Distributed Constraints

José M Vidal

Department of Computer Science and Engineering, University of South Carolina

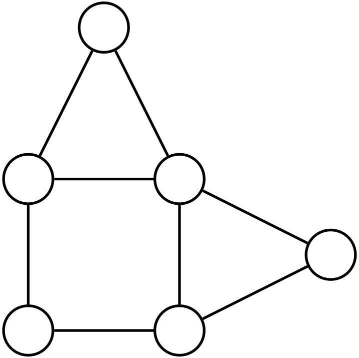
January 15, 2010

Abstract

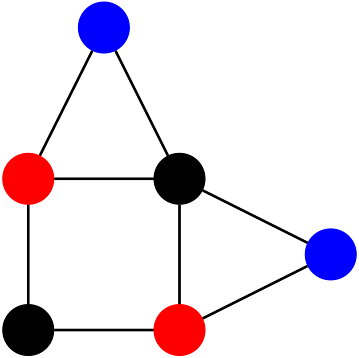
Algorithms for solving distributed constraint problems in multiagent systems. Chapter 2.

Outline

Graph Coloring



Graph Coloring



Constraint Satisfaction Problem (CSP)

Given variables $x_1, x_2, \ldots x_n$ with domains $D_1, D_2, \ldots D_n$ and a set of boolean constraints P of the form $pk(x_{k1}, x_{k2}, \ldots, x_{kj}) \rightarrow \{0, 1\}$, find assignments for all the variables such that no constraints are violated.

Depth First Search for the CSP

DEPTH-FIRST-SEARCH-CSP(i, g)

if $i > n$
  then return g
for $v \in D_i$
  do if setting $x_i \leftarrow v$ does not violate any constraint in P given g
    then $g' \leftarrow$ DEPTH-FIRST-SEARCH-CSP$(i+1, g+\{x_i \leftarrow v\})$
      if $g' \neq \emptyset$
        then return g'

return $\emptyset$

Outline

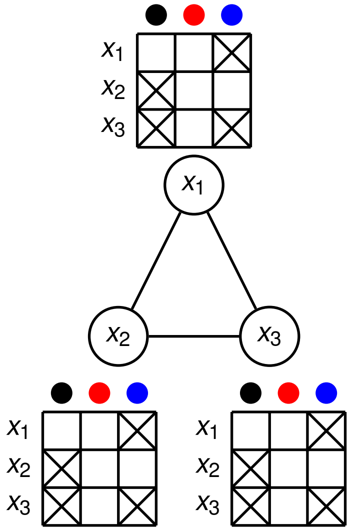
Definition (Distributed Constraint Satisfaction Problem (DCSP))

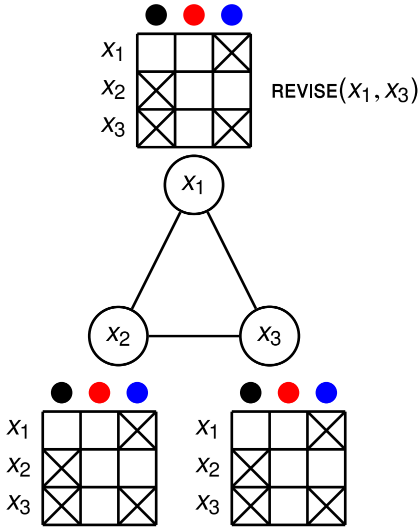
Give each agent one of the variables in a CSP. Agents are responsible for finding a value for their variable and can find out the values of their neighbors' via communication

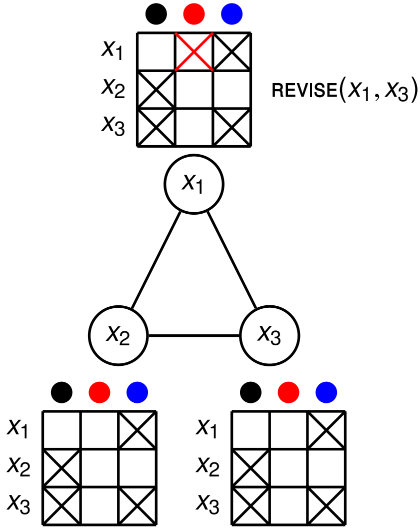
Outline

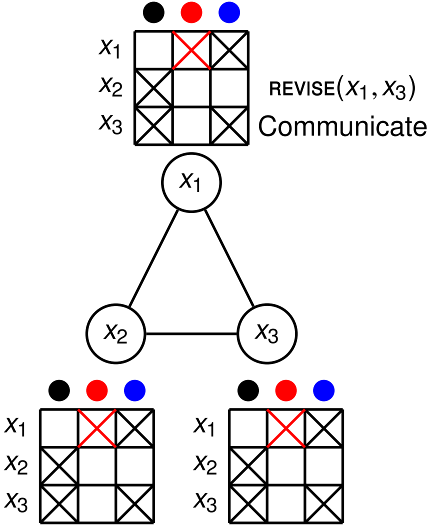
REVISE(x_i, x_j)

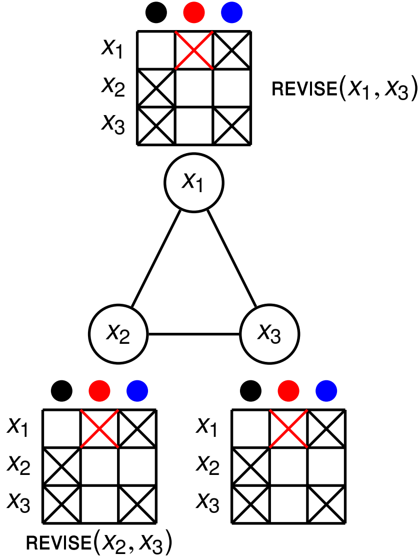
old-domain $\leftarrow D_i$
for $v_i \in D_i$
    do if there is no $v_j \in D_j$ consistent with v_i
        then $D_i \leftarrow D_i - v_i$
if *old-domain* $\neq D_i$
    then $\forall_{k \in \{\text{neighbors of } i\}} k.$HANDLE-NEW-DOMAIN$(i, D_i)$

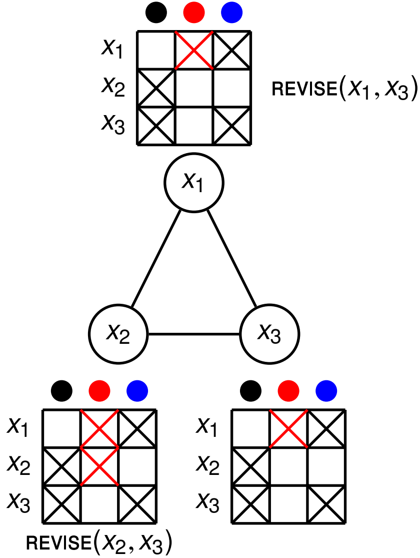

x1
x2
x3
x1
x2
x3
x1
x2
x3
x1
x2
x3
x1
x2
x3

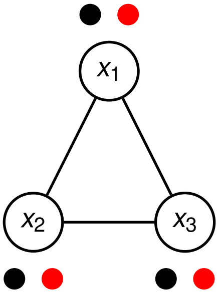

x_1
x_2
x_3
REVISE(x_1, x_3)
x_1
x_2
x_3
x_1
x_2
x_3
x_1
x_2
x_3

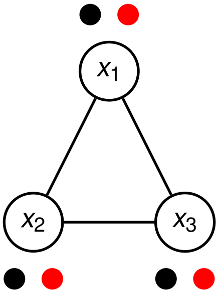

x_1
x_2
x_3
REVISE(x_1, x_3)
x_1
x_2
x_3
x_1
x_2
x_3
x_1
x_2
x_3


x_1
x_2
x_3
REVISE(x_1, x_3)
Communicate
x_1
x_2
x_3
x_1
x_2
x_3
x_1
x_2
x_3


x_1
x_2
x_3
REVISE(x_1, x_3)
x_1
x_2
x_3
x_1
x_2
x_3
x_1
x_2
x_3
REVISE(x_2, x_3)


x_1
x_2
x_3
REVISE(x_1, x_3)
x_1
x_2
x_3
x_1
x_2
x_3
REVISE(x_2, x_3)
x_1
x_2
x_3


x_1
x_2
x_3



Filtering fails to detect no-solution.

Outline

Definition (k-consistency)

Given any instantiation of any $k-1$ variables that satisfy all constraints it is possible to find an instantiation of any kth variable such that all k variable values satisfy all constraints.

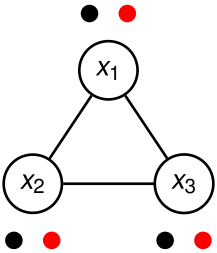
Definition (Strongly k-consistent)

A problem is strongly k-consistent if it is j-consistent for all $j \leq k$.

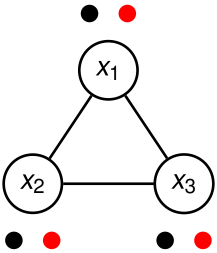
Definition (Hyper-Resolution Rule)

$$\frac{\begin{array}{c} A_1 \vee A_2 \vee \cdots \vee A_m \\ \neg(A_1 \wedge A_{11} \wedge \cdots) \\ \neg(A_2 \wedge A_{21} \wedge \cdots) \\ \vdots \\ \neg(A_m \wedge A_{m1} \wedge \cdots) \end{array}}{\neg(A_{11} \wedge \cdots \wedge A_{21} \wedge \cdots \wedge A_{m1} \wedge \cdots)}.$$

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$
$x_1 = \textcolor{red}{\bullet} \vee x_1 = \bullet$	$x_2 = \textcolor{red}{\bullet} \vee x_2 = \bullet$	$x_3 = \textcolor{red}{\bullet} \vee x_3 = \bullet$
$\neg(x_1 = \textcolor{red}{\bullet} \wedge x_2 = \textcolor{red}{\bullet})$	$\neg(x_1 = \textcolor{red}{\bullet} \wedge x_2 = \textcolor{red}{\bullet})$	$\neg(x_1 = \textcolor{red}{\bullet} \wedge x_2 = \textcolor{red}{\bullet})$
$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$



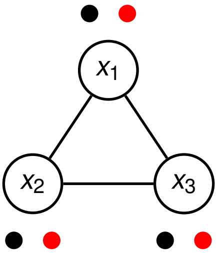
$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$
$x_1 = {\color{red}\bullet} \vee x_1 = \bullet$	$x_2 = {\color{red}\bullet} \vee x_2 = \bullet$	$x_3 = {\color{red}\bullet} \vee x_3 = \bullet$
$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$	$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$	$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$
$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$
${\color{blue}\neg(x_2 = {\color{red}\bullet} \wedge x_3 = {\color{black}\bullet})}$		



$\mathbf{x_1}$

$$\frac{\begin{array}{c} x_1 = {\color{red}\bullet} \vee x_1 = \bullet \\ \neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet}) \\ \neg(x_1 = \bullet \wedge x_3 = \bullet) \end{array}}{\neg(x_2 = {\color{red}\bullet} \wedge x_3 = \bullet)}$$

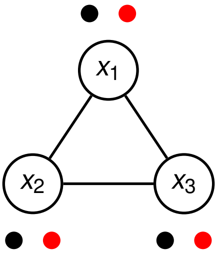
$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$
$x_1 = {\color{red}\bullet} \vee x_1 = \bullet$	$x_2 = {\color{red}\bullet} \vee x_2 = \bullet$	$x_3 = {\color{red}\bullet} \vee x_3 = \bullet$
$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$	$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$	$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$
$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$
${\color{blue}\neg(x_2 =} {\color{red}\bullet} {\color{blue}\wedge x_3 =} \bullet{\color{blue})}$	${\color{blue}\neg(x_2 =} {\color{red}\bullet} {\color{blue}\wedge x_3 =} \bullet{\color{blue})}$	${\color{blue}\neg(x_2 =} {\color{red}\bullet} {\color{blue}\wedge x_3 =} \bullet{\color{blue})}$



$\mathbf{x_1}$

Sends $\neg(x_2 = {\color{red}\bullet} \wedge x_3 = \bullet)$ to x_2 and x_3.

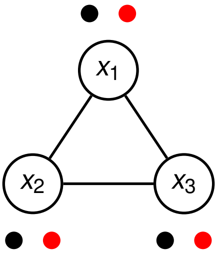
$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$
$x_1 = {\color{red}\bullet} \vee x_1 = \bullet$	$x_2 = {\color{red}\bullet} \vee x_2 = \bullet$	$x_3 = {\color{red}\bullet} \vee x_3 = \bullet$
$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$	$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$	$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$
$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$
${\color{blue}\neg(x_2 = {\color{red}\bullet} \wedge x_3 = \bullet)}$	${\color{blue}\neg(x_2 = {\color{red}\bullet} \wedge x_3 = \bullet)}$	${\color{blue}\neg(x_2 = {\color{red}\bullet} \wedge x_3 = \bullet)}$
	${\color{blue}\neg(x_3 = \bullet)}$	



$\mathbf{x_2}$

$$\frac{\begin{array}{c} x_2 = {\color{red}\bullet} \vee x_2 = \bullet \\ \neg(x_2 = {\color{red}\bullet} \wedge x_3 = \bullet) \\ \neg(x_2 = \bullet \wedge x_3 = \bullet) \end{array}}{\neg(x_3 = \bullet)}$$

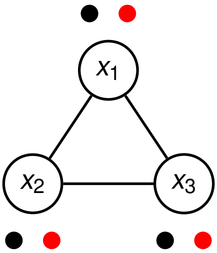
$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$
$x_1 = {\color{red}\bullet} \vee x_1 = \bullet$	$x_2 = {\color{red}\bullet} \vee x_2 = \bullet$	$x_3 = {\color{red}\bullet} \vee x_3 = \bullet$
$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$	$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$	$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$
$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$
$\neg(x_2 = {\color{red}\bullet} \wedge x_3 = \bullet)$	$\neg(x_2 = {\color{red}\bullet} \wedge x_3 = \bullet)$	$\neg(x_2 = {\color{red}\bullet} \wedge x_3 = \bullet)$
$\neg(x_2 = \bullet \wedge x_3 = {\color{red}\bullet})$	$\neg(x_3 = \bullet)$	



$\mathbf{x_1}$

$$\frac{\begin{array}{c} x_1 = {\color{red}\bullet} \vee x_1 = \bullet \\ \neg(x_1 = \bullet \wedge x_2 = \bullet) \\ \neg(x_1 = {\color{red}\bullet} \wedge x_3 = {\color{red}\bullet}) \end{array}}{\neg(x_2 = \bullet \wedge x_3 = {\color{red}\bullet})}$$

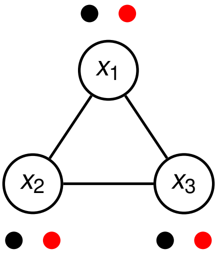
$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$
$x_1 = {\color{red}\bullet} \vee x_1 = \bullet$	$x_2 = {\color{red}\bullet} \vee x_2 = \bullet$	$x_3 = {\color{red}\bullet} \vee x_3 = \bullet$
$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$	$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$	$\neg(x_1 = {\color{red}\bullet} \wedge x_2 = {\color{red}\bullet})$
$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$
$\neg(x_2 = {\color{red}\bullet} \wedge x_3 = \bullet)$	$\neg(x_2 = {\color{red}\bullet} \wedge x_3 = \bullet)$	$\neg(x_2 = {\color{red}\bullet} \wedge x_3 = \bullet)$
$\neg(x_2 = \bullet \wedge x_3 = {\color{red}\bullet})$	$\neg(x_3 = \bullet)$	$\neg(x_2 = \bullet \wedge x_3 = {\color{red}\bullet})$
	$\neg(x_2 = \bullet \wedge x_3 = {\color{red}\bullet})$	



$\mathbf{x_1}$

Sends $\neg(x_2 = \bullet \wedge x_3 = {\color{red}\bullet})$ to x_2 and x_3.

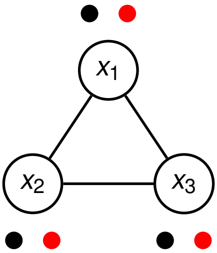
$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$
$x_1 = \textcolor{red}{\bullet} \vee x_1 = \bullet$	$x_2 = \textcolor{red}{\bullet} \vee x_2 = \bullet$	$x_3 = \textcolor{red}{\bullet} \vee x_3 = \bullet$
$\neg(x_1 = \textcolor{red}{\bullet} \wedge x_2 = \textcolor{red}{\bullet})$	$\neg(x_1 = \textcolor{red}{\bullet} \wedge x_2 = \textcolor{red}{\bullet})$	$\neg(x_1 = \textcolor{red}{\bullet} \wedge x_2 = \textcolor{red}{\bullet})$
$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$
$\neg(x_2 = \textcolor{red}{\bullet} \wedge x_3 = \bullet)$	$\neg(x_2 = \textcolor{red}{\bullet} \wedge x_3 = \bullet)$	$\neg(x_2 = \textcolor{red}{\bullet} \wedge x_3 = \bullet)$
$\neg(x_2 = \bullet \wedge x_3 = \textcolor{red}{\bullet})$	$\neg(x_3 = \bullet)$	$\neg(x_2 = \bullet \wedge x_3 = \textcolor{red}{\bullet})$
	$\neg(x_2 = \bullet \wedge x_3 = \textcolor{red}{\bullet})$	
	$\neg(x_3 = \textcolor{red}{\bullet})$	



$\mathbf{x_2}$

$$\begin{array}{c} x_2 = \textcolor{red}{\bullet} \vee x_2 = \bullet \\ \neg(x_2 = \textcolor{red}{\bullet} \wedge x_3 = \textcolor{red}{\bullet}) \\ \neg(x_2 = \bullet \wedge x_3 = \textcolor{red}{\bullet}) \\ \hline \neg(x_3 = \textcolor{red}{\bullet}) \end{array}$$

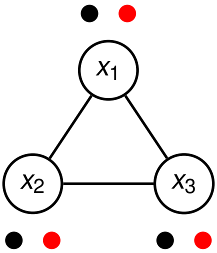
$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$
$x_1 = \textcolor{red}{\bullet} \vee x_1 = \bullet$	$x_2 = \textcolor{red}{\bullet} \vee x_2 = \bullet$	$x_3 = \textcolor{red}{\bullet} \vee x_3 = \bullet$
$\neg(x_1 = \textcolor{red}{\bullet} \wedge x_2 = \textcolor{red}{\bullet})$	$\neg(x_1 = \textcolor{red}{\bullet} \wedge x_2 = \textcolor{red}{\bullet})$	$\neg(x_1 = \textcolor{red}{\bullet} \wedge x_2 = \textcolor{red}{\bullet})$
$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$
$\neg(x_2 = \textcolor{red}{\bullet} \wedge x_3 = \bullet)$	$\neg(x_2 = \textcolor{red}{\bullet} \wedge x_3 = \bullet)$	$\neg(x_2 = \textcolor{red}{\bullet} \wedge x_3 = \bullet)$
$\neg(x_2 = \bullet \wedge x_3 = \textcolor{red}{\bullet})$	$\neg(x_3 = \bullet)$	$\neg(x_2 = \bullet \wedge x_3 = \textcolor{red}{\bullet})$
	$\neg(x_2 = \bullet \wedge x_3 = \textcolor{red}{\bullet})$	$\neg(x_3 = \textcolor{red}{\bullet})$
	$\neg(x_3 = \textcolor{red}{\bullet})$	$\neg(x_3 = \bullet)$



$\mathbf{x_2}$

Sends $\neg(x_3 = \textcolor{red}{\bullet})$ and $\neg(x_3 = \bullet)$ to x_3.

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$
$x_1 = \textcolor{red}{\bullet} \vee x_1 = \bullet$	$x_2 = \textcolor{red}{\bullet} \vee x_2 = \bullet$	$x_3 = \textcolor{red}{\bullet} \vee x_3 = \bullet$
$\neg(x_1 = \textcolor{red}{\bullet} \wedge x_2 = \textcolor{red}{\bullet})$	$\neg(x_1 = \textcolor{red}{\bullet} \wedge x_2 = \textcolor{red}{\bullet})$	$\neg(x_1 = \textcolor{red}{\bullet} \wedge x_2 = \textcolor{red}{\bullet})$
$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$	$\neg(x_1 = \bullet \wedge x_2 = \bullet)$
$\neg(x_2 = \textcolor{red}{\bullet} \wedge x_3 = \bullet)$	$\neg(x_2 = \textcolor{red}{\bullet} \wedge x_3 = \bullet)$	$\neg(x_2 = \textcolor{red}{\bullet} \wedge x_3 = \bullet)$
$\neg(x_2 = \bullet \wedge x_3 = \textcolor{red}{\bullet})$	$\neg(x_3 = \bullet)$	$\neg(x_2 = \bullet \wedge x_3 = \textcolor{red}{\bullet})$
	$\neg(x_2 = \bullet \wedge x_3 = \textcolor{red}{\bullet})$	$\neg(x_3 = \textcolor{red}{\bullet})$
	$\neg(x_3 = \textcolor{red}{\bullet})$	$\neg(x_3 = \bullet)$

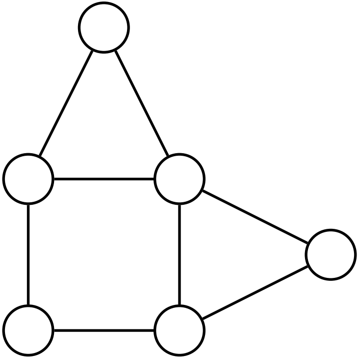


$\mathbf{x_3}$

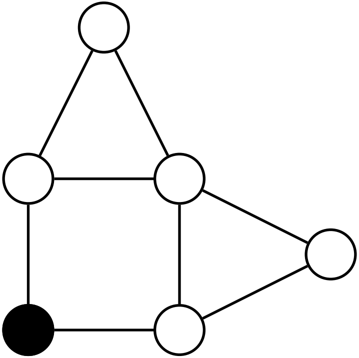
$$\frac{\begin{array}{c} x_3 = \textcolor{red}{\bullet} \vee x_3 = \bullet \\ \neg(x_3 = \textcolor{red}{\bullet}) \\ \neg(x_3 = \bullet) \end{array}}{\text{Contradiction}}$$

Outline

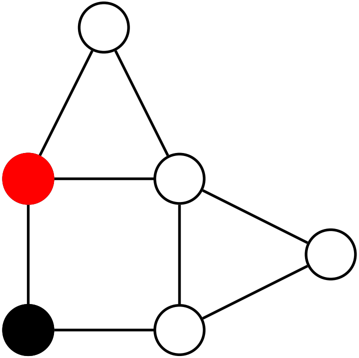
Backtracking



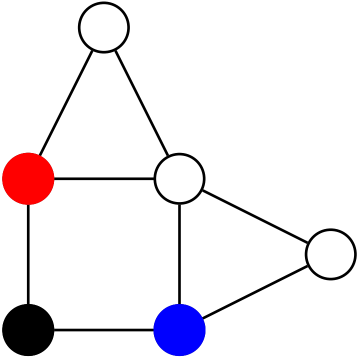
Backtracking



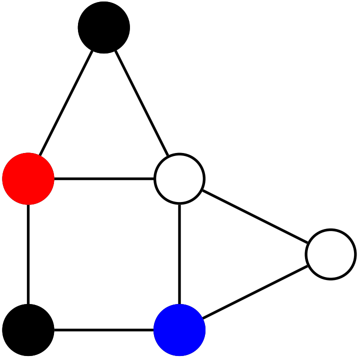
Backtracking



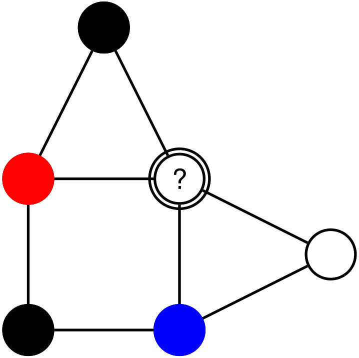
Backtracking



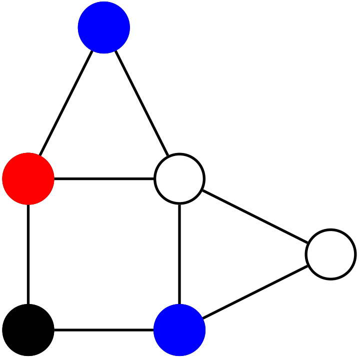
Backtracking



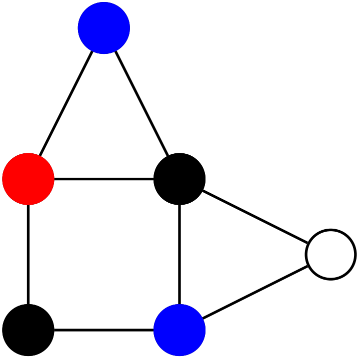
Backtracking



Backtracking



Backtracking



Agent Variables

- *priority*: the agent's fixed priority number. All agents are ordered.
- *local-view*: current values of other agents' variables.
- *current-value*: current value of agent's variable.
- *neighbors*: initially, the set of agents with whom agent shares a constraint.

Remote Calls

- HANDLE-OK?(j, x_j) This message asks the receiver if that assignment does not violate any of his constraints.
- HANDLE-NOGOOD$(j, nogood)$ which means that j is reporting that it can't find a value for his variable because of *nogood*.
- HANDLE-ADD-NEIGHBOR(j) which requests the agent to add some other agent j as its neighbor.

HANDLE-OK?(j, x_j)

$\textit{local-view} \leftarrow \textit{local-view} + (j, x_j)$
CHECK-LOCAL-VIEW$()$

CHECK-LOCAL-VIEW()

if *local-view* and x_i are not consistent
  then if no value in D_i is consistent with *local-view*
    then BACKTRACK()
    else select $d \in D_i$ consistent with *local-view*
      $x_i \leftarrow d$
      $\forall_{k \in neighbors}$ k.HANDLE-OK?(i, x_i)

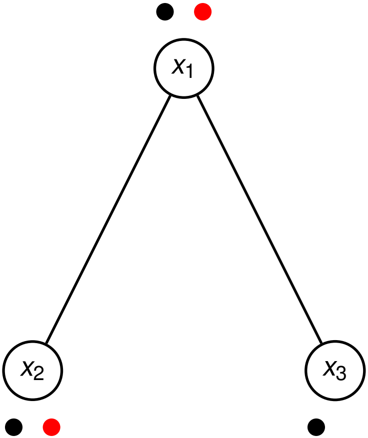
HANDLE-NOGOOD$(j, nogood)$

record *nogood* as a new constraint
for $(k, x_k) \in nogood$ where $k \notin neighbors$
    do k.HANDLE-ADD-NEIGHBOR(i)
        $neighbors \leftarrow neighbors + k$
        $local\text{-}view \leftarrow local\text{-}view + (k, x_k)$
$old\text{-}value \leftarrow x_i$
CHECK-LOCAL-VIEW$()$
if $old\text{-}value \neq x_i$
  then j.HANDLE-OK?(i, x_i)

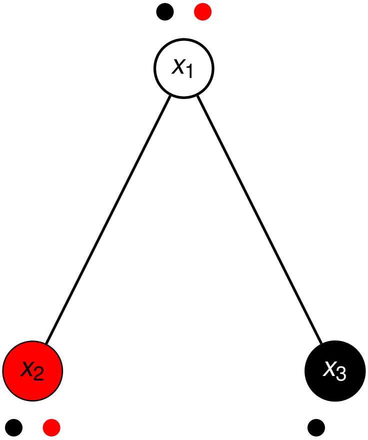
BACKTRACK()

nogoods $\leftarrow \{V \mid V =$ inconsistent subset of *local-view* using hyper-res
if an empty set is an element of *nogoods*
    then broadcast that there is no solution
            terminate this algorithm
for $V \in$ *nogoods*
        do select (j, x_j) where j has lowest priority in V
            j.HANDLE-NOGOOD(i, V)
            local-view $\leftarrow$ *local-view* $- (j, x_j)$
CHECK-LOCAL-VIEW()

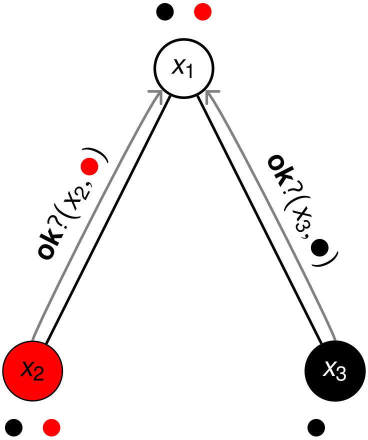
Example



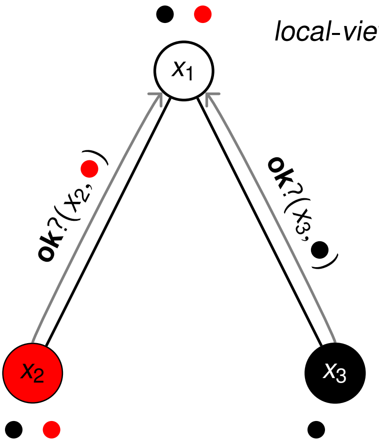
Example



Example

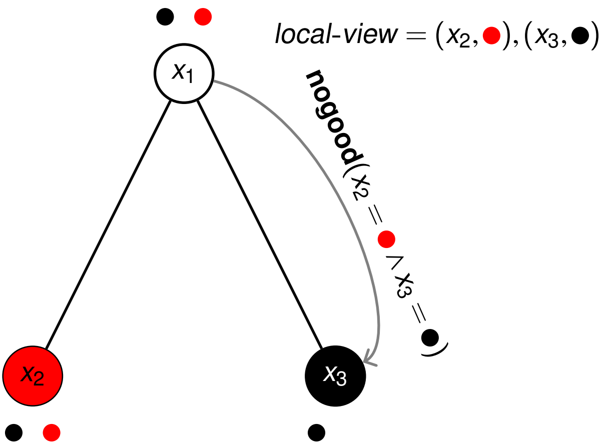


Example

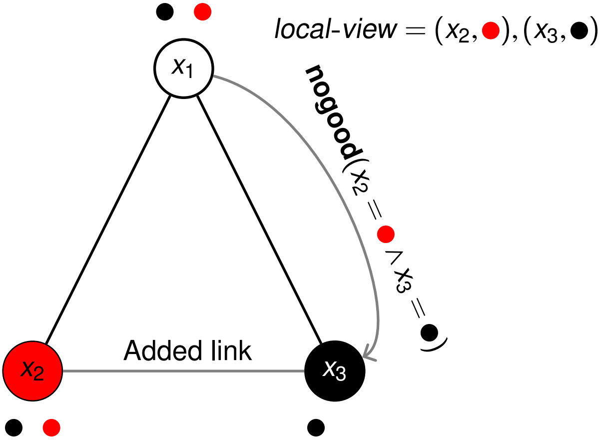


local-view = (x_2, ●), (x_3, ●)

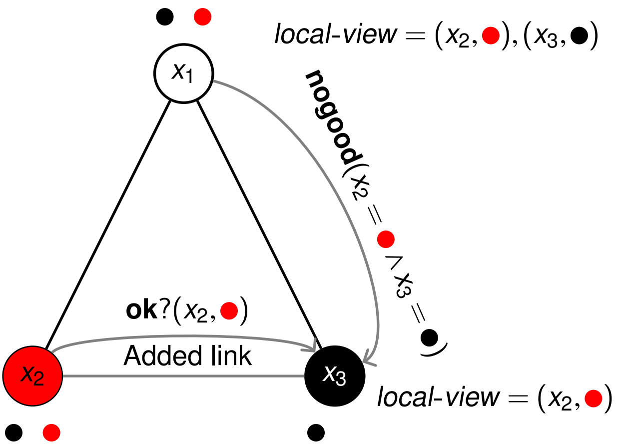
Example



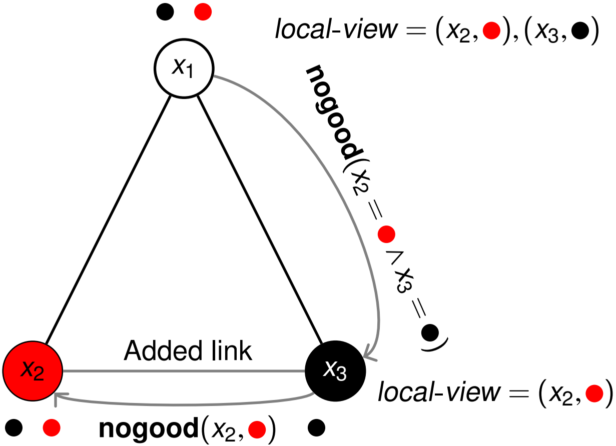
Example


$local\text{-}view = (x_2, \textcolor{red}{\bullet}), (x_3, \bullet)$
x_1
nogood$(x_2 = \textcolor{red}{\bullet} \land x_3 = \bullet)$
x_2
Added link
x_3

Example


x_1
$local\text{-}view = (x_2, \bullet), (x_3, \bullet)$
nogood$(x_2 = \bullet \wedge x_3 = \bullet)$
ok?$(x_2, \bullet)$
Added link
x_2
x_3
$local\text{-}view = (x_2, \bullet)$

Example


local-view = (x2, ●), (x3, ●)
x1
nogood(x2 = ● ∧ x3 = ●)
x2
Added link
x3
local-view = (x2, ●)
nogood(x2, ●)

Theorem (ABT is Complete)

The ABT algorithm always finds a solution if one exists and terminates with the appropriate message if there is no solution.

Proof.

By induction. First show that the agent with the highest priority never enters an infinite loop. Then show that given that all the agents with lower priority that k never fall into an infinite loop then k will not fall into an infinite loop. □

Outline

Asynchronous Weak-Commitment (AWC)

- Use dynamic priorities.
- Change **ok**? messages to include agent's current priority.
- Use min-conflict heuristic.

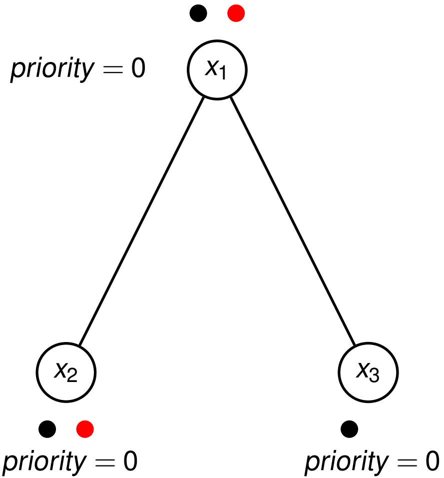
CHECK-LOCAL-VIEW

if x_i is consistent with *local-view*
    then return
if no value in D_i is consistent with *local-view*
    then BACKTRACK()
    else select $d \in D_i$ consistent with *local-view*
        and which minimizes constraint
        violations with lower priority agents.
        $x_i \leftarrow d$
        $\forall_{k \in neighbors}\ k.$HANDLE-OK?$(i, x_i, priority)$

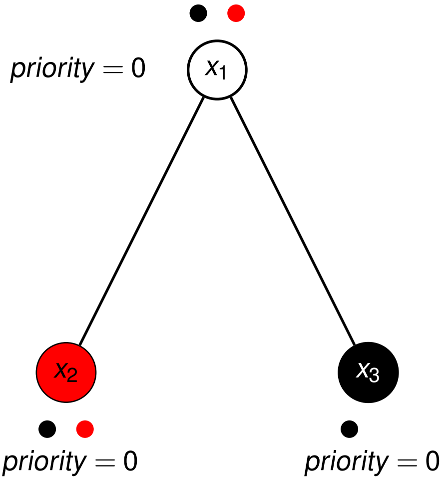
BACKTRACK

generate a nogood V
if V is empty nogood
    then broadcast that there is no solution
            terminate this algorithm
if V is a new nogood
    then $\forall_{(k,x_k)\in V}\ k.\text{HANDLE-NOGOOD}(i, j, \mathit{priority})$
            $\mathit{priority} \leftarrow 1 + \max\{\mathit{neighbors}\text{'priorities}\}$
            select $d \in D_i$ consistent with *local-view*
                    and which minimizes constraint
                    violations with lower priority agents.
            $x_i \leftarrow d$
            $\forall_{k\in \mathit{neighbors}}\ k.\text{HANDLE-OK?}(i, x_i, \mathit{priority})$

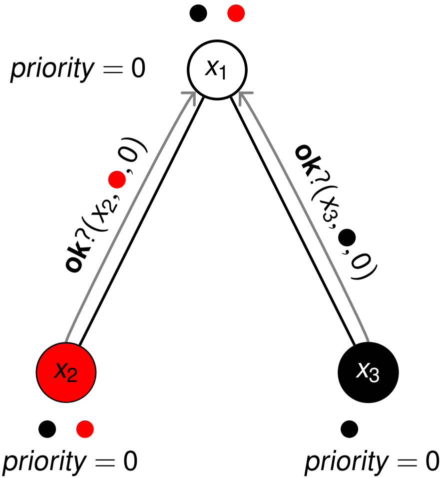
Example



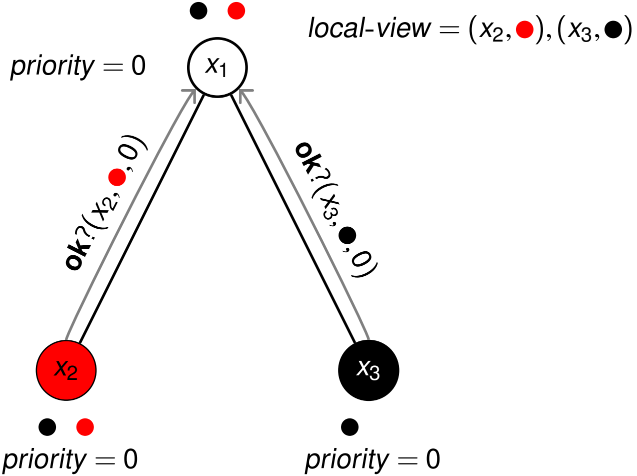
Example



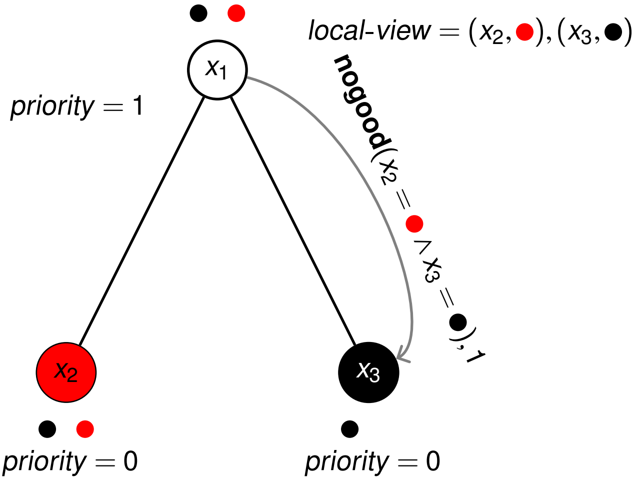
Example



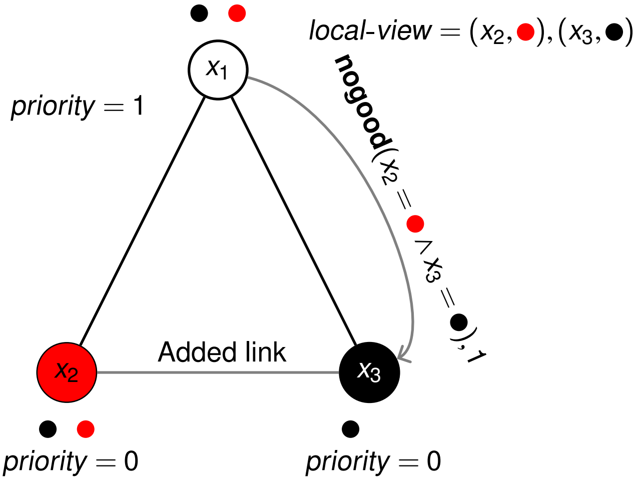
Example


local-view = (x2, ●), (x3, ●)
priority = 0
x1
ok?(x2, ●, 0)
ok?(x3, ●, 0)
x2
x3
priority = 0
priority = 0

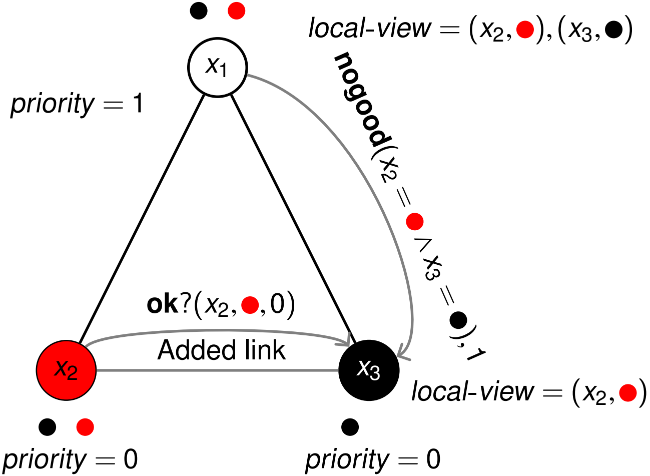
Example


local-view = (x_2, ●), (x_3, ●)
nogood(x_2 = ● ∧ x_3 = ●), 1
x_1
priority = 1
x_2
x_3
priority = 0
priority = 0

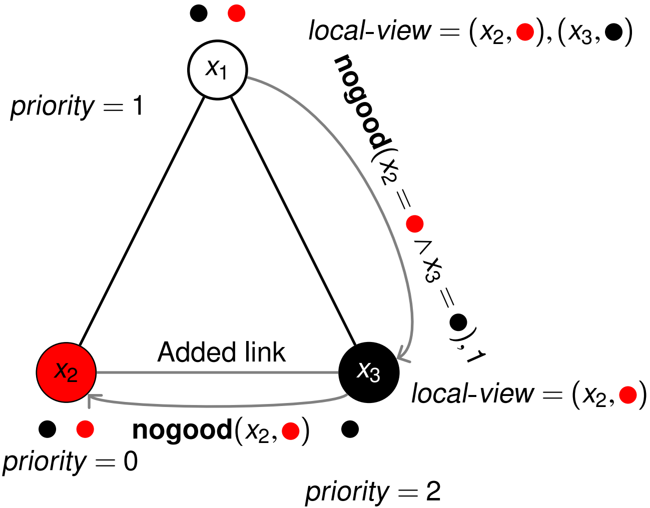
Example


local-view = (x2, ●), (x3, ●)
priority = 1
x1
nogood(x2 = ● ∧ x3 = ●), 1
Added link
x2
x3
priority = 0
priority = 0

Example


local-view = (x_2, ●), (x_3, ●)
x_1
priority = 1
nogood(x_2 = ● ∧ x_3 = ●), 1
ok?(x_2, ●, 0)
Added link
x_2
x_3
local-view = (x_2, ●)
priority = 0
priority = 0

Example


local-view = $(x_2, \bullet), (x_3, \bullet)$
x_1
priority = 1
nogood$(x_2 = \bullet \wedge x_3 = \bullet), 1$
Added link
x_2
x_3
local-view = $(x_2, \bullet)$
nogood$(x_2, \bullet)$
priority = 0
priority = 2

Theorem (AWC is complete)

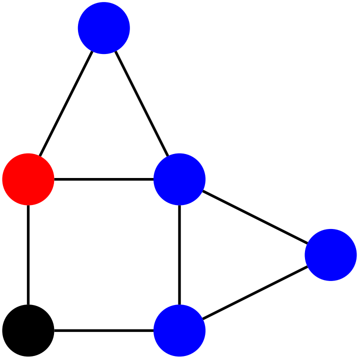
The AWC algorithm always finds a solution if one exists and terminates with the appropriate message if there is no solution.

Proof.

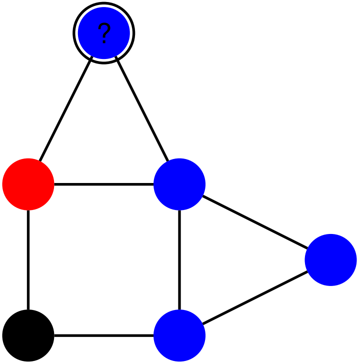
The priority values are changed if and only if a new nogood is found. Since the number of possible nogoods is finite the priority values cannot be changed indefinitely. When the priority values are stable AWC becomes ABT, which is complete. □

Outline

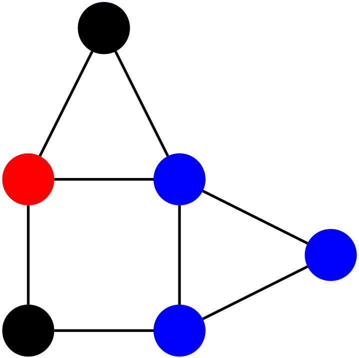
Hill Climbing



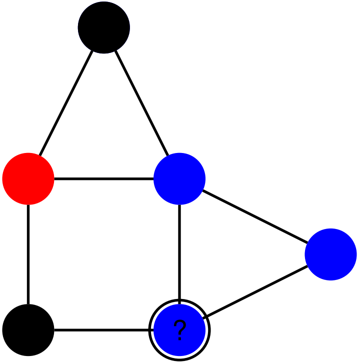
Hill Climbing



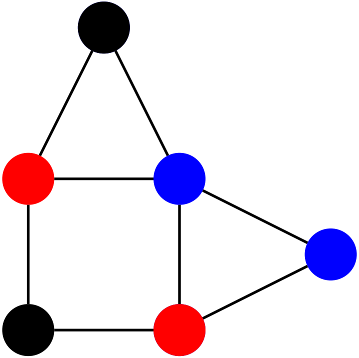
Hill Climbing



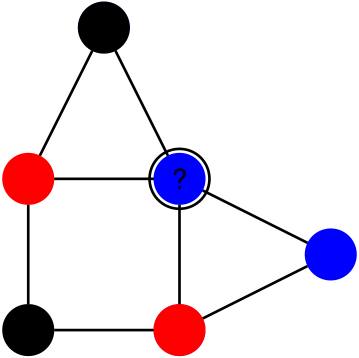
Hill Climbing



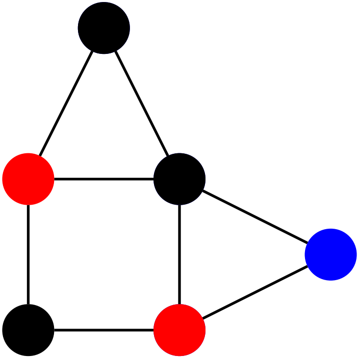
Hill Climbing



Hill Climbing



Hill Climbing



Definition (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 cost for all.

Remote Procedure Calls

- HANDLE-OK?(i, x_i) where i is the agent and x_i is its current value,
- HANDLE-IMPROVE$(i, improve, eval)$ where *improve* is the maximum i could gain by changing to some other color and *eval* is its current cost.

HANDLE-OK?(j, x_j)

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

SEND-IMPROVE()

cost $\leftarrow$ evaluation of x_i given current weights and values.
my-improve $\leftarrow$ possible maximal improvement
new-value $\leftarrow$ value that gives maximal improvement
$\forall_{k \in neighbors}\ k$.HANDLE-IMPROVE($i$, *my-improve*, *cost*)

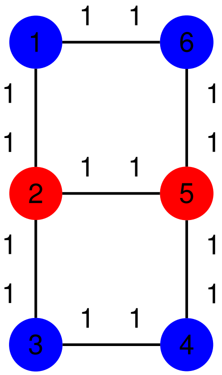
HANDLE-IMPROVE$(j, improve, eval)$

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

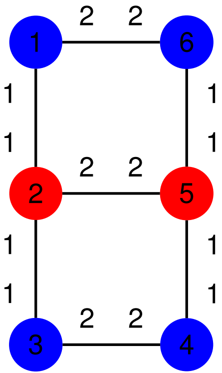
SEND-OK()

if $\forall_{k \in neighbors}\ \textit{my-improve} \geq \textit{received-improve}[k]$
  then $x_i \leftarrow \textit{new-value}$
if $\textit{cost} > 0 \wedge \forall_{k \in neighbors}\ \textit{received-improve}[k] \leq 0$ ▷ quasi-local opt.
  then increase weight of constraint violations
$\forall_{k \in neighbors}\ k$.HANDLE-OK?$(i, x_i)$

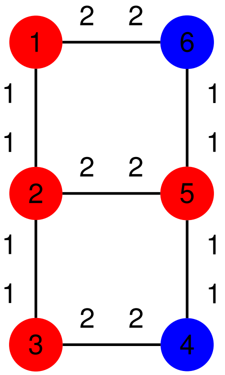
Example



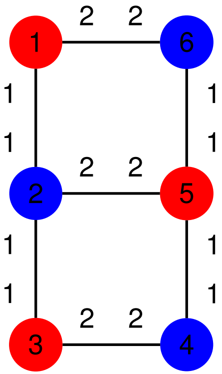
Example



Example



Example



Theorem (Distributed Breakout is not Complete)

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

Proof.

By example. □

Theorem (Distributed Breakout is not Complete)

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

Proof.

By example. □

In practice, its really good.

Outline

Definition (Constraint Optimization Problem (COP))

Given variables $x_1, x_2, \ldots x_n$ with domains $D_1, D_2, \ldots D_n$ and a set of constraints P of the form $pk(x_{k1}, x_{k2}, \ldots, x_{kj}) \rightarrow \Re$, find assignments for all the variables such that the sum of the constraint values is minimized.

BRANCH-AND-BOUND-COP()

$c* \leftarrow \infty$ ▷ Minimum cost found. Global variable.
$g* \leftarrow \emptyset$ ▷ Best solution found. Global variable.
BRANCH-AND-BOUND-COP-HELPER$(1, \emptyset)$
return g^*

BRANCH-AND-BOUND-COP-HELPER(i, g)

if $i = n$
  then if $P(g) < c^*$
    then $g^* \leftarrow g$
      $c^* \leftarrow P(g)$
    return
for $v \in D_i$
  do $g' \leftarrow g + \{x_i \leftarrow v\}$
    if $P(g) < c^*$
      then BRANCH-AND-BOUND-COP-HELPER$(i+1, g')$

Definition (Distributed Constraint Optimization Problem (DCOP))

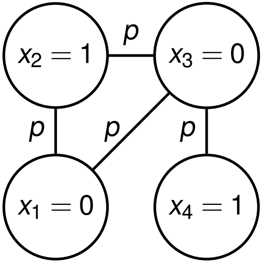
Give each agent one of the variables in a COP. Agents are responsible for finding a value for their variable and can find out the values of their neighbors' via communication

Outline

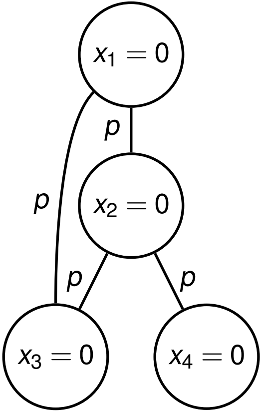
Remote Procedure Calls

- **threshold** tell children how much cost they can incur, ignore anything that costs more than that.
- **value** tell descendants what value agent sets itself to.
- **cost** tell parent lower and upper bounds of cost given the current value assignments of ancestors.

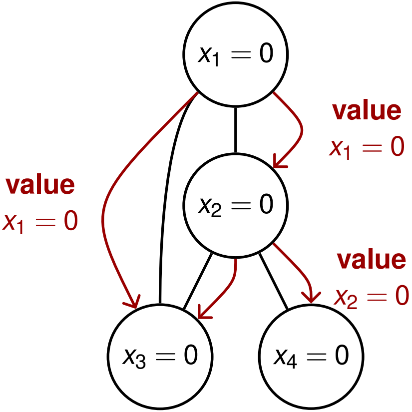
d_i	d_j	$p(d_i, d_j)$
0	0	1
0	1	2
1	0	2
1	1	0



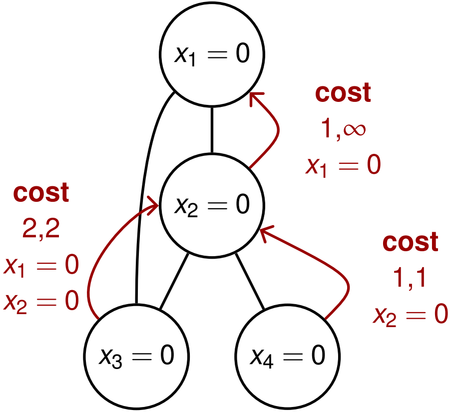
d_i	d_j	$p(d_i, d_j)$
0	0	1
0	1	2
1	0	2
1	1	0



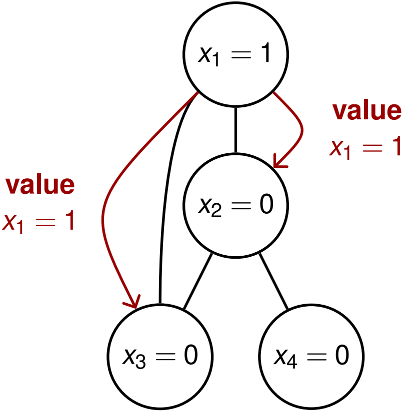
d_i	d_j	$p(d_i, d_j)$
0	0	1
0	1	2
1	0	2
1	1	0



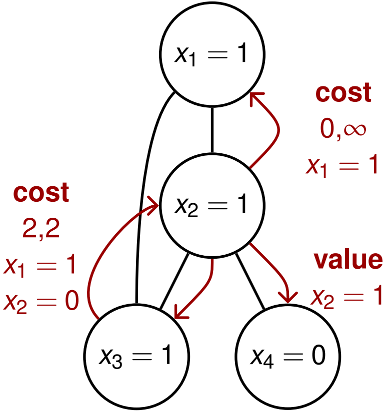
d_i	d_j	$p(d_i, d_j)$
0	0	1
0	1	2
1	0	2
1	1	0



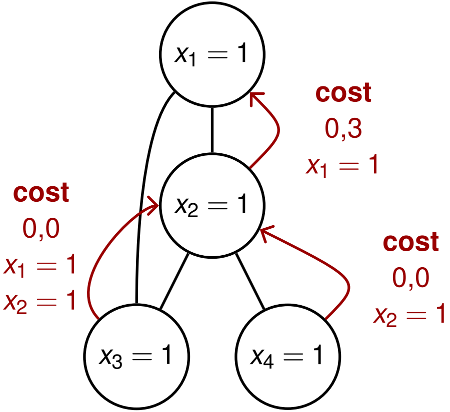
d_i	d_j	$p(d_i, d_j)$
0	0	1
0	1	2
1	0	2
1	1	0



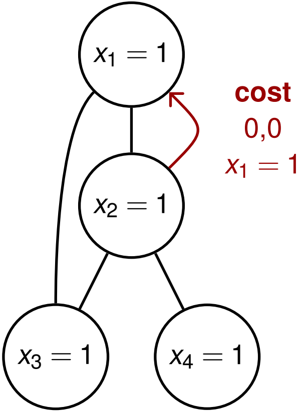
d_i	d_j	$p(d_i, d_j)$
0	0	1
0	1	2
1	0	2
1	1	0



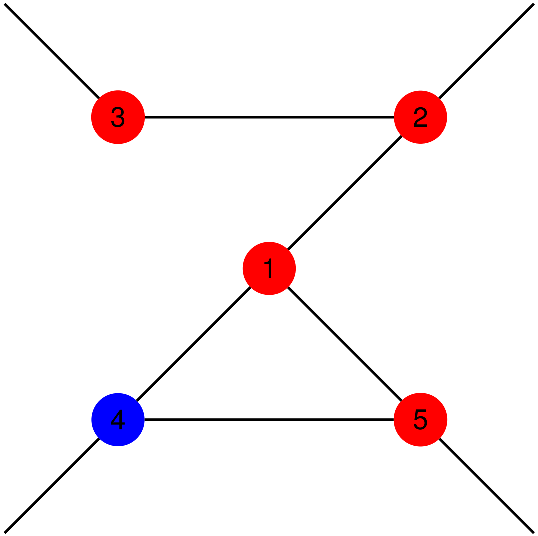
d_i	d_j	$p(d_i, d_j)$
0	0	1
0	1	2
1	0	2
1	1	0

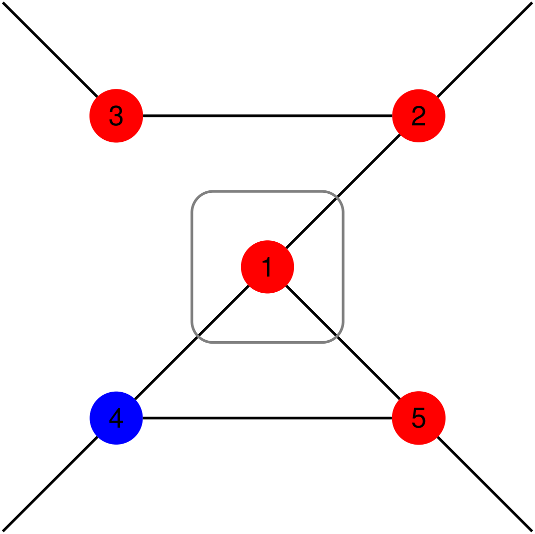


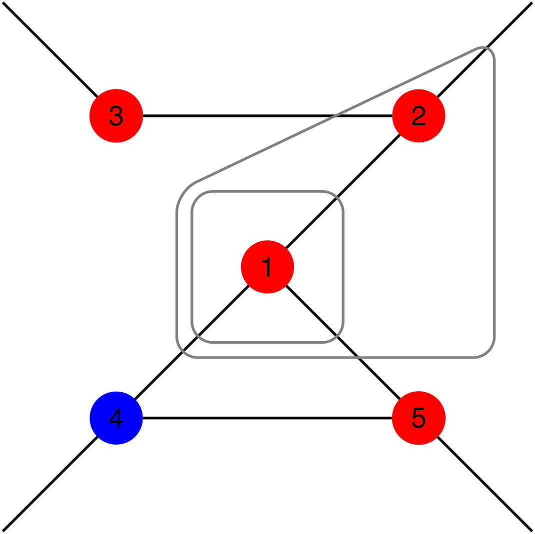
d_i	d_j	$p(d_i, d_j)$
0	0	1
0	1	2
1	0	2
1	1	0

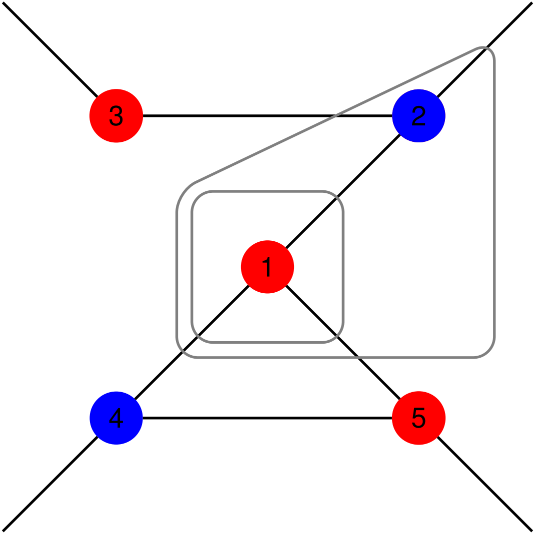


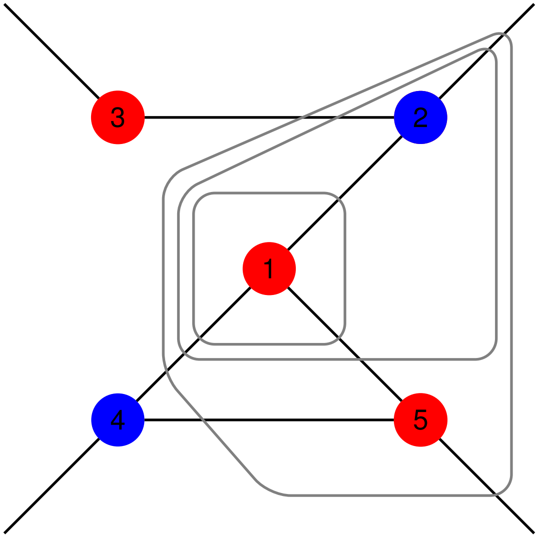
Outline

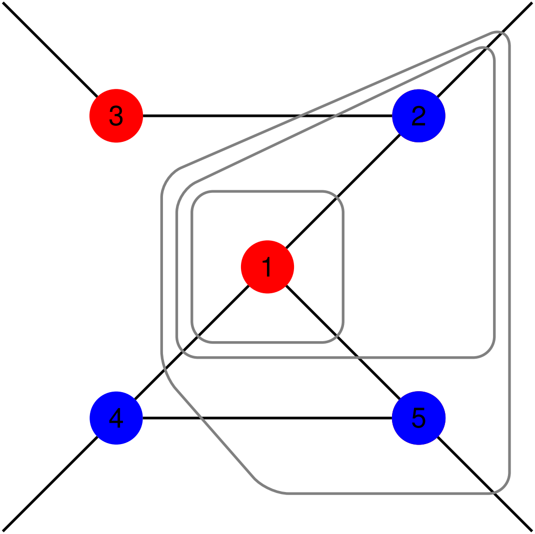

3
2
1
4
5


3
2
1
4
5


3
2
1
4
5


3
2
1
4
5


3
2
1
4
5


3
2
1
4
5

Theorem (APO worst case is centralized search)

In the worst case APO (OptAPO) will make one agent do a completely centralized search of the complete problem space.

Proof.

By example. ☐

Adopt versus OptAPO

- Adopt is better when communications are fast.
- OptAPO is better when communications are slow.
- Both have very bad worst-case but seem to perform well.