



**OPPA European Social Fund  
Prague & EU: We invest in your future.**

---

# A4M33MAS - Multiagent Systems

## Distributed Constraint Optimization

Michal Pechoucek & Michal Jakob  
Department of Computer Science  
Czech Technical University in Prague



In parts based on Multi-agent Constraint Programming, Boi Faltings, Laboratoire d'Intelligence Artificielle, EPFL

# Multiagent Constraint Optimization (DCOP)

Given  $\langle X, D, C, A \rangle$  where:

- $X = \{x_1, \dots, x_n\}$  is a set of  $n$  variables.
- $D = \{d_1, \dots, d_n\}$  is a set of  $n$  domains.
- $C = \{c_1, \dots, c_m\}$  is a set of  $m$  constraints.
- $A = \{a_1, \dots, a_n\}$  is a set of  $n$  agents, not necessarily all different.

Find solution =  $(x_1 = v_1 \in d_1, \dots, x_n = v_n \in d_n)$  such that for all the overall cost of the assignment is minimized

$$\text{Cost}(\{v_1, \dots, v_n\}) = \sum_{\forall c_i \in C} c_i(\{v_1, \dots, v_n\})$$

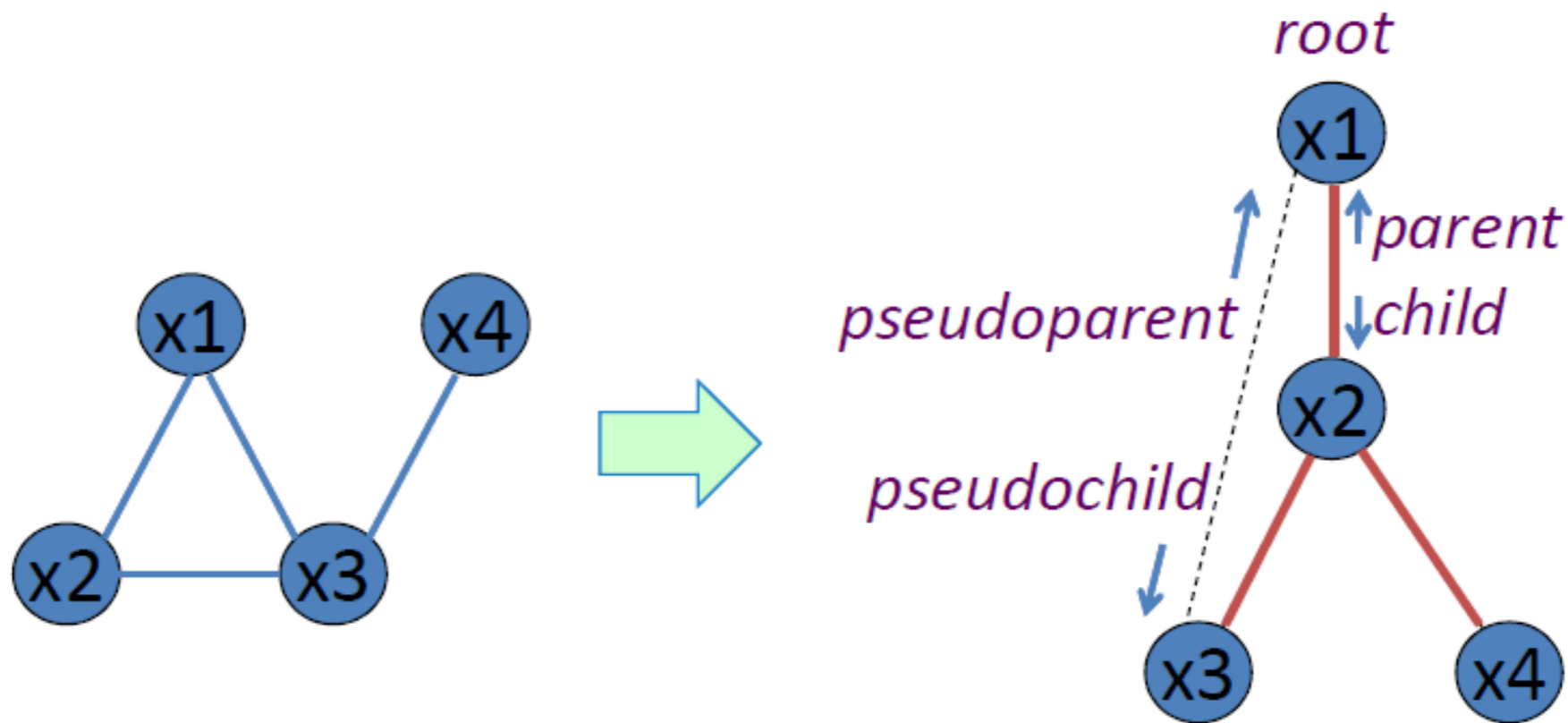
$C$  = represented as a list of **cost** functions on  $1 \dots n$  variables in  $X$  and their values from  $D$ , so that  $\mathcal{P}(X, D) \rightarrow \mathbf{R}$

# ABT for DCOP

- Nogoods give lower bounds on costs.
- Compute total cost of all lower priority agents by summing nogoods.
- Nogood tags must exactly cover all lower-priority variables, otherwise some variables are not counted or counted multiple times.
- If we can prevent this from happening, then ABT works fine for optimization as well.

# ADOPT

- ADOPT assumes that agents are arranged in a DFS tree:
  - constraint graph rooted graph (select a node as root)
  - some links form a tree / others are back edges
  - two constrained nodes must be in the same path to the root by tree links (same branch)
- Every graph admits a DFS tree



# ADOPT: Description

- Asynchronous algorithm
- Each time an agent receives a message:
  - Processes it (the agent may take a new value)
  - Sends VALUE messages to its children and pseudochildren
  - Sends a COST message to its parent
- **View**: set of variable value pairs (as ABT agent view) of ancestor agents, in the same branch. Current context:
  - Updated by each VALUE message

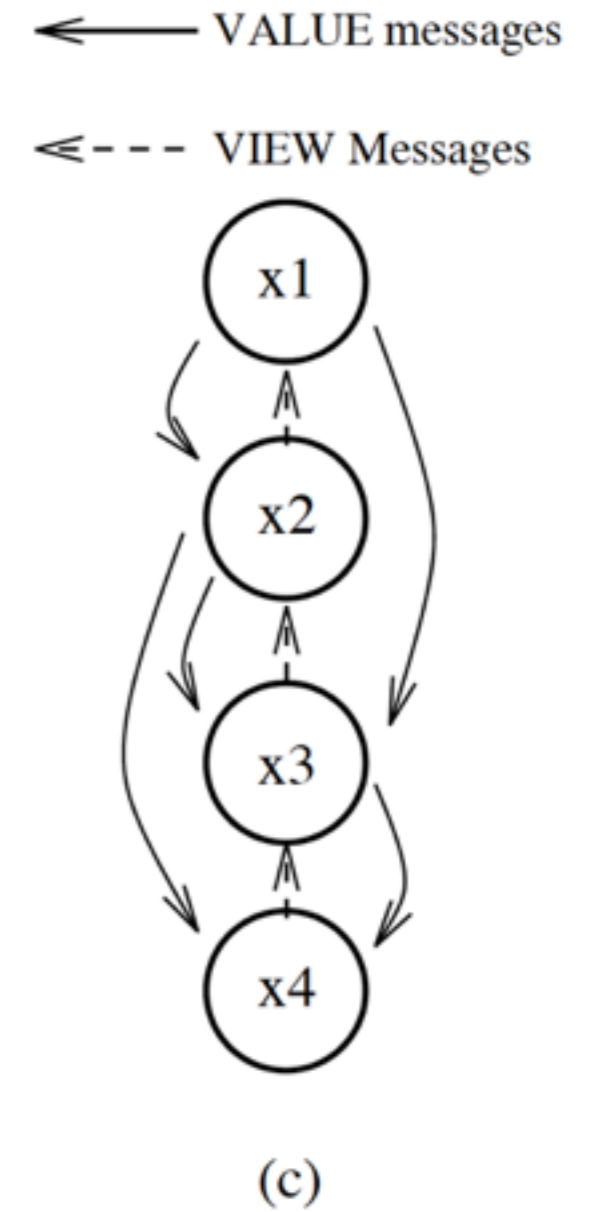
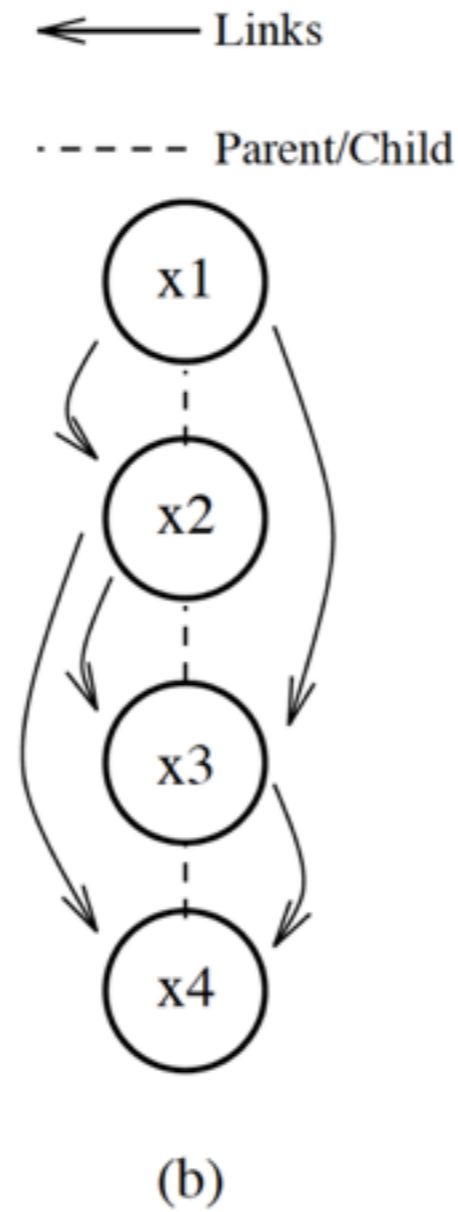
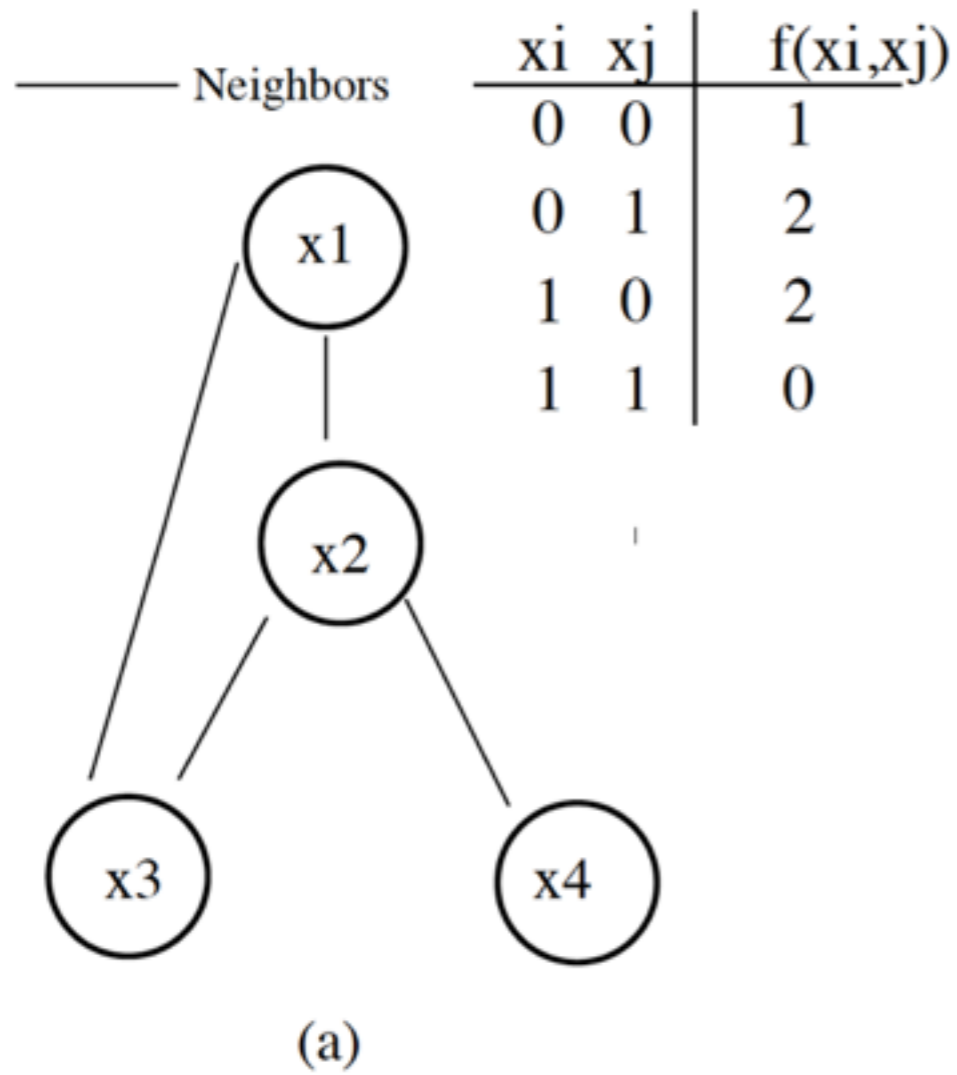
# ADOPT: Description

**Definition:** The *local cost*  $\delta$  incurred at  $x_i$ , wrt to a given view  $vw$  is defined as

$$\delta(x_i, vw) = \sum_{x_j \in V} f_{ij}(d_i, d_j) , \text{ where } x_i \leftarrow d_i, \\ x_j \leftarrow d_j \text{ in } vw$$

**Definition:** A *view* is a set of variable/value pairs of the form  $\{(x_i, d_i), (x_j, d_j) \dots\}$ . A variable can appear in a view no more than once. Two views are *compatible* if they do not disagree on any variable assignment.

# ADOPT: Example





# ADOPT: Messages

- **value**(parent  $\rightarrow$  children & pseudochildren, value):  
parent informs descendants that it has taken value a
- **view**(child  $\rightarrow$  parent, cost, view):  
child informs parent of the best cost of its assignment; attached context to detect obsolescence;

# Simple-ADOPT Algorithm

01

**Initialize:**  $Currentvw \leftarrow \{\}$ ;  $d_i \leftarrow \text{null}$ ;

$\forall d \in D_i :$

$c(d) \leftarrow 0$

hill\_climb;

# Simple-ADOPT Algorithm

**Initialize:**  $Currentvw \leftarrow \{\}$ ;  $d_i \leftarrow \text{null}$ ;

$\forall d \in D_i :$

$c(d) \leftarrow 0$

hill\_climb;

**procedure hill\_climb**

$\forall d \in D_i :$

*#  $e(d)$  is  $x_i$ 's estimate of cost if it chooses  $d$*

$e(d) \leftarrow \delta(x_i, Currentvw \cup \{(x_i, d)\}) + c(d)$ ;

choose  $d$  that minimizes  $e(d)$

prefer current value  $d_i$  for tie;

$d_i \leftarrow d$ ;

SEND (VALUE,  $(x_i, d_i)$ ) to all linked descendents;

SEND (VIEW,  $Currentvw, e(d_i)$ ) to

parent;

# Simple-ADOPT Algorithm

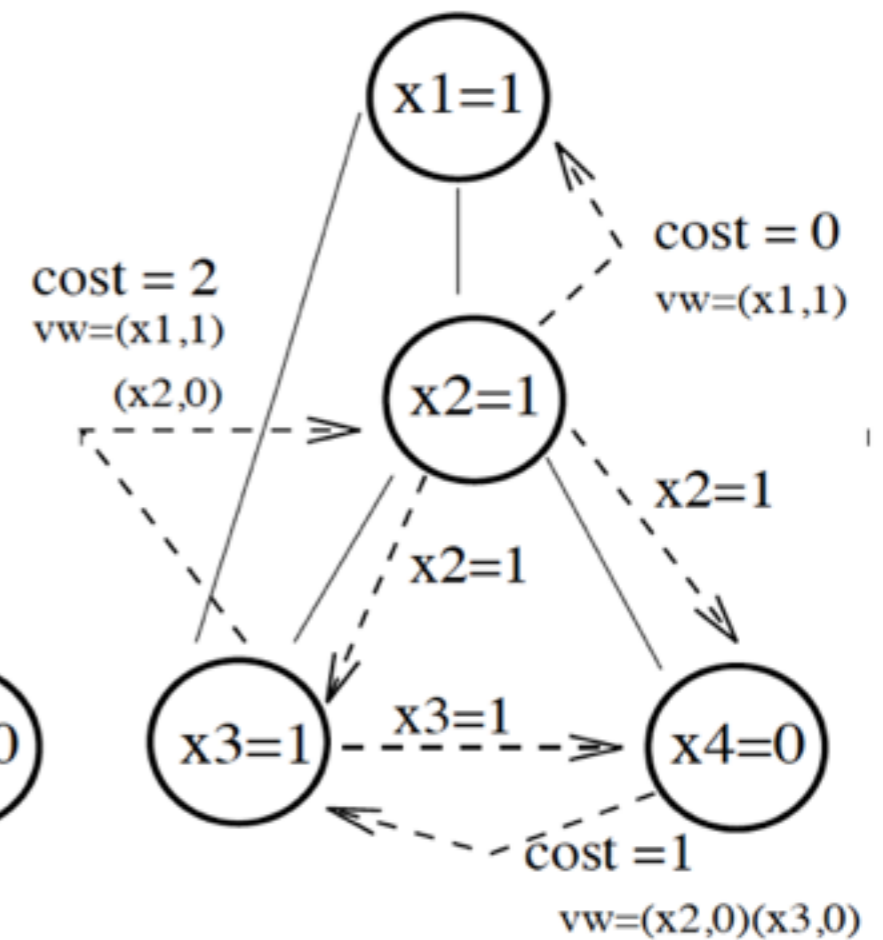
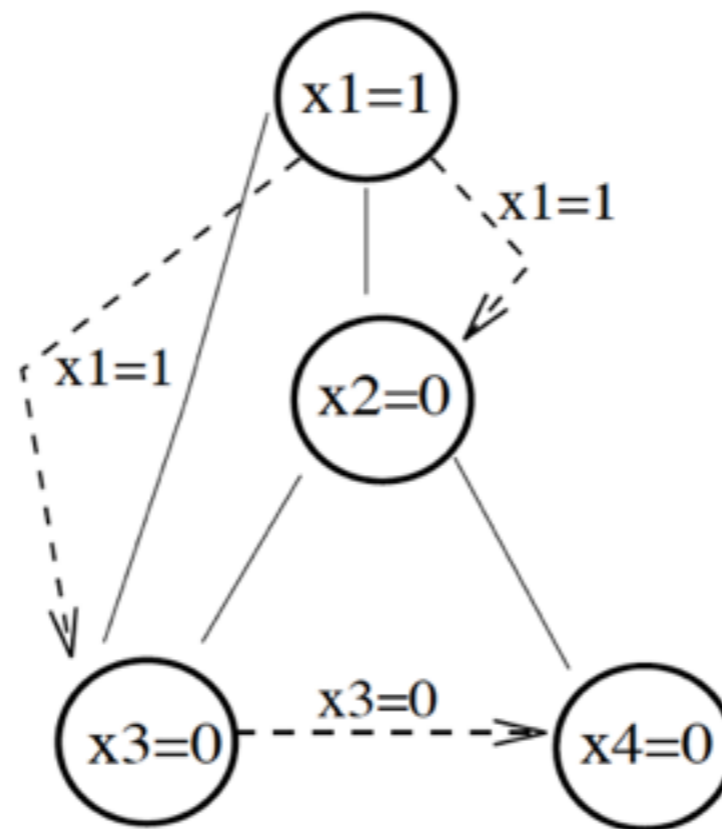
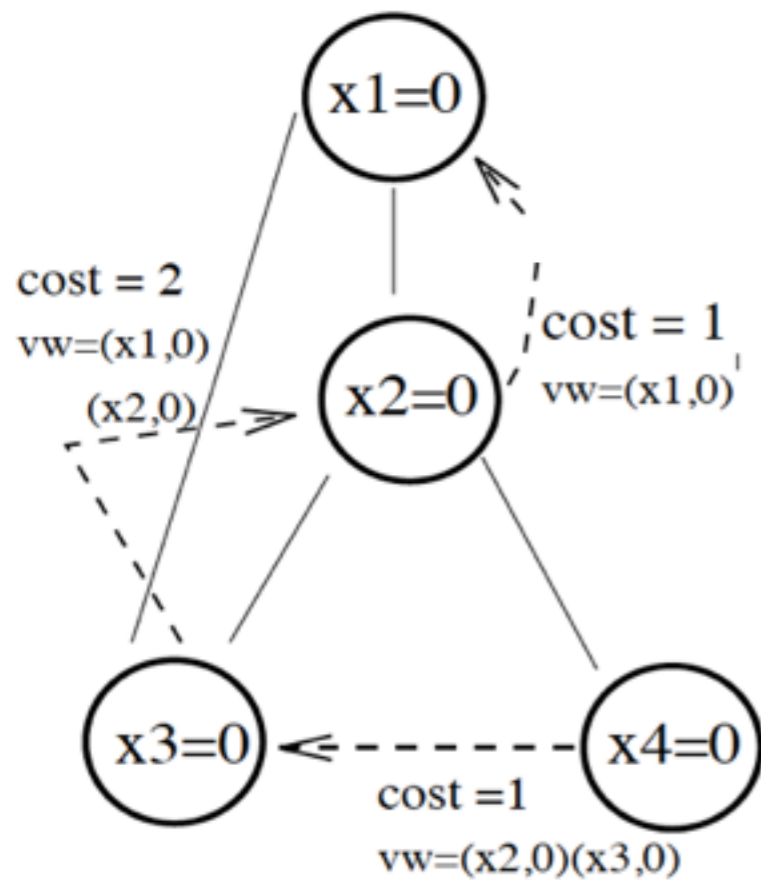
01

**when received (VALUE,  $(x_j, d_j)$ )**  
  add  $(x_j, d_j)$  to *Currentvw*;  
  # context change  
  if *Currentvw* changed then  
     $\forall d \in D_i :$   
       $c(d) \leftarrow 0$   
  end if;  
  hill\_climb;

**when received (VIEW,  $vw, cost$ )**  
   $d \leftarrow$  value of  $x_i$  in  $vw$   
  if  $vw$  is compatible with  
  *Currentvw*  $\cup \{(x_i, d)\}$  then  
     $c(d) \leftarrow \max(c(d), cost);$   
    if  $c(d)$  changed then  
      hill\_climb;  
    end if;  
  end if;

# Simple-ADOPT: Example

$x_i$	$x_j$	$f(x_i, x_j)$
0	0	1
0	1	2
1	0	2
1	1	0



# Simple-ADOPT: Example

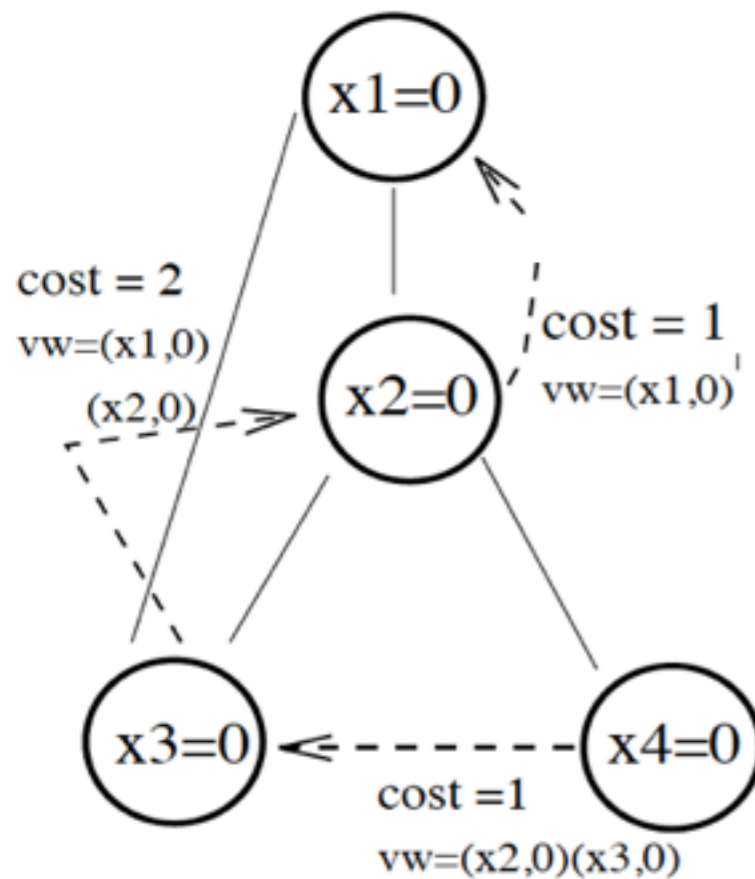
$x_i$	$x_j$	$f(x_i, x_j)$
0	0	1
0	1	2
1	0	2
1	1	0

**Initialize:**  $Currentvw \leftarrow \{\}$ ;  $d_i \leftarrow \text{null}$ ;

$\forall d \in D_i$  :  
 $c(d) \leftarrow 0$   
 hill\_climb;

**procedure hill\_climb**

$\forall d \in D_i$  :  
 #  $e(d)$  is  $x_i$ 's estimate of cost if it chooses  $d$   
 $e(d) \leftarrow \delta(x_i, Currentvw \cup \{(x_i, d)\}) + c(d)$ ;  
 choose  $d$  that minimizes  $e(d)$   
 prefer current value  $d_i$  for tie;  
 $d_i \leftarrow d$ ;  
 SEND (VALUE,  $(x_i, d_i)$ ) to all linked descendents;  
 SEND (VIEW,  $Currentvw, e(d_i)$ ) to  
 parent;



# Simple-ADOPT: Example

$x_i$	$x_j$	$f(x_i, x_j)$
0	0	1
0	1	2
1	0	2
1	1	0

**Initialize:**  $Currentvw \leftarrow \{\}$ ;  $d_i \leftarrow \text{null}$ ;

$\forall d \in D_i$  :  
 $c(d) \leftarrow 0$   
 hill\_climb;

**procedure hill\_climb**

$\forall d \in D_i$  :

#  $e(d)$  is  $x_i$ 's estimate of cost if it chooses  $d$

$e(d) \leftarrow \delta(x_i, Currentvw \cup \{(x_i, d)\}) + c(d)$ ;

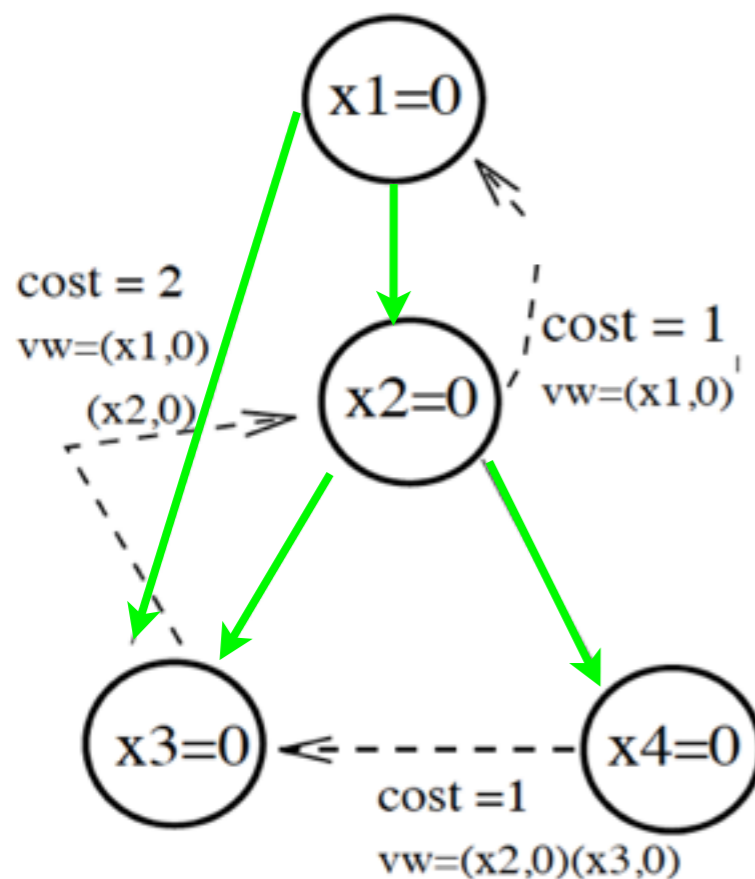
choose  $d$  that minimizes  $e(d)$

prefer current value  $d_i$  for tie;

$d_i \leftarrow d$ ;

SEND (VALUE,  $(x_i, d_i)$ ) to all linked descendents;

SEND (VIEW,  $Currentvw, e(d_i)$ ) to  
 parent;



# Simple-ADOPT: Example

$x_i$	$x_j$	$f(x_i, x_j)$
0	0	1
0	1	2
1	0	2
1	1	0

**Initialize:**  $Currentvw \leftarrow \{\}$ ;  $d_i \leftarrow \text{null}$ ;

$\forall d \in D_i$  :  
 $c(d) \leftarrow 0$   
 hill\_climb;

**procedure hill\_climb**

$\forall d \in D_i$  :

#  $e(d)$  is  $x_i$ 's estimate of cost if it chooses  $d$

$e(d) \leftarrow \delta(x_i, Currentvw \cup \{(x_i, d)\}) + c(d)$ ;

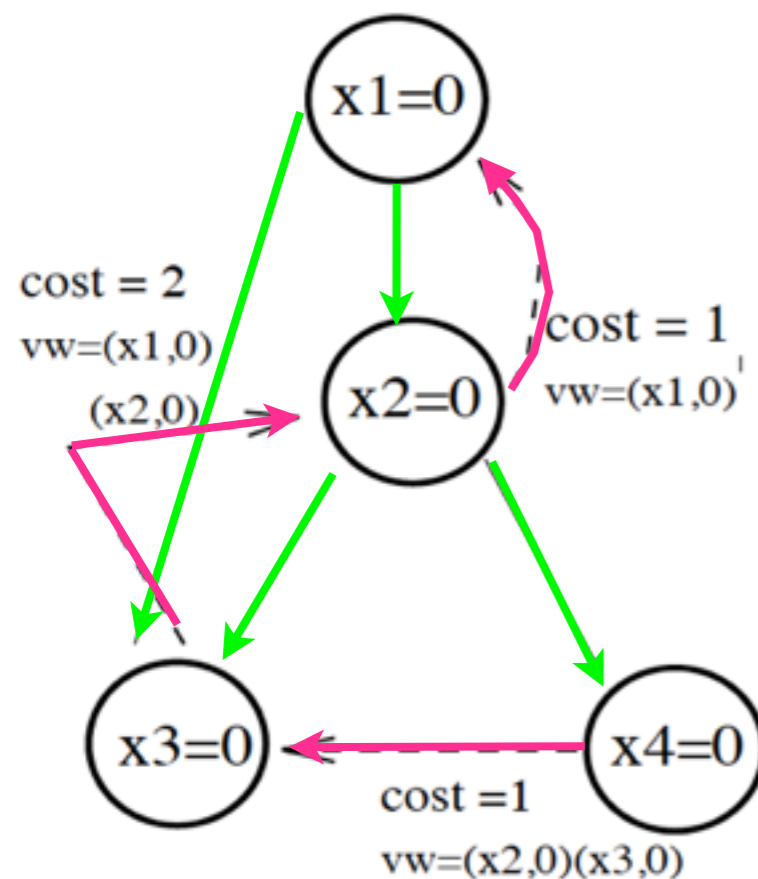
choose  $d$  that minimizes  $e(d)$

prefer current value  $d_i$  for tie;

$d_i \leftarrow d$ ;

SEND (VALUE,  $(x_i, d_i)$ ) to all linked descendents;

SEND (VIEW,  $Currentvw, e(d_i)$ ) to  
 parent;

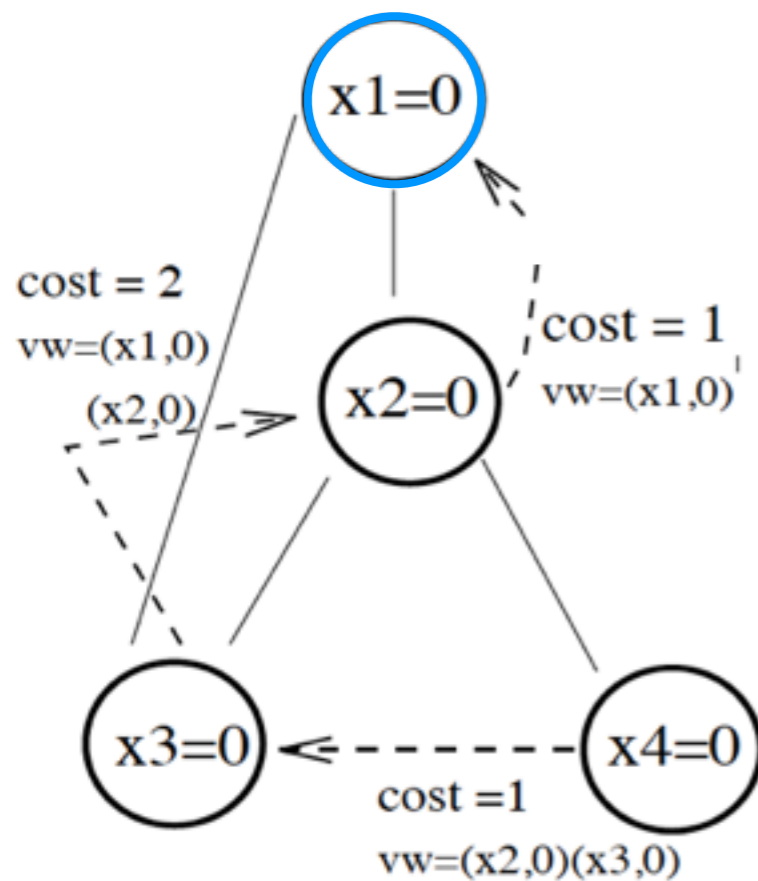




# Simple-ADOPT: Example

$x_i$	$x_j$	$f(x_i, x_j)$
0	0	1
0	1	2
1	0	2
1	1	0

$$cost(x_i, vw) = \sum_{x_j \in V} f_{ij}(d_i, d_j), \text{ where } x_i \leftarrow d_i, \\ x_j \leftarrow d_j \text{ in } vw$$



**when received (VIEW, vw, cost)**

$d \leftarrow$  value of  $x_i$  in  $vw$

if  $vw$  is compatible with

$Currentvw \cup \{(x_i, d)\}$  then

$c(d) \leftarrow \max(c(d), cost);$

if  $c(d)$  changed then

    hill\_climb;

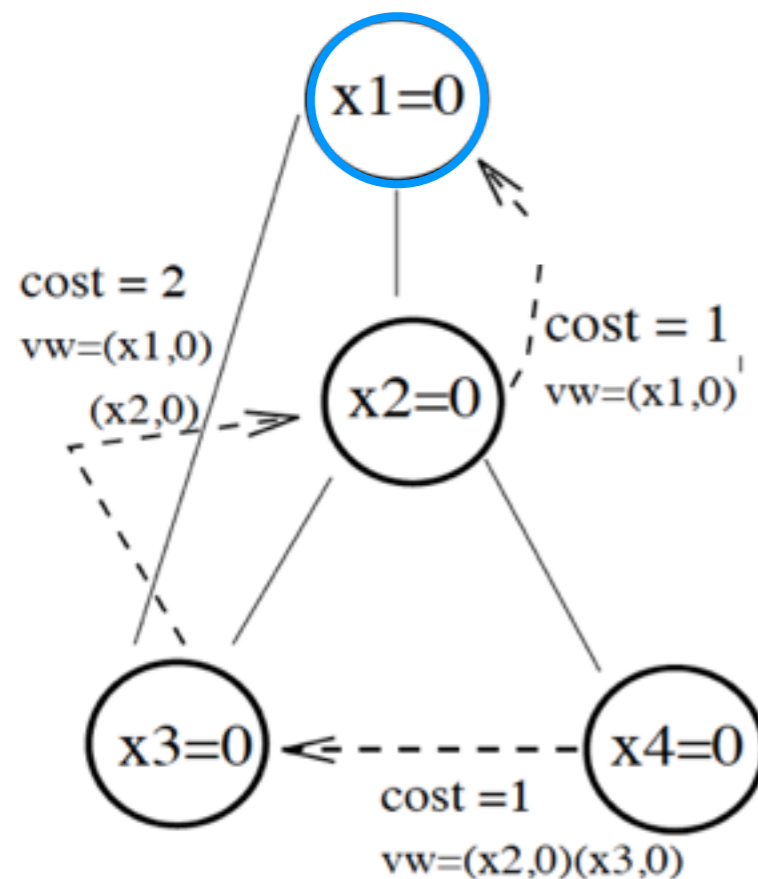
end if;

end if;

# Simple-ADOPT: Example

$x_i$	$x_j$	$f(x_i, x_j)$
0	0	1
0	1	2
1	0	2
1	1	0

$$cost(x_i, vw) = \sum_{x_j \in V} f_{ij}(d_i, d_j), \text{ where } x_i \leftarrow d_i, \\ x_j \leftarrow d_j \text{ in } vw$$



## procedure hill\_climb

$\forall d \in D_i:$

#  $e(d)$  is  $x_i$ 's estimate of cost if it chooses  $d$

$e(d) \leftarrow \delta(x_i, Currentvw \cup \{(x_i, d)\}) + c(d);$

choose  $d$  that minimizes  $e(d)$

prefer current value  $d_i$  for tie;

$d_i \leftarrow d;$

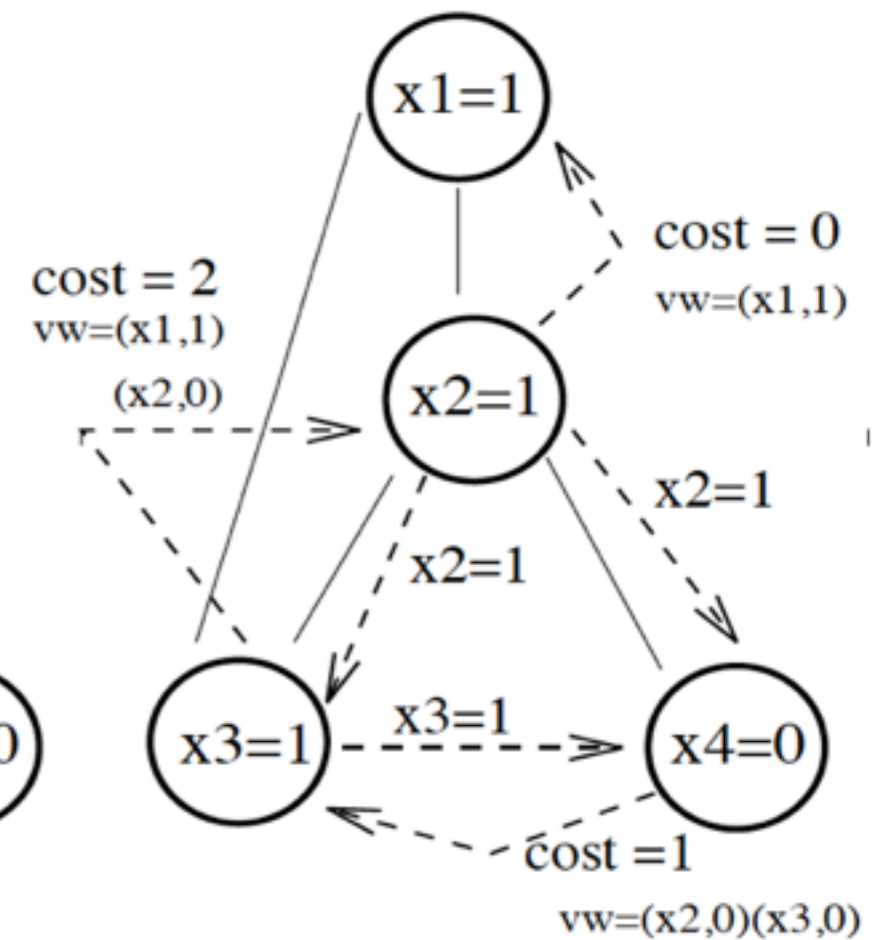
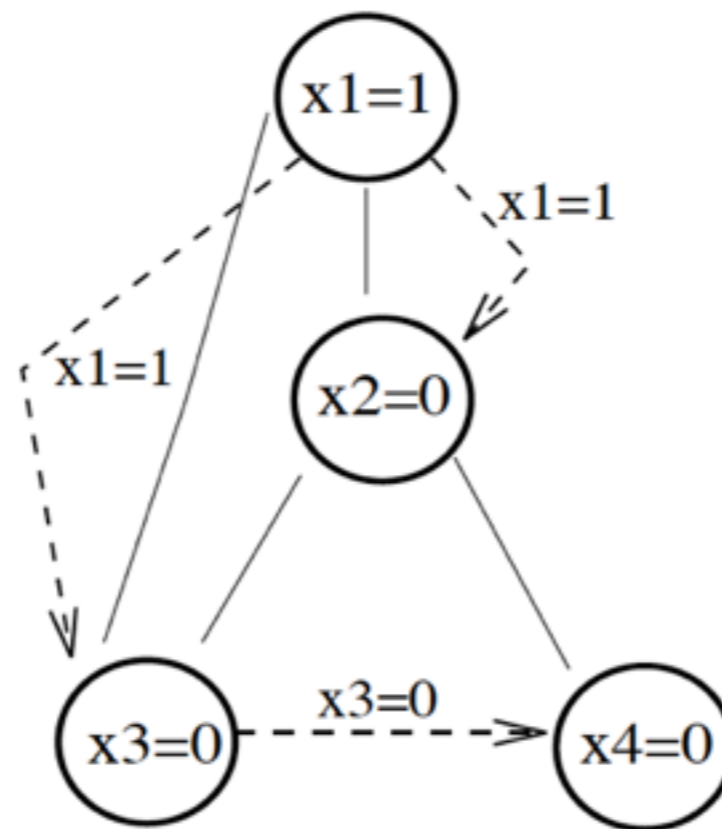
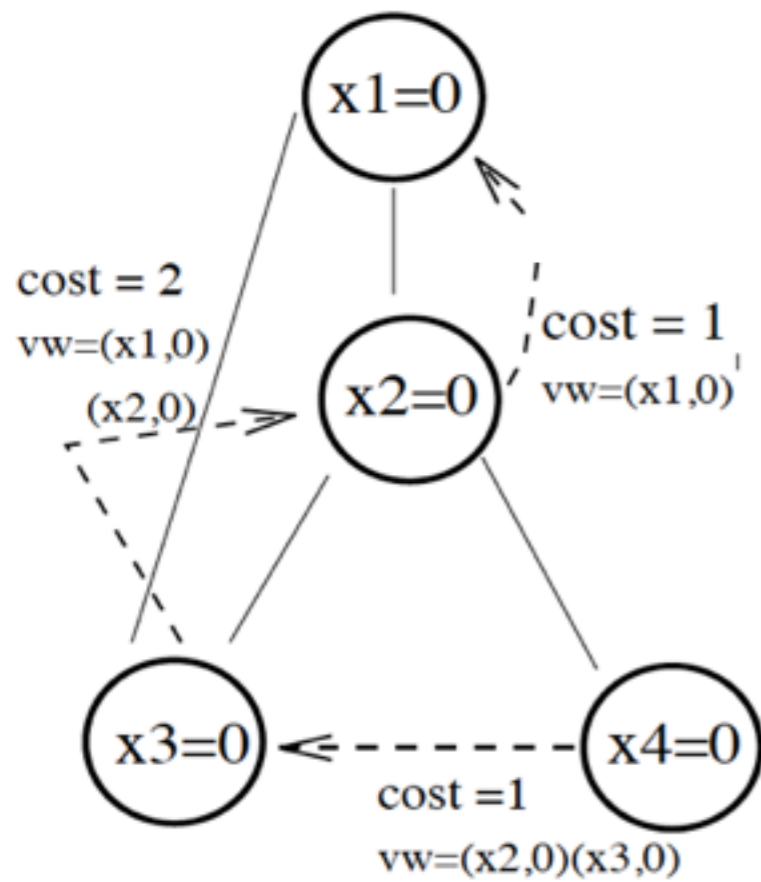
SEND (VALUE,  $(x_i, d_i)$ ) to all linked descendents;

SEND (VIEW,  $Currentvw, e(d_i)$ ) to

parent;

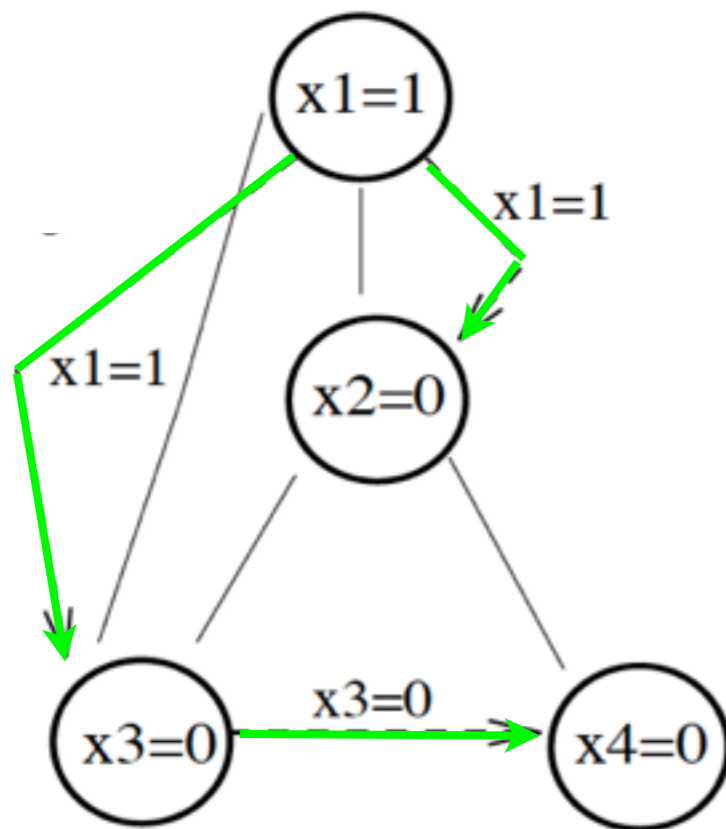
# Simple-ADOPT: Example

$x_i$	$x_j$	$f(x_i, x_j)$
0	0	1
0	1	2
1	0	2
1	1	0



# Simple-ADOPT: Example

$x_i$	$x_j$	$f(x_i, x_j)$
0	0	1
0	1	2
1	0	2
1	1	0



**when received** (**VALUE**,  $(x_j, d_j)$ )

add  $(x_j, d_j)$  to *Currentvw*;  
# context change

if *Currentvw* changed then

$\forall d \in D_i :$

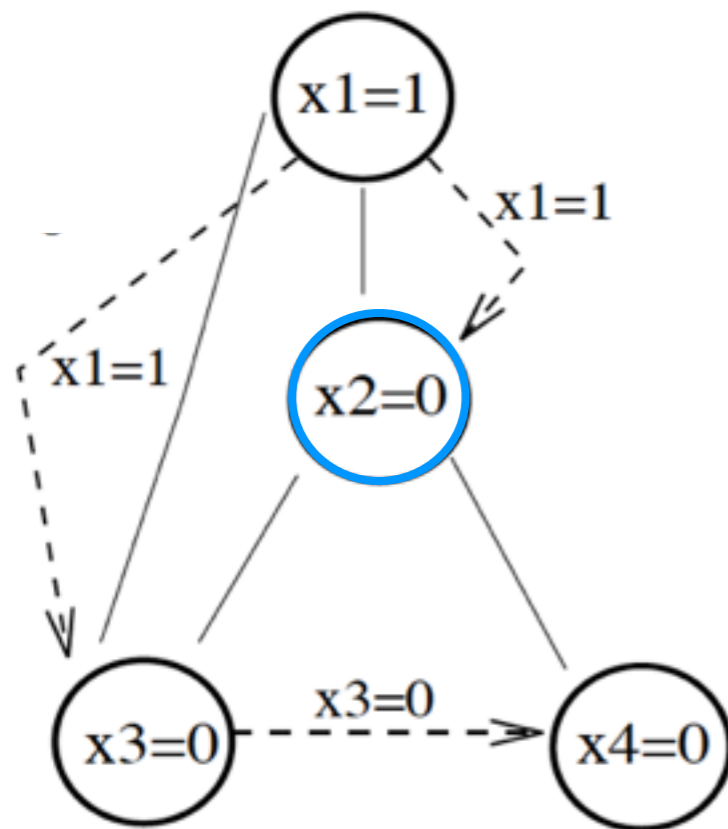
$c(d) \leftarrow 0$

end if;

hill\_climb;

# Simple-ADOPT: Example

$x_i$	$x_j$	$f(x_i, x_j)$
0	0	1
0	1	2
1	0	2
1	1	0



## procedure hill\_climb

$\forall d \in D_i$ :

#  $e(d)$  is  $x_i$ 's estimate of cost if it chooses  $d$

$e(d) \leftarrow \delta(x_i, Currentvw \cup \{(x_i, d)\}) + c(d)$

choose  $d$  that minimizes  $e(d)$

prefer current value  $d_i$  for tie;

$d_i \leftarrow d$ ;

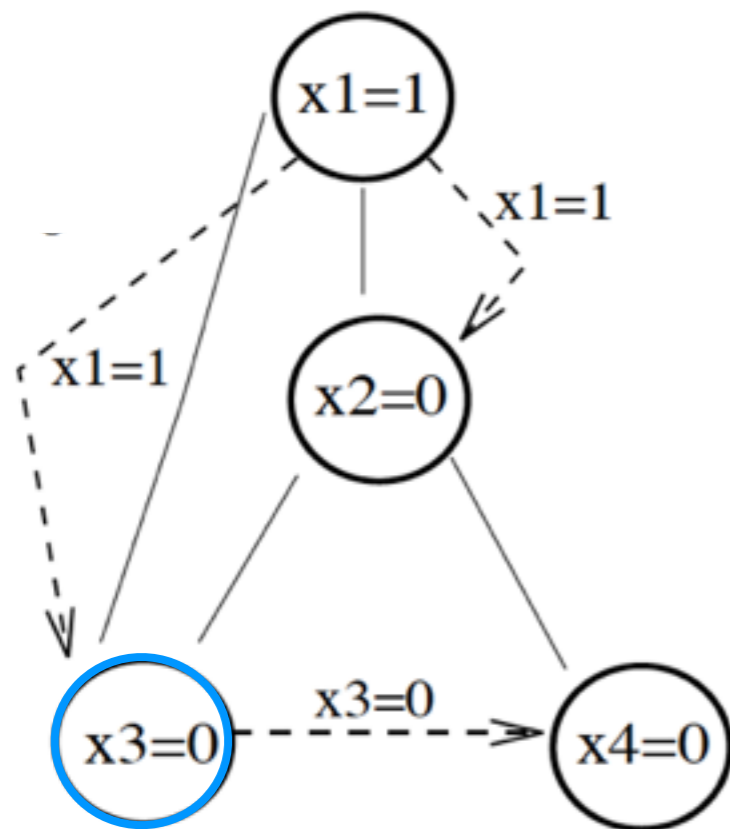
SEND (VALUE,  $(x_i, d_i)$ ) to all linked descendents

SEND (VIEW,  $Currentvw, e(d_i)$ ) to parent;

$$\delta(x_i, vw) = \sum_{x_j \in V} f_{ij}(d_i, d_j), \text{ where } x_i \leftarrow d_i, \\ x_j \leftarrow d_j \text{ in } vw$$

# Simple-ADOPT: Example

$x_i$	$x_j$	$f(x_i, x_j)$
0	0	1
0	1	2
1	0	2
1	1	0



## procedure hill\_climb

$\forall d \in D_i$ :

#  $e(d)$  is  $x_i$ 's estimate of cost if it chooses  $d$

$e(d) \leftarrow \delta(x_i, Currentvw \cup \{(x_i, d)\}) + c(d)$

choose  $d$  that minimizes  $e(d)$

prefer current value  $d_i$  for tie;

$d_i \leftarrow d$ ;

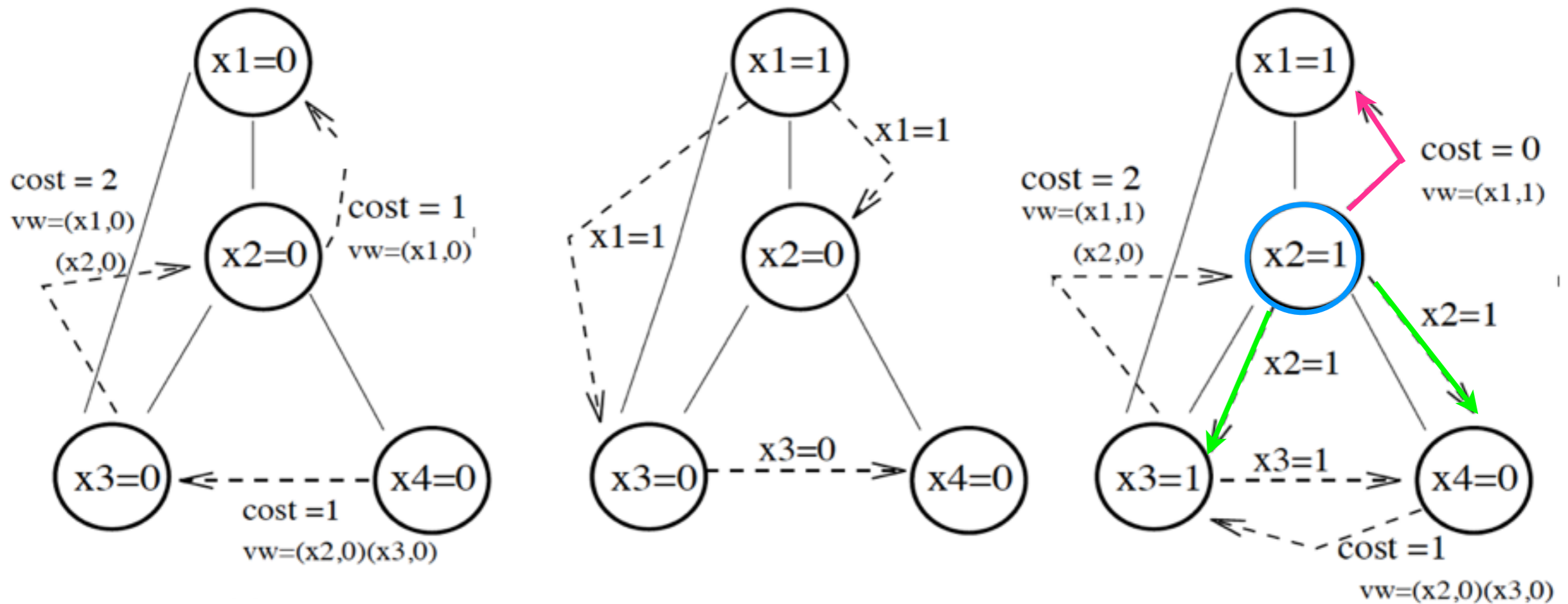
SEND (VALUE,  $(x_i, d_i)$ ) to all linked descendents

SEND (VIEW,  $Currentvw, e(d_i)$ ) to parent;

$$\delta(x_i, vw) = \sum_{x_j \in V} f_{ij}(d_i, d_j), \text{ where } x_i \leftarrow d_i, \\ x_j \leftarrow d_j \text{ in } vw$$

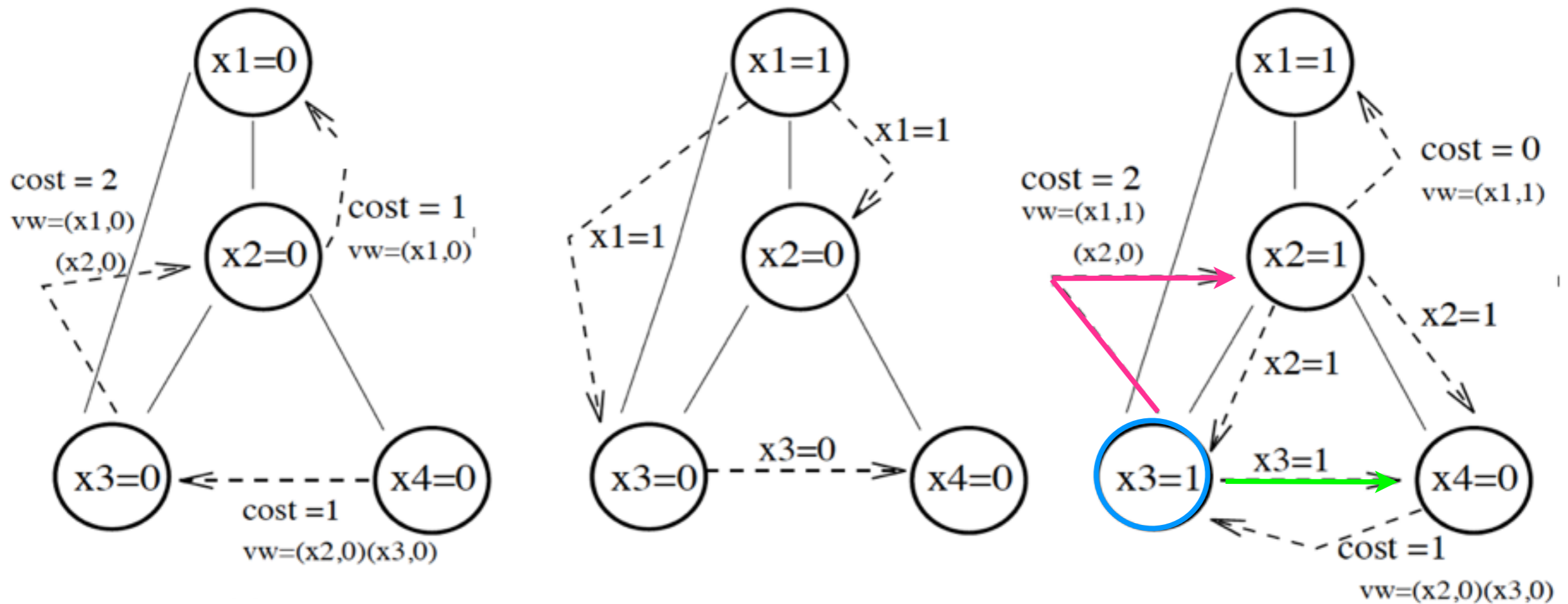
# Simple-ADOPT: Example

$x_i$	$x_j$	$f(x_i, x_j)$
0	0	1
0	1	2
1	0	2
1	1	0



# Simple-ADOPT: Example

$x_i$	$x_j$	$f(x_i, x_j)$
0	0	1
0	1	2
1	0	2
1	1	0

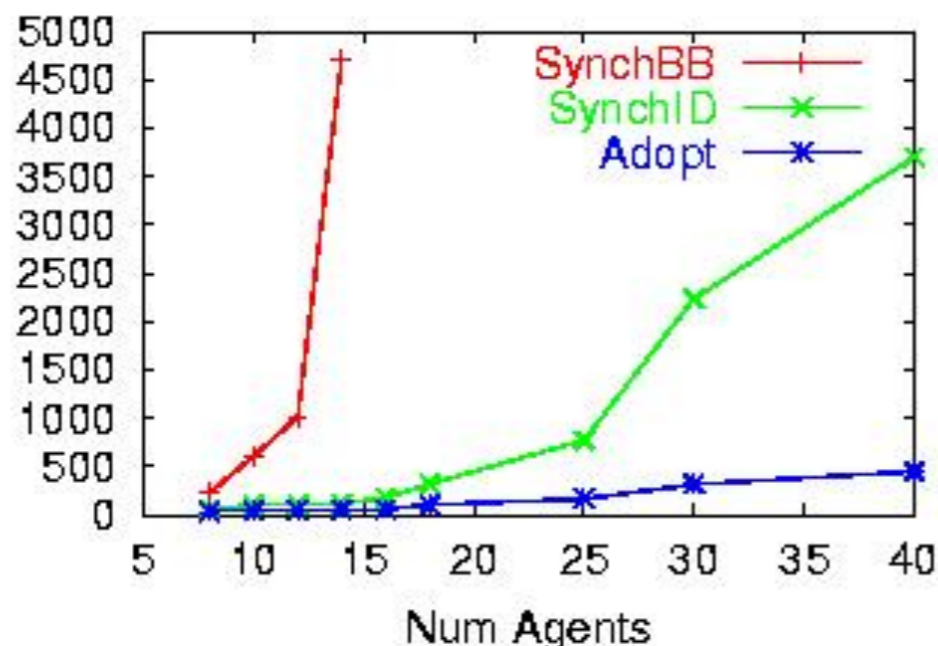




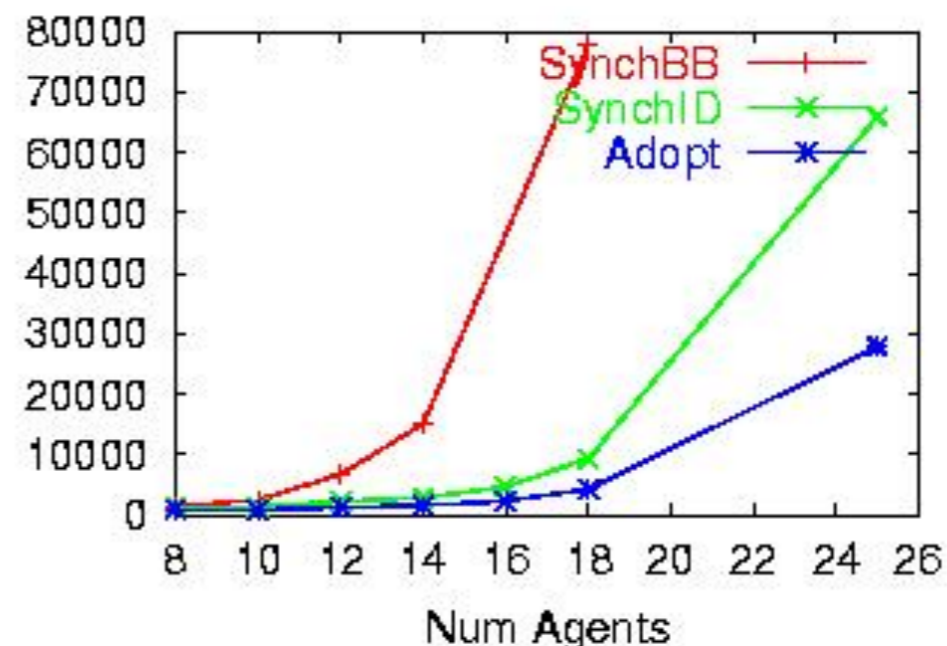
# ADOPT: Properties

- For finite DCOPs with binary non-negative constraints, ADOPT is guaranteed to terminate with the globally optimal solution.
- An ADOPT agent takes the value with minimum cost:
  - Best-first search with eager behavior:
  - Agents may constantly change value
- Graph coloring benchmark:

Avg. number of cycles,  
link density = 2



Avg. number of cycles,  
link density = 3



# ADOPT: Key Ideas

- Optimal, asynchronous algorithm for DCOP
  - polynomial space at each agent
- Weak Backtracking
  - lower bound based search method
  - Parallel search in independent subtrees
- Efficient reconstruction of abandoned solutions
  - backtrack thresholds to control backtracking
- Bounded error approximation
  - sub-optimal solutions faster
  - bound on worst-case performance

# Dynamic Programming Optimization Protocol (DPOP)

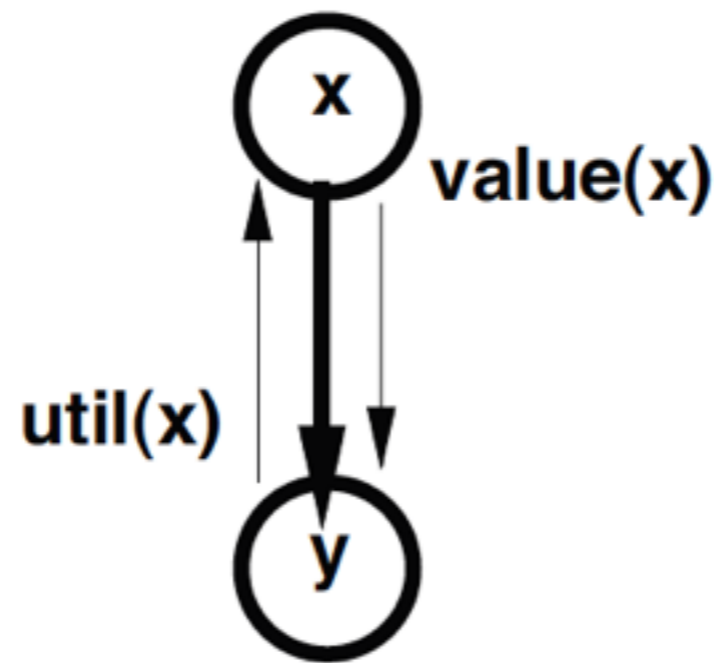
- Principle: replace variables by constraints.
  - Consider variable  $x$  having constraint with  $y$ .
  - For each value of  $x$ , there may be a consistent value of  $y$ .
- ⇒ replace  $y$  by a constraint on  $x$ :

*$x=v$  is allowed if there is a consistent value of  $y$ .*

- Optimization version:

*$utility(x=v) = utility(x=v, y=w)$ ;  $w =$  best possible value of  $y$  given  $x=v$ .*

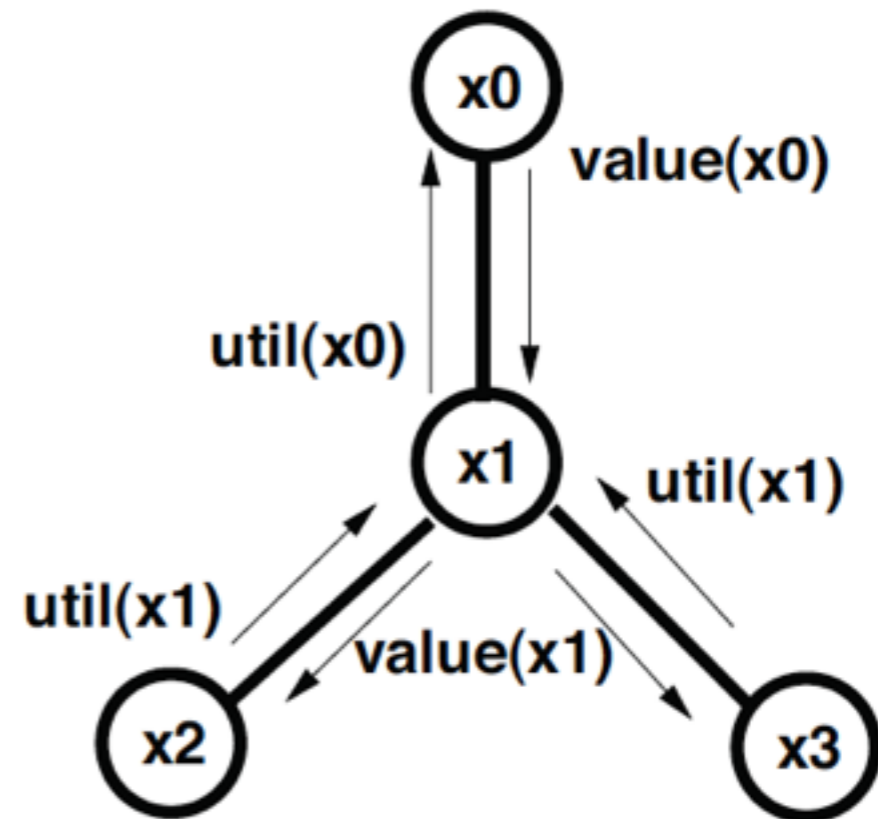
# Dynamic Programming Optimization Protocol (DPOP)



- $y$  sends constraint in  $util(x)$  message.
- ⇒  $x$  can decide (best) value locally.
- $x$  informs  $y$  of value using  $value(x)$  message.

# Dynamic Programming Optimization Protocol (DPOP)

- Rooted tree: every node has at most one parent
- Nodes send UTIL messages to their parents
- Best values of  $x_2, x_3 \Rightarrow$  unary constraint on  $x_1$
- $x_1$  sums up UTIL messages + own constraint  $\Rightarrow$  unary constraint on  $x_0$
- $x_0$  picks best value  $v(x_0)$ ; sends  $\text{value}(x_0=v(x_0)) \rightarrow x_1$
- $x_1$  picks best value given  $x_0$  and informs  $x_2, x_3$



# Dynamic Programming Optimization Protocol (DPOP)

$$c(x_0, x_3)$$

		$x_3$	
		$w$	$b$
$x_0$	$w$	3	0
	$b$	3	3

$$c(x_0, x_1)$$

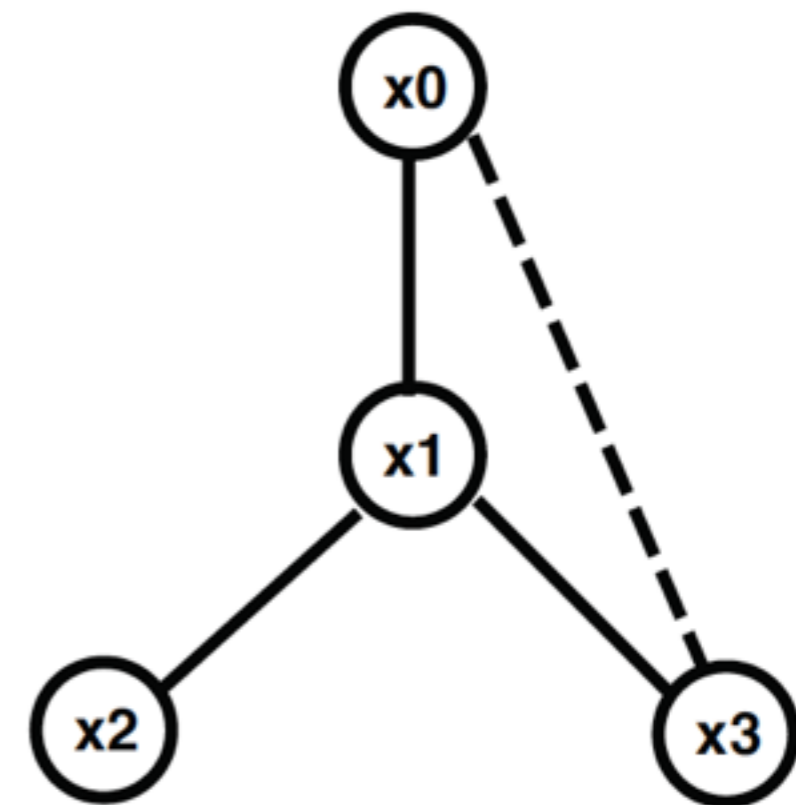
		$x_1$	
		$w$	$b$
$x_0$	$w$	1	0
	$b$	2	2

$$c(x_1, x_2)$$

		$x_2$	
		$w$	$b$
$x_1$	$w$	1	0
	$b$	0	1

$$c(x_1, x_3)$$

		$x_3$	
		$w$	$b$
$x_1$	$w$	2	0
	$b$	0	2



# Dynamic Programming Optimization Protocol (DPOP)

$$c(x_0, x_3)$$

		$x_3$	
		$w$	$b$
$x_0$	$w$	3	0
	$b$	3	3

$$c(x_0, x_1)$$

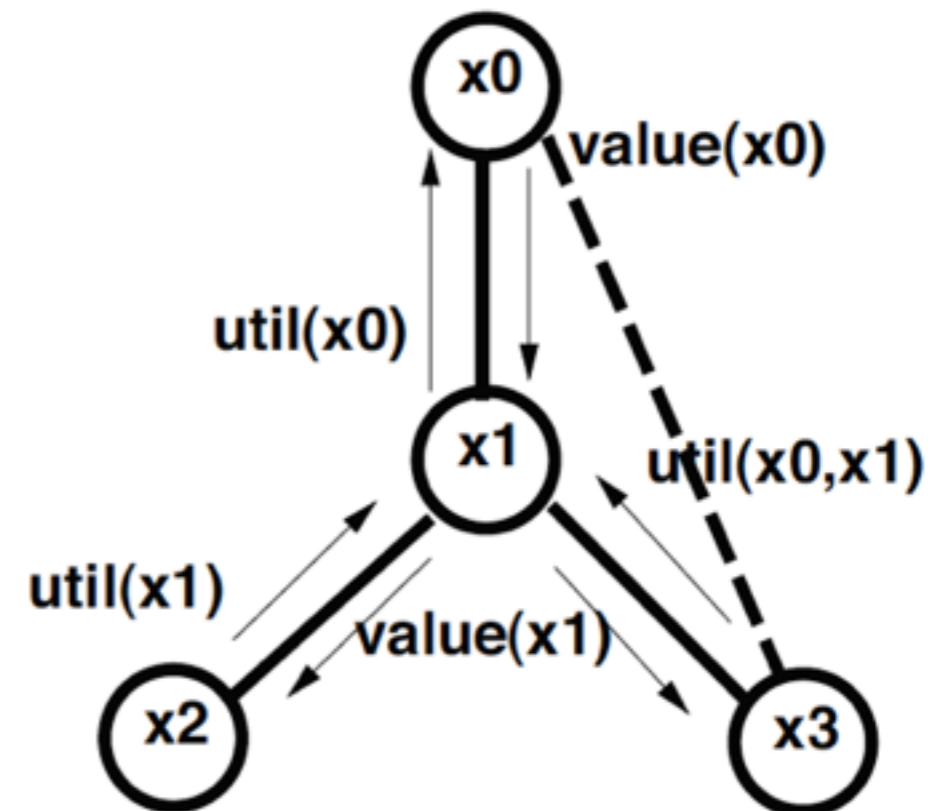
		$x_1$	
		$w$	$b$
$x_0$	$w$	1	0
	$b$	2	2

$$c(x_1, x_2)$$

		$x_2$	
		$w$	$b$
$x_1$	$w$	1	0
	$b$	0	1

$$c(x_1, x_3)$$

		$x_3$	
		$w$	$b$
$x_1$	$w$	2	0
	$b$	0	2



# Dynamic Programming Optimization Protocol (DPOP)

$$c(x_0, x_3)$$

		$x_3$	
$x_0$		$w$	$b$
$w$		3	0
$b$		3	3

$$c(x_0, x_1)$$

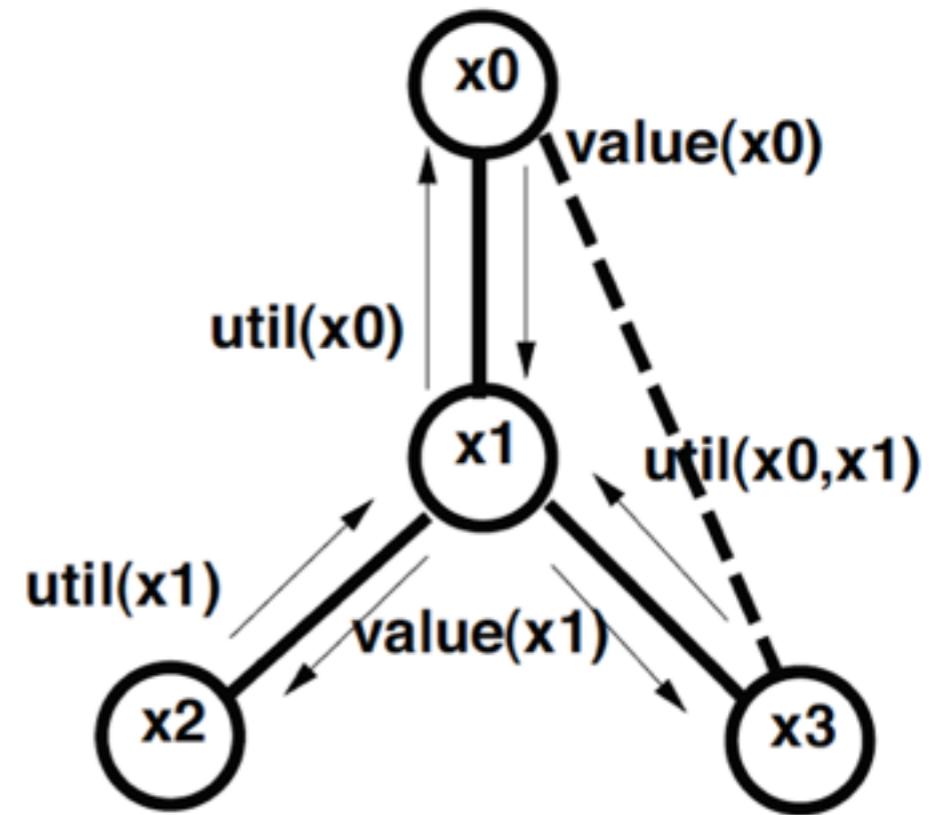
		$x_1$	
$x_0$		$w$	$b$
$w$		1	0
$b$		2	2

$$c(x_1, x_2)$$

		$x_2$	
$x_1$		$w$	$b$
$w$		1	0
$b$		0	1

$$c(x_1, x_3)$$

		$x_3$	
$x_1$		$w$	$b$
$w$		2	0
$b$		0	2



$$UTIL(x_1) = \frac{\begin{matrix} & x_1 \\ w & b \\ \hline 0 & 0 \end{matrix}}$$



# Dynamic Programming Optimization Protocol (DPOP)

$$c(x_0, x_3)$$

		$x_3$	
$x_0$		$w$	$b$
$w$		3	0
$b$		3	3

$$c(x_0, x_1)$$

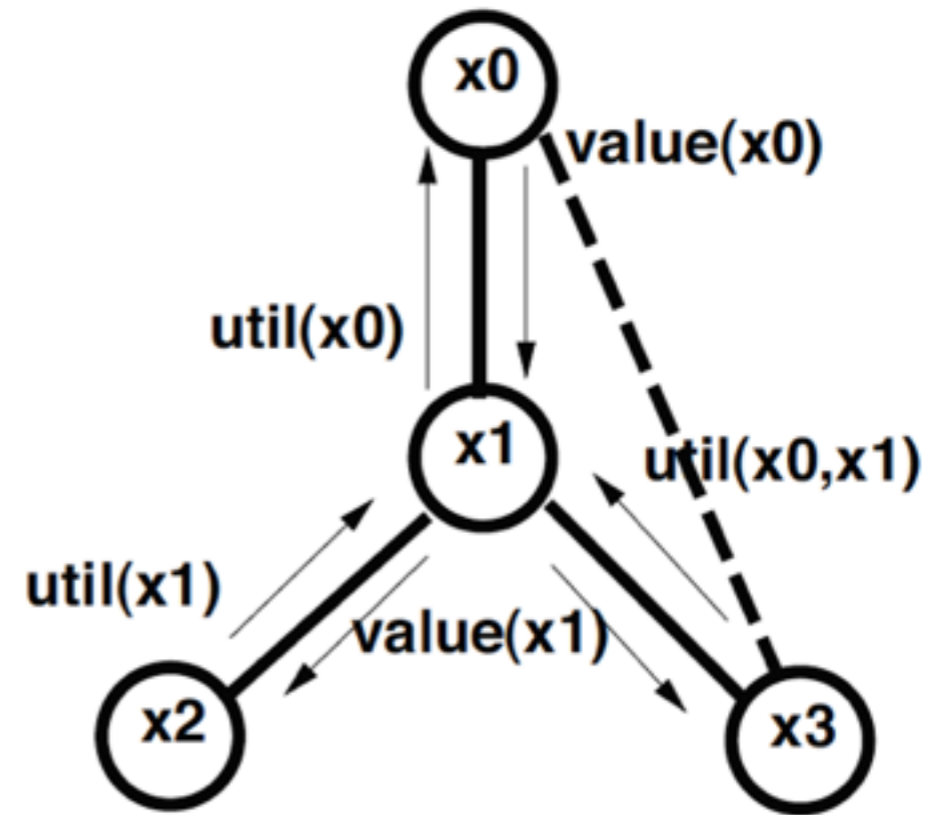
		$x_1$	
$x_0$		$w$	$b$
$w$		1	0
$b$		2	2

$$c(x_1, x_2)$$

		$x_2$	
$x_1$		$w$	$b$
$w$		1	0
$b$		0	1

$$c(x_1, x_3)$$

		$x_3$	
$x_1$		$w$	$b$
$w$		2	0
$b$		0	2



$$UTIL(x_1) = \frac{\begin{array}{c|cc} & x_1 & \\ \hline w & & \\ b & & \end{array}}{\begin{array}{c|cc} & & \\ \hline 0 & & 0 \end{array}}$$

$$UTIL(x_0, x_1) = x_0 \frac{\begin{array}{c|cc} & x_1 & \\ \hline w & & \\ b & & \end{array}}{\begin{array}{c|cc} & & \\ \hline 0 & & 2 \\ 3 & & 3 \end{array}}$$

# Dynamic Programming Optimization Protocol (DPOP)

$$c(x_0, x_3)$$

		$x_3$	
$x_0$		$w$	$b$
$w$		3	0
$b$		3	3

$$c(x_0, x_1)$$

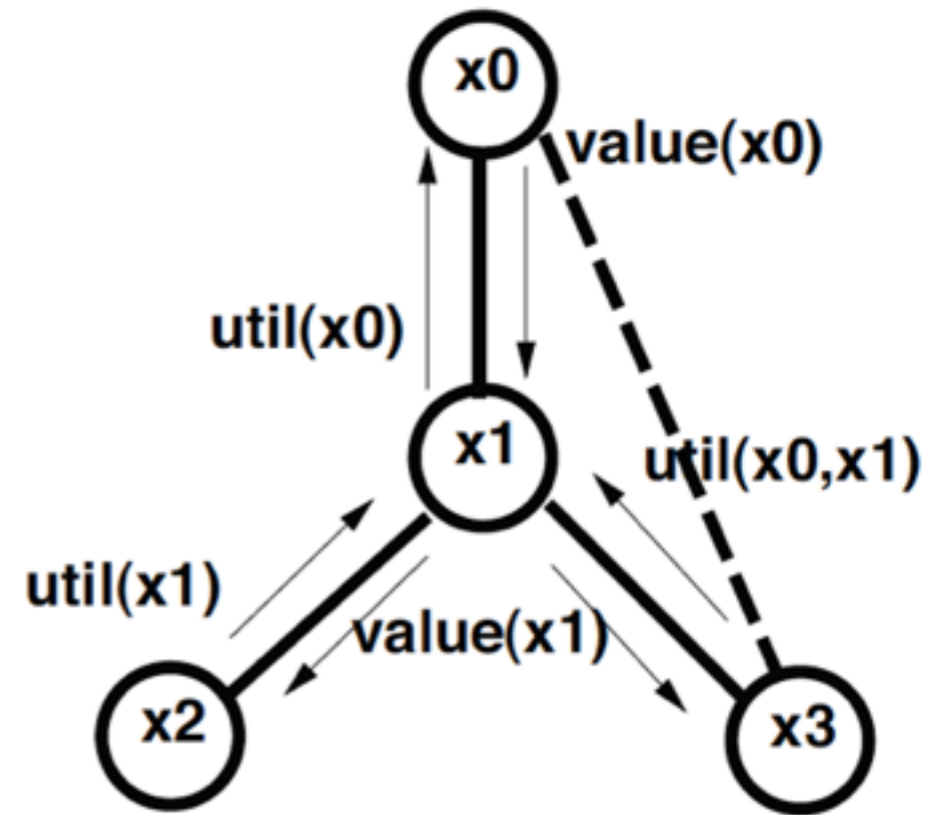
		$x_1$	
$x_0$		$w$	$b$
$w$		1	0
$b$		2	2

$$c(x_1, x_2)$$

		$x_2$	
$x_1$		$w$	$b$
$w$		1	0
$b$		0	1

$$c(x_1, x_3)$$

		$x_3$	
$x_1$		$w$	$b$
$w$		2	0
$b$		0	2



$$UTIL(x_1) = \frac{\begin{array}{c|cc} & x_1 & \\ \hline w & & b \\ b & 0 & 0 \end{array}}{0 \quad 0}$$

$$UTIL(x_0, x_1) = x_0 \frac{\begin{array}{c|cc} & x_1 & \\ \hline w & 0 & 2 \\ b & 3 & 3 \end{array}}{w \quad b}$$

$$UTIL(x_0) = \frac{\begin{array}{c|cc} & x_0 & \\ \hline w & & b \\ 1 & 1 & 3 \end{array}}{1 \quad 3}$$

# Dynamic Programming Optimization Protocol (DPOP)

$$c(x_0, x_3)$$

		$x_3$	
$x_0$		$w$	$b$
$w$		3	0
$b$		3	3

$$c(x_0, x_1)$$

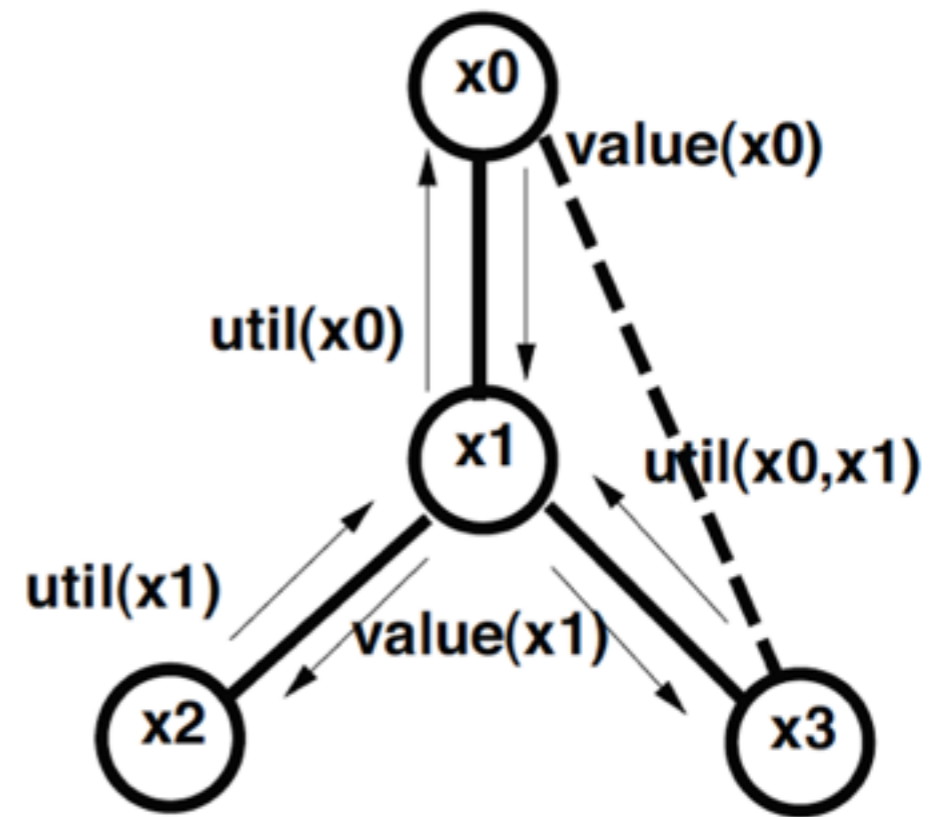
		$x_1$	
$x_0$		$w$	$b$
$w$		1	0
$b$		2	2

$$c(x_1, x_2)$$

		$x_2$	
$x_1$		$w$	$b$
$w$		1	0
$b$		0	1

$$c(x_1, x_3)$$

		$x_3$	
$x_1$		$w$	$b$
$w$		2	0
$b$		0	2



$$UTIL(x_1) = \frac{\begin{array}{c|cc} & x_1 & \\ \hline w & & b \\ b & 0 & 0 \end{array}}{0 \quad 0}$$

$$UTIL(x_0, x_1) = x_0 \frac{\begin{array}{c|cc} & x_1 & \\ \hline w & 0 & 2 \\ b & 3 & 3 \end{array}}$$

$$UTIL(x_0) = \frac{\begin{array}{c|cc} & x_0 & \\ \hline w & & b \\ 1 & 1 & 3 \end{array}}$$

$x_0: w; \text{ send value}(x_0 = w) \rightarrow x_1$

$x_1: w; \text{ send value}(x_0 = w, x_1 = w) \rightarrow x_2, x_3$

# Dynamic Programming Optimization Protocol (DPOP)

- Two messages per variable (UTIL and VALUE).
- ⇒ *number* of messages grows linearly with the size of the problem.
- However, the maximum message *size* grows exponentially with the tree-width of the induced graph.
- In many distributed problems, the tree-width is relatively small.

# Distributed local search

- Drawbacks of systematic search:
  - need variable ordering
  - no anytime behavior: have to wait for termination.
  - often (too) costly.
- Sacrifice completeness  $\Rightarrow$  local search
  - min-conflicts
  - distributed min-conflicts
  - breakout algorithm
  - random sampling

# Min-conflicts

- Assign random value to each variable in parallel (this will conflict with some constraints).
- At each step, find the change in variable assignment which most reduces the number of conflicts.
- Corresponds to search by "hill-climbing".

# Distributed min-conflicts

- Neighbourhood of  $N(x_i)$  = variables connected to  $x_i$  through constraints.
- Change to  $x_i$  can happen asynchronously with others as long as there is no other change in the neighbourhood.  
⇒ two neighbouring agents are not allowed to change simultaneously:
  - highest improvement wins
  - ties broken by fixed ordering
- ⇒ parallel, distributed execution.
  
- also called MGM

# Distributed min-conflicts

- Neighbourhood of  $N(x_i)$  = variables connected to  $x_i$  through constraints.
- Change to  $x_i$  can happen asynchronously with others as long as there is no other change in the neighbourhood.  
⇒ two neighbouring agents are not allowed to change simultaneously:
  - highest improvement wins
  - ties broken by fixed ordering
- ⇒ parallel, distributed execution.
- **CAN GET STUCK IN LOCAL MINIMUM**



# Distributed min-conflicts

- Neighbourhood of  $N(x_i)$  = variables connected to  $x_i$  through constraints.
- Change to  $x_i$  can happen asynchronously with others as long as there is no other change in the neighbourhood.
  - ⇒ two neighbouring agents are not allowed to change simultaneously:
    - highest improvement wins
    - ties broken by fixed ordering
  - ⇒ parallel, distributed execution.

**Definition 2.6** (Quasi-local minimum). *An agent  $x_i$  is in a quasi-local minimum if it is violating some constraint and neither it nor any of its neighbors can make a change that results in lower total cost for the system.*

# Breakout Algorithm

- To escape *local minima* the algorithm identifies *quasi-local minima* and increases the cost of the constraint violations
- Similar to min-conflict, but assign dynamic priority to every conflict (constraint), initially =1
- Modify variable which reduces the most the sum of the priority values of all conflicts.
- When local minimum:
  - increase weight of every existing conflict
- Eventually, new conflicts will have lower weight than existing ones  
⇒ breakout

# Breakout Algorithm: Code

HANDLE-OK?( $j, x_j$ )

```

1  received-ok[ $j$ ]  $\leftarrow$  TRUE
2  agent-view  $\leftarrow$  agent-view + ( $j, x_j$ )
3  if  $\forall_{k \in \text{neighbors}} \text{received-ok}[k] = \text{TRUE}$ 
4      then SEND-IMPROVE()
5       $\forall_{k \in \text{neighbors}} \text{received-ok}[k] \leftarrow \text{FALSE}$ 

```

SEND-IMPROVE()

```

1  new-value  $\leftarrow$  value that gives maximal improvement
2  my-improve  $\leftarrow$  possible maximal improvement
3   $\forall_{k \in \text{neighbors}} k.\text{HANDLE-IMPROVE}(i, \text{my-improve}, \text{cost})$ 

```

HANDLE-IMPROVE( $j, \text{improve}$ )

```

1  received-improve[ $j$ ]  $\leftarrow$  improve
2  if  $\forall_{k \in \text{neighbors}} \text{received-improve}[k] \neq \text{NONE}$ 
3      then SEND-OK
4      agent-view  $\leftarrow$   $\emptyset$ 

```

# Breakout Algorithm: Code

```

2  my-improve ← possible maximal improvement
3   $\forall k \in neighbors$  k.HANDLE-IMPROVE(i, my-improve, cost)

```

HANDLE-IMPROVE(*j*, *improve*)

```

1  received-improve[j] ← improve
2  if  $\forall k \in neighbors$  received-improve[k] ≠ NONE
3      then SEND-OK
4          agent-view ←  $\emptyset$ 
5           $\forall k \in neighbors$  received-improve[k] ← NONE

```

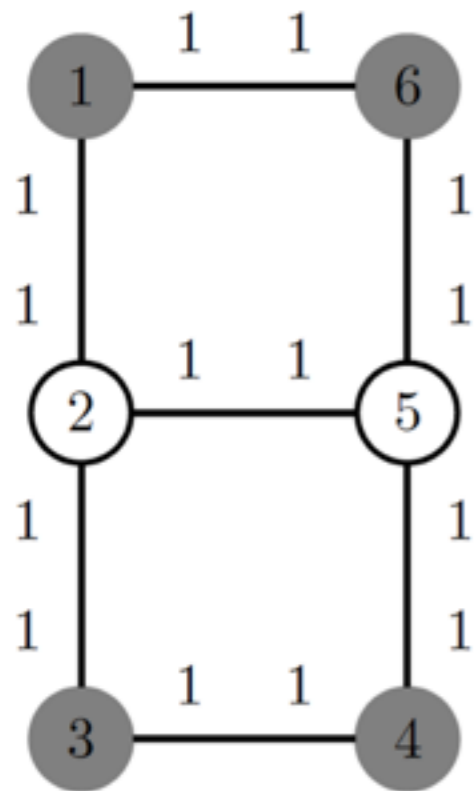
SEND-OK()

```

1  if  $\forall k \in neighbors$  my-improve ≥ received-improve[k]
2      then xi ← new-value
3  if cost > 0  $\wedge \forall k \in neighbors$  received-improve[k] ≤ 0 ▷ quasi-local optimum
4      then increase weight of constraint violations
5   $\forall k \in neighbors$  k.HANDLE-OK?(i, xi)

```

# Breakout Algorithm: Example



HANDLE-OK? ( $j, x_j$ )

```
1  received-ok[ $j$ ]  $\leftarrow$  TRUE
2  agent-view  $\leftarrow$  agent-view + ( $j, x_j$ )
3  if  $\forall k \in \text{neighbors}$  received-ok[ $k$ ] = TRUE
4      then SEND-IMPROVE()
5       $\forall k \in \text{neighbors}$  received-ok[ $k$ ]  $\leftarrow$  FALSE
```

SEND-IMPROVE()

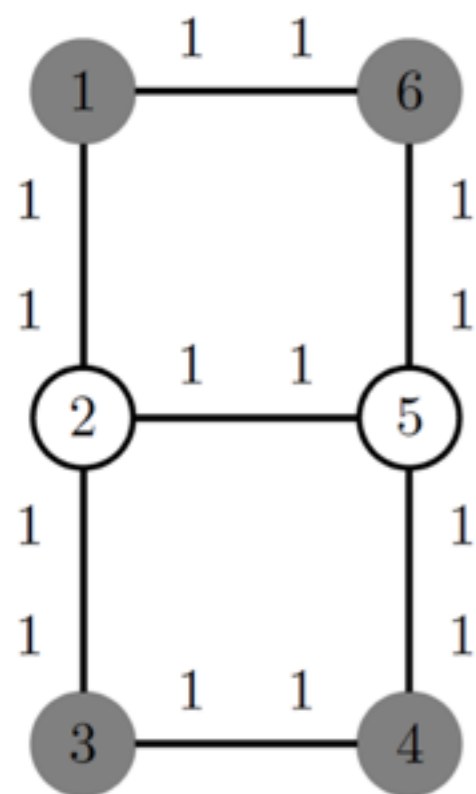
```
1  new-value  $\leftarrow$  value that gives maximal improvement
2  my-improve  $\leftarrow$  possible maximal improvement
3   $\forall k \in \text{neighbors}$   $k$ .HANDLE-IMPROVE( $i, my-improve, cost$ )
```

HANDLE-IMPROVE( $j, improve$ )

```
1  received-improve[ $j$ ]  $\leftarrow$  improve
2  if  $\forall k \in \text{neighbors}$  received-improve[ $k$ ]  $\neq$  NONE
3      then SEND-OK
```

# Breakout Algorithm: Example

- 1  $new-value \leftarrow$  value that gives maximal improvement
- 2  $my-improve \leftarrow$  possible maximal improvement
- 3  $\forall k \in neighbors$   $k.HANDLE-IMPROVE(i, my-improve, cost)$



$HANDLE-IMPROVE(j, improve)$

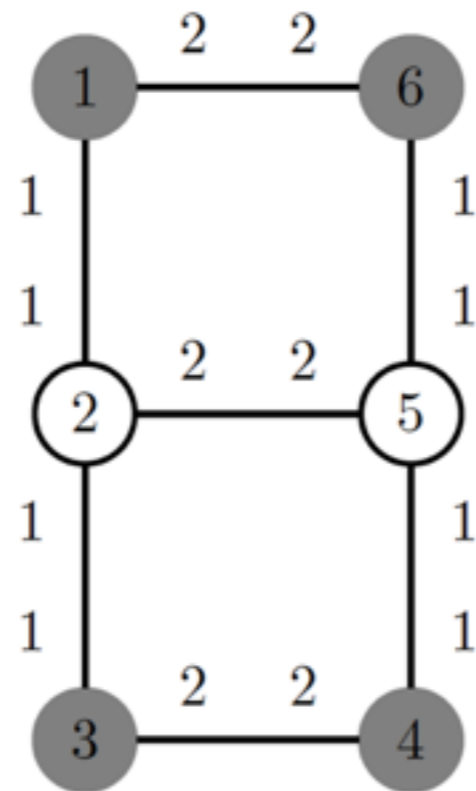
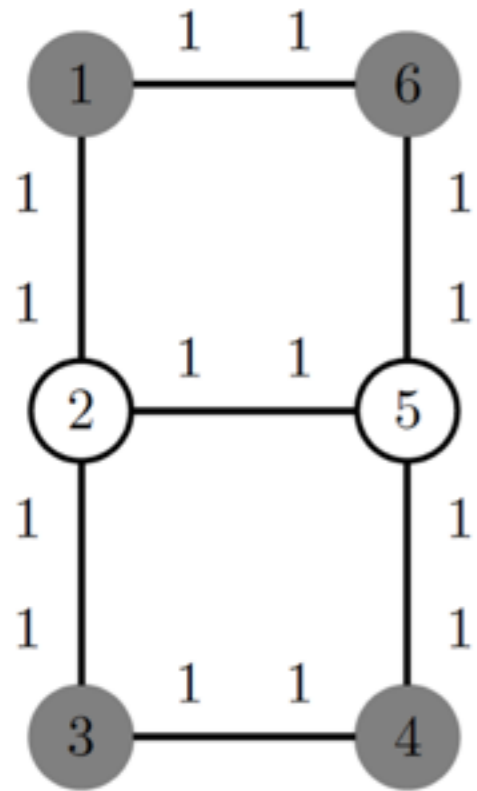
- 1  $received-improve[j] \leftarrow improve$
- 2 **if**  $\forall k \in neighbors$   $received-improve[k] \neq NONE$
- 3     **then** SEND-OK
- 4          $agent-view \leftarrow \emptyset$
- 5          $\forall k \in neighbors$   $received-improve[k] \leftarrow NONE$

SEND-OK()

- 1 **if**  $\forall k \in neighbors$   $my-improve \geq received-improve[k]$
- 2     **then**  $x_i \leftarrow new-value$
- 3 **if**  $cost > 0 \wedge \forall k \in neighbors$   $received-improve[k] \leq 0 \triangleright$
- 4     **then** increase weight of constraint violations
- 5  $\forall k \in neighbors$   $k.HANDLE-OK?(i, x_i)$

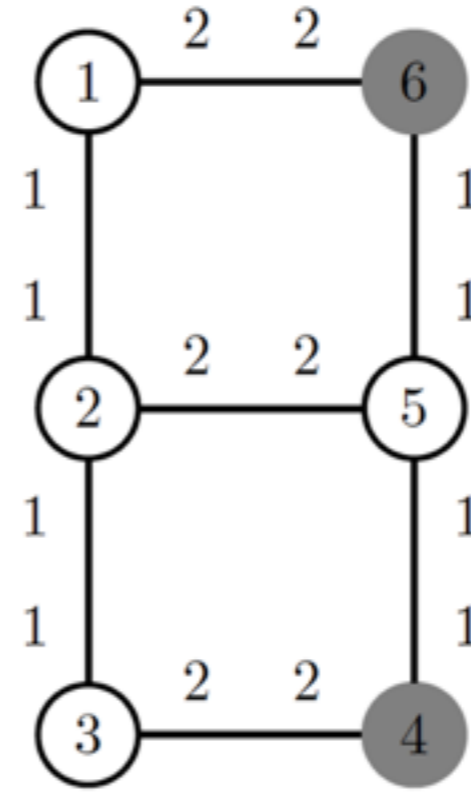
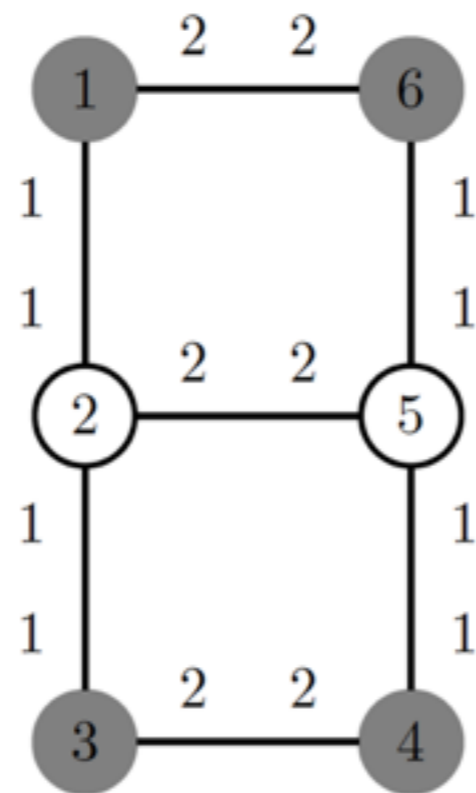
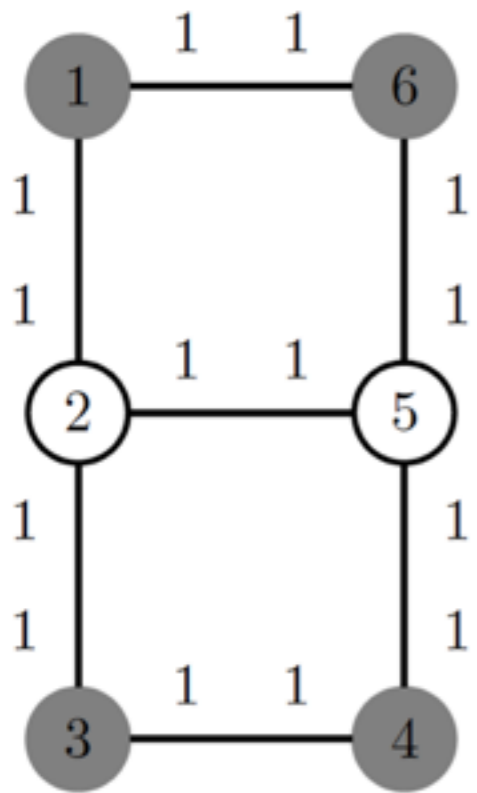
# Breakout Algorithm: Example

01



# Breakout Algorithm: Example

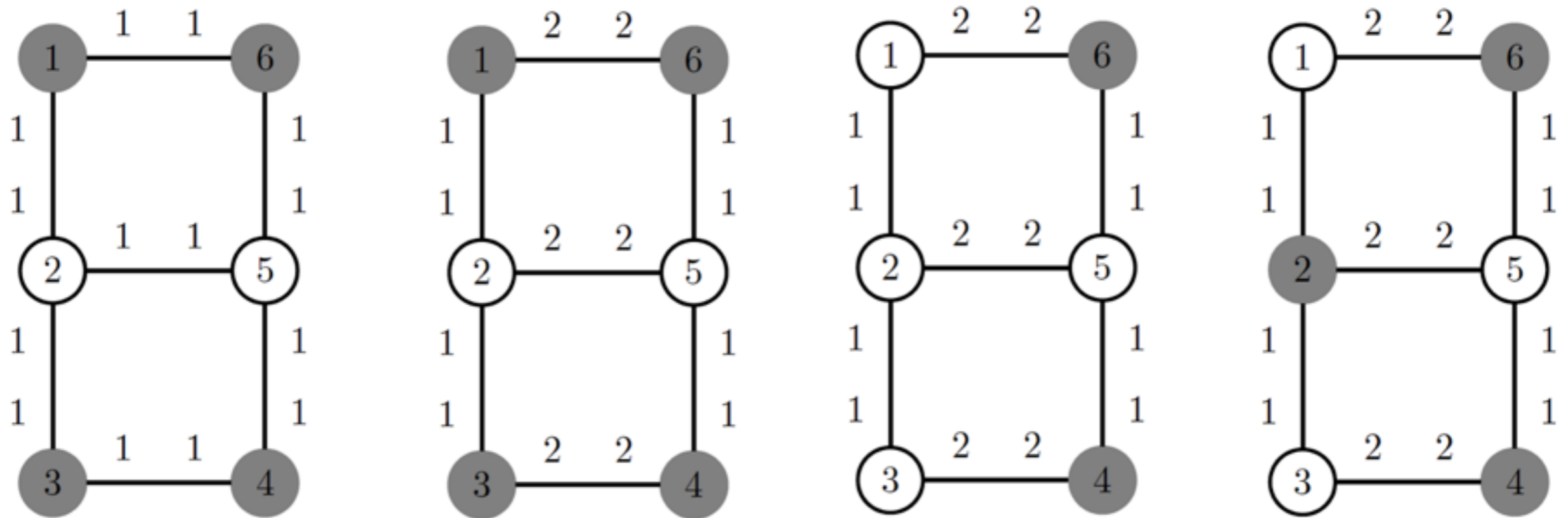
01





# Breakout Algorithm: Example

01



# Breakout Algorithm: Properties

- **Theorem (Distributed Breakout is not Complete)**

Distributed breakout can get stuck in local minimum. Therefore, there are cases where a solution exists and it cannot find it.



**OPPA European Social Fund  
Prague & EU: We invest in your future.**

---