CS 1571 Introduction to AI
Lecture 6

Informed search methods

Milos Hauskrecht
milos@cs.pitt.edu
5329 Sennott Square

Announcements

Homework assignment 2 is out

- Due on Thursday, September 20, 2012 before the class
- **Two parts:**
  - Pen and pencil part
  - Programming part (Puzzle 8): informed search methods

Course web page:

http://www.cs.pitt.edu/~milos/courses/cs1571/

Search methods

- **Uninformed search methods**
  - **Breadth-first search (BFS)**
  - **Depth-first search (DFS)**
  - **Iterative deepening (IDA)**
  - **Bi-directional search**
  - **Uniform cost search**
- **Informed (or heuristic) search methods:**
  - **Best first search with the heuristic function**

CS 1571 Intro to AI M. Hauskrecht

Best-first search

Best-first search

- Driven by the evaluation function $f(n)$ to guide the search.
- incorporates a **heuristic function** $h(n)$ in $f(n)$
- heuristic function measures a potential of a state (node) to reach a goal

Special cases (differ in the design of evaluation function):

- **Greedy search**

$$f(n) = h(n)$$

- **A* algorithm**

$$f(n) = g(n) + h(n)$$

+ **iterative deepening** version of A* : **IDA***

CS 1571 Intro to AI M. Hauskrecht

A* search

- The problem with the **greedy search** is that it can keep expanding paths that are already very expensive.
- The problem with the **uniform-cost search** is that it uses only past exploration information (path cost), no additional information is utilized
- **A* search**

$$f(n) = g(n) + h(n)$$

$g(n)$ - cost of reaching the state

$h(n)$ - estimate of the cost from the current state to a goal

$f(n)$ - estimate of the path length

- **Additional A*condition**: admissible heuristic

$$h(n) \le h^*(n) \quad \text{for all } n$$

Optimality of A*

- In general, a heuristic function $h(n)$:

  Can overestimate, be equal or underestimate the true distance of a node to the goal $h^*(n)$
- **Admissible heuristic condition**
  - **Never overestimate the distance to the goal !!!**

$$h(n) \le h^*(n) \quad \text{for all } n$$

Example: the straight-line distance in the travel problem never overestimates the actual distance

Admissible heuristics

- Heuristics can be designed using relaxed versions of the original problem
- **Example:** the 8-puzzle problem

Initial position

4	5	
6	1	8
7	3	2

Goal position

1	2	3
4	5	6
7	8	

- **Admissible heuristics:**
  1. number of misplaced tiles
  2. Sum of distances of all tiles from their goal positions (Manhattan distance)

Admissible heuristics

Heuristics 1: number of misplaced tiles

Initial position

4	5	
6	1	8
7	3	2

Goal position

1	2	3
4	5	6
7	8	

h(n) for the initial position: ?

Admissible heuristics

Heuristics 1: number of misplaced tiles

Initial position

4	5	
6	1	8
7	3	2

Goal position

1	2	3
4	5	6
7	8	

h(n) for the initial position: 7

Admissible heuristics

- **Heuristic 2:** Sum of distances of all tiles from their goal positions (Manhattan distance)

Initial position

4	5	
6	1	8
7	3	2

Goal position

1	2	3
4	5	6
7	8	

h(n) for the initial position:

Admissible heuristics

- **Heuristic 2:** Sum of distances of all tiles from their goal positions (Manhattan distance)

Initial position

4	5	
6	1	8
7	3	2

Goal position

1	2	3
4	5	6
7	8	

h(n) for the initial position:
2 + 3 + 3 + 1 + 1 + 2 + 0 + 2 = 14

For tiles: 1 2 3 4 5 6 7 8

Admissible heuristics

- We can have multiple admissible heuristics for the same problem
- **Dominance:** Heuristic function h_1 dominates h_2 if

$$\forall n \quad h_1(n) \geq h_2(n)$$

- **Combination:** two or more admissible heuristics can be combined to give a new admissible heuristics
  - Assume two admissible heuristics h_1, h_2

Then: $h_3(n) = \max(h_1(n), h_2(n))$

is admissible

Iterative deepening algorithm (IDA)

- Based on the idea of the limited-depth search, but
- It resolves the difficulty of knowing the depth limit ahead of time.

Idea: try all depth limits in an increasing order.

That is, search first with the depth limit $l=0$, then $l=1$, $l=2$, and so on until the solution is reached

Iterative deepening combines advantages of the depth-first and breadth-first search with only moderate computational overhead

Properties of IDA

- **Completeness:** **Yes.** The solution is reached if it exists. (the same as BFS)
- **Optimality:** **Yes**, for the shortest path. (the same as BFS)
- **Time complexity:**

$$O(1) + O(b^1) + O(b^2) + \ldots + O(b^d) = O(b^d)$$

  exponential in the depth of the solution *d*

  worse than BFS, but asymptotically the same

- **Memory (space) complexity:**

$$O(db)$$

  much better than BFS

IDA*

Iterative deepening version of A*

- Progressively increases the **evaluation function limit** (instead of the depth limit)
- Performs **limited-cost depth-first search** for the current evaluation function limit
  - Keeps expanding nodes in the depth-first manner up to the evaluation function limit
- **Problem:** the amount by which the evaluation limit should be progressively increased

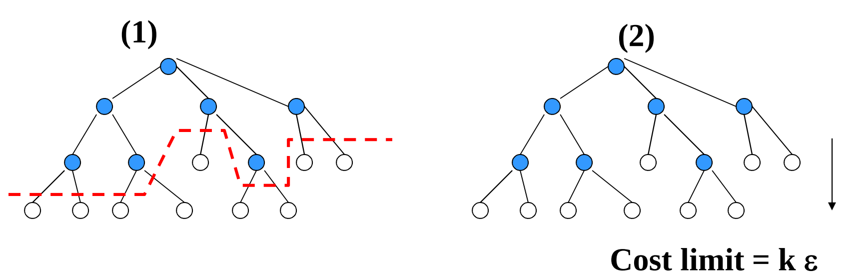
IDA*

Problem: the amount by which the evaluation limit should be progressively increased

Solutions:

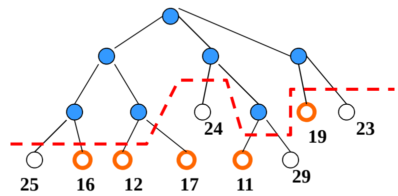
(1) peak over the previous step boundary to guarantee that in the next cycle some number of nodes are expanded

(2) Increase the limit by a fixed cost increment – say ε



IDA*

Solution 1: **peak over the previous step boundary to guarantee that in the next cycle more nodes are expanded**

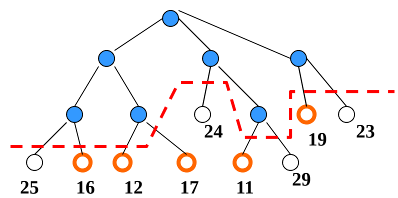


Properties:

- the choice of the new cost limit influences how many nodes are expanded in each iteration
- Assume I choose a limit such that at least 5 new nodes are examined in the next DFS run
- What is the problem here?

IDA*

Solution 1: **peak over the previous step boundary to guarantee that in the next cycle more nodes are expanded**

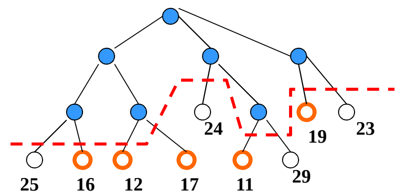


Properties:

- the choice of the new cost limit influences how many nodes are expanded in each iteration
- Assume I choose a limit such that at least 5 new nodes are examined in the next DFS run
- What is the problem here?
  We may find a sub-optimal solution
  - **Fix:** **?**

IDA*

Solution 1: **peak over the previous step boundary to guarantee that in the next cycle more nodes are expanded**



Properties:

- the choice of the new cost limit influences how many nodes are expanded in each iteration
- Assume I choose a limit such that at least 5 new nodes are examined in the next DFS run
- What is the problem here?

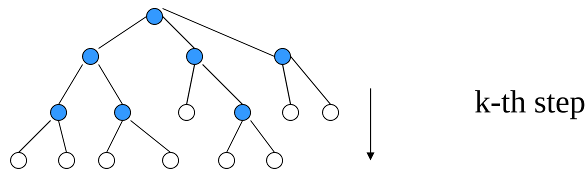
  We may find a sub-optimal solution

  – **Fix:** complete the search up to the limit to find the best

M. Hauskrecht

IDA*

Solution 2: **Increase the limit by a fixed cost increment (ε)**



Cost limit = k ε

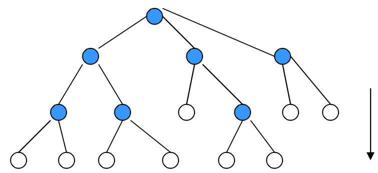
Properties:

– What is bad?

M. Hauskrecht

IDA*

Solution 2: **Increase the limit by a fixed cost increment (ε)**



k-th step

Cost limit = k ε

Properties:

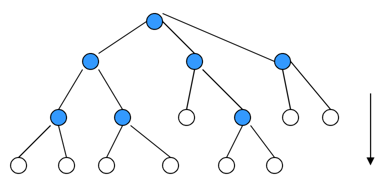
What is bad? Too many or too few nodes expanded – no control of the number of nodes

What is the quality of the solution?

M. Hauskrecht

IDA*

Solution 2: **Increase the limit by a fixed cost increment (ε)**



k-th step

Cost limit = k ε

Properties:

What is bad? Too many or too few nodes expanded – no control of the number of nodes

What is the quality of the solution?

– The solution found first may differ by < **ε** from the optimal solution

M. Hauskrecht

next

Constraint satisfaction search

Search problem

A search problem:

- **Search space (or state space):** a set of objects among which we conduct the search;
- **Initial state:** an object we start to search from;
- **Operators (actions):** transform one state in the search space to the other;
- **Goal condition:** describes the object we search for

- **Possible metric on the search space:**
  - measures the quality of the object with respect to the goal

Constraint satisfaction problem (CSP)

Two types of search:

- **path search** (a path from the initial state to a state satisfying the goal condition)
- **configuration search (**a configuration satisfying goal conditions)

Constraint satisfaction problem (CSP)

= **a configuration search problem** where**:**

- A **state** is defined by **a set of variables and their values**
- **Goal condition** is represented by **a set constraints on possible variable values**

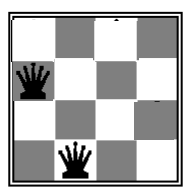
Special properties of the CSP lead to special procedures we can design to solve them

Example of a CSP: N-queens

Goal: n queens placed in non-attacking positions on the board

Variables:

- Represent queens, one for each column:
  - Q_1, Q_2, Q_3, Q_4
- Values:
  - Row placement of each queen on the board {1, 2, 3, 4}



$Q_1 = 2, Q_2 = 4$

Constraints: $Q_i \neq Q_j$ Two queens not in the same row

$|Q_i - Q_j| \neq |i - j|$ Two queens not on the same diagonal

Satisfiability (SAT) problem

Determine whether a sentence in the conjunctive normal form (CNF) is satisfiable (can evaluate to true)

- Used in the propositional logic (covered later)

$$(P \vee Q \vee \neg R) \wedge (\neg P \vee \neg R \vee S) \wedge (\neg P \vee Q \vee \neg T) \ldots$$

Variables:

- Propositional symbols (P, R, T, S)
- Values: *True, False*

Constraints:

- Every conjunct must evaluate to true, at least one of the literals must evaluate to true

$$(P \vee Q \vee \neg R) \equiv True, (\neg P \vee \neg R \vee S) \equiv True, \ldots$$

Other real world CSP problems

Scheduling problems:

- E.g. telescope scheduling
- High-school class schedule

Design problems:

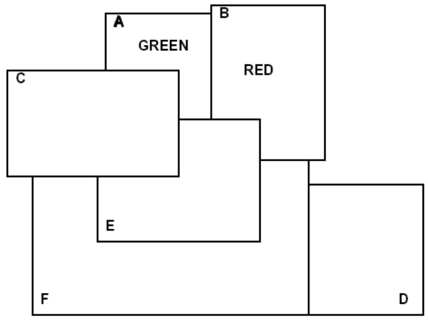
- Hardware configurations
- VLSI design

More complex problems may involve:

- **real-valued variables**
- **additional preferences on variable assignments** – the optimal configuration is sought

Exercise: Map coloring problem

Color a map using k different colors such that no adjacent countries have the same color



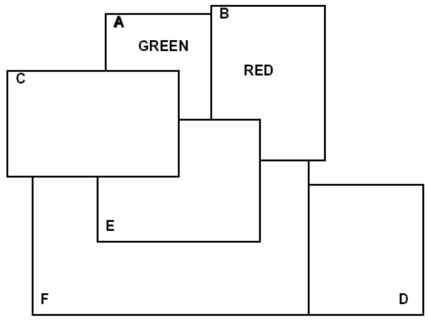
Variables: ?

- Variable values: ?

Constraints: ?

Map coloring

Color a map using k different colors such that no adjacent countries have the same color



Variables:

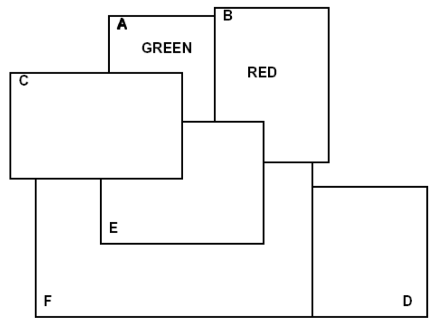
- Represent countries
  - A, B, C, D, E
- Values:
  - K -different colors

  {Red, Blue, Green,..}

Constraints: ?

Map coloring

Color a map using k different colors such that no adjacent countries have the same color



Variables:

- Represent countries
  - A, B, C, D, E
- Values:
  - K -different colors

  {Red, Blue, Green,..}

Constraints: $A \neq B, A \neq C, C \neq E$, etc

An example of a problem with **binary constraints**

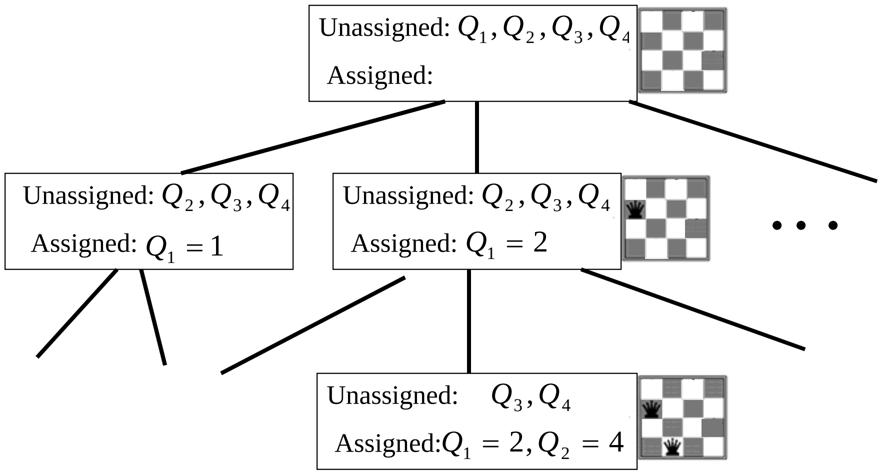
Constraint satisfaction as a search problem

A formulation of the search problem:

- **States.** Assignment (partial or complete) of values to variables.
- **Initial state.** No variable is assigned a value.
- **Operators.** Assign a value to one of the unassigned variables.
- **Goal condition.** All variables are assigned, no constraints are violated.

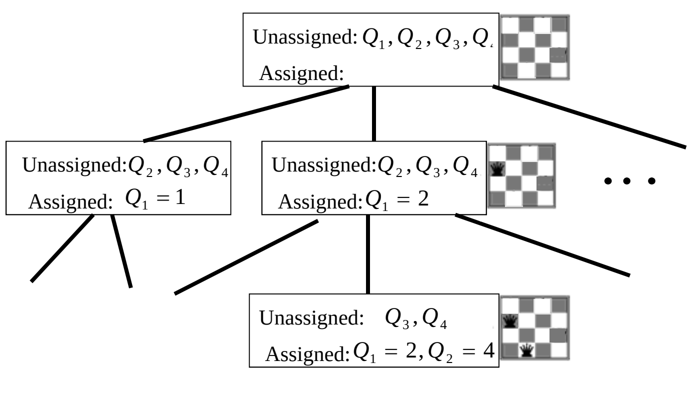
- Constraints can be **represented**:
  - **Explicitly** by a set of allowable values
  - **Implicitly** by a function that tests for the satisfaction of constraints

Search strategies for solving CSP



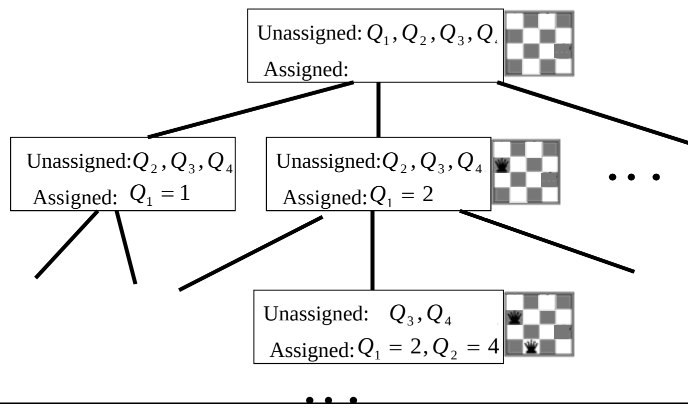
Search strategies for solving CSP

- **Maximum depth of the tree (m): ?**
- **Depth of the solution (d) : ?**
- **Branching factor (b) : ?**



Search strategies for solving CSP

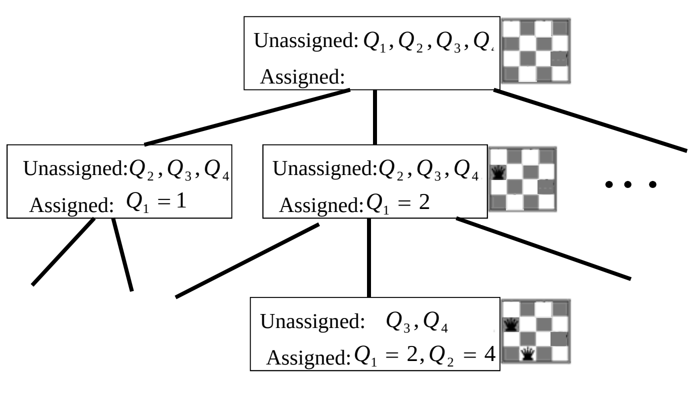
- **Maximum depth of the tree:** Number of variables in the CSP
- **Depth of the solution:** Number of variables in the CSP
- **Branching factor:** if we fix the order of variable assignments the branch factor depends on the number of their values



Search strategies for solving CSP

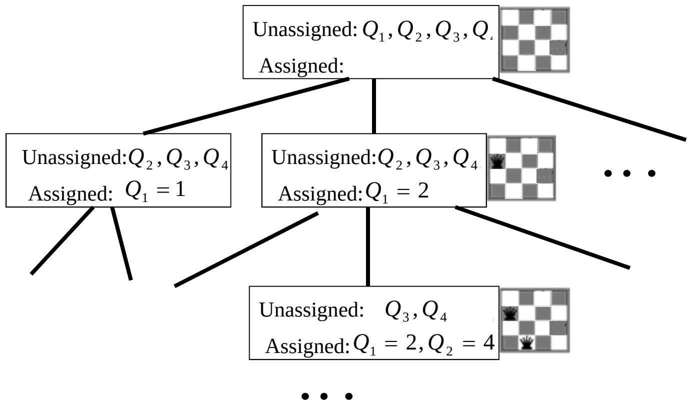
- **What search algorithm to use: ?**

  Depth of the tree = Depth of the solution=number of vars



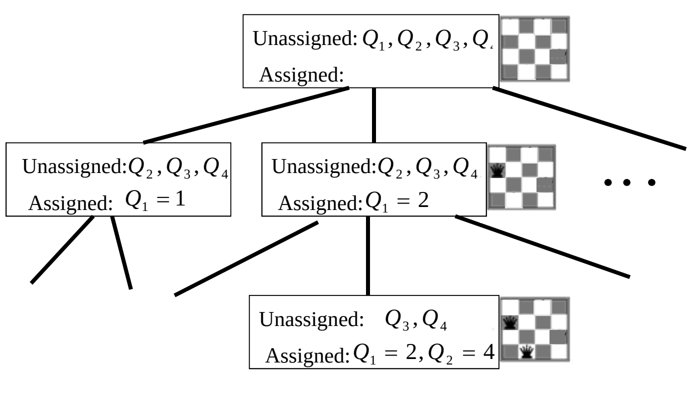
Search strategies for solving CSP

- **What search algorithm to use: ?**



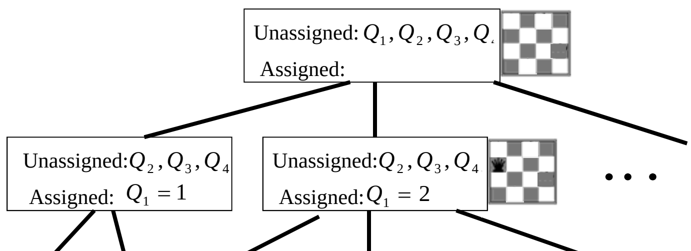
Search strategies for solving CSP

- **What search algorithm to use: Depth first search !!!**
  - Since we know the depth of the solution
  - We do not have to keep large number of nodes in queues



Search strategies for solving CSP

- **What search algorithm to use: Depth first search !!!**
  - Since we know the depth of the solution
  - We do not have to keep large number of nodes in queues



Depth-first search strategy for CSP is also referred to as **backtracking**

Constraint consistency

Question:

- **When to check the constraints defining the goal condition?**
- The violation of constraints can be checked:
  - at the end (for the leaf nodes)
  - for each node of the search tree during its generation or before its expansion

Checking the constraints for intermediate nodes:

- More efficient: cuts branches of the search tree early

Constraint consistency

Assuring consistency of constraints:

- Current **variable assignments** together **with constraints restrict remaining legal values of unassigned variables**
- The remaining **legal and illegal values of variables may be inferred** (effect of constraints propagates)
- To prevent "blind" exploration we can keep track of the remaining legal values, so we know when the constraints are violated and when to terminate the search

Constraint propagation

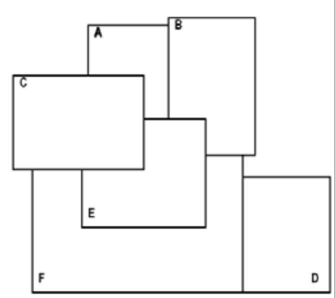
A **state** (more broadly) is defined by a set of variables, their values and a list of legal and illegal assignments for unassigned variables

Legal and illegal assignments can be represented via: **equations** (value assignments) and **disequations (list of invalid assignments)**

Example: map coloring

Equation $A = \text{Red}$

Disequation $C \neq \text{Red}$



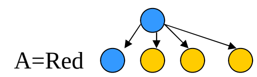
Constraints + assignments
can entail new equations and disequations

$A = \text{Red} \quad \rightarrow \quad B \neq \text{Red}$

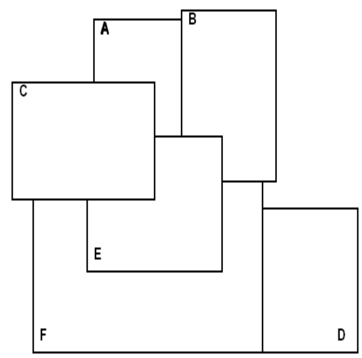
Constraint propagation: the process of inferring of new equations and disequations from existing equations and disequations

Constraint propagation

- Assign A=Red



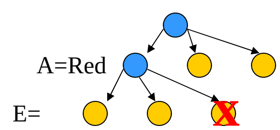
	Red	Blue	Green
A	✓		
B			
C			
D			
E			
F			



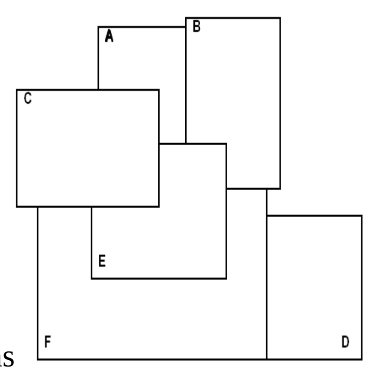
✓ - equations ✗ - disequations

Constraint propagation

- Assign A=Red



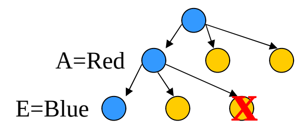
	Red	Blue	Green
A	✓		
B	✗		
C	✗		
D			
E	✗		
F			



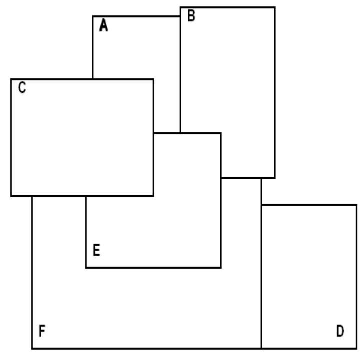
✓ - equations ✗ - disequations

Constraint propagation

- Assign E=Blue

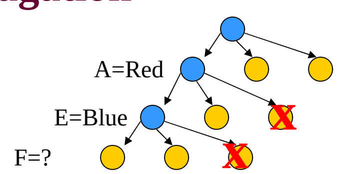


	Red	Blue	Green
A	✓		
B	✗		
C	✗		
D			
E	✗	✓	
F			

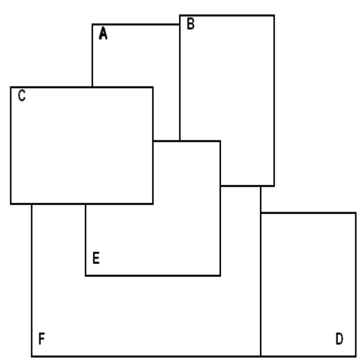


Constraint propagation

- Assign E=Blue



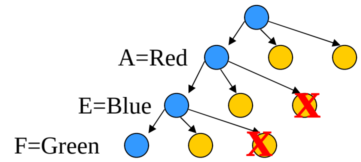
	Red	Blue	Green
A	✓	✗	
B	✗	✗	
C	✗	✗	
D			
E	✗	✓	
F		✗	

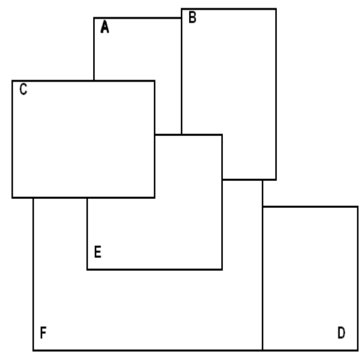


Constraint propagation

- Assign F=Green

	Red	Blue	Green
A	✓	✗	
B	✗	✗	
C	✗	✗	
D			
E	✗	✓	
F		✗	✓

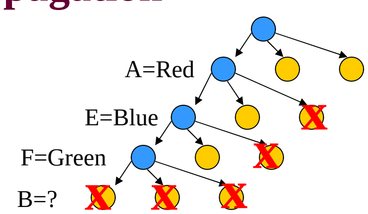


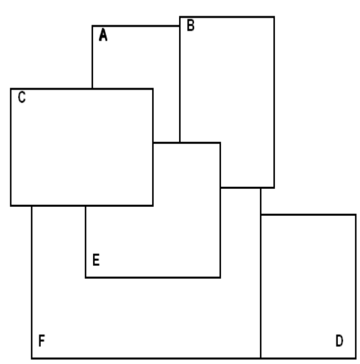


Constraint propagation

- Assign F=Green

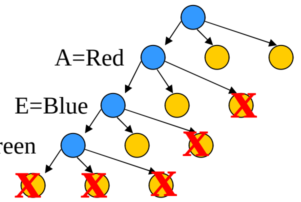
	Red	Blue	Green
A	✓	✗	
B	✗	✗	✗
C	✗	✗	✗
D			✗
E	✗	✓	✗
F		✗	✓



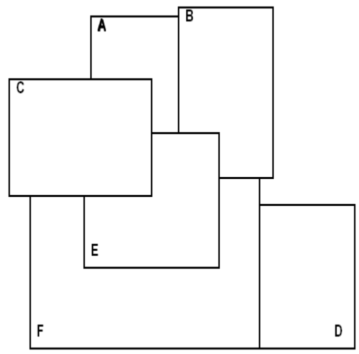


Constraint propagation

- Assign F=Green



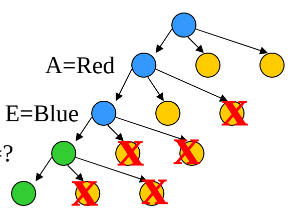
	Red	Blue	Green
A	✓	✗	
B	✗	✗	✗
C	✗	✗	✗
D			✗
E	✗	✓	✗
F		✗	✓



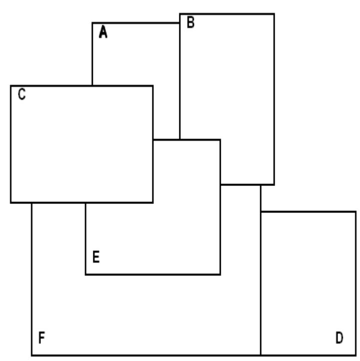
Conflict !!! No legal assignments available for B and C

Constraint propagation

- We can derive remaining legal values through propagation



	Red	Blue	Green
A	✓	✗	
B	✗	✗	✓
C	✗	✗	✓
D			
E	✗	✓	
F		✗	



B=Green

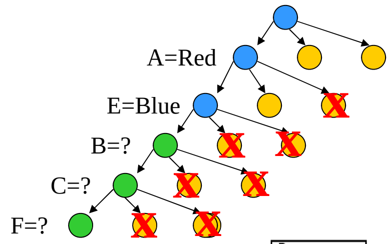
C=Green

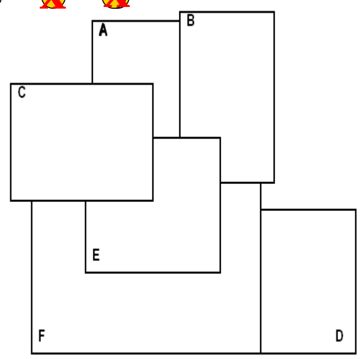
Constraint propagation

- We can derive remaining legal values through propagation

	Red	Blue	Green
A	✓	✗	✗
B	✗	✗	✓
C	✗	✗	✓
D	✗		
E	✗	✓	✗
F	✓	✗	✗

B=Green
C=Green → **F=Red**





Constraint propagation

- We can derive remaining legal values through propagation

	Red	Blue	Green
A	✓	✗	✗
B	✗	✗	✓
C	✗	✗	✓
D	✗		
E	✗	✓	✗
F	✓	✗	✗

B=Green
C=Green → **F=Red**