# Machine Intelligence

## Lecture 3: Constraint satisfaction problems

Thomas Dyhre Nielsen

*Aalborg University*

# Tentative course overview

**Topics:**

- Introduction
- Search-based methods
- **Constraint satisfaction problems**
- Logic-based knowledge representation
- Representing domains endowed with uncertainty.
- Bayesian networks
- Machine learning
- Planning
- Reinforcement learning
- Multi-agent systems

# Features and Possible Worlds

# Features and Variables

Describing the world (environment) by features:

| Name (**Algebraic Variable**) | Domain |
|---|---|
| *Symbol_on_square_1* | $\{1, 2, 3, 4, 5, 6, 7, 8, \textit{empty}\}$ |
| *Robot_battery* | $\{\textit{full}, \textit{half}, \textit{empty}\}$ |
| *Robot_position* | $\{r131, \ldots, 0111\}$ |
| *Coffee_ready* | $\{\textit{true}, \textit{false}\}$ |
| *No._of_undelivered_packages* | $\{1, 2, 3, \ldots\}$ |
| *Temperature* | $[-25, 40]$ |

- We will be mostly concerned with (algebraic) variables that have a *finite* domain.
- Special interest: **boolean** variable with domain $\{\textit{true}, \textit{false}\}$
- Numerical variables can be approximated:

$$\begin{array}{rcl} \{1, 2, 3, \ldots\} & \mapsto & \{1, 2, 3, 4, 5, > 5\} \\ [-25, 40] & \mapsto & \{-25, -24, \ldots, -1, 0, 1, \ldots 40\} \end{array}$$

### From variables to possible worlds

A **possible world** for a set of variables is an assignment of a value to each variable.

### Connection with state spaces

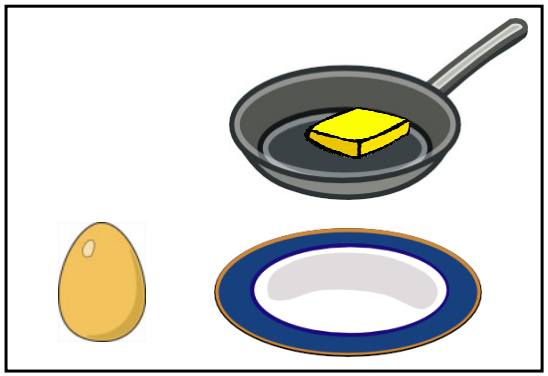
The set of all possible worlds for a given set of variables defines a state space (we can also call a possible world simply a state).

# Example: Cooking

Variables:

| | |
|---|---|
| *egg* | $\{$*whole*, *broken*$\}$ |
| *butter_in* | $\{$*pan*, *plate*, *table*$\}$ |
| *egg_in* | $\{$*pan*, *plate*, *table*$\}$ |

One out of $2 \cdot 3 \cdot 3 = 18$ possible worlds:



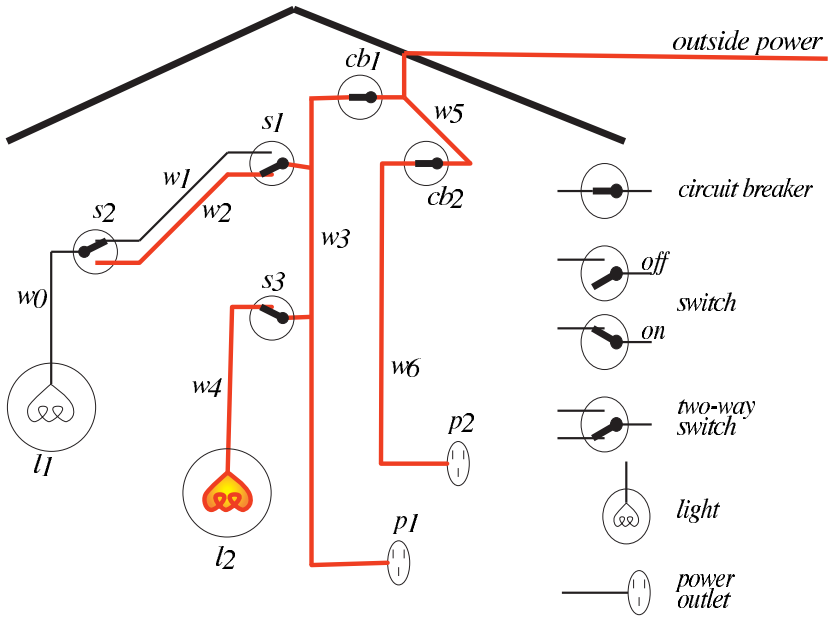
*egg=whole*
*butter_in=pan*
*egg_in=table*

# Example: Electrical

Variables:

| | | | |
|---|---|---|---|
| $S_1$_pos | {up, down} | $S_1$_st | {ok, broken, short} |
| $S_2$_pos | {up, down} | $S_2$_st | {ok, broken, short} |
| $W_1$_st | {ok, broken} | $W_1$_current | {yes, no} |
| ... | ... | ... | ... |

One out of many possible worlds:




$S_1$_pos = down
$S_1$_st = ok
$S_2$_pos = up
$S_2$_st = ok
$W_1$_st = ok
$W_1$_current = no
...

# Example: Schedule

Variables:

| | |
|---|---|
| *Teacher_MI* | $\{PD, MJ, TDN\}$ |
| *Time_MI* | $\{$*Mo_m*, *Mo_a*, . . . , *Fr_m*, *Fr_a*$\}$ |
| *Room_MI* | $\{0.2.12, 0.2.13, 0.2.90\}$ |
| *Teacher_AD* | $\{PD, MJ, TDN\}$ |
| *Time_AD* | $\{$*Mo_m*, *Mo_a*, . . . , *Fr_m*, *Fr_a*$\}$ |
| *Room_AD* | $\{0.2.12, 0.2.13, 0.2.90\}$ |

One possible world:

| Mo | Tue | Wed | Thu | Fr |
|---|---|---|---|---|
| | MI, TDN, 0.2.13 | | | |
| | | | AD, PD, 0.2.90 | |

*Teacher_MI*=TDN
*Time_MI*=*Tue_m*
*Room_MI*=0.2.13
*Teacher_AD*=PD
*Time_AD*=*Thu_a*
*Room_AD*=0.2.90

# Constraint Satisfaction Problems

A **constraint** is a condition on the values of variables in a possible world.

## Extensional Constraint Specification

Explicitly list all allowed (or disallowed) combination of values:

| *Teacher_MI* | *Time_MI* | *Room_MI* | *Teacher_AD* | *Time_AD* | *Room_AD* |
|---|---|---|---|---|---|
| PD | *Mo_m* | 0.2.12 | PD | *Mo_a* | 0.2.12 |
| PD | *Mo_m* | 0.2.12 | PD | *Mo_a* | 0.2.13 |
| ... | ... | ... | ... | ... | ... |

*Not* on the list of allowed possible worlds:

| *Teacher_MI* | *Time_MI* | *Room_MI* | *Teacher_AD* | *Time_AD* | *Room_AD* |
|---|---|---|---|---|---|
| PD | *Mo_m* | 0.2.12 | PD | *Mo_m* | 0.2.12 |
| PD | *Mo_m* | 0.2.12 | MJ | *Mo_m* | 0.2.12 |
| ... | ... | ... | ... | ... | ... |

## Intensional Constraint Specification

Use logical expressions:

$$Teacher\_AD = Teacher\_MI \rightarrow Time\_AD \neq Time\_MI$$
$$Time\_AD = Time\_MI \rightarrow Room\_AD \neq Room\_MI$$

# Example: Sudoku

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A1 | A2 | 1 | A4 | A5 | A6 | A7 | A8 | A9 |
| B1 | B2 | 2 | B4 | 3 | B6 | B7 | B8 | 4 |
| C1 | C2 | C3 | 5 | C5 | C6 | 6 | C8 | 7 |
| 5 | D2 | D3 | 1 | 4 | D6 | D7 | D8 | D9 |
| E1 | 7 | E3 | E4 | E5 | E6 | E7 | 2 | E9 |
| F1 | F2 | F3 | F4 | 7 | 8 | F7 | F8 | 9 |
| 8 | G2 | 7 | G4 | G5 | 9 | G7 | G8 | G9 |
| 4 | H2 | H3 | H4 | 6 | H6 | 3 | H8 | H9 |
| I1 | I2 | I3 | I4 | I5 | I6 | 5 | I8 | I9 |

Constraints:

$$A1 = 2 \vee A2 = 2 \vee A4 = 2 \vee A5 = 2 \vee A6 = 2 \vee A7 = 2 \vee A8 = 2 \vee A9 = 2$$
$$A1 = 3 \vee A2 = 3 \vee A4 = 3 \vee A5 = 3 \vee A6 = 3 \vee A7 = 3 \vee A8 = 3 \vee A9 = 3$$
$$\dots$$
$$A1 = 3 \vee A2 = 3 \vee B1 = 3 \vee B2 = 3 \vee C1 = 3 \vee C2 = 3 \vee C3 = 3$$
$$\dots$$

# Constraint Satisfaction Problem

A **Constraint Satisfaction Problem (CSP)** is given by

- a set of variables
- a set of constraints (usually intensional)

A **solution** to a CSP consists of a possible world that satisfies all the constraints (also called a **model** of the constraints).

Other tasks:

- Determine the number of models of the constraints
- Find an *optimal* model (given also a value function on possible worlds).
- ...

# CSP as State Space Problem

A CSP can be represented as a state space problem:

- States are all partial assignments of values to variables that are consistent with the constraints
- For a state $s$: select some variable $V$ not assigned a value in $s$, and let the neighbors of $s$ be all states that assign a value to $V$ (if any exist).
- The start state is the state that does not assign any values
- A goal state is a state that assigns values to all variables

# CSP as State Space Problem

A CSP can be represented as a state space problem:

- States are all partial assignments of values to variables that are consistent with the constraints
- For a state $s$: select some variable $V$ not assigned a value in $s$, and let the neighbors of $s$ be all states that assign a value to $V$ (if any exist).
- The start state is the state that does not assign any values
- A goal state is a state that assigns values to all variables

## Solving the CSP

- A solution to the state space problem is a path with a goal state at the end: a solution to the CSP problem
- To solve the state space problem need only be able to:
  - enumerate all partial assignments that assign a value to one more variable than $s$
  - check whether a partial assignment is consistent with the constraints

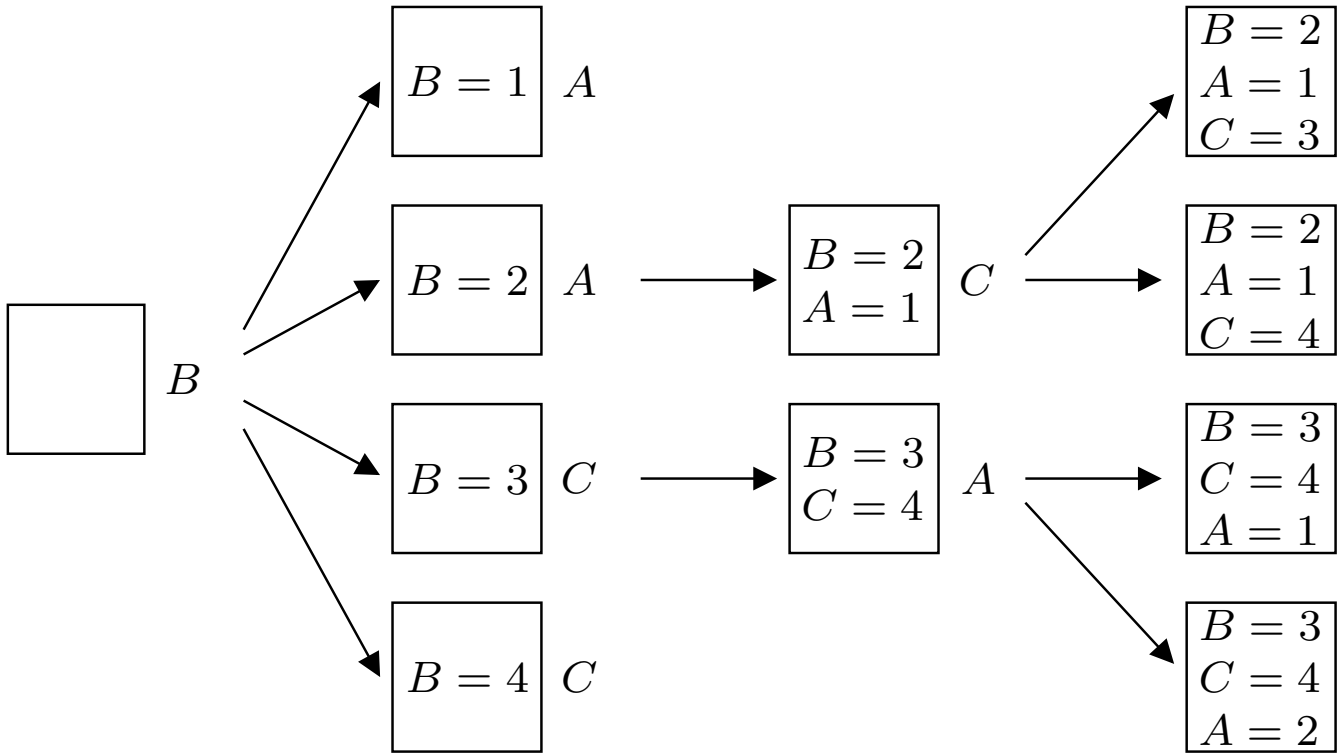
  (that is sufficient to implement the *get_neighbors* and *goal* functions needed in the generic search algorithm)

# Example [PM 4.13]

Variables $A, B, C$; all with domain $\{1, 2, 3, 4\}$.

Constraints: $A < B$, $B < C$.

State Space Graph (showing at each node which variable was selected to generate the neighbors):




- The state space graph is a tree (= search tree)

# Consistency Algorithms

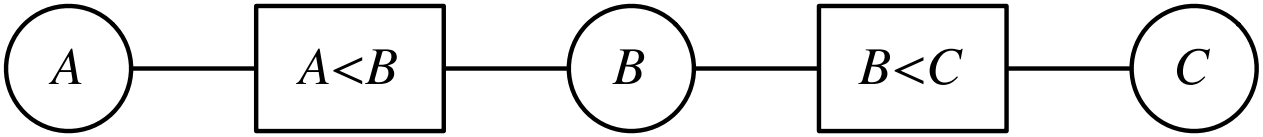
**Example:** Variables $A, B, C$; all with domain $\{1, 2, 3, 4\}$.

Constraints: $A < B$, $B < C$.

**Observation:** There is no solution with $A = 4$.

**Approach** to solving CSPs: iteratively eliminate value assignments that cannot be part of a solution.

# Constraint Network




The **constraint network** for a CSP consists of

- One (oval) node for each variable $X$
- One (rectangular) node for each constraint $c$
- An (undirected) arc $\langle X, c \rangle$ between every constraint and every variable involved in the constraint

With each variable node $X$ is associated a **(reduced) domain** $D_X$:

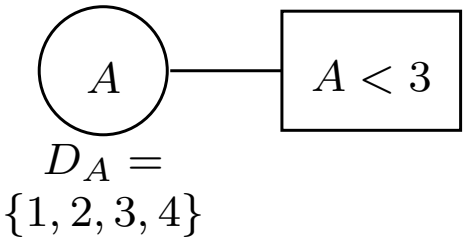
- Initially the domain of the variable
- Reduced by successively deleting values that cannot be part of a solution

An arc $\langle X, c \rangle$ is **arc consistent**, if

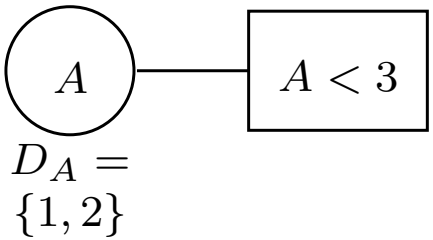
- for all $x \in D_X$ there exists values $y_1, \ldots, y_k$ for the other variables involved in $c$, such that $x, y_1, \ldots, y_k$ is consistent with $c$.

A constraint network is **arc consistent**, if all its arcs are arc consistent.

## Examples

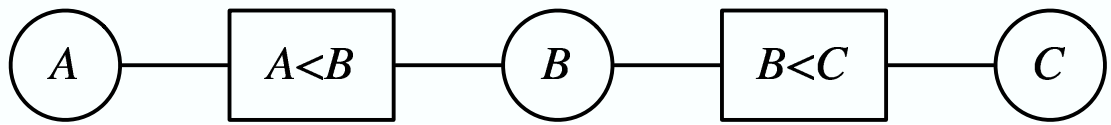



Not arc consistent




Arc consistent

# Arc Consistency Examples

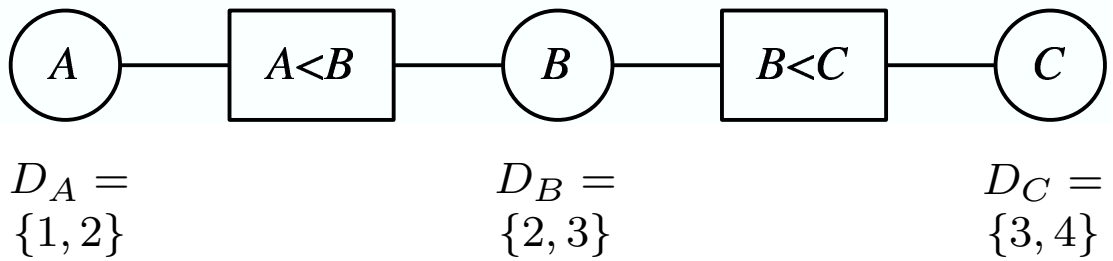



$D_A = \{1, 2\}$

$D_B = \{2, 3\}$

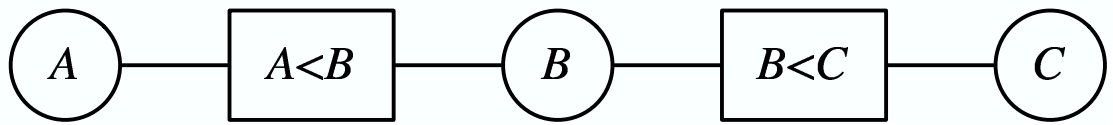
$D_C = \{3, 4\}$

# Arc Consistency Examples




Arc consistent. Not every combination of values from $D_A, D_B, D_C$ is a solution!

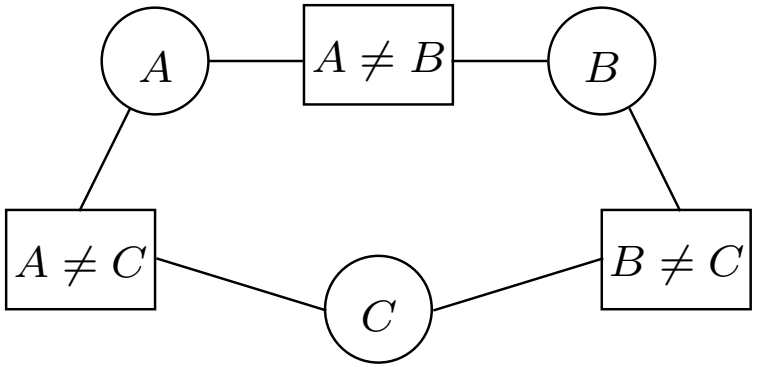
# Arc Consistency Examples




$D_A = \{1, 2\}$ $D_B = \{2, 3\}$ $D_C = \{3, 4\}$

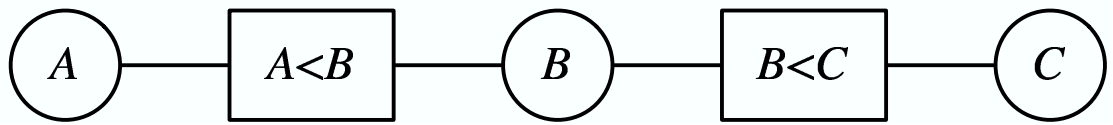
Arc consistent. Not every combination of values from $D_A, D_B, D_C$ is a solution!

$D_A = \{0, 1\}$ $D_B = \{0, 1\}$




$D_C = \{0, 1\}$

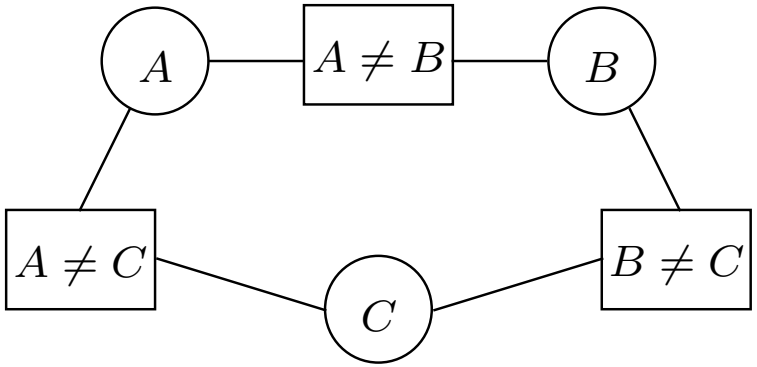
# Arc Consistency Examples




$D_A = \{1, 2\}$ $D_B = \{2, 3\}$ $D_C = \{3, 4\}$

Arc consistent. Not every combination of values from $D_A, D_B, D_C$ is a solution!

$D_A = \{0, 1\}$ $D_B = \{0, 1\}$




$D_C = \{0, 1\}$

Arc consistent. There exists no solution!

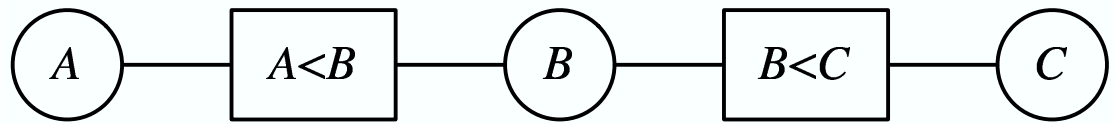
## Algorithm Outline

1. *To-do-arcs*= all arcs in constraint network // Potentially inconsistent arcs
2. **while** *To-do-arcs* $\neq \emptyset$
3. select and delete one arc $\langle X, c \rangle$ from *To-do-arcs*
4. make arc consistent by deleting values from $D_X$, if necessary
5. **if** values were deleted: add all other arcs $\langle Z, c' \rangle$ ($c \neq c'$, $X \in dom(c')$) to *To-do-arcs*

## Algorithm Outline

1. *To-do-arcs*= all arcs in constraint network // Potentially inconsistent arcs
2. **while** *To-do-arcs* $\neq \emptyset$
3. select and delete one arc $\langle X, c \rangle$ from *To-do-arcs*
4. make arc consistent by deleting values from $D_X$, if necessary
5. **if** values were deleted: add all other arcs $\langle Z, c' \rangle$ ($c \neq c'$, $X \in dom(c')$) to *To-do-arcs*

## Example



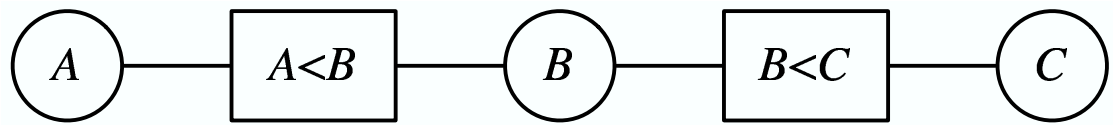

$D_A = \{1, 2, 3, 4\}$ $D_B = \{1, 2, 3, 4\}$ $D_C = \{1, 2, 3, 4\}$

- *To-do-arcs* $= \{\langle A, A < B \rangle, \langle B, A < B \rangle, \langle B, B < C \rangle, \langle C, B < C \rangle\}$
- Selecting $\langle A, A < B \rangle$: For $A = 4$, no value of $B$ satisfies $4 < B$.
  - Remove $\langle A, A < B \rangle$ from *To-do-arcs*.
  - Update $D_A$.

## Algorithm Outline

1. *To-do-arcs*= all arcs in constraint network // Potentially inconsistent arcs
2. **while** *To-do-arcs* $\neq \emptyset$
3. select and delete one arc $\langle X, c \rangle$ from *To-do-arcs*
4. make arc consistent by deleting values from $D_X$, if necessary
5. **if** values were deleted: add all other arcs $\langle Z, c' \rangle$ ($c \neq c'$, $X \in dom(c')$) to *To-do-arcs*

## Example



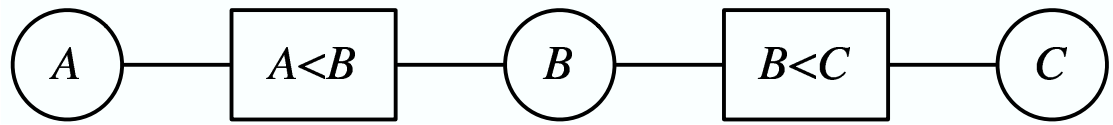

$D_A = \{1, 2, 3\}$ $D_B = \{1, 2, 3, 4\}$ $D_C = \{1, 2, 3, 4\}$

- *To-do-arcs* $= \{\langle B, A < B \rangle, \langle B, B < C \rangle, \langle C, B < C \rangle\}$

## Algorithm Outline

1. *To-do-arcs*= all arcs in constraint network // Potentially inconsistent arcs
2. **while** *To-do-arcs* $\neq \emptyset$
3. select and delete one arc $\langle X, c \rangle$ from *To-do-arcs*
4. make arc consistent by deleting values from $D_X$, if necessary
5. **if** values were deleted: add all other arcs $\langle Z, c' \rangle$ ($c \neq c'$, $X \in dom(c')$) to *To-do-arcs*

## Example



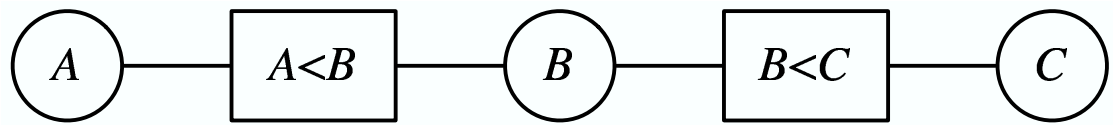

$D_A = \{1, 2, 3\}$ $D_B = \{1, 2, 3, 4\}$ $D_C = \{1, 2, 3, 4\}$

- *To-do-arcs* $= \{\langle B, A < B \rangle, \langle B, B < C \rangle, \langle C, B < C \rangle\}$
- Selecting $\langle B, A < B \rangle$: $B = 1$ can be pruned.
  - Remove $\langle B, A < B \rangle$ from *To-do-arcs*.
  - Update $D_B$.

## Algorithm Outline

1. *To-do-arcs*= all arcs in constraint network // Potentially inconsistent arcs
2. **while** *To-do-arcs* $\neq \emptyset$
3. select and delete one arc $\langle X, c \rangle$ from *To-do-arcs*
4. make arc consistent by deleting values from $D_X$, if necessary
5. **if** values were deleted: add all other arcs $\langle Z, c' \rangle$ ($c \neq c'$, $X \in dom(c')$) to *To-do-arcs*

## Example



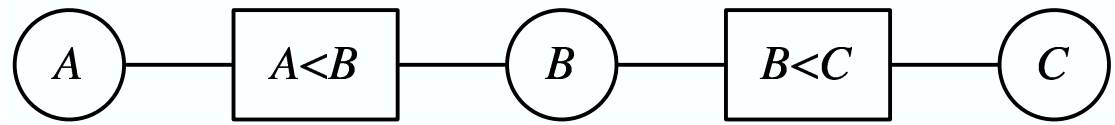

$D_A = \{1, 2, 3\}$ $D_B = \{2, 3, 4\}$ $D_C = \{1, 2, 3, 4\}$

- *To-do-arcs* $= \{\langle B, B < C \rangle, \langle C, B < C \rangle\}$

## Algorithm Outline

1. *To-do-arcs*= all arcs in constraint network // Potentially inconsistent arcs
2. **while** *To-do-arcs* $\neq \emptyset$
3. select and delete one arc $\langle X, c\rangle$ from *To-do-arcs*
4. make arc consistent by deleting values from $D_X$, if necessary
5. **if** values were deleted: add all other arcs $\langle Z, c'\rangle$ ($c \neq c'$, $X \in dom(c')$) to *To-do-arcs*

## Example



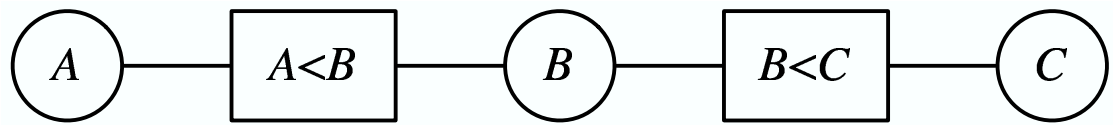

$D_A = \{1, 2, 3\}$ $D_B = \{2, 3, 4\}$ $D_C = \{1, 2, 3, 4\}$

- *To-do-arcs* $= \{\langle B, B < C\rangle, \langle C, B < C\rangle\}$
- Selecting $\langle B, B < C\rangle$: $B = 4$ can be pruned.
  - Add $\langle A, A < B\rangle$ to *To-do-arcs* and remove $\langle B, B < C\rangle$.
  - Update $D_B$.

## Algorithm Outline

1. *To-do-arcs*= all arcs in constraint network // Potentially inconsistent arcs
2. **while** *To-do-arcs* $\neq \emptyset$
3. select and delete one arc $\langle X, c \rangle$ from *To-do-arcs*
4. make arc consistent by deleting values from $D_X$, if necessary
5. **if** values were deleted: add all other arcs $\langle Z, c' \rangle$ ($c \neq c'$, $X \in dom(c')$) to *To-do-arcs*

## Example



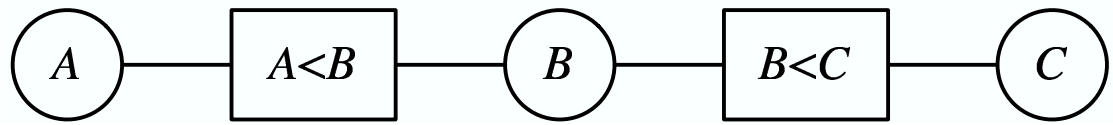

$D_A = \{1, 2, 3\}$ $D_B = \{2, 3\}$ $D_C = \{1, 2, 3, 4\}$

- *To-do-arcs* $= \{\langle A, A < B \rangle, \langle C, B < C \rangle\}$

## Algorithm Outline

1. *To-do-arcs*= all arcs in constraint network // Potentially inconsistent arcs
2. **while** *To-do-arcs* $\neq \emptyset$
3. select and delete one arc $\langle X, c \rangle$ from *To-do-arcs*
4. make arc consistent by deleting values from $D_X$, if necessary
5. **if** values were deleted: add all other arcs $\langle Z, c' \rangle$ ($c \neq c'$, $X \in dom(c')$) to *To-do-arcs*

## Example



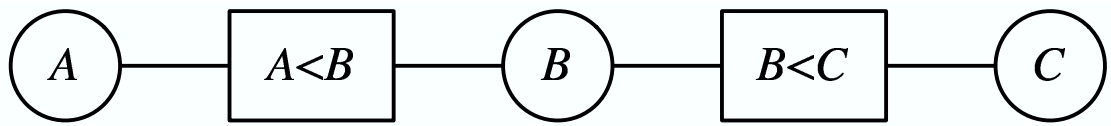

$D_A = \{1, 2, 3\}$ $D_B = \{2, 3\}$ $D_C = \{1, 2, 3, 4\}$

- *To-do-arcs* $= \{\langle A, A < B \rangle, \langle C, B < C \rangle\}$
- Selecting $\langle A, A < B \rangle$: $A = 3$ can be pruned.
  - Remove $\langle A, A < B \rangle$ from *To-do-arcs*.
  - Update $D_A$.

## Algorithm Outline

1. *To-do-arcs*= all arcs in constraint network // Potentially inconsistent arcs
2. **while** *To-do-arcs* $\neq \emptyset$
3. select and delete one arc $\langle X, c \rangle$ from *To-do-arcs*
4. make arc consistent by deleting values from $D_X$, if necessary
5. **if** values were deleted: add all other arcs $\langle Z, c' \rangle$ ($c \neq c'$, $X \in dom(c')$) to *To-do-arcs*

## Example



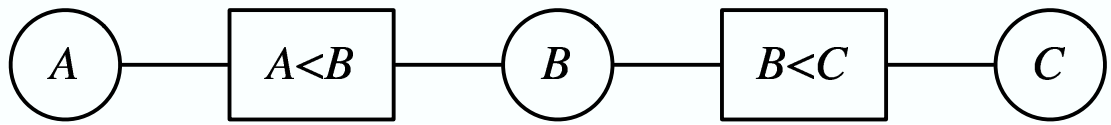

$D_A = \{1, 2\}$ $D_B = \{2, 3\}$ $D_C = \{1, 2, 3, 4\}$

- *To-do-arcs* $= \{\langle C, B < C \rangle\}$

## Algorithm Outline

1. *To-do-arcs*= all arcs in constraint network // Potentially inconsistent arcs
2. **while** *To-do-arcs* $\neq \emptyset$
3. select and delete one arc $\langle X, c \rangle$ from *To-do-arcs*
4. make arc consistent by deleting values from $D_X$, if necessary
5. **if** values were deleted: add all other arcs $\langle Z, c' \rangle$ ($c \neq c'$, $X \in dom(c')$) to *To-do-arcs*

## Example



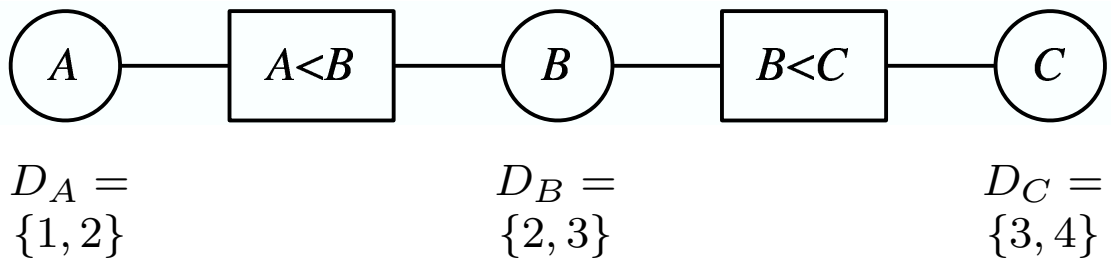

$D_A = \{1, 2\}$ $D_B = \{2, 3\}$ $D_C = \{1, 2, 3, 4\}$

- *To-do-arcs* $= \{\langle C, B < C \rangle\}$
- Selecting $\langle C, B < C \rangle$: $C = 1$ and $C = 2$ can be pruned.
  - Remove $\langle B, B < C \rangle$ from *To-do-arcs*.
  - Update $D_C$.

## Algorithm Outline

1. *To-do-arcs*= all arcs in constraint network // Potentially inconsistent arcs
2. **while** *To-do-arcs* $\neq \emptyset$
3. select and delete one arc $\langle X, c \rangle$ from *To-do-arcs*
4. make arc consistent by deleting values from $D_X$, if necessary
5. **if** values were deleted: add all other arcs $\langle Z, c' \rangle$ ($c \neq c'$, $X \in dom(c')$) to *To-do-arcs*

## Example




- *To-do-arcs* $= \{\}$
- DONE!

## Algorithm Outline

1. *To-do-arcs*= all arcs in constraint network // Potentially inconsistent arcs
2. **while** *To-do-arcs* $\neq \emptyset$
3. select and delete one arc $\langle X, c \rangle$ from *To-do-arcs*
4. make arc consistent by deleting values from $D_X$, if necessary
5. **if** values were deleted: add all other arcs $\langle Z, c' \rangle$ ($c \neq c'$, $X \in dom(c')$) to *To-do-arcs*

## Algorithm Outcomes

Algorithm is guaranteed to terminate. Result independent of order in which arcs are processed. Possible cases at termination:

- $D_X = \emptyset$ for some $X$: CSP has no solution
- $D_X$ contains exactly one value for each $X$: CSP has unique solution, given by the $D_X$ values.
- Other: if the CSP has a solution, then the solution can only consist of current $D_X$ values.

# Variable Elimination

# Idea

- Arc Consistency: simplify problem by eliminating values
- Variable Elimination: simplify problem by eliminating variables

# Relational Operations

Variable Elimination operates on extensional (table) representations of constraints:

$A < B$:

| $A$ | $B$ |
|---|---|
| 1 | 2 |
| 1 | 3 |
| 1 | 4 |
| 2 | 3 |
| 2 | 4 |
| 3 | 4 |

$B < C$:

| $B$ | $C$ |
|---|---|
| 1 | 2 |
| 1 | 3 |
| 1 | 4 |
| 2 | 3 |
| 2 | 4 |
| 3 | 4 |

Algorithm requires **projection** and **join** operations on tables.

# Projection

Projection of a table:

| Course | Year | Student | Grade |
|---|---|---|---|
| cs322 | 2008 | fran | 77 |
| cs111 | 2009 | billie | 88 |
| cs111 | 2009 | jess | 78 |
| cs444 | 2008 | fran | 83 |
| cs322 | 2009 | jordan | 92 |

$\xrightarrow{\pi \{ Student, Year \}}$

| Student | Year |
|---|---|
| fran | 2008 |
| billie | 2009 |
| jess | 2009 |
| jordan | 2009 |

Given two tables $r_1, r_2$ for variables $vars_1, vars_2$. The **join** is the table $r_3 = r_1 \bowtie r_2$ for variables $vars_1 \cup vars_2$ that

- contains all tuples, which restricted to $vars_1$ are in $r_1$, and restricted to $vars_2$ are in $r_2$.

**Example**

| Course | Year | Student | Grade |
|---|---|---|---|
| cs322 | 2008 | fran | 77 |
| cs111 | 2009 | billie | 88 |
| cs111 | 2009 | jess | 78 |
| cs444 | 2008 | fran | 83 |
| cs322 | 2009 | jordan | 92 |

$\bowtie$

| Course | Year | TA |
|---|---|---|
| cs322 | 2008 | yuki |
| cs111 | 2009 | sam |
| cs111 | 2009 | chris |
| cs322 | 2009 | yuki |

=

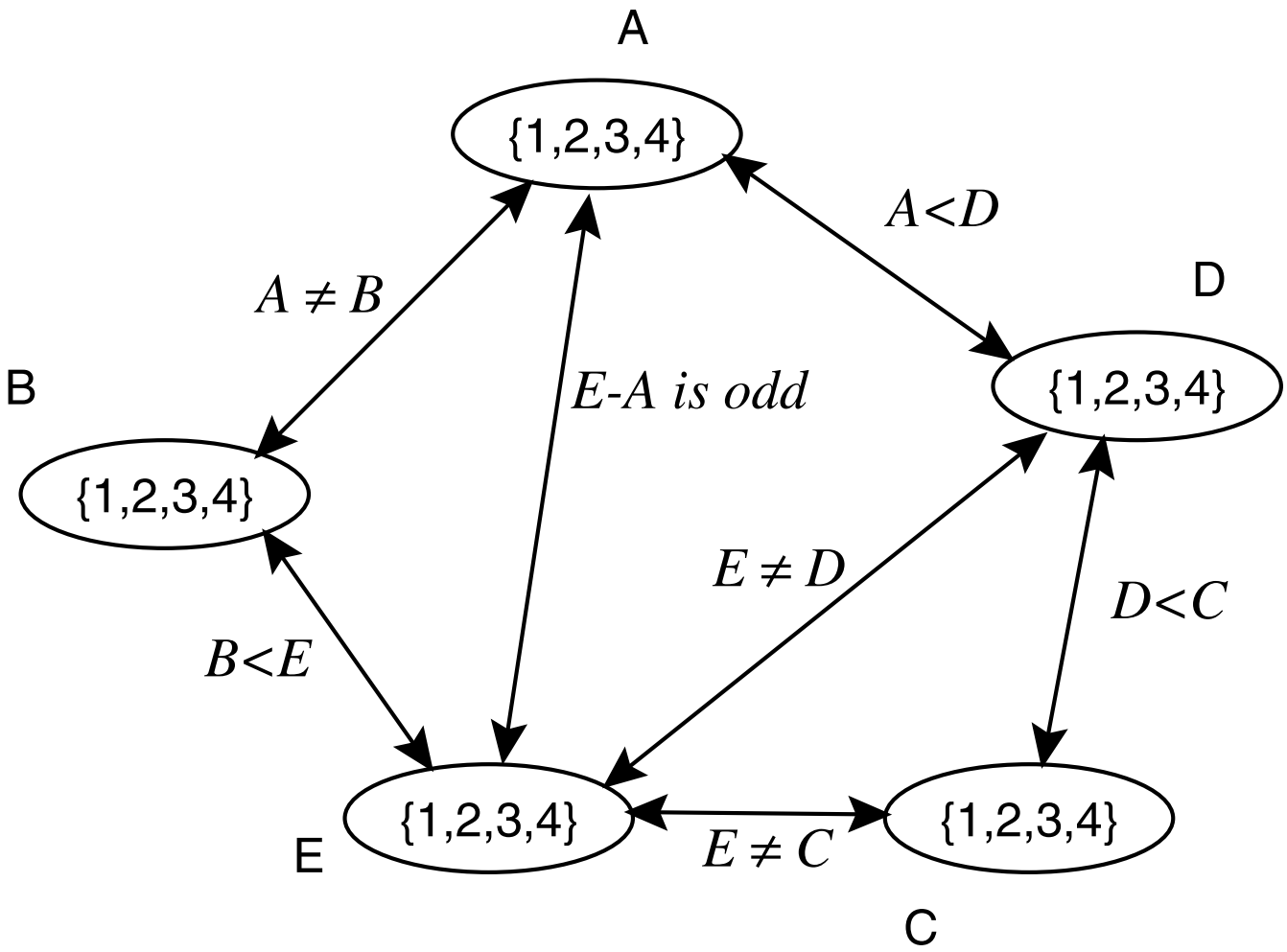
| Course | Year | Student | Grade | TA |
|---|---|---|---|---|
| cs322 | 2008 | fran | 77 | yuki |
| cs111 | 2009 | billie | 88 | sam |
| cs111 | 2009 | jess | 78 | sam |
| cs111 | 2009 | billie | 88 | chris |
| cs111 | 2009 | jess | 78 | chris |
| cs322 | 2009 | jordan | 92 | yuki |

## Algorithm Outline

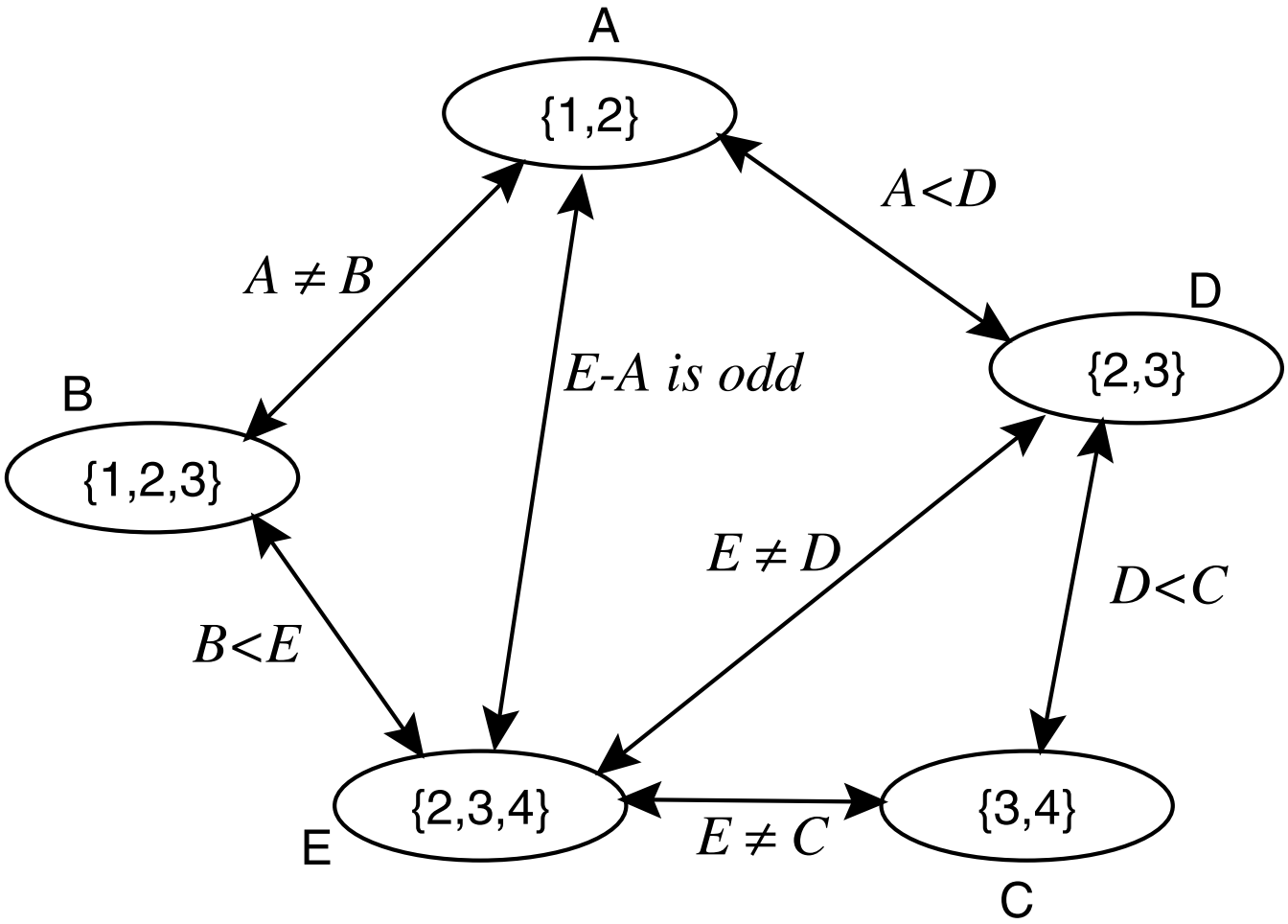
1. **Input:** $C$: set of constraints on variables *vars*
2. **while** $C$ contains more than one element
3. select a variable $X \in$ *vars*
4. delete $X$ from *vars*
5. remove all constraints involving $X$ from $C$ and construct their join
6. **if** *vars* is not empty
7. project the join onto the variables other than $X$
8. add the (projected) join to $C$

Intuition: the constraint constructed in line 5. summarizes the effect that all the constraints involving $X$ have on variables other than $X$.

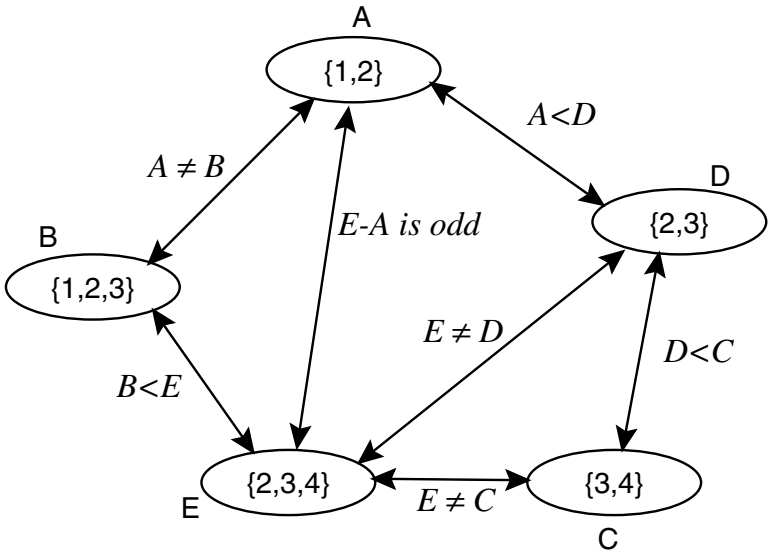
# Example network



**. . . now arc-consistent**



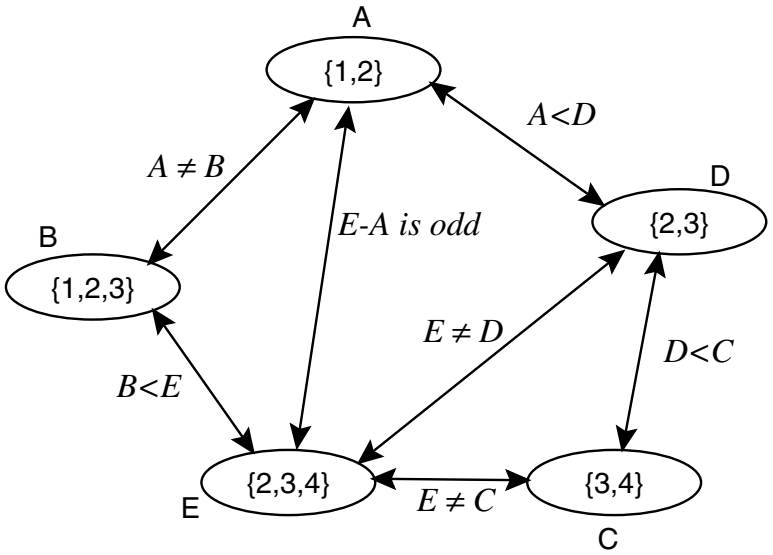
## Example: eliminating $C$




| $r_1 : C \neq E$ | $C$ | $E$ |
|---|---|---|
| | 3 | 2 |
| | 3 | 4 |
| | 4 | 2 |
| | 4 | 3 |

| $r_2 : C > D$ | $C$ | $D$ |
|---|---|---|
| | 3 | 2 |
| | 4 | 2 |
| | 4 | 3 |

## Example: eliminating $C$

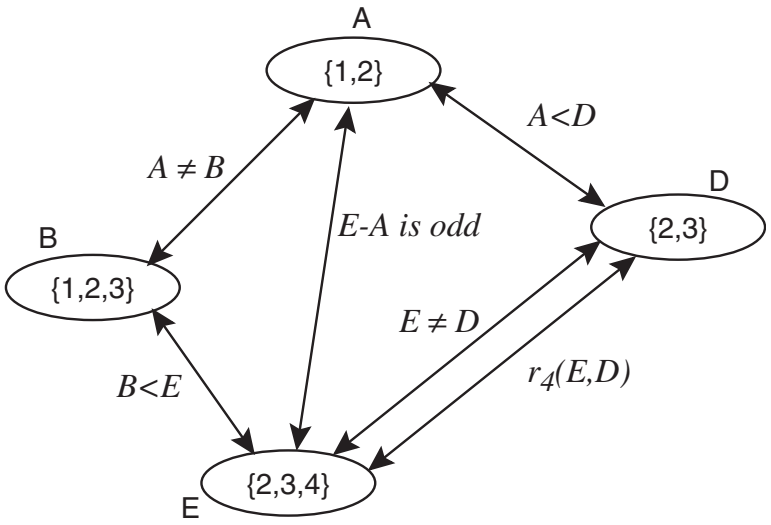



| $r_1 : C \neq E$ | $C$ | $E$ |
|---|---|---|
| | 3 | 2 |
| | 3 | 4 |
| | 4 | 2 |
| | 4 | 3 |

| $r_2 : C > D$ | $C$ | $D$ |
|---|---|---|
| | 3 | 2 |
| | 4 | 2 |
| | 4 | 3 |

| $r_3 : r_1 \bowtie r2$ | $C$ | $D$ | $E$ |
|---|---|---|---|
| | 3 | 2 | 2 |
| | 3 | 2 | 4 |
| | 4 | 2 | 2 |
| | 4 | 2 | 3 |
| | 4 | 3 | 2 |
| | 4 | 3 | 3 |

## Example: eliminating $C$




| $r_1 : C \neq E$ | $C$ | $E$ |
|---|---|---|
| | 3 | 2 |
| | 3 | 4 |
| | 4 | 2 |
| | 4 | 3 |

| $r_2 : C > D$ | $C$ | $D$ |
|---|---|---|
| | 3 | 2 |
| | 4 | 2 |
| | 4 | 3 |

| $r_3 : r_1 \bowtie r2$ | $C$ | $D$ | $E$ |
|---|---|---|---|
| | 3 | 2 | 2 |
| | 3 | 2 | 4 |
| | 4 | 2 | 2 |
| | 4 | 2 | 3 |
| | 4 | 3 | 2 |
| | 4 | 3 | 3 |

| $r_4 : \pi_{\{D,E\}} r_3$ | $D$ | $E$ |
|---|---|---|
| | 2 | 2 |
| | 2 | 3 |
| | 2 | 4 |
| | 3 | 2 |
| | 3 | 3 |

$\hookrightarrow$ new constraint

## Properties

- The algorithm terminates
- The CSP has a solution if and only if the final constraint is non-empty
- The set of all solutions can be generated by joining the final constraint with the intermediate "summarizing" constraints generated in line 5.
- Algorithm operates on extensional constraint representations, therefore
  - constraints must not contain too many tuples (initial and constructed constraints)
- Worst case: VE is not more efficient than enumerating all possible worlds and checking whether they are solutions.

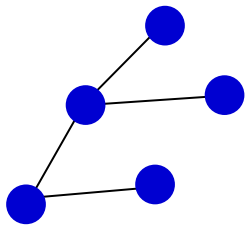
## Properties

- The algorithm terminates
- The CSP has a solution if and only if the final constraint is non-empty
- The set of all solutions can be generated by joining the final constraint with the intermediate "summarizing" constraints generated in line 5.
- Algorithm operates on extensional constraint representations, therefore
  - constraints must not contain too many tuples (initial and constructed constraints)
- Worst case: VE is not more efficient than enumerating all possible worlds and checking whether they are solutions.

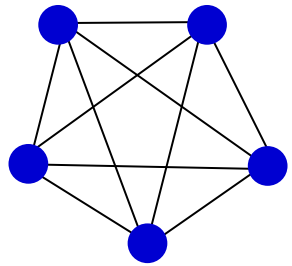
## Constraint Graph

Consider the graph where

- there is one node for each variable
- two variables are connected when they appear together in one constraint



*sparse*



*dense*

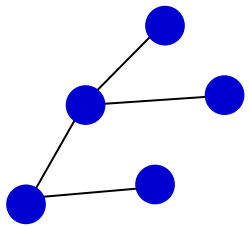
## Properties

- The algorithm terminates
- The CSP has a solution if and only if the final constraint is non-empty
- The set of all solutions can be generated by joining the final constraint with the intermediate "summarizing" constraints generated in line 5.
- Algorithm operates on extensional constraint representations, therefore
  - constraints must not contain too many tuples (initial and constructed constraints)
- Worst case: VE is not more efficient than enumerating all possible worlds and checking whether they are solutions.

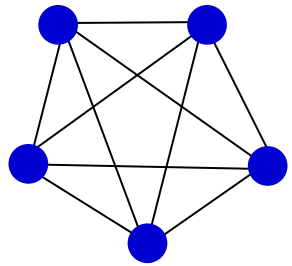
## Constraint Graph

Consider the graph where

- there is one node for each variable
- two variables are connected when they appear together in one constraint



*sparse*



*dense*

Then: VE will work better if the constraint graph is sparsely connected

# Local Search

# Systematic vs. Local Search

So far: all methods systematically explored the state space (possible worlds).

Problem: Time and space when search space is large.

Local Search approach:

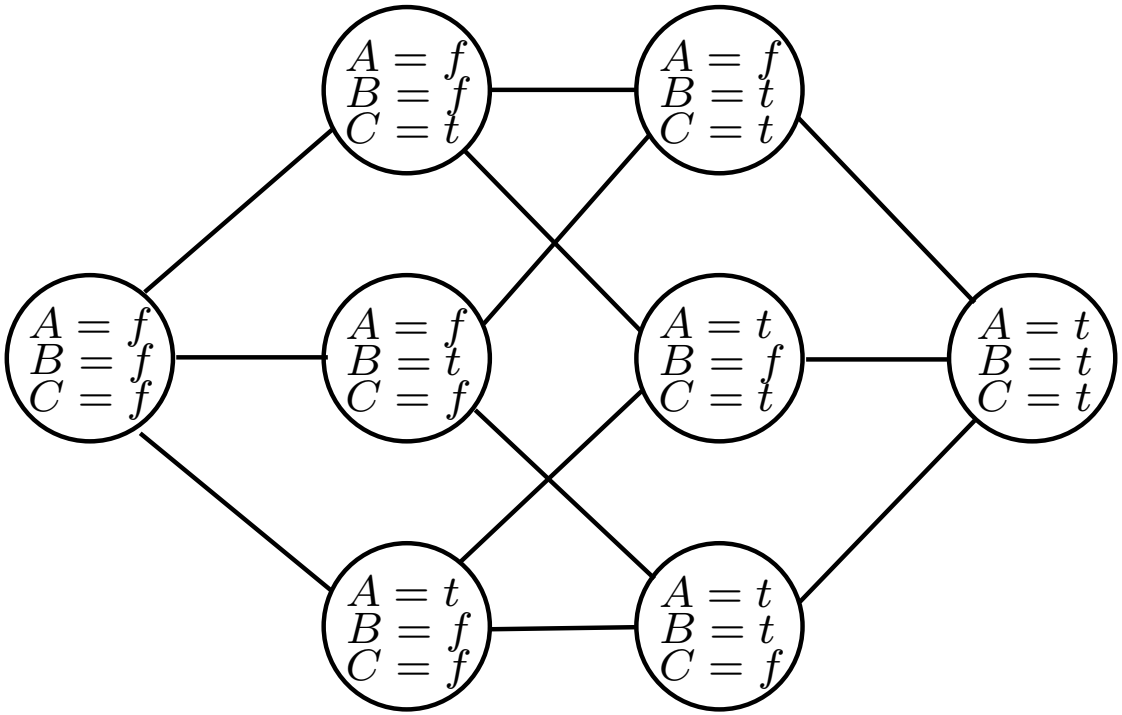
- explore state space without 'bookkeeping' (where have we been? what still needs to be explored?).
- no success/termination guarantees
- in practice, often the only thing that works

# State Space Graph for CSP

(Another) state space graph representation for CSPs:

- Nodes are possible worlds
- Neighbors are possible worlds that differ in the value of exactly one variable

State space graph for 3 boolean variables:

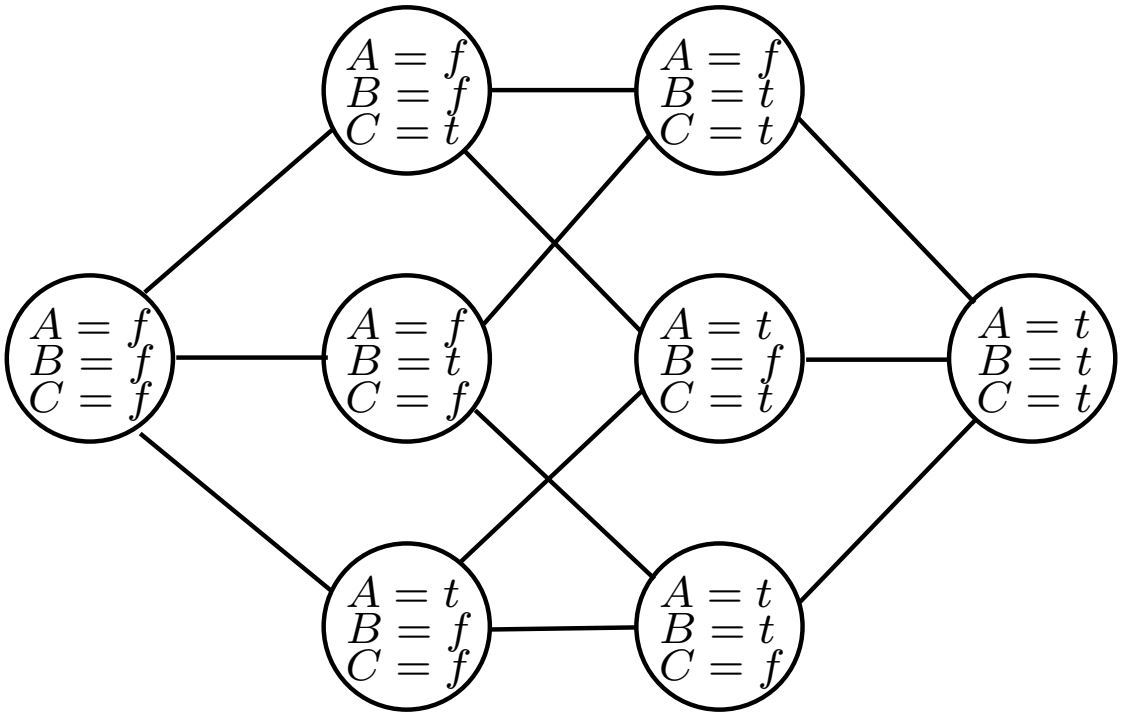


## Algorithm Outline

1. Select some node in state space graph as *current state*
2. **while** *current state* is not a solution
3. *current state* = some neighbor of *current state*

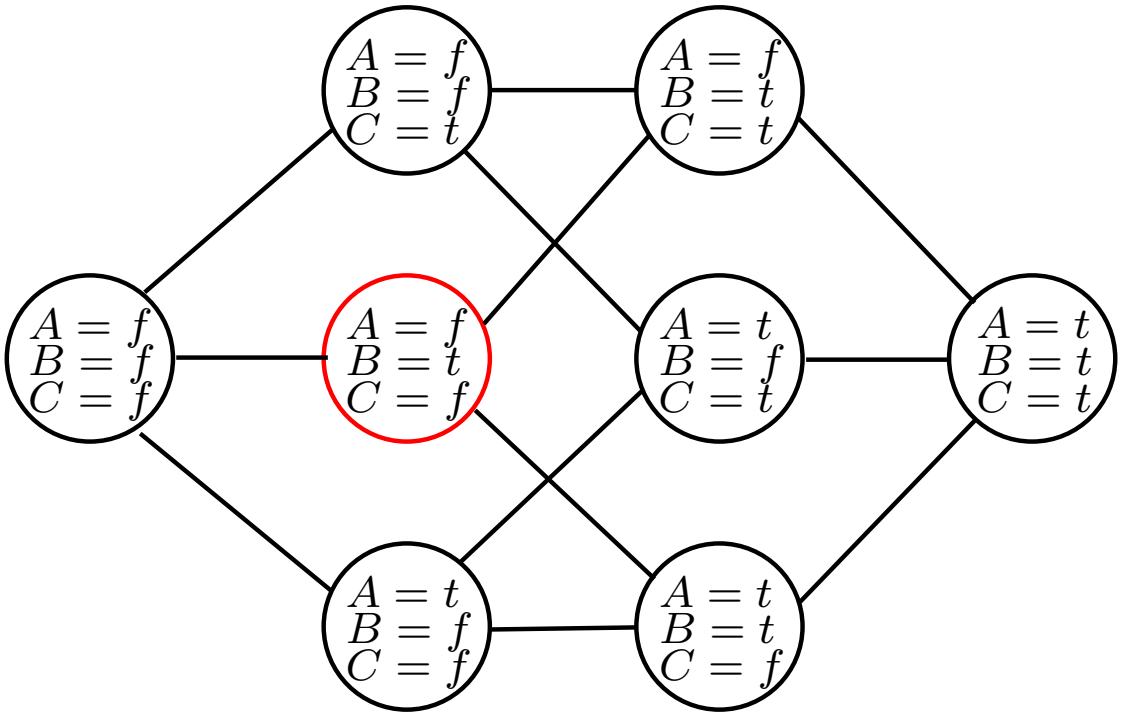
## Algorithm Outline

1.Select some node in state space graph as *current state*
2.**while** *current state* is not a solution
3. *current state* = some neighbor of *current state*



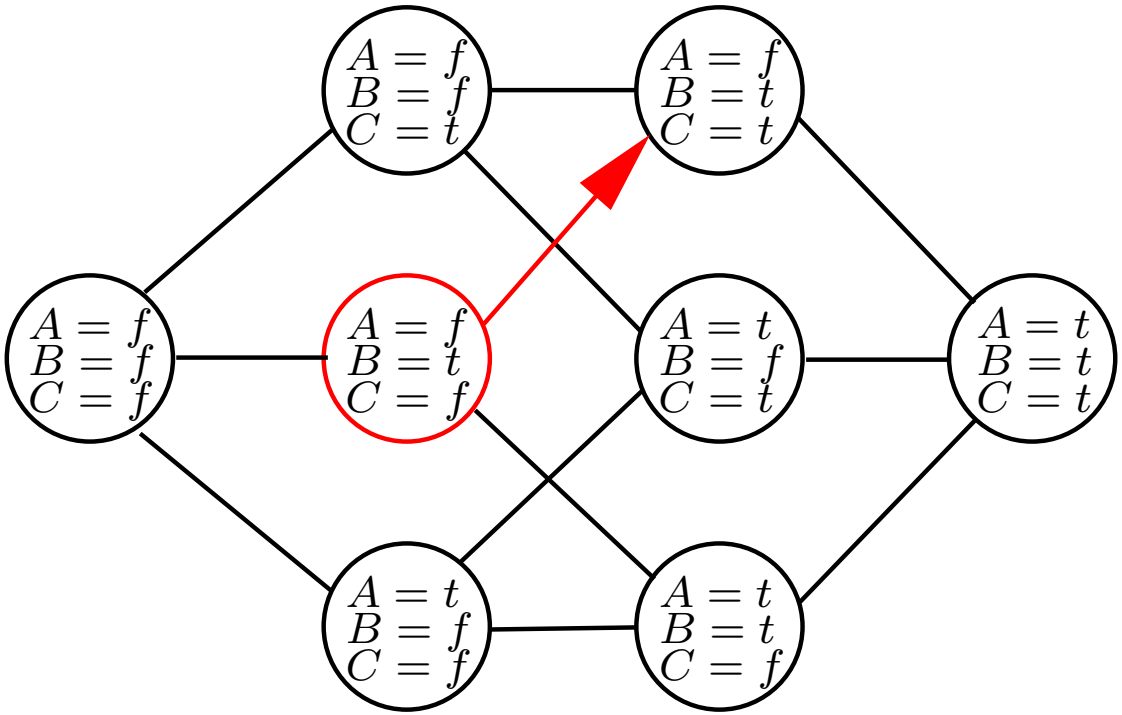
## Algorithm Outline

1. Select some node in state space graph as *current state*
2. **while** *current state* is not a solution
3. *current state* = some neighbor of *current state*



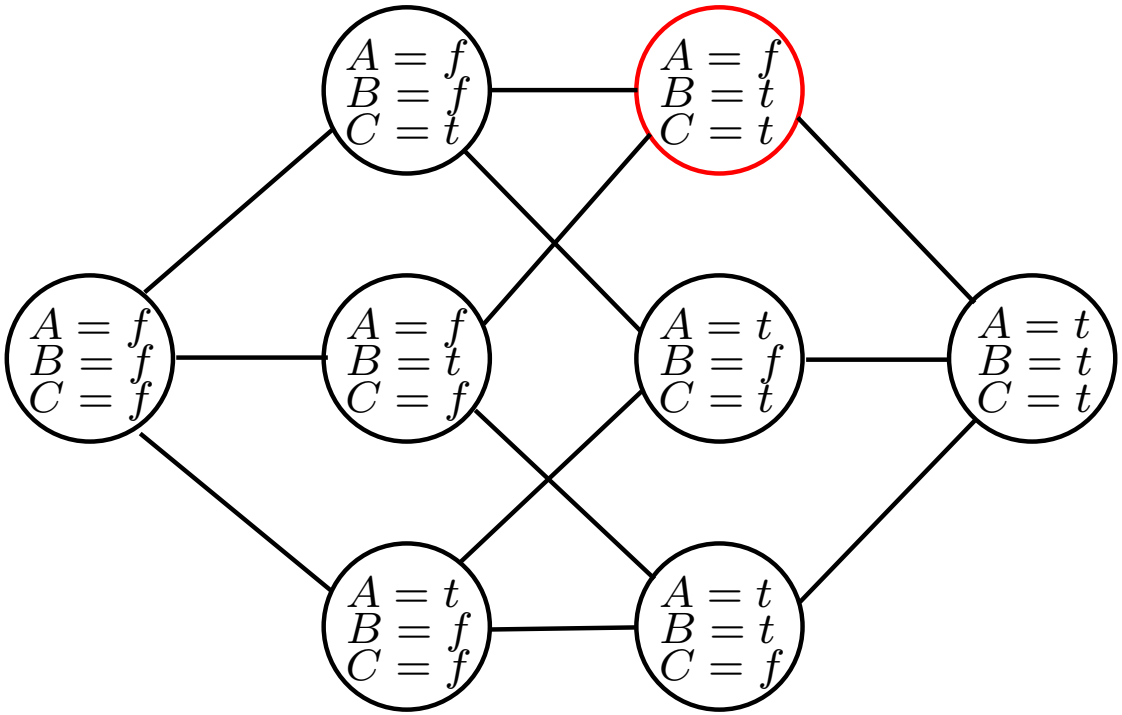
## Algorithm Outline

1.Select some node in state space graph as *current state*
2.**while** *current state* is not a solution
3. *current state* = some neighbor of *current state*



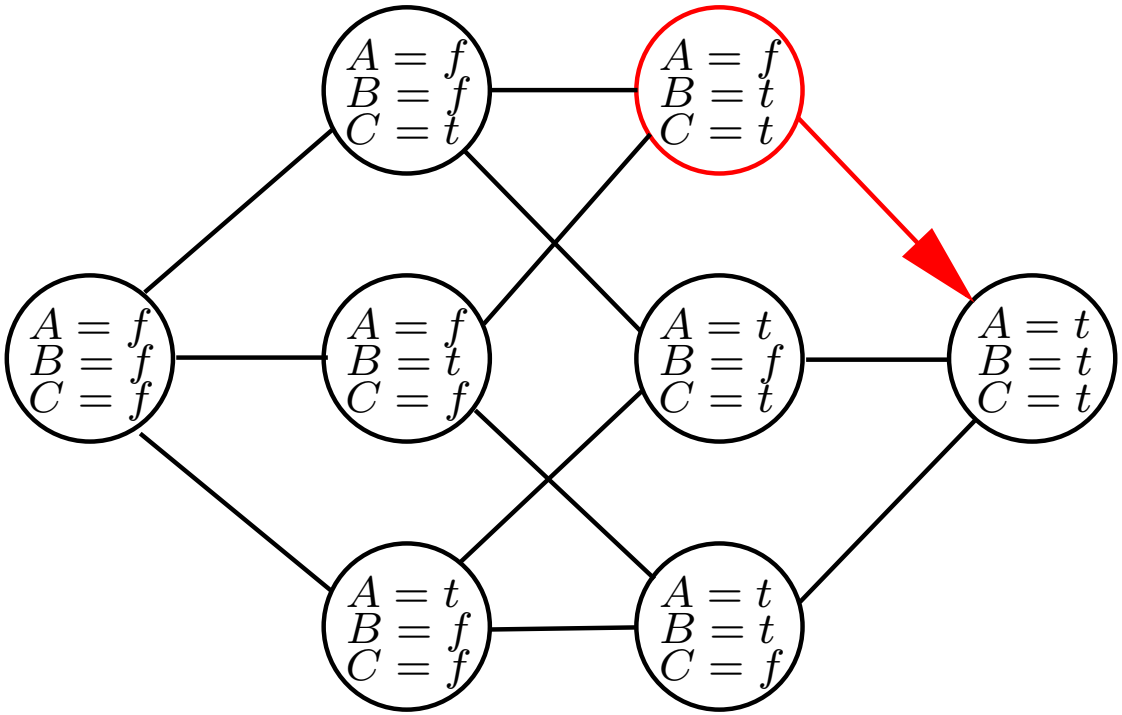
## Algorithm Outline

1.Select some node in state space graph as *current state*
2.**while** *current state* is not a solution
3. *current state* = some neighbor of *current state*



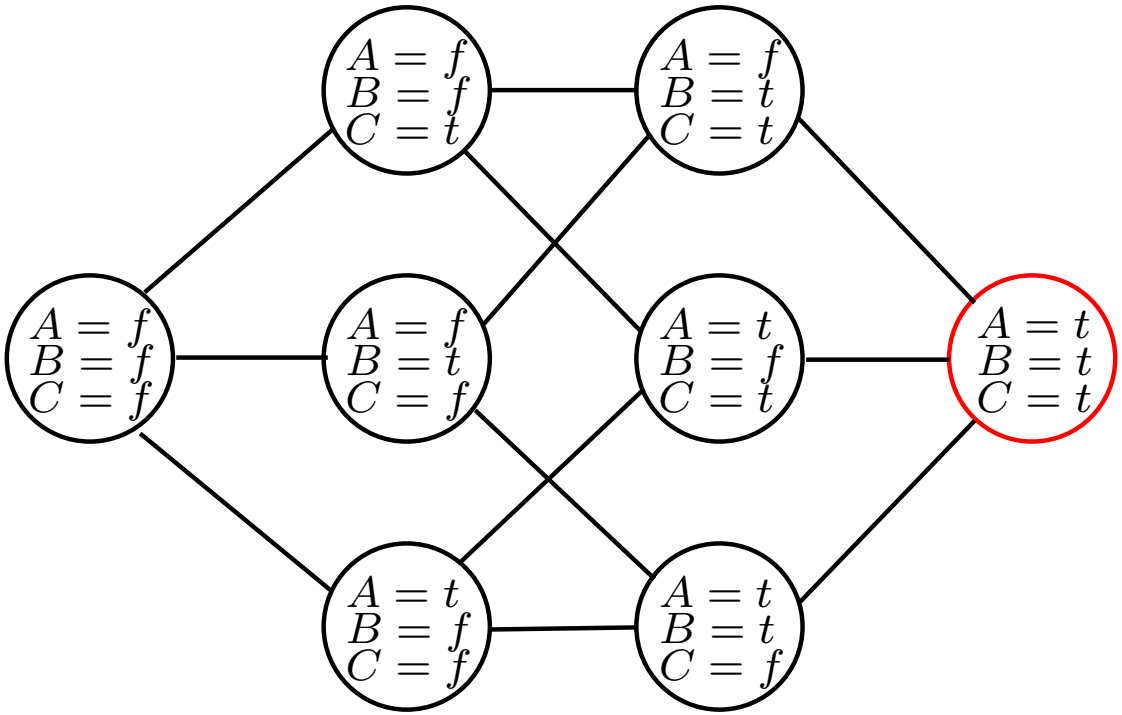
## Algorithm Outline

1. Select some node in state space graph as *current state*
2. **while** *current state* is not a solution
3. *current state* = some neighbor of *current state*



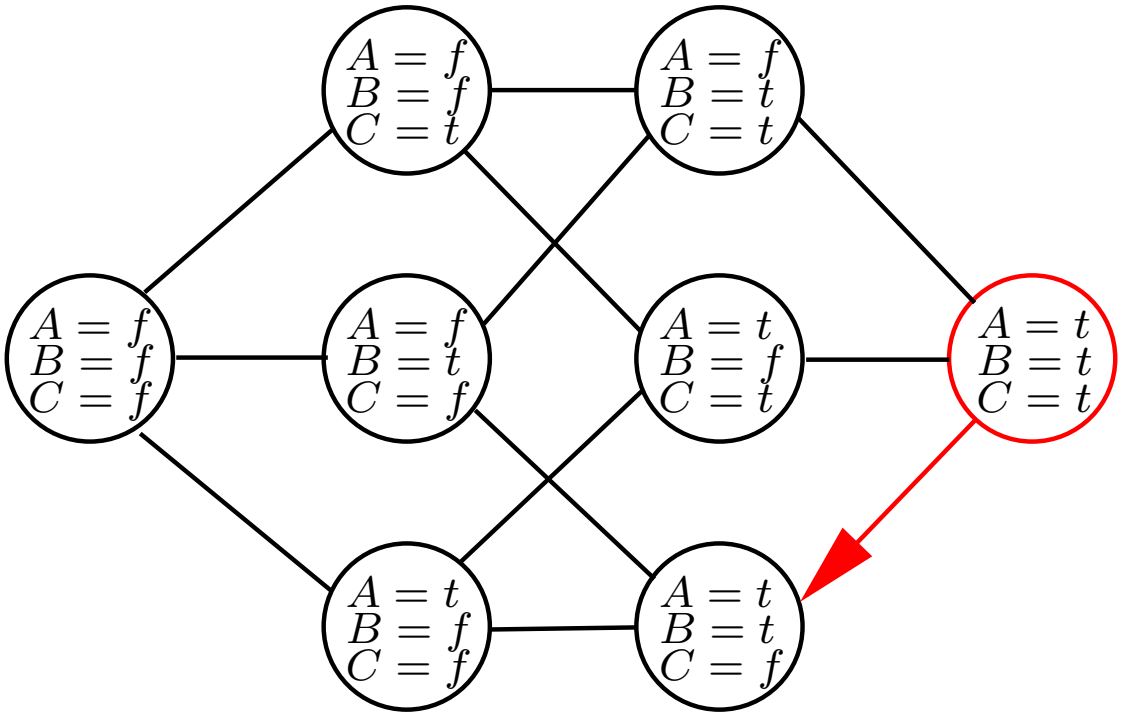
## Algorithm Outline

1.Select some node in state space graph as *current state*
2.**while** *current state* is not a solution
3. *current state* = some neighbor of *current state*



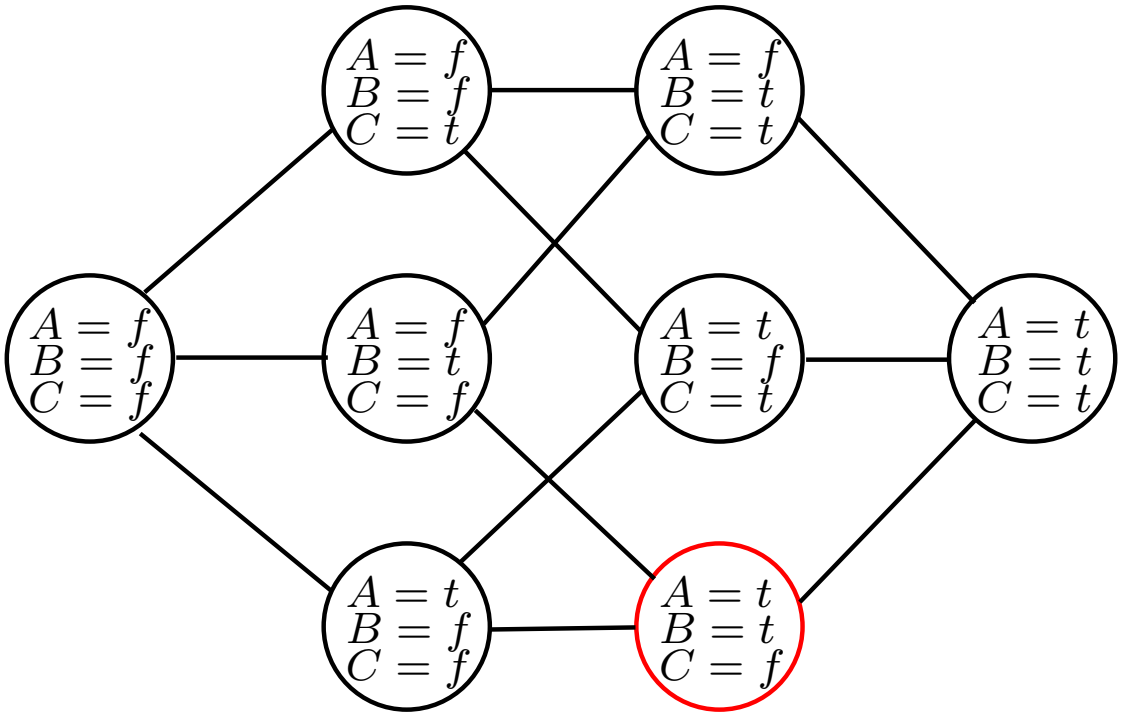
## Algorithm Outline

1. Select some node in state space graph as *current state*
2. **while** *current state* is not a solution
3. *current state* = some neighbor of *current state*



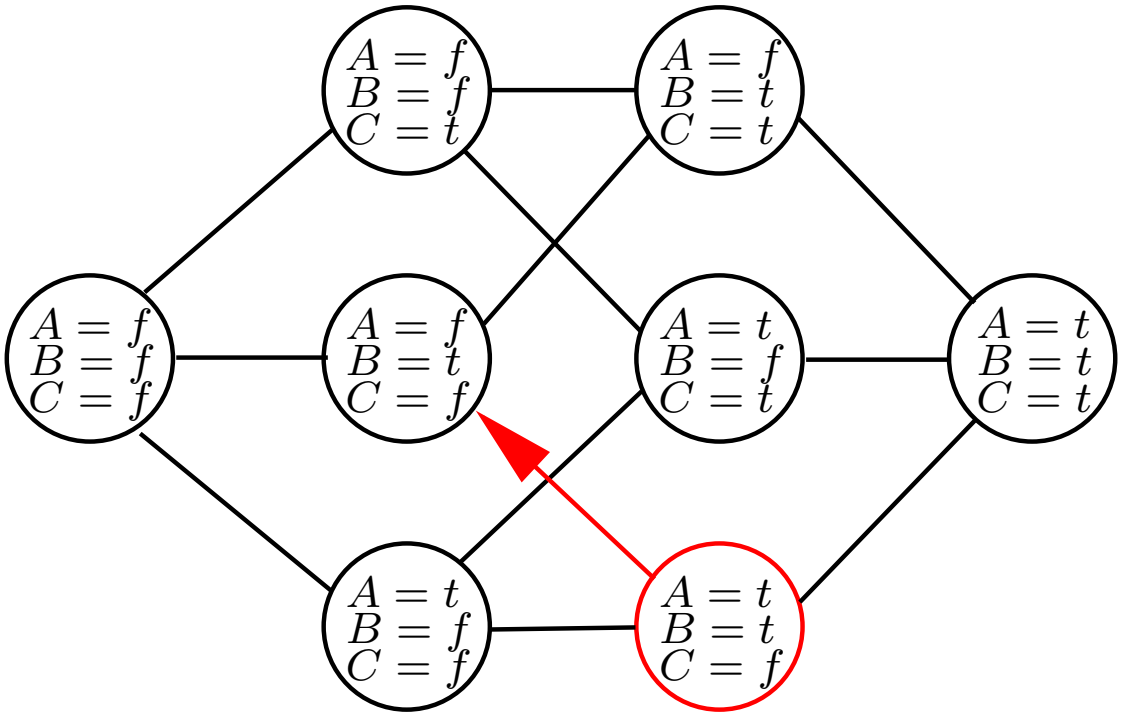
## Algorithm Outline

1.Select some node in state space graph as *current state*
2.**while** *current state* is not a solution
3. *current state* = some neighbor of *current state*



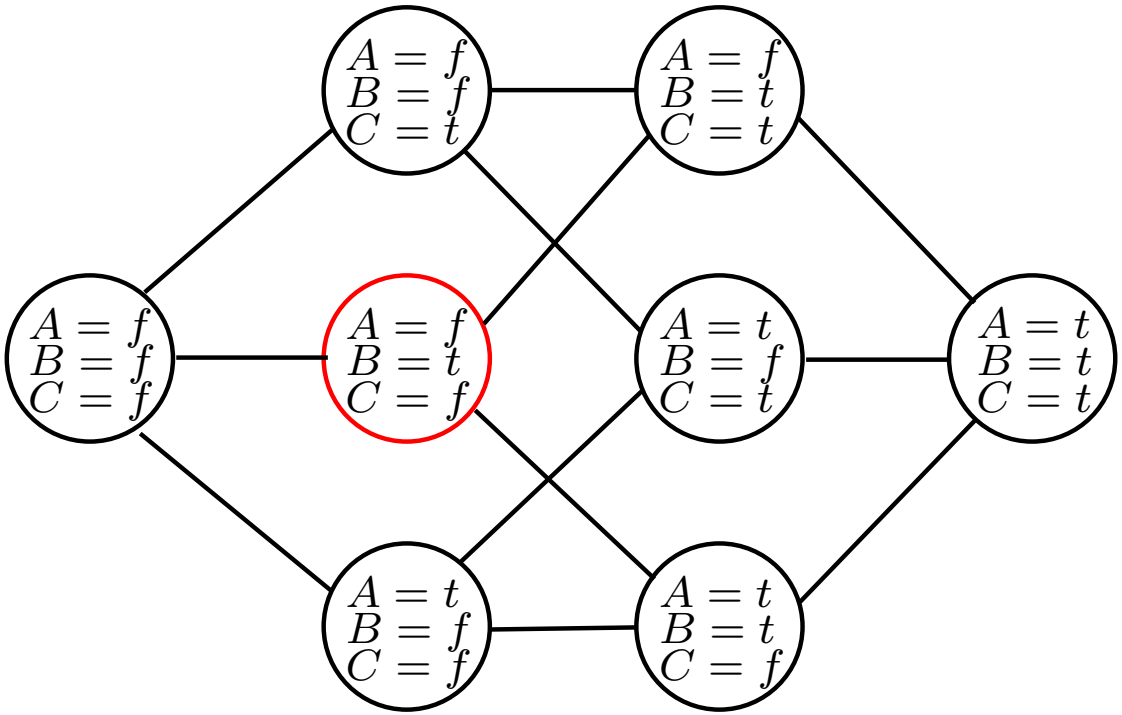
## Algorithm Outline

1. Select some node in state space graph as *current state*
2. **while** *current state* is not a solution
3. *current state* = some neighbor of *current state*



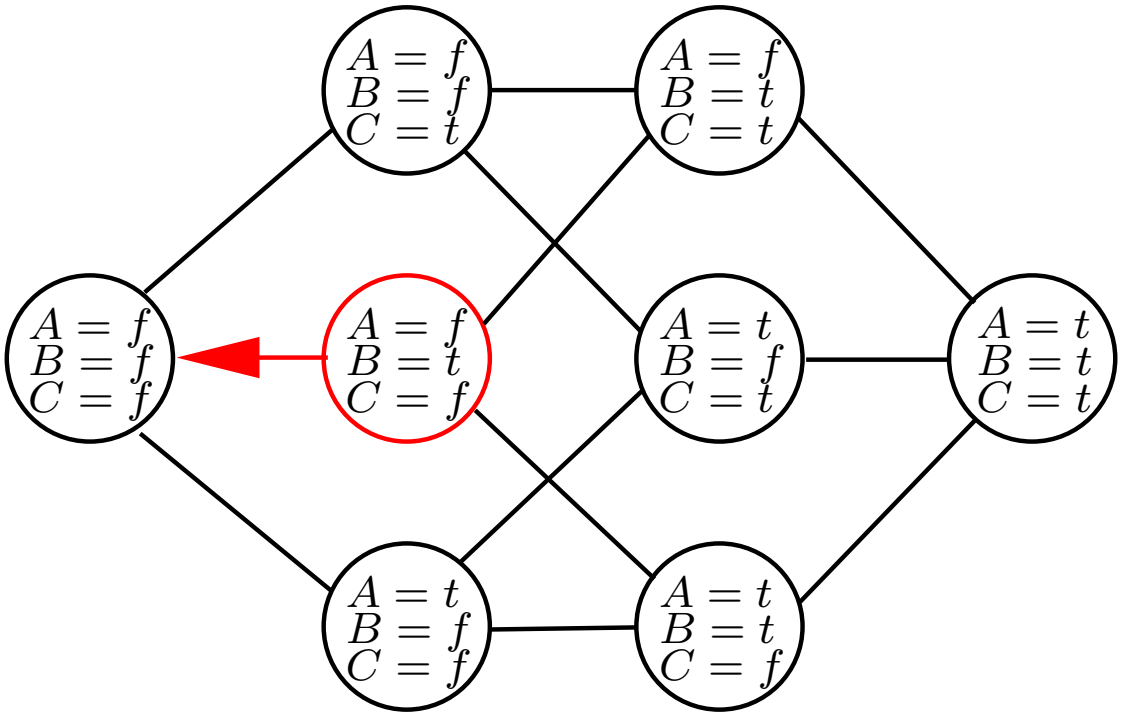
## Algorithm Outline

1. Select some node in state space graph as *current state*
2. **while** *current state* is not a solution
3. *current state* = some neighbor of *current state*



## Algorithm Outline

1.Select some node in state space graph as *current state*
2.**while** *current state* is not a solution
3. *current state* = some neighbor of *current state*



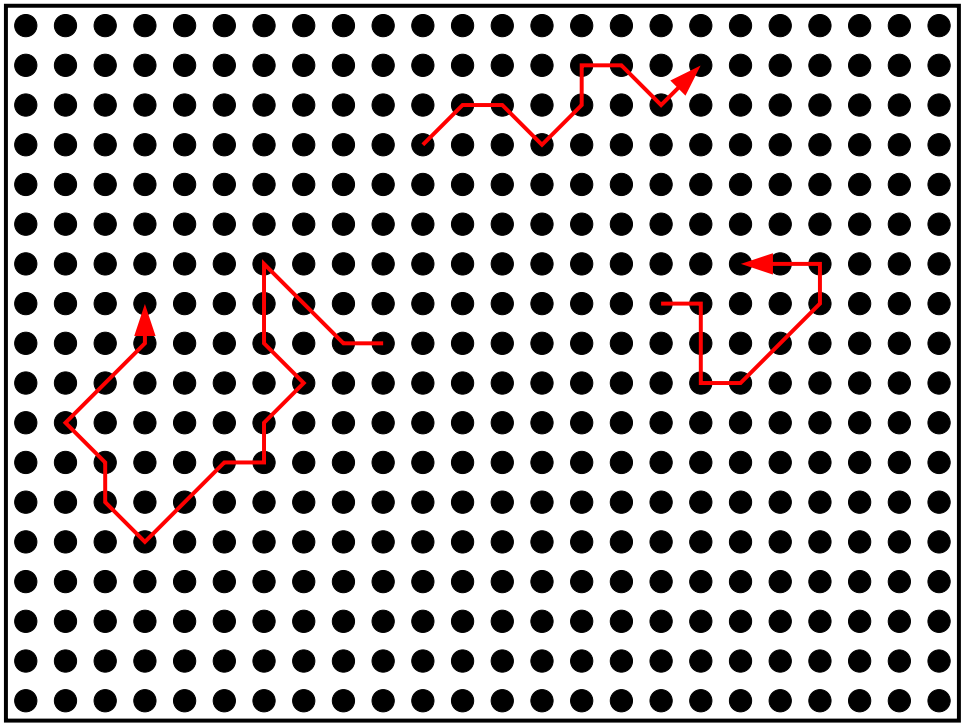
# Random Search

- Make choices in line 1. and 3. completely random
- “Random walk”
- Unlikely to find a solution if state space large with only few solutions

**Greedy Search** or **Hill Climbing**:

- Use an *evaluation function* on states
- Example for evaluation function: number of constraints not satisfied by state
- Always choose neighbor with minimal evaluation function value
- Terminates when all neighbors have higher value than current state: current state is a **local minimum**.

Possible greedy search paths starting from different states:



## Problem

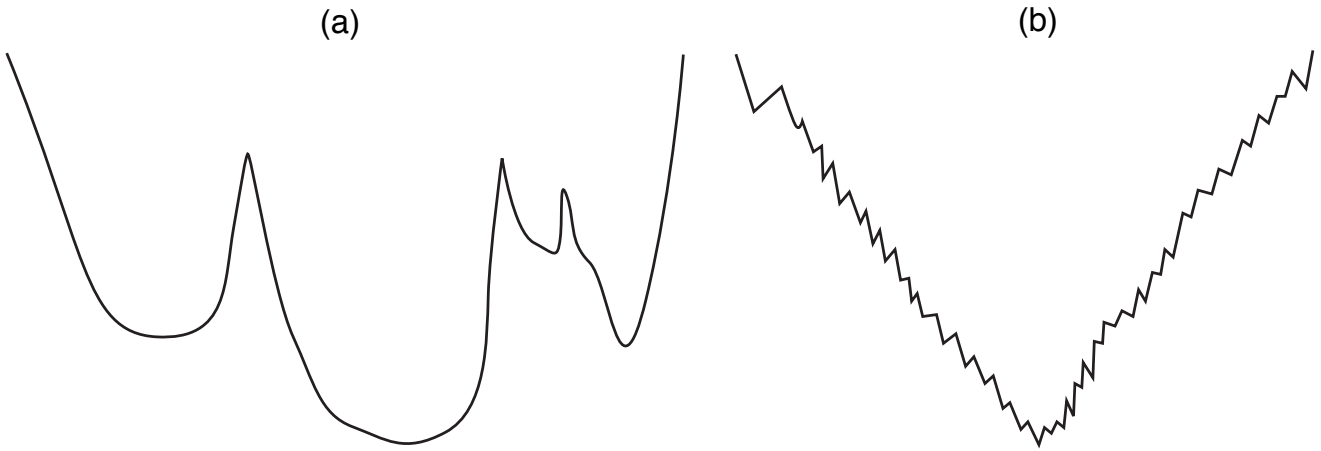
- Search terminates with local minimum of evaluation function. This may not be a solution to the CSP.

## Problem

- Search terminates with local minimum of evaluation function. This may not be a solution to the CSP.

## Solution Approaches

- Random restarts: repeat greedy search with several randomly chosen initial states
- Random moves: combine greedy moves with random steps

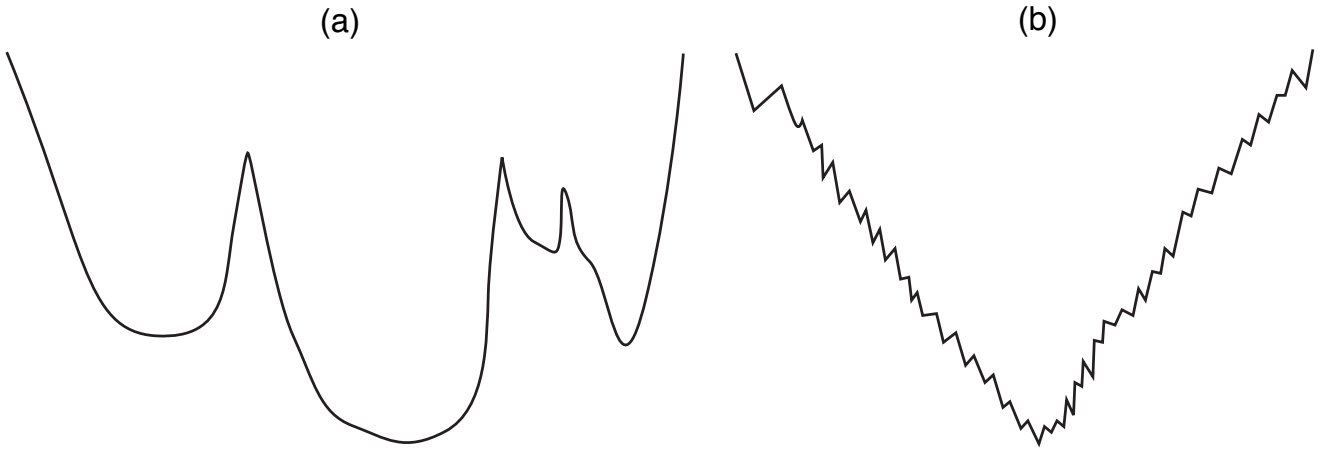


## Problem

- Search terminates with local minimum of evaluation function. This may not be a solution to the CSP.

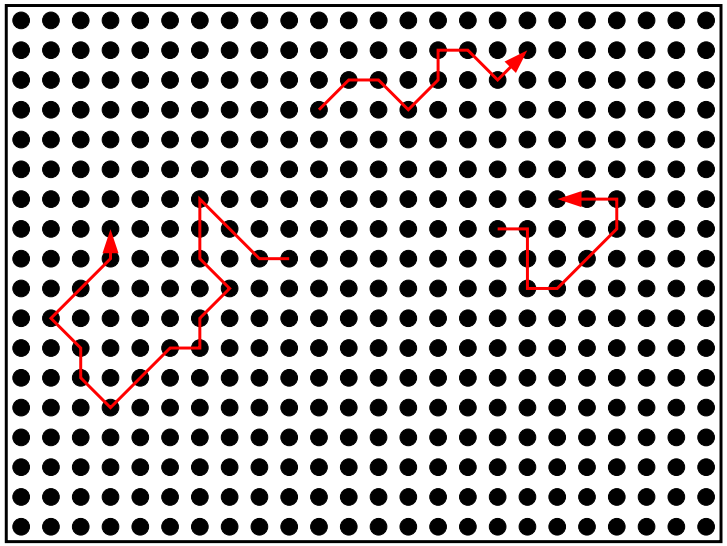
## Solution Approaches

- Random restarts: repeat greedy search with several randomly chosen initial states
- Random moves: combine greedy moves with random steps
- **Example (a)**: Small number of random restarts will find global minimum
- **Example (b)**: Make random move when local minimum reached



# Local search

- Maintain an assignment of a value to each variable.
- At each step, select a “neighbor” of the current assignment (e.g., one that improves some heuristic value).
- Stop when a satisfying assignment is found, or return the best assignment found.



Requires:

- What is a neighbor?
- Which neighbor should be selected?

# Most improving step

## Principle

Select the variable-value pair that gives the highest improvement.

## Naive approach

- Linearly scan all variables and for each value of each variable determine the improvement (how many fewer constrains are violated).

# Most improving step

## Principle

Select the variable-value pair that gives the highest improvement.

## Naive approach

- Linearly scan all variables and for each value of each variable determine the improvement (how many fewer constrains are violated).

## Alternative

- Maintain a priority queue with variable-value pairs not part of the current assignment.
- Weight$\langle X, v \rangle =$ *eval(current assignment)* $-$ *eval(current assignment but with* $X = v$*)*.
- If $X$ is given a new value, update the weight of all pairs participating in a changed constraint.

## Principle

- First: choose variable
- Second: choose state

## Data structure

- Maintain priority queue of variables; weight is the number of participating conflicts.
- After selecting a variable, pick the value minimizes the number of conflicts.
- Update weights of variables that participate in a conflict that is changed.

## Algorithm

- Pick a variable at random and a new value at random.
- If it is an improvement, adopt it.
- If it isn't an improvement, adopt it probabilistically depending on a temperature parameter, $T$.
  - With current assignment $n$ and proposed assignment $n'$ we move to $n'$ with probability

$$e^{(h(n')-h(n))/T}$$

- Reduce the temperature.

## Algorithm

- Pick a variable at random and a new value at random.
- If it is an improvement, adopt it.
- If it isn't an improvement, adopt it probabilistically depending on a temperature parameter, $T$.
  - With current assignment $n$ and proposed assignment $n'$ we move to $n'$ with probability

$$e^{(h(n')-h(n))/T}$$

- Reduce the temperature.

## Probability of accepting a change

| Temperature | 1-worse | 2-worse | 3-worse |
|---|---|---|---|
| 10 | 0.91 | 0.81 | 0.74 |
| 1 | 0.37 | 0.14 | 0.05 |
| 0.25 | 0.02 | 0.0003 | 0.000005 |
| 0.1 | 0.00005 | 0 | 0 |

## Algorithm

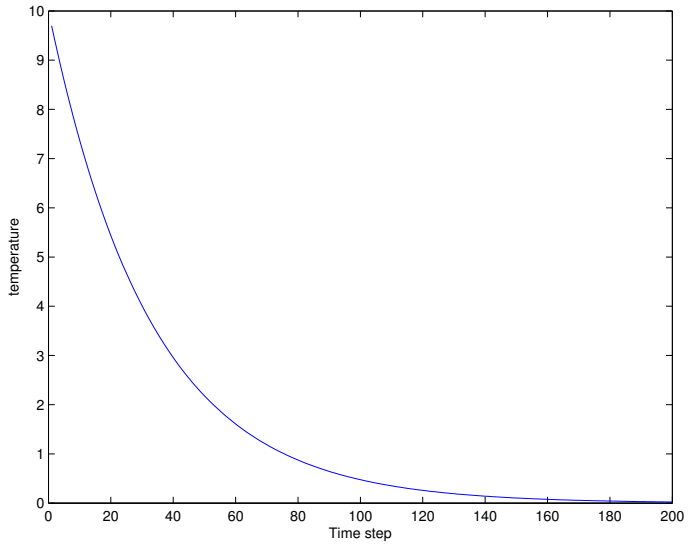
- Pick a variable at random and a new value at random.
- If it is an improvement, adopt it.
- If it isn't an improvement, adopt it probabilistically depending on a temperature parameter, $T$.
  - With current assignment $n$ and proposed assignment $n'$ we move to $n'$ with probability

$$e^{(h(n')-h(n))/T}$$

- Reduce the temperature.

## Probability of accepting a change

| Temperature | 1-worse | 2-worse | 3-worse |
|---|---|---|---|
| 10 | 0.91 | 0.81 | 0.74 |
| 1 | 0.37 | 0.14 | 0.05 |
| 0.25 | 0.02 | 0.0003 | 0.000005 |
| 0.1 | 0.00005 | 0 | 0 |




$$y = 10 \cdot 0.97^x$$

# Propositional Logic Basics

## Previously ...

Intensional representation of constraints:

$$A < B$$
$$\mathit{Teacher\_AD} = \mathit{Teacher\_MI} \rightarrow \mathit{Time\_AD} \neq \mathit{Time\_MI}$$
$$\ldots$$

CSP algorithms (arc-consistency algorithm) need to perform certain operations:

- *test* whether a certain value for one variable is *consistent* with a given constraint (and certain values for other variables)

To implement this:

- need a **formal language for representing constraints**

## Propositional Logic

- provides a formal language for representing constraints on *binary variables*.

## Atomic Propositions

*Boolean variables* are now seen as **atomic propositions**. Convention: start with lowercase letter.

| Constraints | Logic |
|---|---|
| $A = \mathit{true}$ | $a$ |
| $A = \mathit{false}$ | $\neg a$ |

## Propositions

Using **logical connectives** more complex propositions are constructed:

| | |
|---|---|
| $\neg p$ | **not** $p$ |
| $(p \wedge q)$ | $p$ **and** $q$ |
| $(p \vee q)$ | $p$ **or** $q$ |
| $(p \rightarrow q)$ | $p$ **implies** $q$ |

A *set* of propositions is also called a **Knowledge Base**

## Example

"If **it rains** I'll **take my umbrella**, or I'll **stay home**"

## Atomic Propositions

*Boolean variables* are now seen as **atomic propositions**. Convention: start with lowercase letter.

| Constraints | Logic |
|---|---|
| $A = \mathit{true}$ | $a$ |
| $A = \mathit{false}$ | $\neg a$ |

## Propositions

Using **logical connectives** more complex propositions are constructed:

| | |
|---|---|
| $\neg p$ | **not** $p$ |
| $(p \wedge q)$ | $p$ **and** $q$ |
| $(p \vee q)$ | $p$ **or** $q$ |
| $(p \rightarrow q)$ | $p$ **implies** $q$ |

A *set* of propositions is also called a **Knowledge Base**

## Example

"If **it rains** I'll **take my umbrella**, or I'll **stay home**"

$$\mathit{rains} \rightarrow (\mathit{umbrella} \vee \mathit{home})$$

An **interpretation** $\pi$ for a set of atomic propositions $a_1, a_2, \ldots, a_n$ is an assignment of a truth value to each proposition (= possible world when atomic propositions seen as boolean variables):

$$\pi(a_i) \in \{\textit{true}, \textit{false}\}$$

An interpretation defines a truth value for all propositions:

| $\pi(p)$ | $\pi(\neg p)$ |
|---|---|
| *true* | *false* |
| *false* | *true* |

| $\pi(p)$ | $\pi(q)$ | $\pi(p \wedge q)$ |
|---|---|---|
| *true* | *true* | *true* |
| *true* | *false* | *false* |
| *false* | *true* | *false* |
| *false* | *false* | *false* |

| $\pi(p)$ | $\pi(q)$ | $\pi(p \vee q)$ |
|---|---|---|
| *true* | *true* | *true* |
| *true* | *false* | *true* |
| *false* | *true* | *true* |
| *false* | *false* | *false* |

| $\pi(p)$ | $\pi(q)$ | $\pi(p \rightarrow q)$ |
|---|---|---|
| *true* | *true* | *true* |
| *true* | *false* | *false* |
| *false* | *true* | *true* |
| *false* | *false* | *true* |

## Models

A **model** of a proposition (a knowledge base) is an interpretation in which the proposition (all the propositions in the knowledge base) is true.

Propositions as constraints: a model is a possible world that satisfies the constraint.

## Logical consequence

A proposition $g$ is a **logical consequence** of a knowledge base *KB*, if every model of *KB* is a model of $g$. Written:

$$KB \models g$$

(whenever *KB* is true, then $g$ also is true).

## Example

*KB*=$\{man \rightarrow mortal, man\}$. Then

$$KB \models mortal$$

# Simple Example

$$KB = \begin{cases} p \leftarrow q. \\ q. \\ r \leftarrow s. \end{cases}$$

| | $p$ | $q$ | $r$ | $s$ |
|---|---|---|---|---|
| $I_1$ | *true* | *true* | *true* | *true* |
| $I_2$ | *false* | *false* | *false* | *false* |
| $I_3$ | *true* | *true* | *false* | *false* |
| $I_4$ | *true* | *true* | *true* | *false* |
| $I_5$ | *true* | *true* | *false* | *true* |

Model?

# Simple Example

$$KB = \begin{cases} p \leftarrow q. \\ q. \\ r \leftarrow s. \end{cases}$$

| | $p$ | $q$ | $r$ | $s$ | Model? |
|---|---|---|---|---|---|
| $I_1$ | *true* | *true* | *true* | *true* | is a model of $KB$ |
| $I_2$ | *false* | *false* | *false* | *false* | not a model of $KB$ |
| $I_3$ | *true* | *true* | *false* | *false* | is a model of $KB$ |
| $I_4$ | *true* | *true* | *true* | *false* | is a model of $KB$ |
| $I_5$ | *true* | *true* | *false* | *true* | not a model of $KB$ |

# Simple Example

$$KB = \begin{cases} p \leftarrow q. \\ q. \\ r \leftarrow s. \end{cases}$$

| | $p$ | $q$ | $r$ | $s$ | Model? |
|---|---|---|---|---|---|
| $I_1$ | *true* | *true* | *true* | *true* | is a model of $KB$ |
| $I_2$ | *false* | *false* | *false* | *false* | not a model of $KB$ |
| $I_3$ | *true* | *true* | *false* | *false* | is a model of $KB$ |
| $I_4$ | *true* | *true* | *true* | *false* | is a model of $KB$ |
| $I_5$ | *true* | *true* | *false* | *true* | not a model of $KB$ |

Which of $p, q, r, q$ logically follow from KB?

# Simple Example

$$KB = \begin{cases} p \leftarrow q. \\ q. \\ r \leftarrow s. \end{cases}$$

| | $p$ | $q$ | $r$ | $s$ | Model? |
|---|---|---|---|---|---|
| $I_1$ | *true* | *true* | *true* | *true* | is a model of $KB$ |
| $I_2$ | *false* | *false* | *false* | *false* | not a model of $KB$ |
| $I_3$ | *true* | *true* | *false* | *false* | is a model of $KB$ |
| $I_4$ | *true* | *true* | *true* | *false* | is a model of $KB$ |
| $I_5$ | *true* | *true* | *false* | *true* | not a model of $KB$ |

Which of $p, q, r, q$ logically follow from KB?

$$KB \models p,\ KB \models q,\ KB \not\models r,\ KB \not\models s$$