ steps [Friedmann].

Ludwig’s Algorithm – Bland’s rule

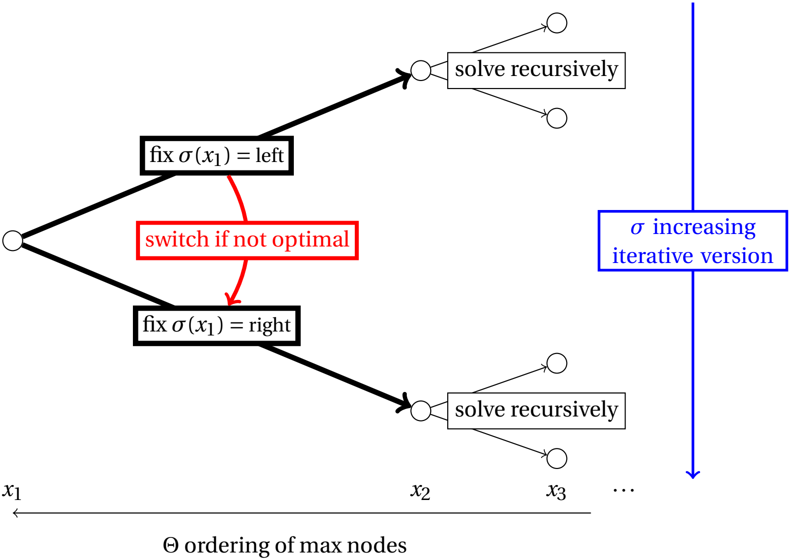
Ludwig's Algorithm : recursive version

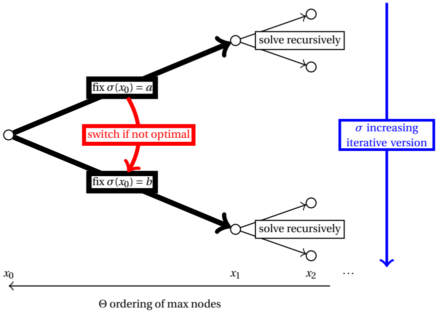


Ludwig's Algorithm : recursive version



Ludwig's Algorithm : recursive version





random position of x_0 in the set of nodes (for a "technical" order)

$$\underbrace{x, x, x, \cdots x, x}_{\text{will never be switched again in the second subtree}}, \boxed{x_0}, \cdots x, x, x$$

- Recursive formula with n max nodes on the average number of iterations Φ

$$\Phi(n) \leq \Phi(n-1) + 1 + \frac{1}{n} \sum_{i=0}^{n-1} \Phi(i).$$

- $2^{c.\sqrt{n}}$ iterations on average instead of 2^n.

Ludwig's Algorithm : randomized Bland's rule on SSGs

Ludwig's Algorithm (Ludwig, 1995). Iterative simplified version.

input : **binary** SSG

· start with an initial MAX strategy σ

·**Pick randomly and uniformly a total order Θ on *max*-nodes**

while σ *is not optimal (check Bellman eq.)* **do**

  · switch σ **at the first switchable node in order Θ**

  · update the values of σ (against a best response)

return σ

Theorem (Ludwig)

The expected number of strategies considered by this algorithm is at most $e^{2\sqrt{n}}$.

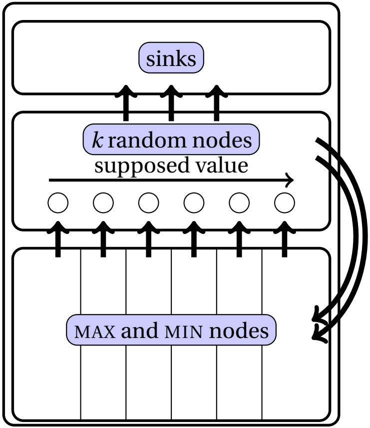
n is the number of MAX -nodes (at least 2^n strategies)

Gimbert & Horn framework – few random nodes

Framework in which we want to apply Ludwig's technique

Main idea

To solve an SSG you only need to know the ordering of the values for random nodes



- Gimbert and Horn (2007)
- if *the ordering of the values of random nodes* is known, then the resulting game is *deterministic*
- $\mathcal{T}_k$: set of total orders on $1, 2, \cdots k$
- algorithm : enumerate/iterate $\mathcal{T}_k$ and check for optimality conditions.

$$k! \approx 2^{O(k \log(k))} \text{ iterations}$$

Using the two techniques together

(Finally) Our main result

With a clever iteration on strategies using a dichtomic partition, Ludwig reduces the average number of iterations from 2^n to $2^{c\sqrt{n}}$.

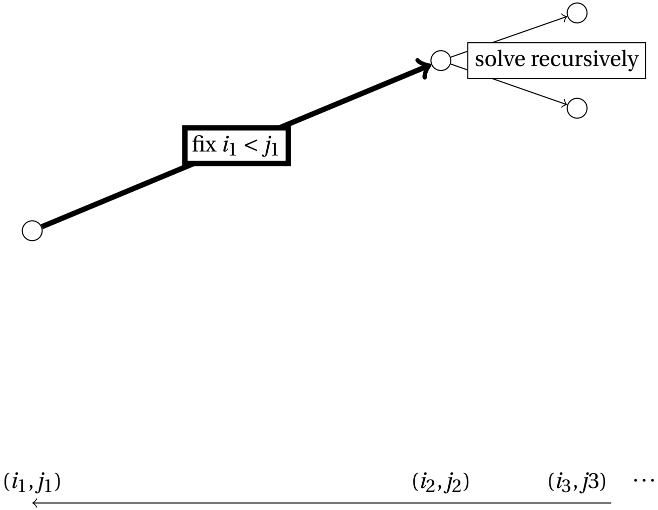
Gimbert and Horn enumerates $k! \approx 2^{ck\log(k)}$ orders of set $\mathcal{T}_k$

Find a “pivot” operation on $\mathcal{T}_k$, with an ad-hoc dichotomic enumeration

Then we can enumerate $\mathcal{T}_k$ in time

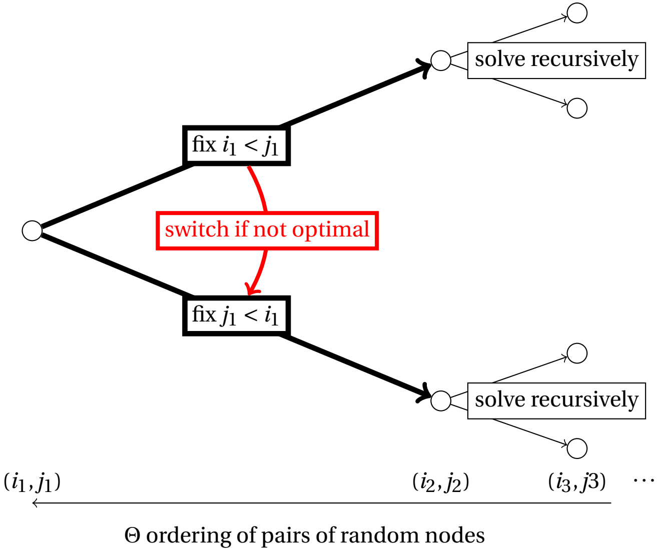
$$2^{c.\sqrt{k^2}} = 2^{c.k}$$

Ludwig's Like Algorithm on orders : recursive version

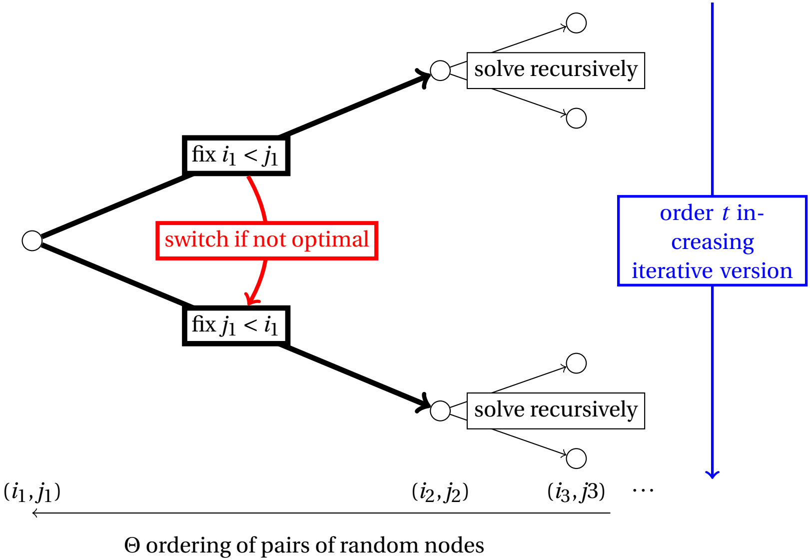


Θ ordering of pairs of random nodes

Ludwig's Like Algorithm on orders : recursive version



Ludwig's Like Algorithm on orders : recursive version

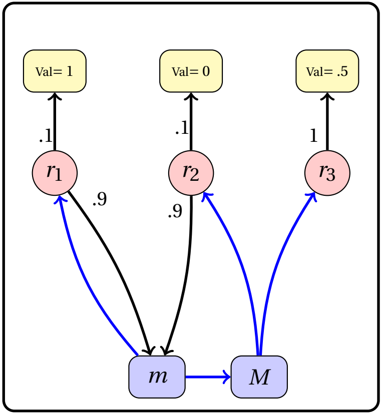


The auxilliary graph

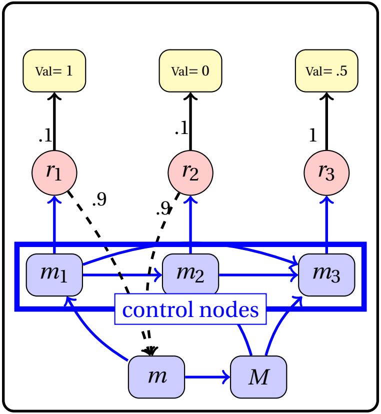
To a graph G and an order t on the k random nodes associate $G[t]$:

- the set of sinks, *max*-nodes and *ran*-nodes remain the same as in G;
- For every $1 \leq i \leq k$, add a *min*-node denoted i to $G[t]$, which we call *control node* and add an arc (i, r_i) ;
- For every $(i, j) \in t$, $i \neq j$, add the arc (i, j) to $G[t]$;
- For every arc $(x, r_i) \in A$, remove this arc and add an arc (x, i).

Auxiliary graph – order t=(1,2,3)



G initial SGG



$G[t]$ with $t = (1, 2, 3)$ total order

The auxilliary graph

To a graph G and an order t on the k random nodes associate $G[t]$:

- the set of sinks, *max*-nodes and *ran*-nodes remain the same as in G;
- For every $1 \leq i \leq k$, add a *min*-node denoted i to $G[t]$, which we call *control node* and add an arc (i, r_i) ;
- For every $(i, j) \in t$, $i \neq j$, add the arc (i, j) to $G[t]$;
- For every arc $(x, r_i) \in A$, remove this arc and add an arc (x, i).

Lemma

(i) *optimal values of control nodes* $i \in [1, k]$ *in* $G[t]$ *are nondecreasing along* t *;*

(ii) *the game* $G[t]$ *can be solved in polynomial time.*

(iii) *for the "optimal" order* t*, optimal strategies in* $G[t]$ *coincide with optimal strategies* G *;*

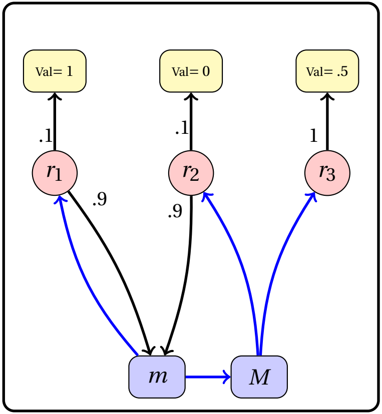
Main algorithm : iterative version

1. Prior to the execution of the algorithm, choose randomly and uniformly an order Θ on the set of all $\frac{k(k-1)}{2}$ unordered pairs of control nodes.
2. *Pivot selection rule* and *pivot operation* on orders that yields an order improvement algorithm.

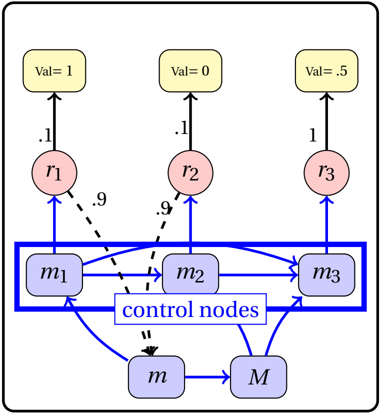
Theorem

The Algorithm computes optimal order in at most $e^{\sqrt{2}\cdot k}$ *expected steps.*

A run of our algorithm

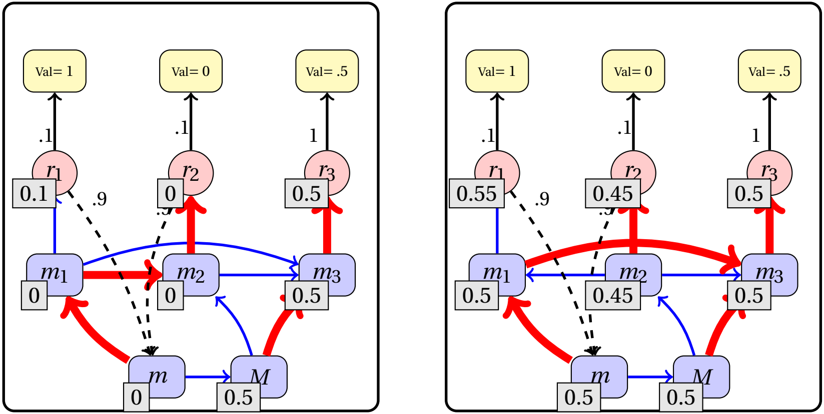


G initial SGG



$G[t]$ with $t = (1,2,3)$ total order

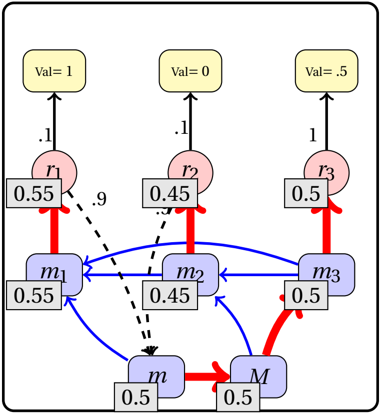
order Θ on pairs : $\{1,2\},\{1,3\},\{2,3\}$



order : 1,2,3
Value intervals : [1,2] [3]

order : 2,1,3
Value intervals : [2], [1,3]

order Θ on pairs : $\{1,2\},\{1,3\},\{2,3\}$



order : 2,3,1

Value intervals : [2] [3] [1]

Optimal order !