Unit II

Search Methodologies

→ Search is a method that can be used by computer to examine a problem space ~~the~~ in order to find goal.

→ A problem space can also be considered to be a search space because in order to ~~search~~ for the solve the problem, we will search the space for a goal state.

Two approaches

* Data driven Search :- Starts from an initial state and use actions that are allowed to move forward until a goal is reached. aka forward chaining.

* Goal driven Search: Search starts at the goal and work back toward a start state, by seeing what moves could have led to the goal state. aka backward chaining.

* Goal driven search is useful when goal can be clearly specified eg. Medical diagnosis.

* Data driven search is most useful when initial data is provided and it is not very clear what the goal is. eg Making deductions about Stars and planets using astronomical data as it doesn't necessarily take us to a conclusion.

Generate & Test

→ Simplest approach, involves generating each node in the search

POCO
SHOT ON POCO M2 PRO

extremly large will not sidetrack the search.

-) like BFS, will always find the path with least steps through the tree.

Maths shit or something

depth - d, branching factor - b, no of nodes is given as

1 root node

b nodes in the first layer

b^2 nodes in the 2^{nd} layer

⋮

b^n nodes in the n^{th} layer.

$\therefore$ Total = $1+b+b^2+b^3+b^4+\ldots+b^n$

GP equal to $\frac{1-b^{d+1}}{1-b}$ eg.1 for d=2, b=2 $\frac{1-8}{1-2} = 7$ nodes.

$\therefore$ Using dfs or bfs, we need to examine 7 nodes.

But in DFID, due to examining each node more than once

$$(d+1)+b(d)+b^2(d-1)+b^3(d-2)+\ldots+b^d$$

$\therefore$ Time Complexity (DFID) = $O(b^d)$

Space complexity = $O(bd)$

eg.2. for d=4 b=10, no of nodes

$\frac{1-10^5}{1-10} = 11,111$ nodes

POCO

SHOT ON POCO M2 PRO

DOMS Page No.
Date / /

Examples

1) Hill Climbing

Example of informed search method because it uses info about the search space to search in a reasonably efficient manner.

Steepest Ascent Hill Climbing

-> Always check around in all four directions and choose the position that is highest.

-> Apply heuristic to the search tree

Algorithm.

```
Function hill ( )
{
    queue = [ ] ;
    state = root_node ;
    while (true)
    {
        if is_goal (state)
            then return SUCESS
        else
        {
            sort (sucessors (state));
            add_to_front_of_queue (sucessors (state));
        }
        if queue == [ ]
            then report FAILURE ;
        state = queue [0]; // state = first item in queue
        remove_first_item_from (queue);
    }
```

POCO
SHOT ON POCO M2 PRO

15	H	I, J, K, L, M, N, O	H has no successors, so we have nothing to add to the queue in this state, or in fact for any subsequent states
16	I	J, K, L, M, N, O	
17	J	K, L, M, N, O	
18	K	L, M, N, O	
19	L	M, N, O	SUCCESS: A goal state has been reached.

Web Spidering

- Assumption - Majority of the web is connected (can get from one page to another through a series of (even infinite) ~~the~~ links.
- On avg. branching factor is reasonably low.
- The search tree representing the connected web is hugeeeeeee. thus a breadth first search approach would need prohibitively large storage requirements.
- Depth Search first also not sensible as some paths might have ∞ depth.
- Thus we need to use a combination of approaches, with emphasis on frequently changed & important pages.

DFID - DF Iterative Deepening (aka IDS)

- Performing DFS repeatedly starting with a dfs limited to a depth of one, then a depth of two and so on.
- Wasteful in terms of no of steps
- Advantage - Combines efficiency of memory use of dfs with the advantage that branches of the search tree that are ∞ or

POCO
SHOT ON POCO M2 PRO

space and testing it to see if it is a goal node.

→ Simplest form of brute force search, it assumes no addnal knowledge other than how to traverse the search tree and how to identify leaf nodes & goal nodes, & it search every node until a goal is found.

→ Three properties
1. It must be complete
2. It must be non redundant
3. It must be well informed.

~~Deep~~ Depth Search First

→ Follows each path to the greatest depth before moving to the next path.
→ If goal ~~tree~~ node is ~~fond~~, found, return succes, else backtrace to the next highest node that has ~~an~~ ~~ex~~ unexplored path.
→ Uses chronological backtracing to move back up the search tree once a dead end has been found.

→ It undoes choices in reverse order of the time the decisions were originally made.

→ Usually used by computer for search problems such as locating files on a disk, or by search engines for spidering the internet.

Illustration.

POCO
SHOT ON POCO M2 PRO

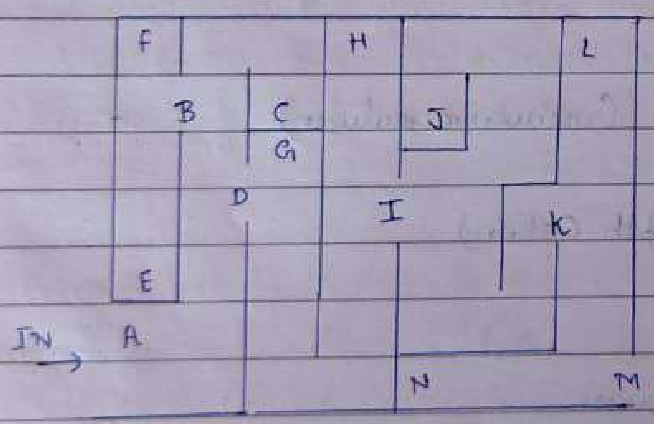
DOMS Page No.
Date / /

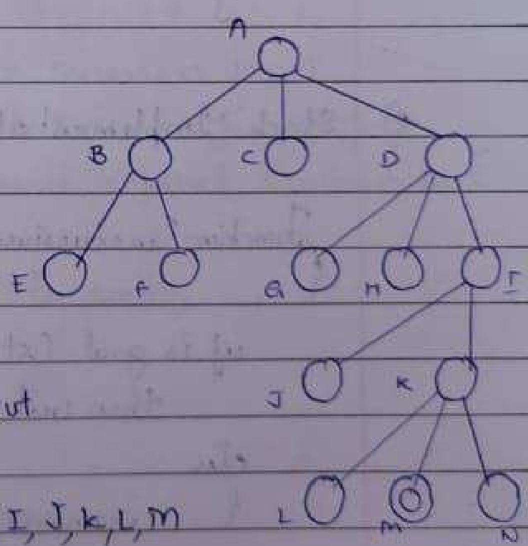
→ Irrevocable methods often find suboptimal solutions to problems as they tend to be fooled by local optimal solution that ~~goo~~ look good locally but are less favorable when compared when compared with other solution elsewhere in the search tree.

Use of DFS

1. Maze Traversing

eg. A is the entrance
M is the exit
C, E, F, G, H, J, L & N are dead ends.
B, D, I & K are decision making points.





path to be followed :- A, B, E, F, C, D, G, H, I, J, K, L, M

→ Here, instead of blindly searching for a path, the person needs to keep his left hand on the left edge of the ~~a~~ maze wall

2. Implementation of DFS & BFS

PTO

POCO
SHOT ON POCO M2 PRO

Page No.
Date / /

Tracing for the maze thingy using queue

Step	State	Queue	Notes
1	A	empty	The q starts out empty, initial state element is the root node, (A)
2	A	B, C	The sucessors of A are added.
3	B	C	
4	B	D, E, C	enqueue b's sucessors.
5	D	E, C	
6	D	H, I, E, C	
7	H	I, E, C	H has no sucessors, don't enqueue anything new.
8	I	E, C	I has no sucesson, don't enqueue anything new.
9	E	C	
10	E	J, K, C	
11	J	K, C	No sucessors for J
12	k	C	No sucessors for k, full branch explored, traceback to C
13	C	empty	~~E~~ q is empty
14	C	F, G	
15	F	G	
16	F	L M G	
17	L	M, G.	SUCCESS

3. Breadth Search first

```
function breadth ()
{
    queue = [ ] ;
    state = root node ;
```

POCO
SHOT ON POCO M2 PRO

No of nodes examined by dfid

$(4+1) + (10 \times 4) + (100 \times 3) + (1000 \times 2) + (10{,}000 \times 1) = 12345$ nodes

Thus for large tree, its good as it has optimality of ~~df~~ bfs and space efficiency of dfs and it perfoms almost equal no of searches as dfs / bfs.

⇒ Heuristics for Search

Heuristics is a function which is used in Informed search and it finds the most promising path. It takes the current state of the agent as the input and produces the estimation of how close the agent is from the goal.

→ eg for two nodes m & n & function f, if $f(m) < f(n)$ m is more likely to be an optimal path.

→ lower the heuristic value, better the optimality

Informed and Uninformed Methods

→ Informed → heuristic is informed if it uses additional information about nodes that have not yet been explored to decide which nodes to examine next.

→ Search methods that use heuristics are informed & not blind.

A heuristic h is said to be more informed than another, j, if h(node) ≤ j(node) for all nodes in the search space.

POCO
SHOT ON POCO M2 PRO

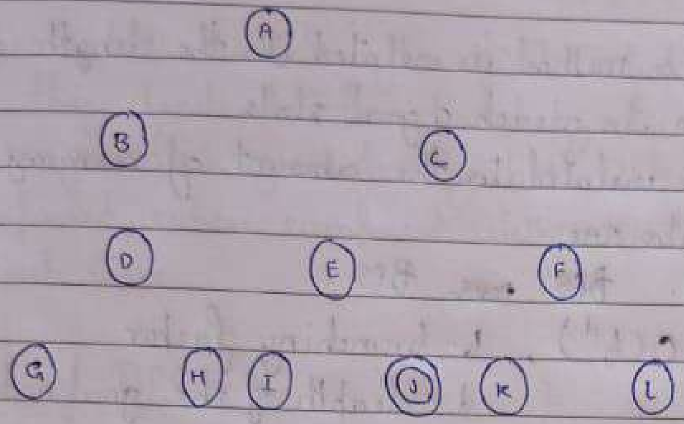
1) Queue Implementation

```
Function ~~deep~~ depth()
{
    queue = [ ] ;
    state = root node ;
    while (true )
    {
        if is goal( state)
            then return SUCCESS
        else add to the front of queue ( successors ( state));
        if queue == [ ]
            then report FAILURE ;
        state = queue[0] ;
        remove first item from (queue);
    }
}
```

2) Stack Implementation (recursive nature)

```
Function recursive depth (state)
{
    if is goal (state)
        then return sucess
    else
    {   remove from stack ( state);
        add to stack ( sucessors (state))
    }
    while (stack != [ ])
    {   if recursive depth( stack [0] == sucess
            then return SUCCESS ;
        remove first item from (stack) ;
    }
}  return failure ;
```

POCO
SHOT ON POCO M2 PRO



Breadth Search First

→ Traverses a tree by breadth rather than the depth

→ Its a poor idea in trees where all paths lead to a goal node with similar path lengths. (Thus DFS is prefered)

→ Comparison of DFS & BFS

Senario.	DFS	BFS
i) Some paths are extreamly long, or even ∞	performs badly	performs well
ii) All paths are similar length.	performs well	performs well
iii) All paths are of similar length, and all paths lead to a goal state	performs well	Waste of time & money
iv) High branching factor	performace depends on other factors	performs poorly

Properties of Search Methods

POCO
SHOT ON POCO M2 PRO

```
while (True)
{
    if is-goal (state)
        then return sucess
    else add-to back of queue (sucessors (state));
    if queue == []
        then report FAILURE;
    state = queue [0];
    remove first item from (queue);
}
}
```

Same Example (Maze) using BFS.

Step	State	Queue	Notes
1	A	empty	The queue starts out empty and the initial state is the root node, which is A.
2	A	B, C	The two descendents of A are added to the queue.
3	B	C	
4	B	C, D, E	The two descendents of the current state B are added to the back of the queue.
5	C	D, E	
6	C	D, E, F, G	
7	D	E, F, G	
8	D	E, F, G, H, I	
9	E	F, G, H, I	
10	E	F, G, H, I, J, K	
11	F	G, H, I, J, K	
12	F	G, H, I, J, K, L, M	
13	G	H, I, J, K, L, M	
14	G	H, I, J, K, L, M, N, O	

POCO
SHOT ON POCO M2 PRO

1) Complexity
a) Time complexity of a method is related to the length of the method would take to reach a goal state
b) Space complexity is related to the amount of memory that the method needs to use.
Eg. ~~Depth~~ Search first. ~~BFS~~ ~~DFS~~ BFS
Time Complexity :- $O(b^d)$, b - branching factor
d - depth of the goal node in the tree
Space → DFS Very efficient because it needs to store information about the path it is currently examining, not very time efficient

2) Completeness
A search tree is described as complete if it is guaranteed to find a goal state if one exists.

DFS isn't complete, coz it might have to search a tree of infinite depth, while BFS is.

3) Optimality
→ A search method is optimal if it is guaranteed to find the best solution that exists. (find path involving least no. of steps.
→ An algorithm is said to be admissible if it is guaranteed to find the best solution.
→ BFS is optimal, DFS isn't as the first solution it finds might be the worst.

4) Irrevocability
→ Methods that use backtracing as known as tentative
→ Methods that do not, thereby examining just one path are called irrevocable.

POCO
SHOT ON POCO M2 PRO

Choosing a Good Heuristic

→ A heuristic that reduces the number of nodes that need to be examined in the search tree is a good heuristic.
→ Heuristic with higher efficiency of running is a good heuristic.

Development of Heuristic

The 8 puzzle.

Random	7	6		Goal	1	2	3
Start	4	3	1	State	8		4
State	2	5	8		7	6	5

→ Typical depth → 20 (usually needs around 20 steps)
→ branching factor → if the blank square is
i) in middle → 4
ii) if on the edge → 3
iii) if in a corner → 2.

→ Heuristic must not overestimate the cost of changing from a given state to the goal state, such heuristics are known as admissible.

Monotonicity

A search method is defined as monotone if it always reaches a given node by the shortest possible path. A monotone search must be admissible provided there is only one goal state.

POCO
SHOT ON POCO M2 PRO

CS 1571 Introduction to AI
Lecture 7

Constraint satisfaction search

Milos Hauskrecht

milos@cs.pitt.edu

5329 Sennott Square

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

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***

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

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

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 !!!**

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

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

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

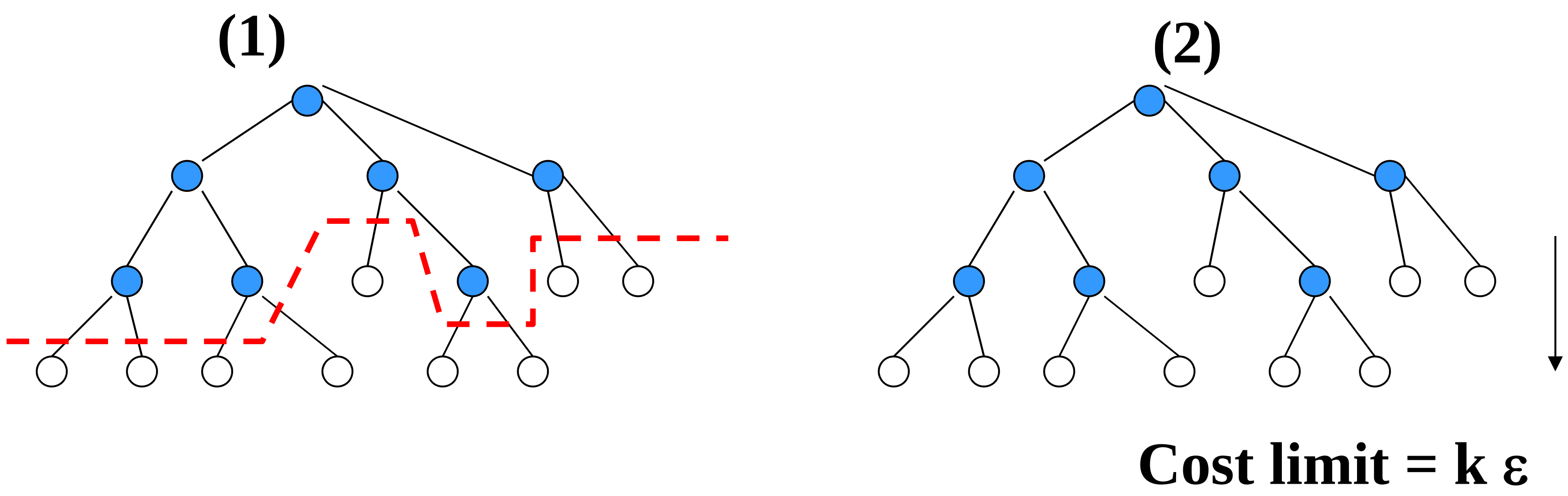
IDA*

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

Solutions:

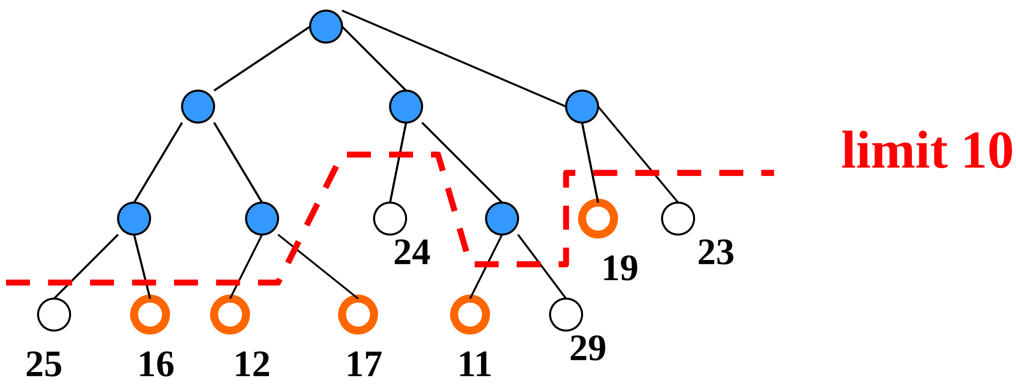
(1) peak over the previous step boundary to guarantee that in the next cycle some number of nodes are expanded

(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

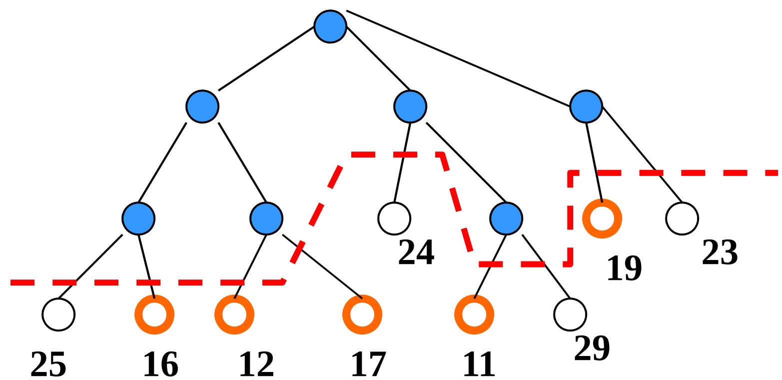


Properties:

- 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:

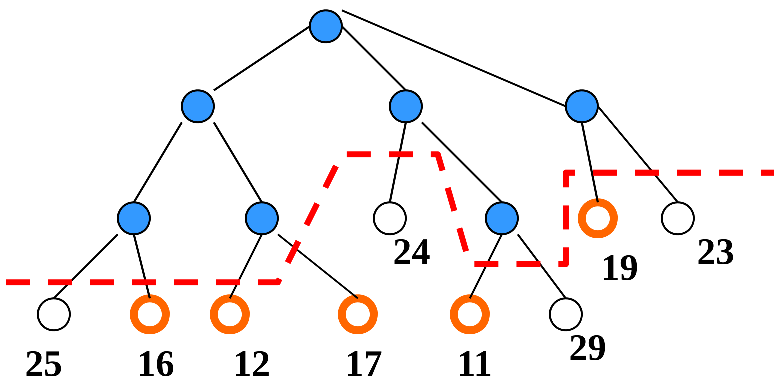
- the choice of the new cost limit influences how many nodes are expanded in each iteration
- Assume I choose a limit such that at least 5 new nodes are examined in the next DFS run
- What is the problem here?

  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:

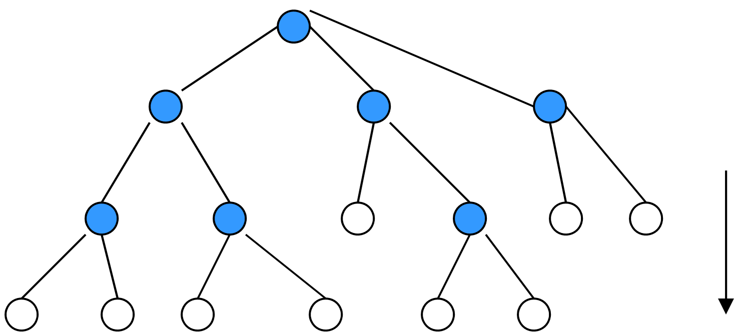
- the choice of the new cost limit influences how many nodes are expanded in each iteration
- Assume I choose a limit such that at least 5 new nodes are examined in the next DFS run
- What is the problem here?

  We may find a sub-optimal solution

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

IDA*

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



k-th step

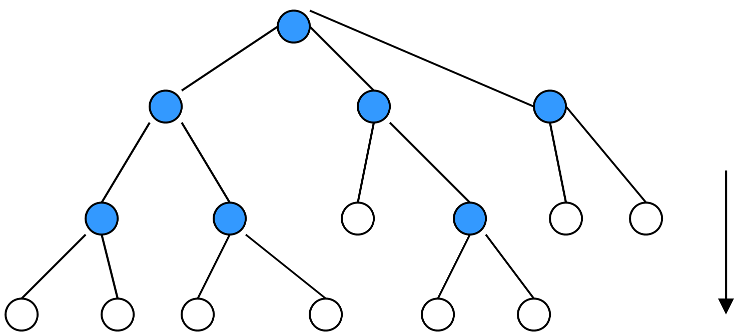
Cost limit = k ε

Properties:

– What is bad?

IDA*

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



k-th step

Cost limit = k ε

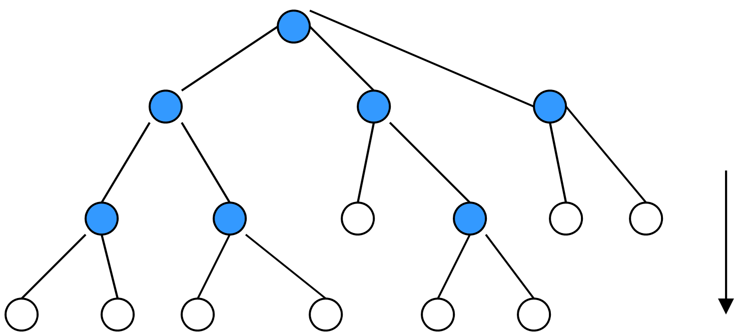
Properties:

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

What is the quality of the solution?

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

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 search procedures we can design to solve them

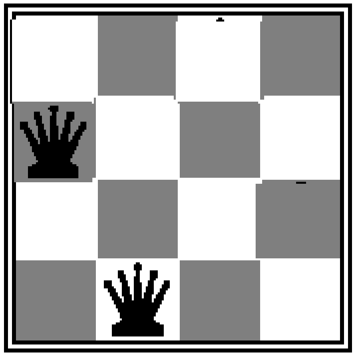
Example of a CSP: N-queens

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:

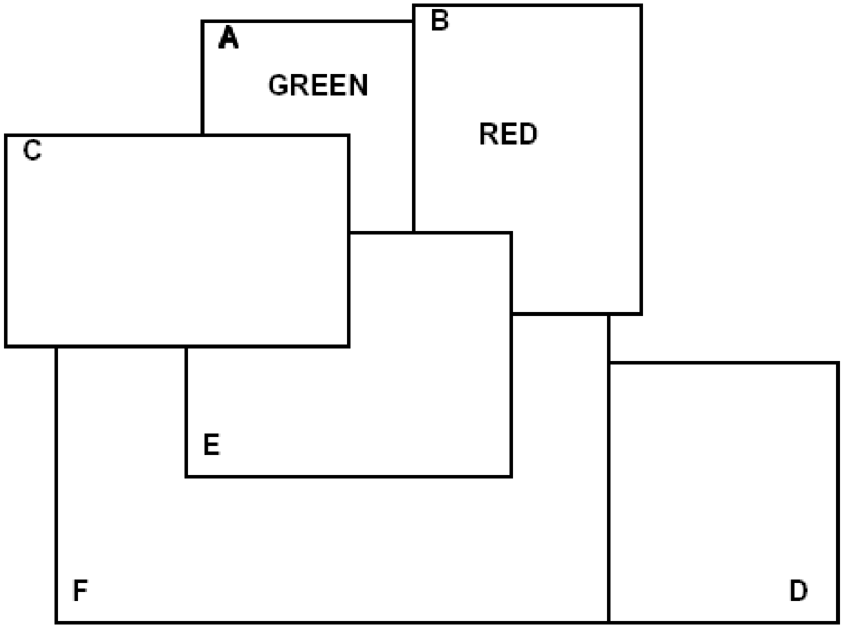
- Hardware configurations
- VLSI design

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: ?

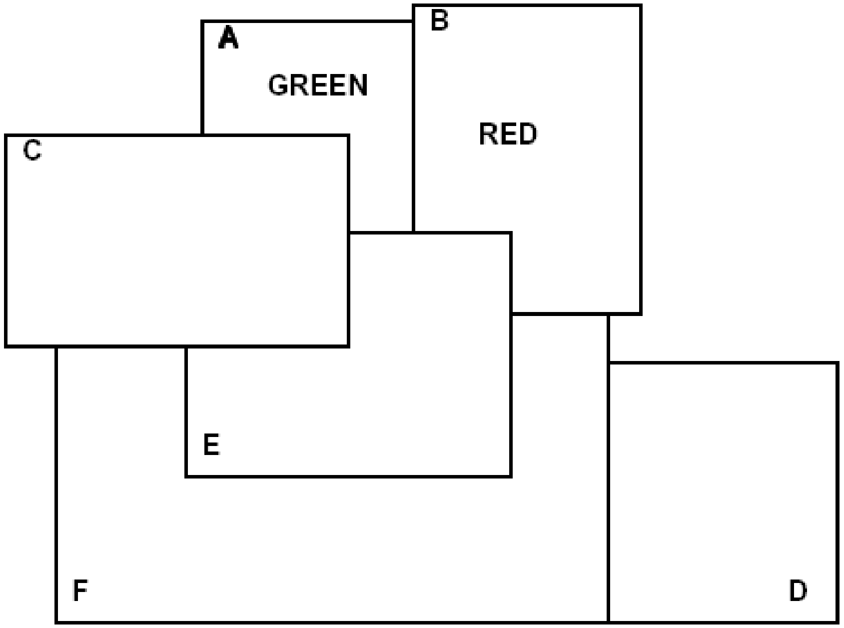
Map coloring

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: ?

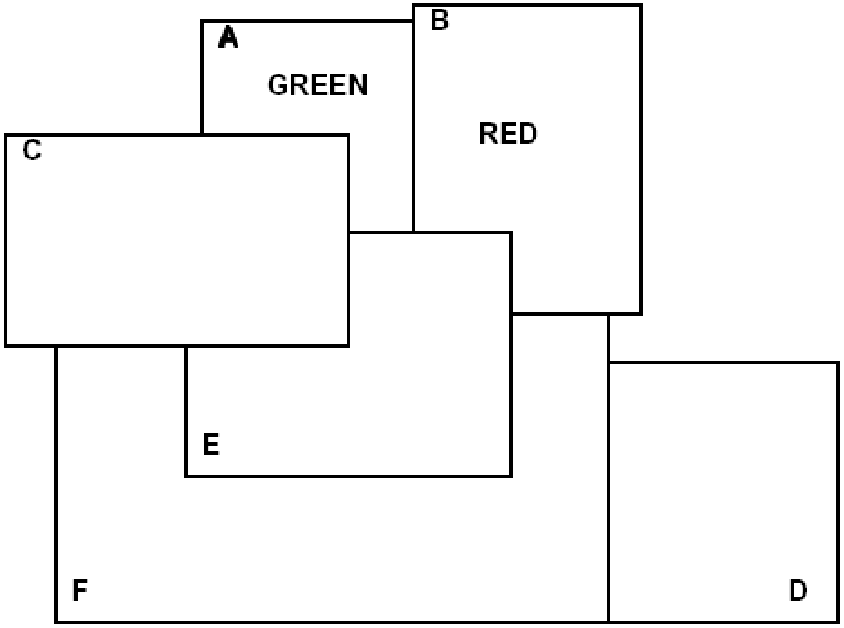
Map coloring

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**

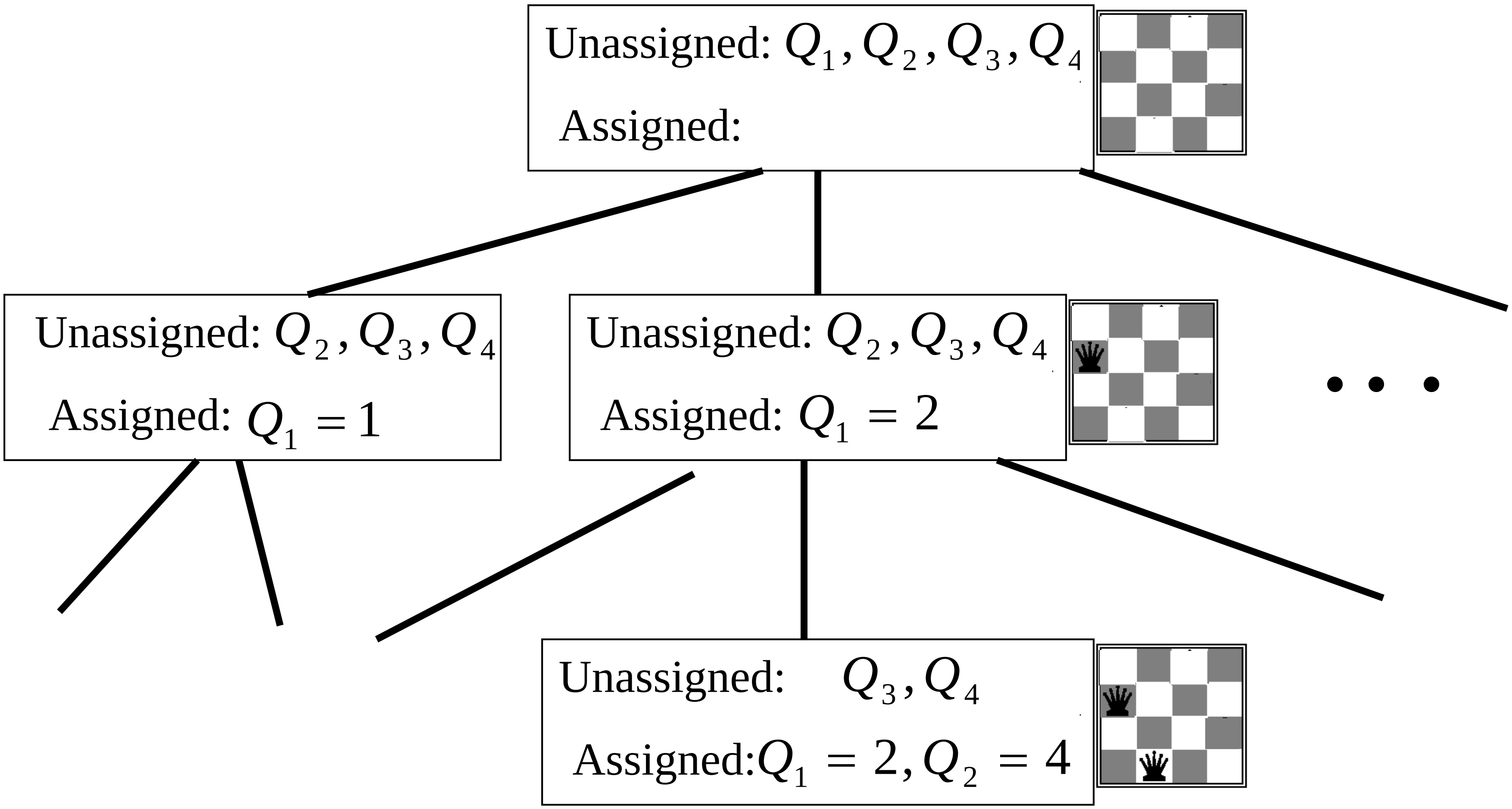
Constraint satisfaction as a search problem

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.

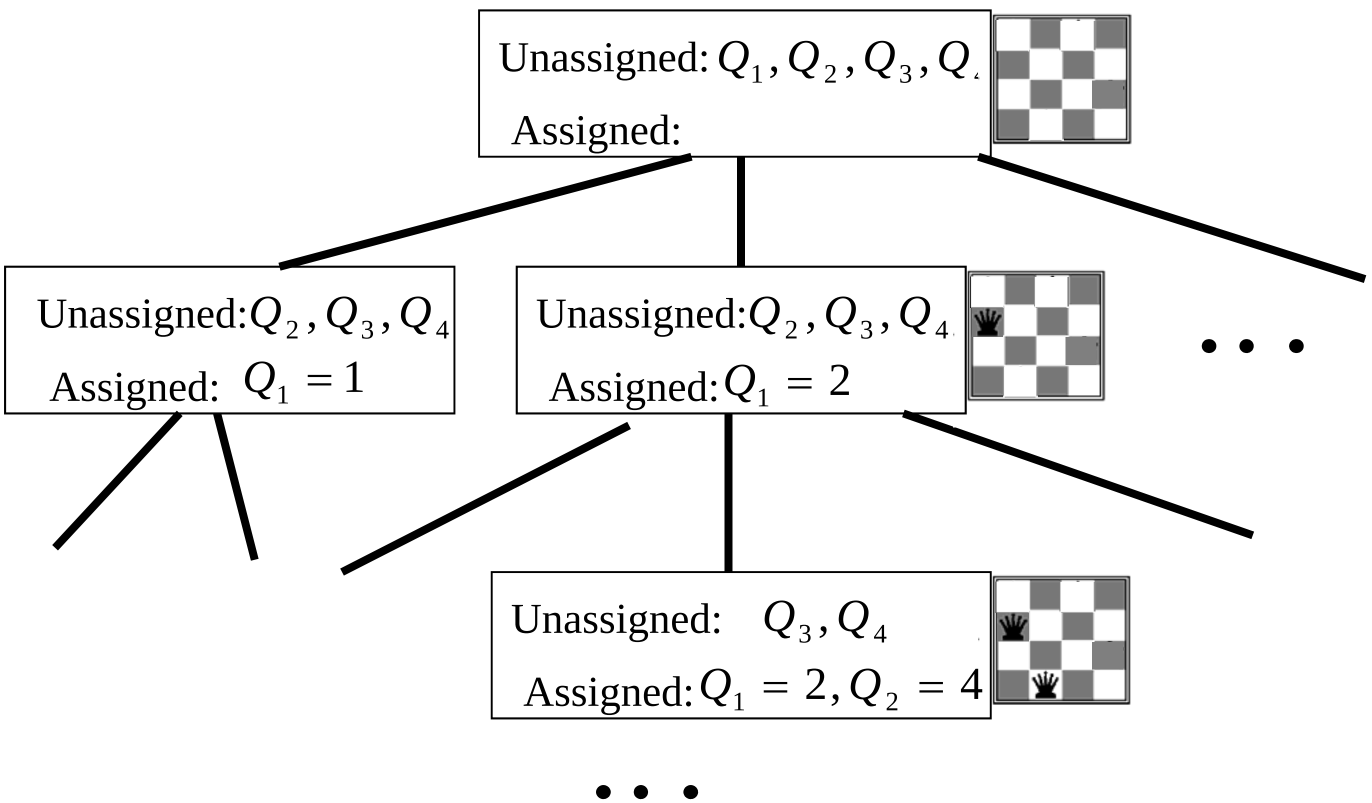
- **Constraints** can be **represented**:
  - **Explicitly** by a set of allowable values
  - **Implicitly** by a function that tests for the satisfaction of constraints

Search strategies for solving CSP



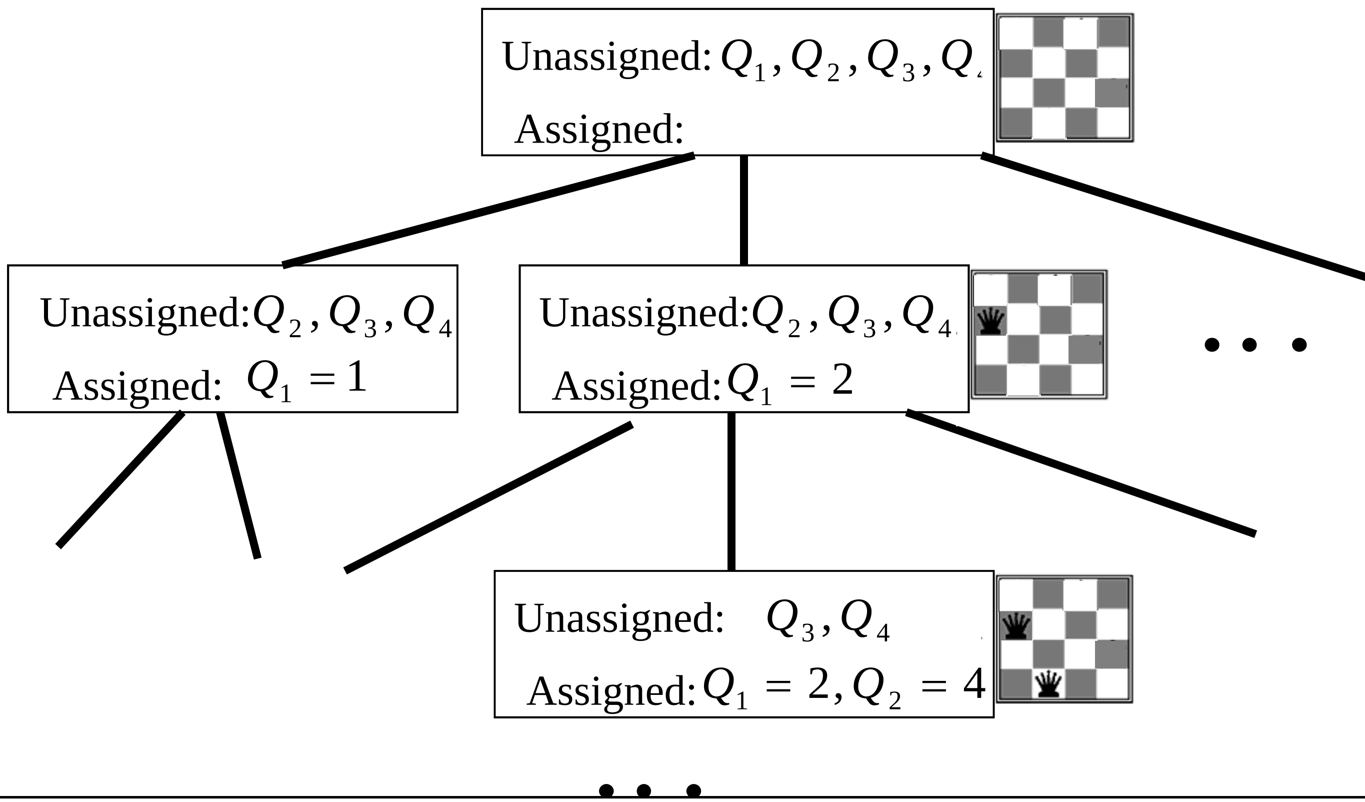
Search strategies for solving CSP

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



Search strategies for solving CSP

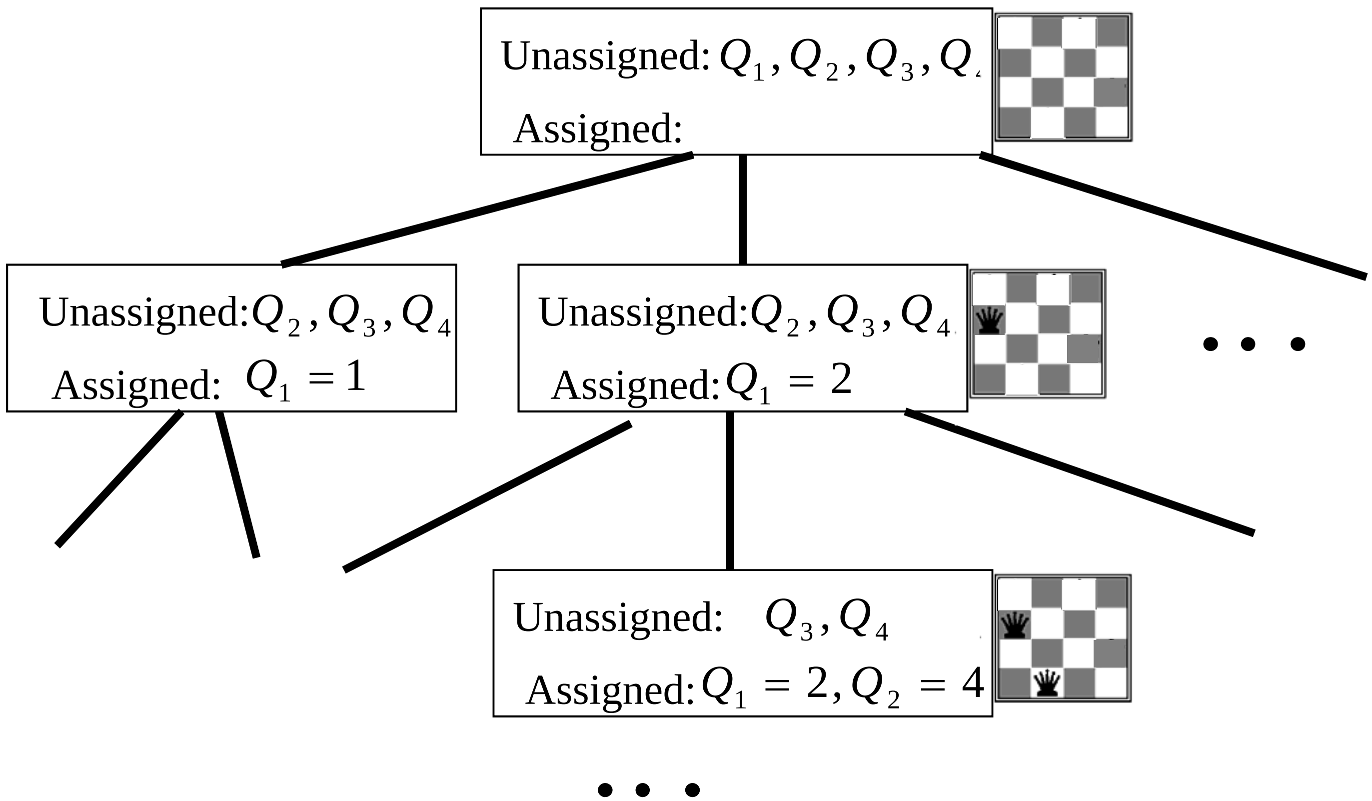
- **Maximum depth of the tree:** Number of variables in the CSP
- **Depth of the solution:** Number of variables in the CSP
- **Branching factor:** if we fix the order of variable assignments the branch factor depends on the number of their values



Search strategies for solving CSP

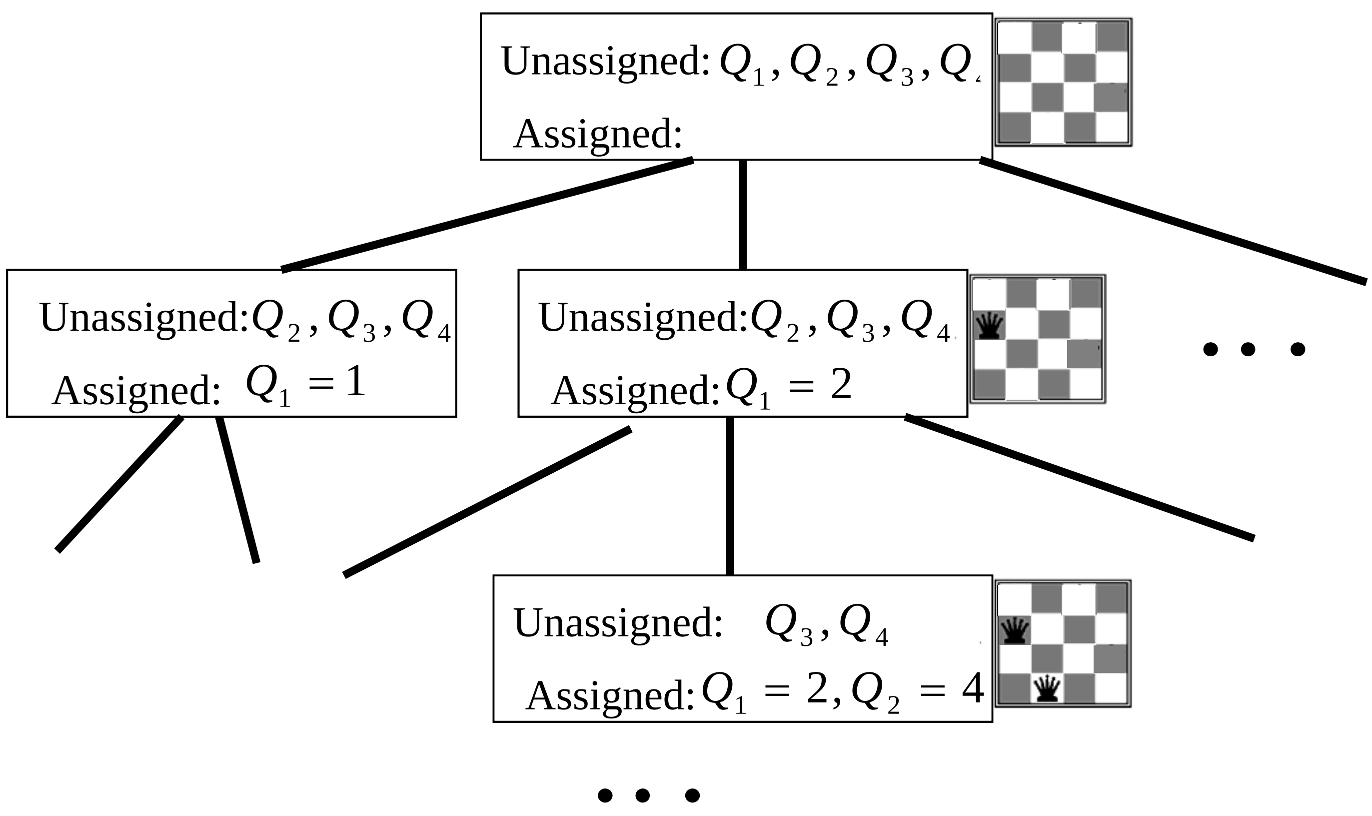
- **What search algorithm to use: ?**

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



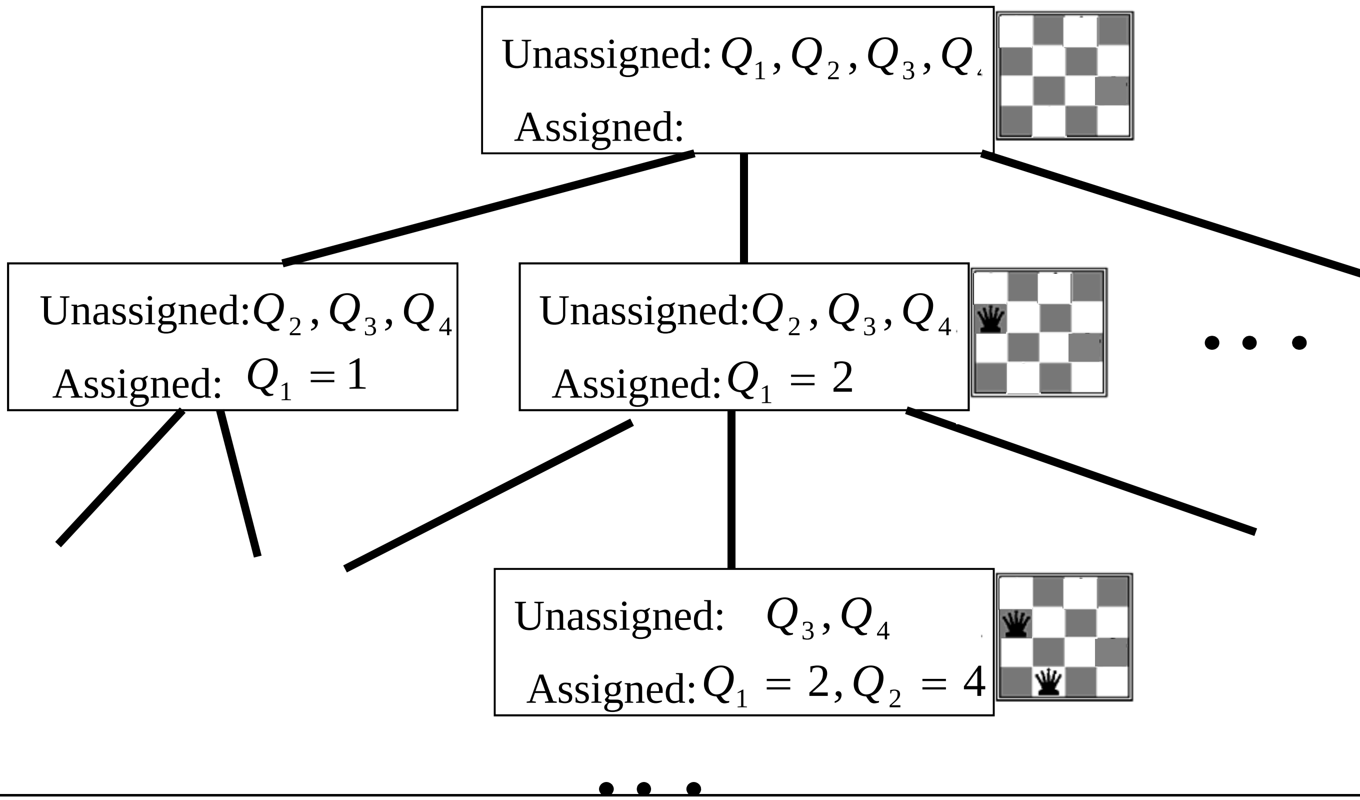
Search strategies for solving CSP

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



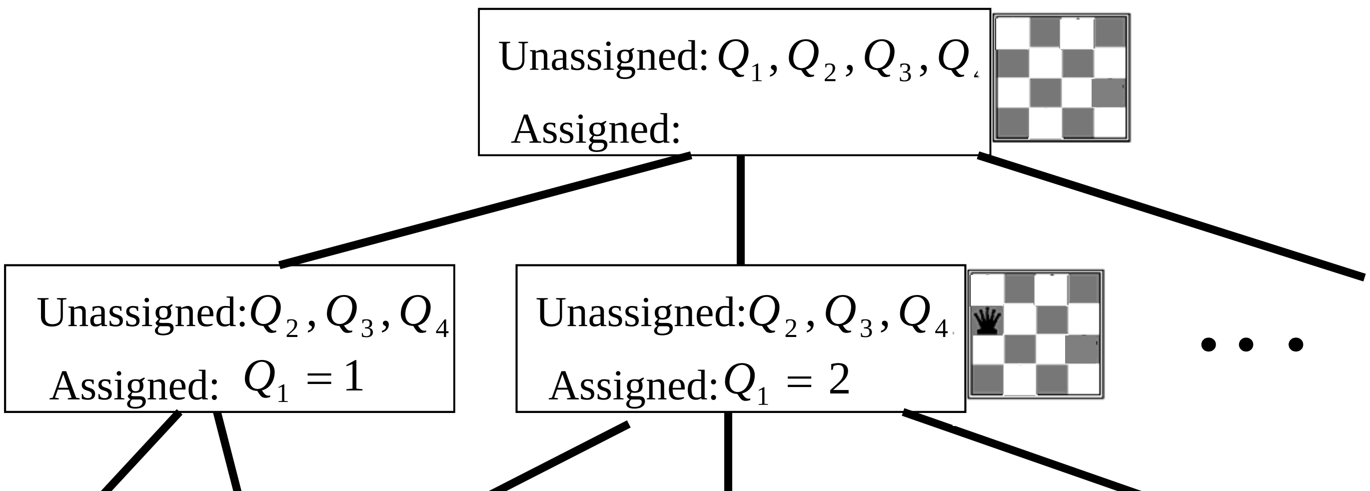
Search strategies for solving CSP

- **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

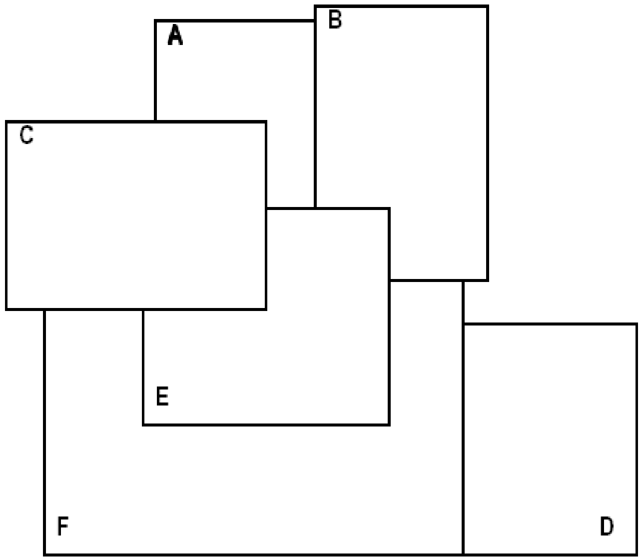
A **state** (more broadly) is defined:

- by a set of assigned variables, their values and
- a list of legal and illegal assignments for unassigned variables

Legal and illegal assignments can be represented:

- **equations** (value assignments) and
- **disequations (list of invalid assignments)**

$$A = \text{Red, Blue} \qquad C \neq \text{Red}$$



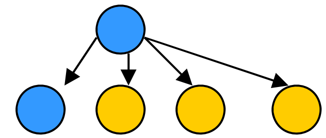
Constraints + assignments
can entail new equations and disequations

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

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

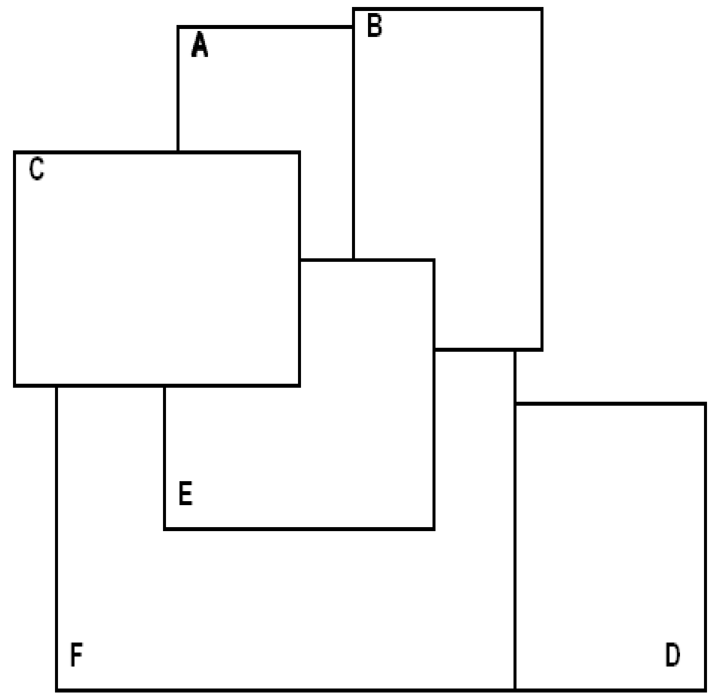
Constraint propagation

A=Red



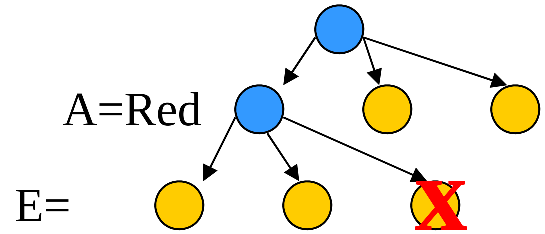
- Assign A=Red

	Red	Blue	Green
A	✓		
B			
C			
D			
E			
F			



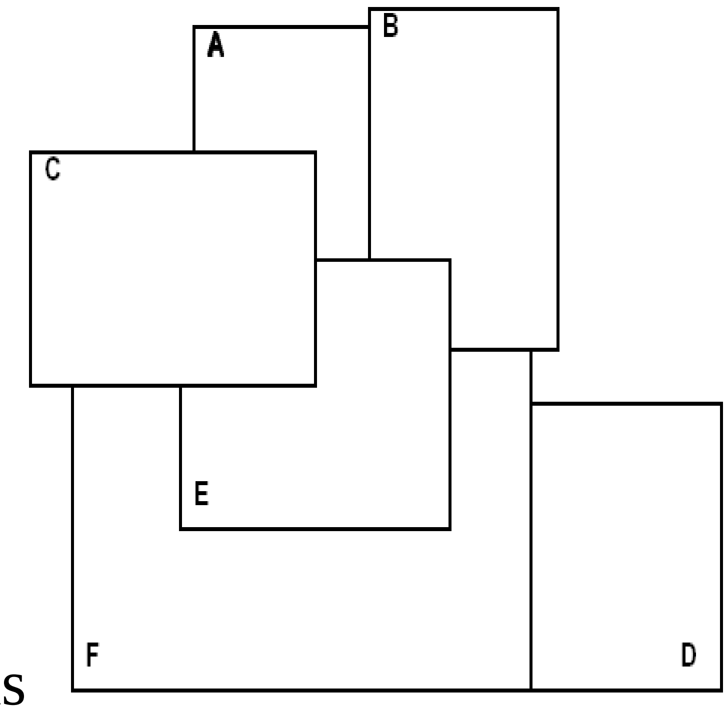
✓ - equations ✗ - disequations

Constraint propagation



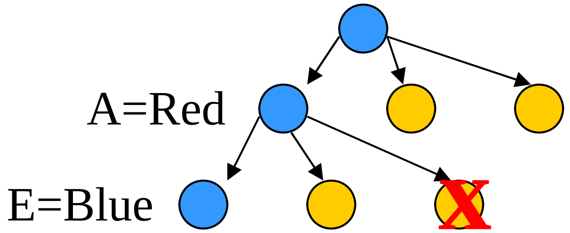
- Assign A=Red

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



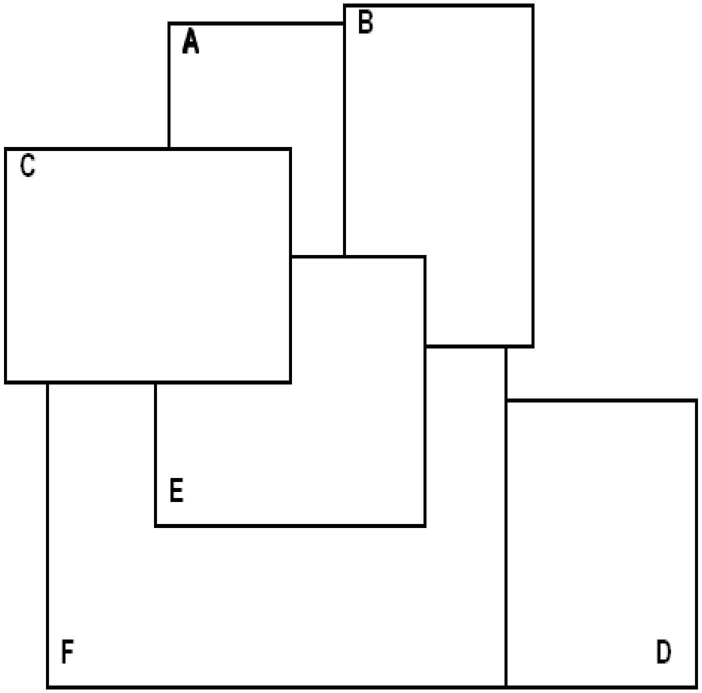
✓ - equations ✗ - disequations

Constraint propagation



- Assign E=Blue

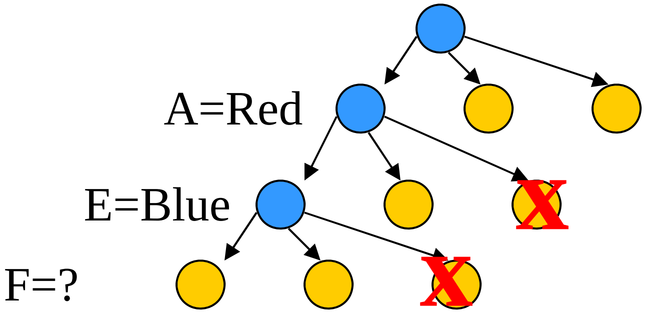
	Red	Blue	Green
A	✓		
B	✗		
C	✗		
D			
E	✗	✓	
F			

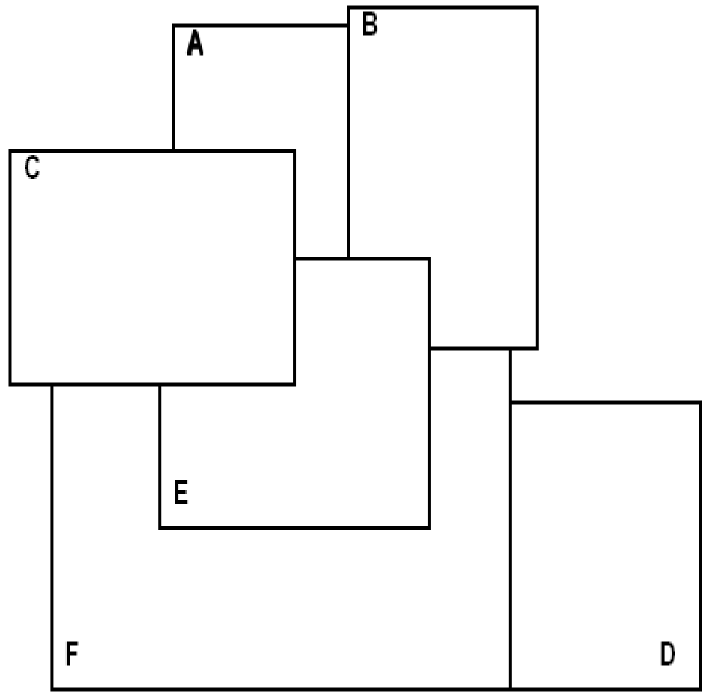


Constraint propagation

- Assign E=Blue

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

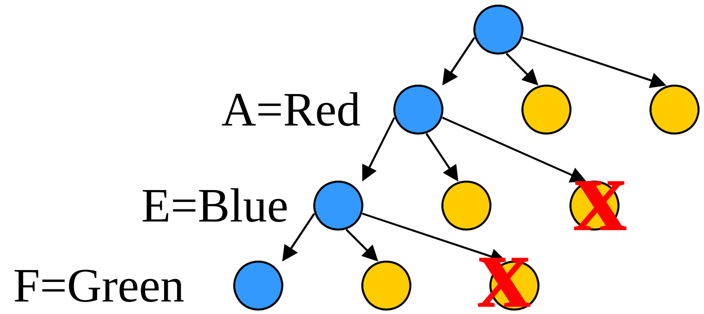


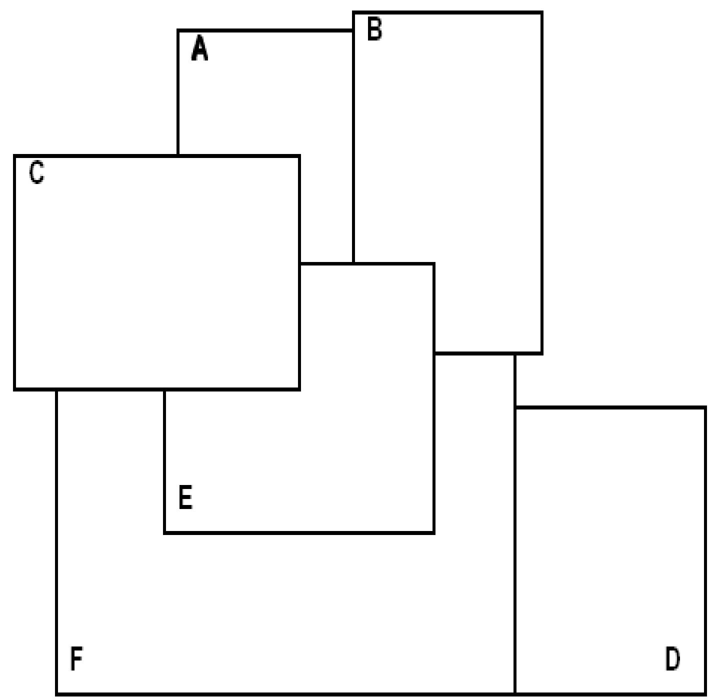


Constraint propagation

- Assign F=Green

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

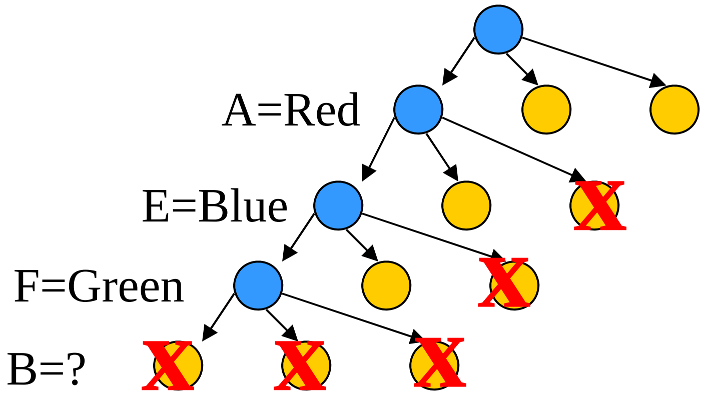


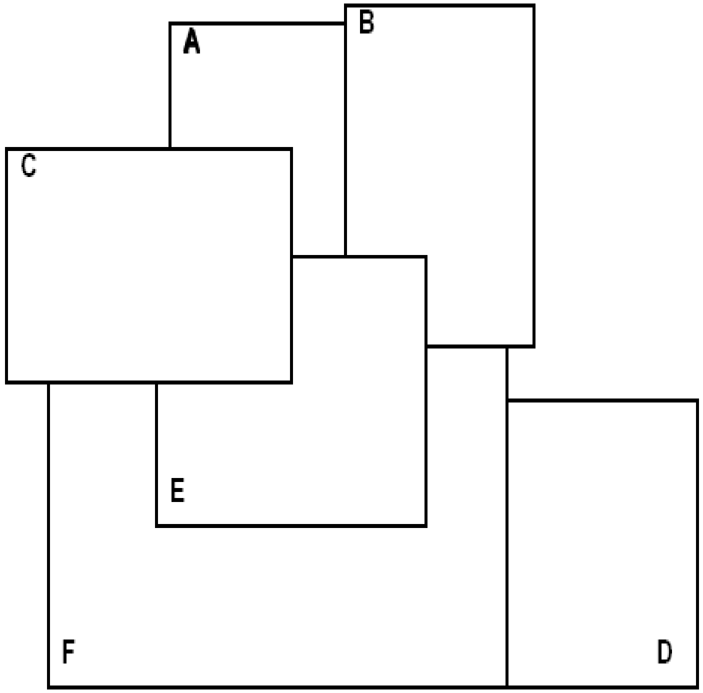


Constraint propagation

- Assign F=Green

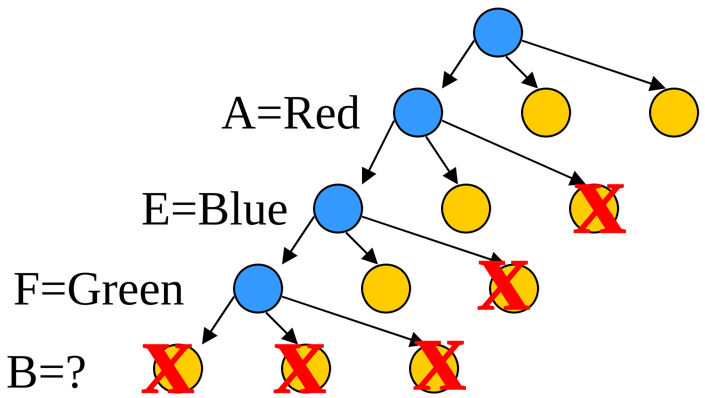
	Red	Blue	Green
A	✓	✗	
B	✗	✗	✗
C	✗	✗	✗
D			✗
E	✗	✓	✗
F		✗	✓



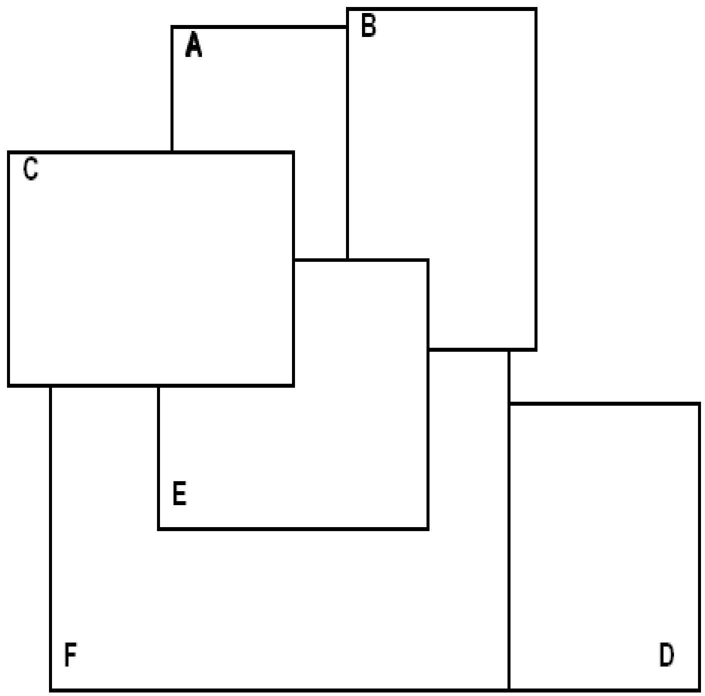


Constraint propagation

- Assign F=Green



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



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

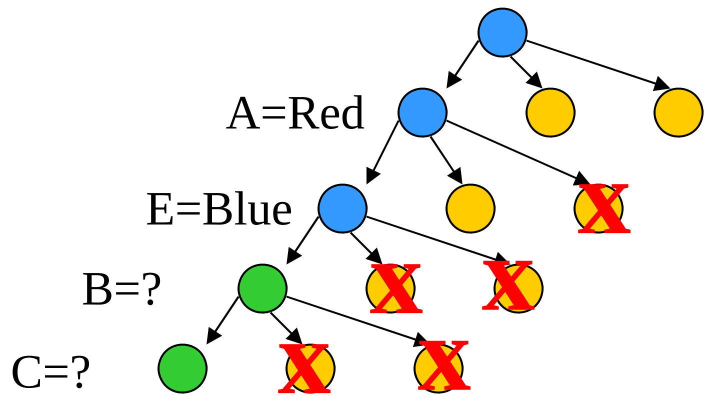
Constraint propagation

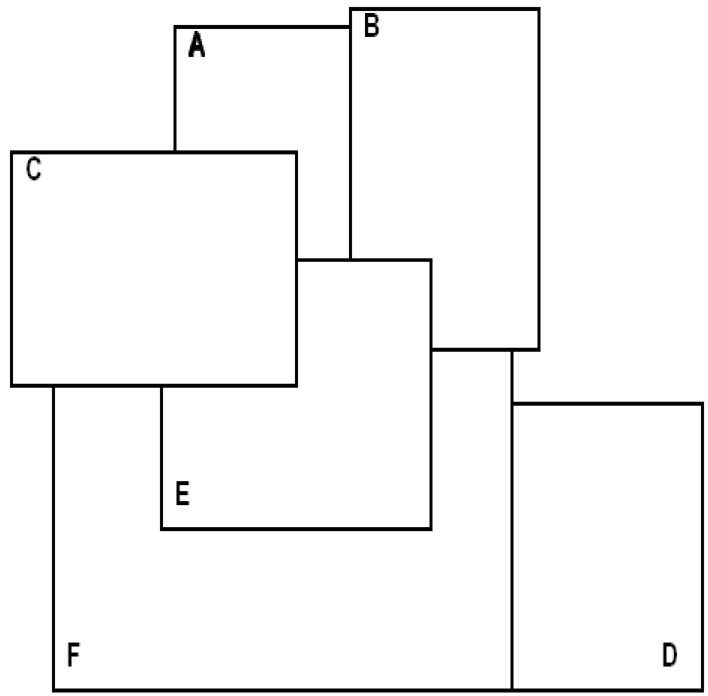
- We can derive remaining legal values through propagation

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

B=Green

C=Green



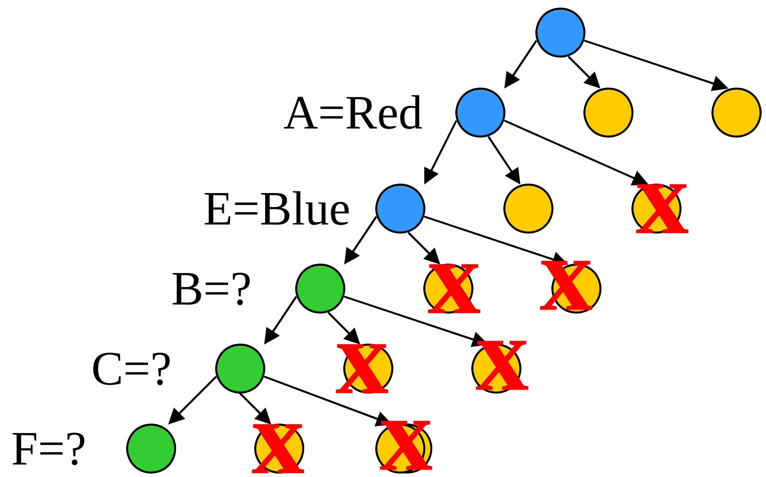


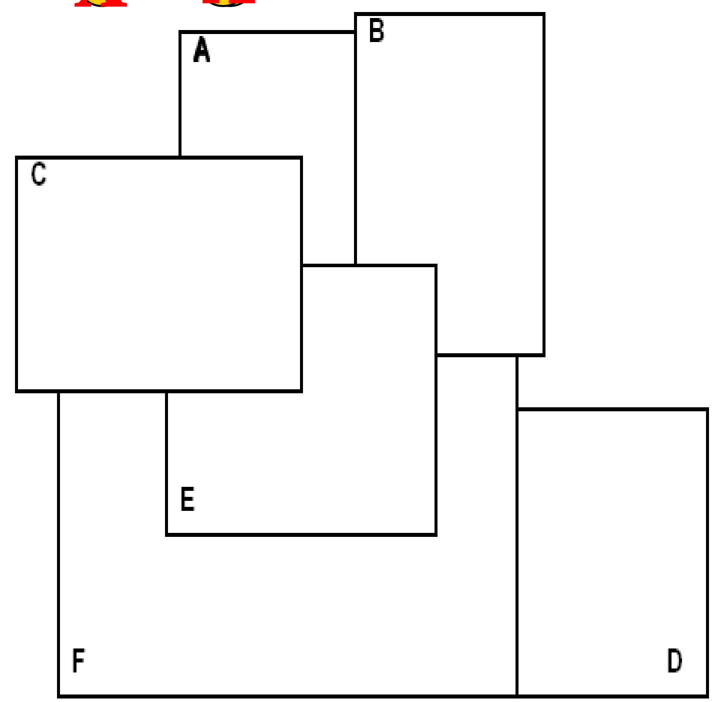
Constraint propagation

- 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**