UNIT – 2

SEARCHING FOR SOLUTIONS

1. Search Tree

- A search tree is a conceptual tool used to visualize the possible sequences of actions starting from the initial state.
- **Initial State (Root Node)**: The starting point of the problem. For example, in a route-finding problem, the initial state might be "In(Arad)."
- **Branches (Actions)**: The possible actions you can take from any given state. For example, from Arad, you can drive to Sibiu, Timisoara, or Zerind.
- **Nodes (States)**: Each node represents a state in the problem space. For instance, driving from Arad to Sibiu results in the state "In(Sibiu)."

Example: Finding a Route from Arad to Bucharest

- **Start**: The root node is "In(Arad)."
- **Goal Check**: First, we check if this is the goal state. It is not, so we expand the node.
- **Expanding the Node**: From Arad, you can travel to Sibiu, Timisoara, or Zerind. These are the child nodes.
  - In(Arad) -> In(Sibiu)
  - In(Arad) -> In(Timisoara)
  - In(Arad) -> In(Zerind)
- **Frontier**: The set of leaf nodes (nodes with no children) available for expansion. Initially, the frontier includes Sibiu, Timisoara, and Zerind.

Expanding Nodes and Choosing Paths

- The essence of the search process is choosing which node to expand next. If we choose Sibiu:
- **Goal Check**: We check if Sibiu is the goal state. It is not.
- **Expanding Sibiu**: Sibiu connects to Arad, Fagaras, Oradea, and Rimnicu Vilcea.
  - In(Sibiu) -> In(Arad)
  - In(Sibiu) -> In(Fagaras)
  - In(Sibiu) -> In(Oradea)
  - In(Sibiu) -> In(Rimnicu Vilcea)
- **New Frontier**: The frontier now includes Timisoara, Zerind, Arad (from Sibiu), Fagaras, Oradea, and Rimnicu Vilcea.

2. Redundant Paths in Problem Solving

- ***Example: 8-Queens Problem***
- **Original Formulation**:
  - In the 8-queens problem, the goal is to place 8 queens on a chessboard such that no two queens threaten each other.
  - If we allow a queen to be placed in any column, each state with n queens can be reached by n! different paths.
  - This means there are many redundant paths, making the search inefficient.
- **Improved Formulation**: If we reformulate the problem such that each new queen is placed **in the leftmost empty column**, each state can be reached by only one path. This eliminates redundant paths, simplifying the search process
- **This formulation reduces the 8-queens state space from 1.8×1014 to just 2,057, and solutions are easy to find.**

Problems with Reversible Actions

- In some problems, such as **route-finding** and **sliding-block puzzles,** redundant paths are unavoidable because actions are reversible.

Example: Route-Finding on a Rectangular Grid

- **Grid Structure**: In a rectangular grid, each state (or cell) has four possible successors (up, down, left, right). This leads to a large number of possible paths.
- **Depth and Complexity**: A search tree of depth d including repeated states has 4^d leaves, but only about $2d^2$distinct states within d steps of any given state.
  - For d=20, this results in about a trillion nodes but only about 800 distinct states.

Impact of Redundant Paths

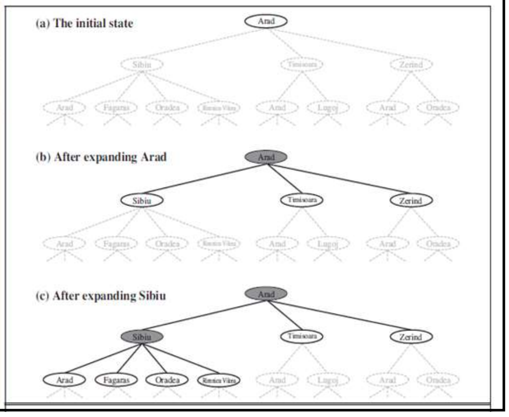
- Following redundant paths can **cause a tractable problem to become intractable**, even for algorithms that avoid infinite loops.

3. Handling Redundant and Loopy Paths

- **Redundant Paths**: Paths that lead to the same state but are less efficient. For instance, reaching Sibiu via Arad-Zerind-Oradea is longer than directly from Arad.
- **Loopy Paths**: Paths that revisit the same state, such as Arad-Sibiu-Arad. These make the search tree infinite and are unnecessary because the direct path is always better.

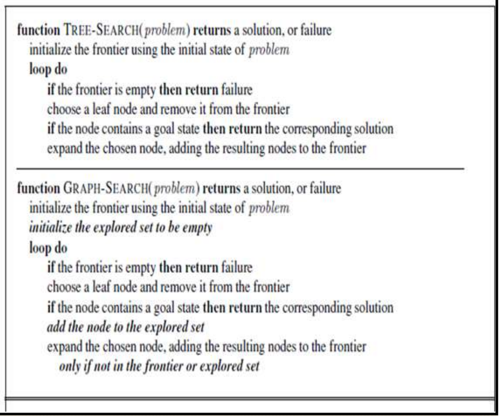
Avoiding Redundant and Loopy Paths

- To avoid exploring redundant or loopy paths, we use two concepts:
- **Explored Set (Closed List)**: Keeps track of all nodes that have been expanded. Any newly generated node that matches a node in the explored set or frontier can be discarded.
- **Graph-Search Algorithm**: Enhances the tree-search algorithm by using the explored set. This ensures each state is considered only once, avoiding redundant paths.



General Process

- **Initialize**: Start with the initial state in the frontier.
- **Expand**: Expand nodes in the frontier, generating new states.
- **Check Goal**: If a node is the goal state, return the solution.
- **Update Frontier**: Add new states to the frontier unless they are in the explored set or already in the frontier.
- **Repeat**: Continue expanding nodes until a solution is found or no more states can be expanded.
- By systematically exploring the state space, the algorithm will eventually find the optimal solution if it exists.



```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     add the node to the explored set
     expand the chosen node, adding the resulting nodes to the frontier
       only if not in the frontier or explored set
```

INFRASTRUCTURE FOR SEARCH ALGORITHMS

Components of a Search Node

- Each node in the search tree contains the following four components:
- **n.STATE**: Represents the state in the state space that the node corresponds to.
- **n.PARENT**: The node in the search tree that generated this node.
- **n.ACTION**: The action applied to the parent to generate this node.
- **n.PATH-COST**: The cost, typically denoted by g(n)g(n)g(n), of the path from the initial state to this node, as indicated by the parent pointers.

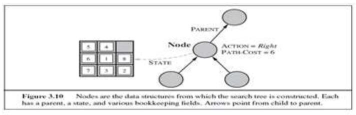
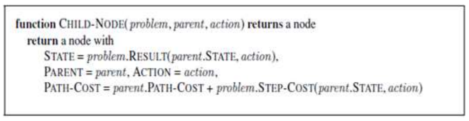


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Creating Child Nodes

- The function CHILD-NODE is used to generate a child node from a given parent node and an action. It computes the components for the child node using the parent node's information:



```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
     STATE = problem.RESULT(parent.STATE, action),
     PARENT = parent, ACTION = action,
     PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

- **Problem**: You are navigating a grid where each move (up, down, left, right) has a cost of 1.
- **Parent Node**:
  - **STATE**: (2, 3)
  - **PARENT**: Node corresponding to (2, 2)
  - **ACTION**: "Move Right"
  - **PATH-COST**: 3
  - **Action**: "Move Down"
- Using the CHILD-NODE function:
  - **STATE**:
    - problem.RESULT((2, 3), "Move Down") = (3, 3)
  - **PARENT**:
    - parent = Node corresponding to (2, 3)
  - **ACTION**:
    - action = "Move Down"
  - **PATH-COST**:
    - parent.PATH-COST = 3
    - problem.STEP-COST((2, 3), "Move Down") = 1
    - parent.PATH-COST + problem.STEP-COST((2, 3), "Move Down") = 3 + 1 = 4

Managing the Frontier

- The frontier is managed using a queue, which supports the following operations:
- **EMPTY?(queue)**: Returns true if the queue is empty.
- **POP(queue)**: Removes and returns the first element of the queue.
- **INSERT(element, queue)**: Inserts an element into the queue.

Types of Queues

- **FIFO Queue**: First-In-First-Out; the oldest element is popped first.
- **LIFO Queue (Stack)**: Last-In-First-Out; the newest element is popped first.
- **Priority Queue**: Elements are popped based on their priority, according to some ordering function.

Explored Set (Closed List)

- The explored set keeps track of all expanded nodes to avoid revisiting states and generating redundant paths.
- It is often implemented with a hash table for efficient insertion and lookup.
- **Example: Search Algorithm in Action**
- Consider a route-finding problem where you need to find the shortest path from city Arad to Bucharest.
- **Initial State**:
  - **STATE**: In(Arad)
  - **PARENT**: None (root node)
  - **ACTION**: None
  - **PATH-COST**: 0
  - **Frontier**: {In(Arad)}
  - **Explored Set**: {}

- First Expansion: Expand In(Arad)
  - Generate children: In(Sibiu), In(Timisoara), In(Zerind)
  - Update nodes:
    - In(Sibiu) with parent In(Arad), action "Arad to Sibiu", path-cost 140
    - In(Timisoara) with parent In(Arad), action "Arad to Timisoara", path-cost 118
    - In(Zerind) with parent In(Arad), action "Arad to Zerind", path-cost 75
  - **Frontier**: {In(Sibiu), In(Timisoara), In(Zerind)}
  - **Explored Set**: {In(Arad)}
- Next Expansion: Expand In(Zerind)
  - Generate children: In(Oradea)
  - Update nodes:
    - In(Oradea) with parent In(Zerind), action "Zerind to Oradea", path-cost 146
  - **Frontier**: {In(Sibiu), In(Timisoara), In(Oradea)}
  - **Explored Set**: {In(Arad), In(Zerind)}
- Continue Expansion: Expand In(Sibiu), generating In(Fagaras), In(Rimnicu Vilcea), etc., updating frontier and explored set accordingly.

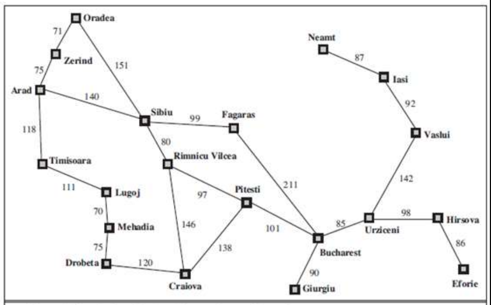


Figure 3.2 A simplified road map of part of Romania.

MEASURING PROBLEM-SOLVING PERFORMANCE

- We can evaluate an algorithm's performance in four ways:
- COMPLETENESS: Is the algorithm guaranteed to find a solution when there is one?
- OPTIMALITY: Does the strategy find the optimal solution?
- TIME COMPLEXITY: How long does it take to find a solution?
- SPACE COMPLEXITY: How much memory is needed to perform the search?

- Complexity is typically measured with respect to the size of the problem, often represented by the state space graph with vertices **|V| and edges |E|.** This is appropriate when the graph is an explicit data structure.
- In AI, however, the graph is often represented implicitly by the initial state, actions, and transition model, and can be infinite. Therefore, complexity is expressed in terms of:
  - **Branching factor b**: Maximum number of successors of any node.
  - **Depth d**: Depth of the shallowest goal node.
  - **Maximum path length m**: Longest path in the state space.
- **Time complexity** is measured by the number of nodes generated during the search.
- **Space complexity** is measured by the maximum number of nodes stored in memory.
- For search on a tree, time and space complexity are straightforward, but for a graph, it depends on the redundancy of paths.
- **Search cost** mainly depends on time complexity but can also include memory usage.
- **Total cost** combines search cost and the path cost of the solution found.

UNINFORMED (BLIND) SEARCH STRATEGIES

- These are methods used to explore a problem space without any additional information beyond what is given in the problem definition.
- These strategies do not use any domain-specific knowledge and can only:
  - **Generate Successors**: Create new states from the current state by applying available actions.
  - **Distinguish Goal States**: Identify if a state is a goal state or not.
- Uninformed search strategies are characterized by the order in which they expand nodes.

Some common uninformed search strategies include

- **Breadth-First Search (BFS)**: Expands the shallowest nodes first.
- **Depth-First Search (DFS)**: Expands the deepest unexpanded node first.
- **Uniform-Cost Search (UCS)**: Expands the node with the lowest path cost first.
- **Depth-Limited Search (DLS)**: Depth-first search with a depth limit.
- **Iterative Deepening Search (IDS)**: Repeatedly applies depth-limited search with increasing depth limits.

1. Breadth-First Search (BFS)

- Breadth-First Search (BFS) is a simple search strategy that explores all nodes at the present depth level before moving on to nodes at the next depth level.
  - Start from the root node.
  - Expand all the root's successors.
  - Expand all nodes at the next depth level, and so on.

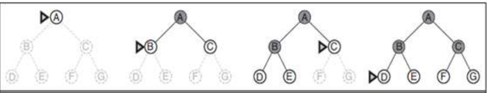
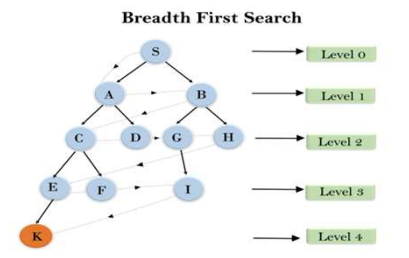
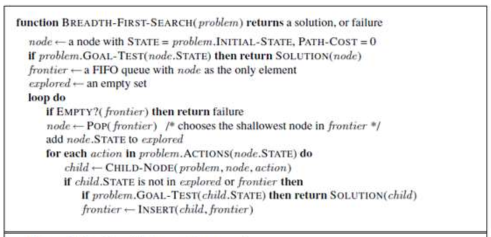


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.



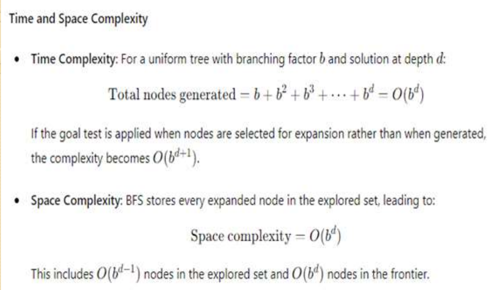
S--->A--->B---->C--->D---->G--->H--->E---->F---->I---->K



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the shallowest node in frontier */
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
              frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

- New nodes (deeper than their parents) are added to the back of the queue.
- Older nodes (shallower than new nodes) are expanded first.
- The goal test is applied when nodes are generated, not when they are expanded, ensuring the shallowest path to each node on the frontier.



Time and Space Complexity

- **Time Complexity:** For a uniform tree with branching factor b and solution at depth d:

$$\text{Total nodes generated} = b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

If the goal test is applied when nodes are selected for expansion rather than when generated, the complexity becomes $O(b^{d+1})$.

- **Space Complexity:** BFS stores every expanded node in the explored set, leading to:

$$\text{Space complexity} = O(b^d)$$

This includes $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier.

Drawbacks

- *First, the memory requirements are a bigger problem for breadth-first search than is the execution time.*
- *Second, for deep solutions, BFS can become impractical due to exponential growth in the number of nodes.*

2. Depth-First Search (DFS) Strategy

- DFS explores the deepest nodes in the current frontier of the search tree first.
- Once a node is expanded, if it has no successors, the search backs up to the next deepest node with unexplored successors.

Implementation:

- DFS can be implemented using a LIFO (Last In, First Out) queue, meaning the most recently generated node is chosen for expansion.
- It can also be implemented recursively, where a function calls itself on each child node in turn.

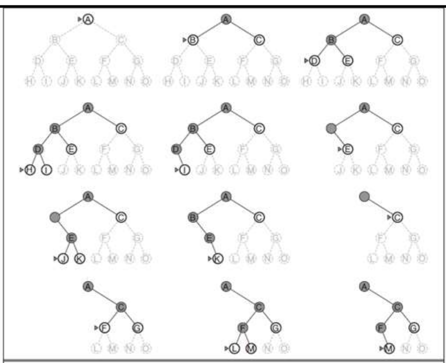


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Types of DFS

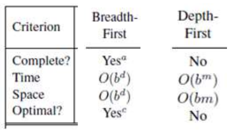
- **Graph-Search Version:**
  - Avoids repeated states and redundant paths.
  - Complete in finite state spaces (eventually expands every node).
- **Tree-Search Version:**
  - Does not avoid repeated states and can get stuck in loops (e.g., Arad-Sibiu loop).
  - Not complete in infinite state spaces if an infinite non-goal path is encountered.

Properties

- **Completeness:**
  - Graph-search is complete in finite state spaces.
  - Tree-search is not complete in some cases (e.g., infinite loops).
- **Optimality:**
  - DFS is non-optimal. It might not find the shortest path or best solution.
- **Time Complexity:**
  - For tree-search, DFS may generate all **O(b^m) nodes**, where b is the branching factor and m is the maximum depth.
- **Space Complexity:**
  - Graph-search: No significant advantage in space.
  - Tree-search: Stores only a single path from root to leaf and unexpanded sibling nodes, requiring **O(bm) space.**

- **Memory Efficiency:** DFS is preferred in memory-limited situations because it requires much less space than Breadth-First Search (BFS).
- **Backtracking Search:** A variant of DFS that generates only one successor at a time, needing only O(m) memory.

- DFS is essential in AI for tasks such as constraint satisfaction, propositional satisfiability, and logic programming due to its lower memory requirements compared to BFS.



Criterion	Breadth-First	Depth-First
Complete?	Yes[a]	No
Time	$O(b^d)$	$O(b^m)$
Space	$O(b^d)$	$O(bm)$
Optimal?	Yes[c]	No

INFORMED (HEURISTIC) SEARCH STRATEGIES

- Informed search strategies use problem-specific knowledge beyond the basic problem definition to find solutions more efficiently than uninformed strategies.

Best-First Search

Definition:

- Best-first search is a search algorithm where a node is selected for expansion based on an evaluation function f(n).
- The node with the lowest evaluation f(n)is expanded first.
- Best-first search can be implemented using either the general TREE-SEARCH or GRAPH-SEARCH algorithms.

- The informed search algorithm is more useful for large search space.
- Informed search algorithm uses the idea of heuristic, so it is also called **Heuristic search**.

Heuristics function: Heuristic 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 its input and produces the estimation of how close agent is from the goal.
- The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time.
- Heuristic function estimates how close a state is to the goal.

- **A heuristic function h(n)** estimates the cost of the cheapest path from the state at node n to a goal state.
- Unlike the cost function g(n), h(n) depends only on the state at node n, not on the path taken to reach that node.
- ***Cost Function g(n):*** Cumulative and path-dependent, reflecting the actual cost from the start node to the current node.
- ***Heuristic Function h(n):*** Estimate and state-dependent, reflecting an estimated cost from the current node to the goal, independent of the path taken to the current node.
- Heuristic functions are used **to incorporate additional problem-specific knowledge** into the search algorithm.
- They are often **arbitrary, nonnegative**, problem-specific functions, with the constraint that if n is a goal node, then $h(n)=0$.
  - In the Romania example, the heuristic function might estimate the cost of the cheapest path from Arad to Bucharest by using the straight-line distance between the two cities.

Two Ways to Use Heuristic Information

- Heuristic information can be used in different ways to guide the search. Two common approaches are:
- **Greedy Best-First Search:**
  - Uses the heuristic function h(n) directly as the evaluation function f(n).
  - Selects nodes with the lowest heuristic value for expansion, prioritizing nodes that appear closer to the goal.
- ***A Search:****
  - Combines the cost function g(n) (the cost to reach node n) and the heuristic function h(n) to form the evaluation function: f(n)=g(n)+h(n)
  - Balances the cost incurred so far and the estimated cost to the goal, providing a more balanced search approach.

Greedy Best-First Search: Expands the node closest to the goal using heuristic f(n)=h(n).
Romania Example: Uses straight-line distance to Bucharest as the heuristic.
Efficiency: Finds a solution quickly without expanding unnecessary nodes.
Non-Optimality: The found path may be longer than the optimal path due to the greedy approach.
Incompleteness: In finite state spaces, can get stuck in loops or dead ends.
Complexity: Tree version worst-case is O(b^m) graph search version is complete in finite spaces.
Heuristic Quality: Good heuristics can substantially reduce time and space complexity.

Informed Search

- Hill climbing
- A* Algorithm
- Alpha-Beta Pruning

Hill Climbing

- Hill climbing algorithm is a local search algorithm which **continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem**. It terminates when it reaches a peak value where no neighbour has a higher value.
- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems.
- One of the widely discussed examples of Hill climbing algorithm is **Traveling-salesman Problem** in which we need to minimize the distance travelled by the salesman.
- It is also called **greedy local search** as it only looks to its good immediate neighbour state and not beyond that**.**

Features of Hill Climbing

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

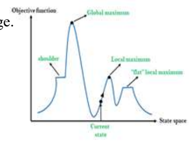
Hill Climbing Different regions in the State Space Diagram:

Local Maximum: Local maximum is a state which is better than its neighbour states, but there is also another state which is higher than it.

Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.

Current state: It is a state in a landscape diagram where an agent is currently present.

Flat local maximum: It is a flat space in the landscape where all the neighbour agents of current states have the same value.
Ridge: It is a region that is higher than its neighbors but itself has a slope. It is a special kind of local maximum.
Shoulder: It is a plateau that has an uphill edge.



Problems in different regions in Hill climbing

Hill climbing cannot reach the optimal/best state(global maximum) if it enters any of the following regions :

- **Local maximum:** At a local maximum all neighboring states have a value that is worse than the current state. Since hill-climbing uses a greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.

To overcome the local maximum problem: Utilize the backtracking technique. Maintain a list of visited states. If the search reaches an undesirable state, it can backtrack to the previous configuration and explore a new path.

Problems in different regions in Hill climbing

- **Plateau:** On the plateau, all neighbors have the same value. Hence, it is not possible to select the best direction.

To overcome plateaus: Make a big jump. Randomly select a state far away from the current state. Chances are that we will land in a non-plateau region.

- **Ridge:** Any point on a ridge can look like a peak because movement in all possible directions is downward. Hence the algorithm stops when it reaches this state.

To overcome Ridge: In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

Applications of Hill Climbing Algorithm

- **Machine Learning:** Hill climbing can be used for hyper parameter tuning in machine learning algorithms, finding the best combination of hyper parameters for a model.
- **Robotics:** In robotics, hill climbing can help robots navigate through physical environments, adjusting their paths to reach a destination.
- **Network Design:** It can be used to optimize network topologies and configurations in telecommunications and computer networks.
- **Game Playing:** In game playing AI, hill climbing can be employed to develop strategies that maximize game scores.
- **Natural Language Processing:** It can optimize algorithms for tasks like text summarization, machine translation, and speech recognition.

Hill Climbing Algorithm

1. Evaluate the initial state. If it is also goal state then return it, otherwise continue with the initial state as the current state.
2. Loop until the solution is found or until there are no new operators to be applied in the current state
   a) Select an operator that has not yet been applied to the current state and apply it to produce new state
   b) Evaluate the new state
      i. If it is a goal state then return it and quit
      ii. If it is not a goal state but it is better than the current state, then make it as current state
      iii. If it is not better than the current state, then continue in loop.

Hill Climbing Algorithm

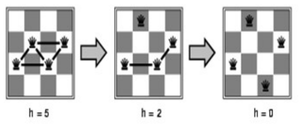
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
       neighbor ← a highest-valued successor of current
       if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
       current ← neighbor
```

Problem: https://www.youtube.com/watch?v=wM4n12FHelM

4-Queens

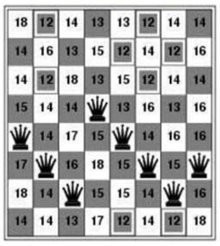
- **States:** 4 queens in 4 columns (256 states)
- **Neighborhood Operators:** move queen in column
- **Evaluation / Optimization function:** h(n) = number of attacks
- **Goal test:** no attacks, i.e., h(G) = 0

Initial state (guess).



Local search: Because we only consider local changes to the state at each step. We generally make sure that series of local changes can reach all possible states.

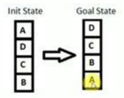
Hill-climbing search: 8-queens problem



- Need to convert to an optimization problem
- *h* = number of pairs of queens that are attacking each other
- *h* = *17* for the above state

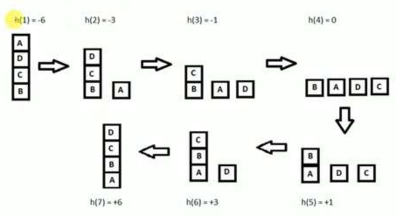
Hill Climbing Algorithm - Example

- To understand the concept easily, we will take up a very simple example



- Let's define such function *h:*
- ***h(x)* = +1 for all the blocks in the support structure if the block is correctly positioned otherwise -1 for all the blocks in the support structure.**

Hill Climbing Algorithm - Example



Types of Hill Climbing techniques

- Simple Hill Climbing
- Steepest-Ascent hill-climbing
- Stochastic hill Climbing

Simple Hill Climbing

- Simple hill climbing is the simplest way to implement a hill climbing algorithm. **It only evaluates the neighbour node state at a time and selects the first one which optimizes current cost and set it as a current state.**
- It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state.
  - Less time consuming
  - Less optimal solution and the solution is not guaranteed

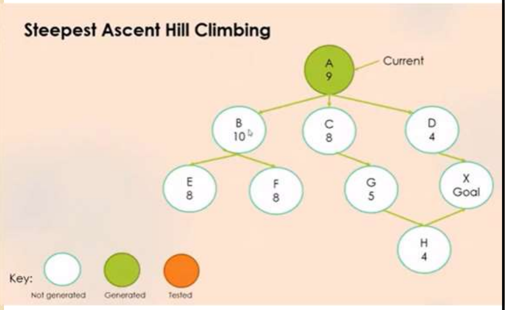
Steepest-Ascent hill-climbing

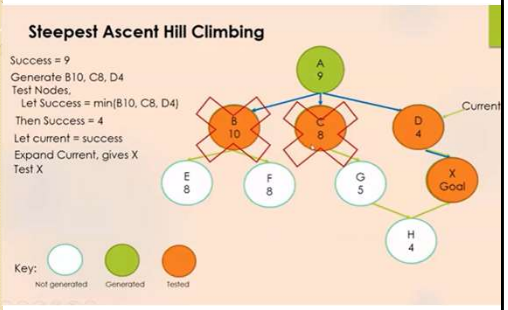
- The steepest-Ascent algorithm is a variation of simple hill climbing algorithm.
- This algorithm examines all the neighbouring nodes of the current state and selects one neighbour node which is closest to the goal state.
- This algorithm consumes more time as it searches for multiple neighbours.

Algorithm

- **Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- **Step 2:** Loop until a solution is found or the current state does not change.
  - Let SUCC be a state such that any successor of the current state will be better than it.
  - For each operator that applies to the current state:
    - Apply the new operator and generate a new state.
    - Evaluate the new state.
    - If it is goal state, then return it and quit, else compare it to the SUCC.
    - If it is better than SUCC, then set new state as SUCC.
    - If the SUCC is better than the current state, then set current state to SUCC.
- **Step 5:** Exit.

Example





Steepest Ascent Hill Climbing

Success = 9
Generate B10, C8, D4
Test Nodes,
  Let Success = min(B10, C8, D4)
Then Success = 4
Let current = success
Expand Current, gives X
Test X

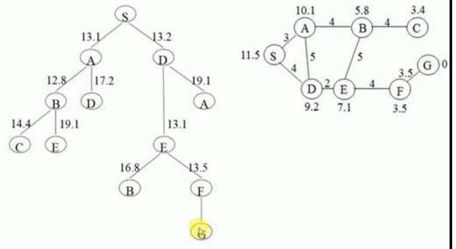
Stochastic hill climbing

- Stochastic hill climbing does not examine for all its neighbour before moving.
- Rather, this search algorithm selects one neighbour node at random and decides whether to choose it as a current state or examine another state.

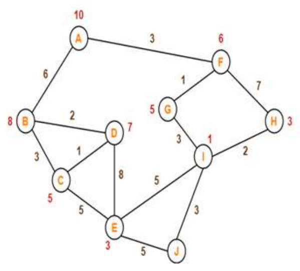
A* Searching Algorithm

- It is a searching algorithm that is used to find the shortest path between an initial and a final point.
- It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.
- Another aspect that makes A* so powerful is the use of weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.

A* (Star) Search in Artificial Intelligence



Graph that we will work on… A is Initial and J is final state



Problem Example

- **Step-01:**
- ✓ We start with node A.
- ✓ Node B and Node F can be reached from node A.
- A* Algorithm calculates f(B) and f(F).
- ✓ f(B) = 6 + 8 = 14
- ✓ f(F) = 3 + 6 = 9
- **Since f(F) < f(B), so it decides to go to node F.**

Path- A → F

Problem Example

- Node G and Node H can be reached from node F.
- A* Algorithm calculates f(G) and f(H).
- f(G) = (3+1) + 5 = 9
- f(H) = (3+7) + 3 = 13

- Since f(G) < f(H), so it decides to go to node G.

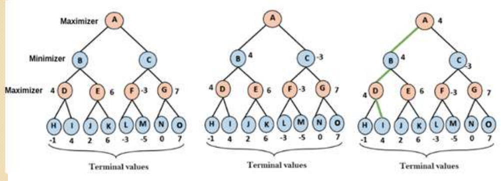
Path- A → F → G

Alpha-Beta Pruning

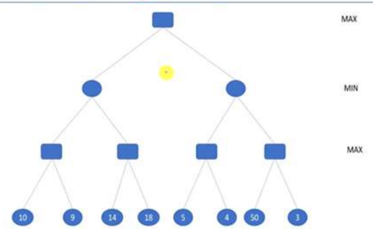
- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree.
- Since we cannot eliminate the exponent, but we can cut it to half.
- Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**.
- This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**.
- It is also called as **Alpha-Beta Algorithm**.

- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:
  - ✓ **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.
  - ✓ **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

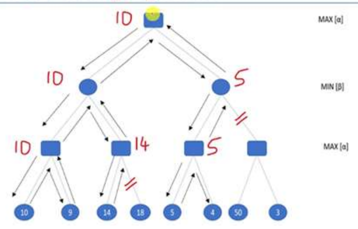
Mini-Max algorithm



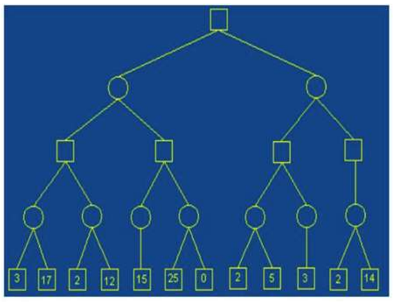
Mini Max Search Algorithm



Alpha Beta Pruning Algorithm

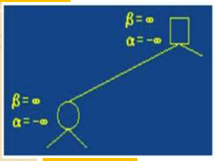


Working of Alpha-Beta Pruning Example

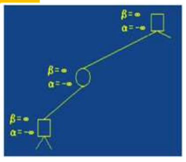


Continued…

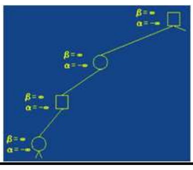
Step-1



Step-2

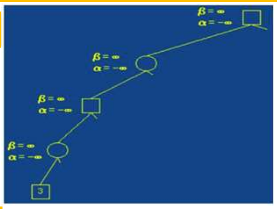


Step-3

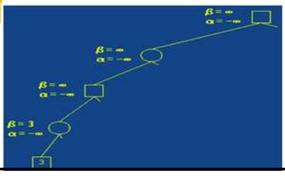


Continued…

Step-4

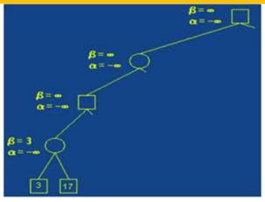


Step-5

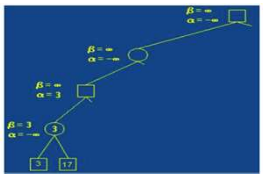


Continued…

Step-6

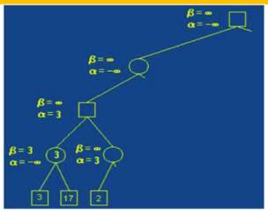


Step-7

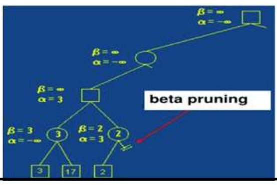


Continued…

Step-8

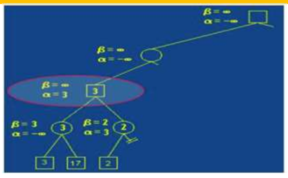


Step-9

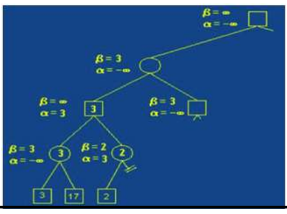


Continued…

Step-10

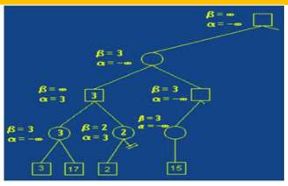


Step-12

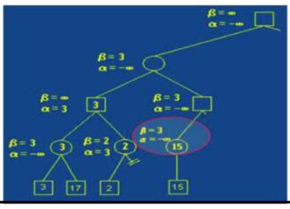


Continued…

Step-13

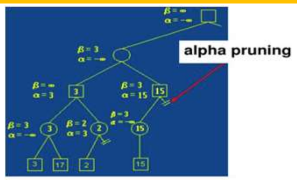


Step-14

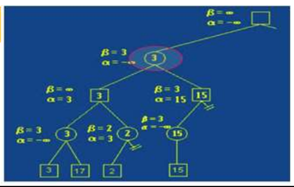


Continued…

Step-15

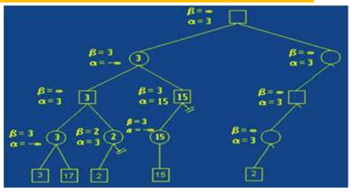


Step-16

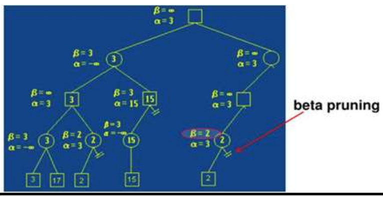


Continued…

Step-17

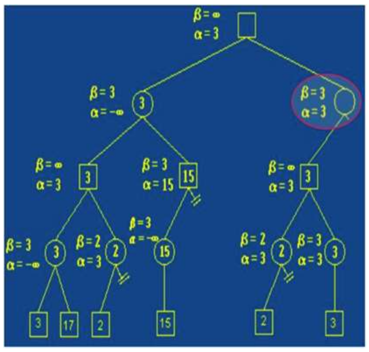


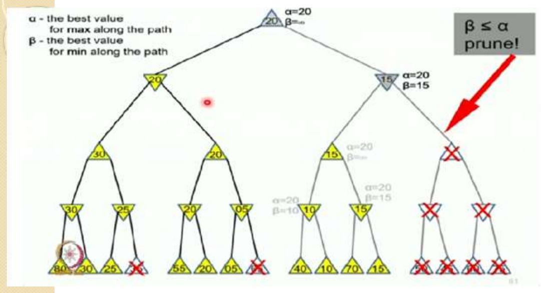
Step-18



Continued…

Final Step





CSP (Constraint Satisfaction Problem)

Standard search problem:

- **State**: In a standard search problem, the state is considered a "black box," which means it can be any kind of data structure that supports specific operations such as goal tests, evaluation, and finding successors.

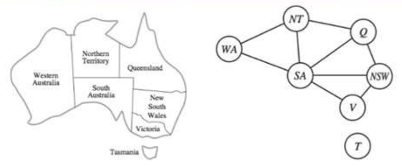
Constraint Satisfaction Problem:

- **State**: In a CSP, the state is defined by a set of variables Xi where each variable can take values from a specific domain Di.
- **Goal test**: The goal test in a CSP is a set of constraints. These constraints specify the allowable combinations of values for subsets of variables.

- CSPs provide a straightforward way to formally represent problems using **variables, domains, and constraints.**
- CSPs enable the use of powerful general-purpose algorithms that are often more effective than standard search algorithms for solving problems defined by constraints.
- Each constraint Ci consists of a pair <scope, rel >where scope is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on.

- To solve a CSP, we need to define a state space and the notion of a solution.
- Each state in a CSP is defined by an **assignment** of values to some or all of the variables, {Xi = vi,Xj = vj , . . .}.
- An assignment that does not violate any constraints is called a **consistent** or legal assignment.
- A **complete assignment** is one in which every variable is assigned, and a **solution** to a CSP is a consistent, complete assignment.
- A **partial assignment** is one that assigns values to only some of the variables.
- It can be helpful to visualize a CSP as a **constraint graph**.
- The nodes of the graph correspond to variables of the problem, and a link connects any two variables that participate in a constraint.

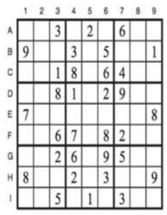
Constraint Satisfaction Probelm
Map Coloring



- Color each region either red,green or blue
- No adjacent region can have the same color

- $X = \{SA, NSW, NT, Q, WA, V\}$
- $D = \{red, blue, green\} foreach X_i \in X\}$
- $C = \{\langle(\forall X_i, X_j such\ that\ X_i\ touches\ X_j), (Color(X_i) \neq Color(X_j))\rangle\}$

CSPs
Sudoku



	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Crypt arithmetic puzzle

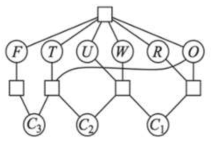
- Classic cryptarithmetic puzzle, where each letter represents a unique digit from 0 to 9.
- The problem to be solved is the equation TWO+TWO=FOUR
- The goal is to find the digit each letter represents such that the equation holds true and all digits are unique.
- The relationships between the variables in a graphical format, which helps to visualize the constraints and the dependencies between the letters.
- Each circle represents a variable (F, T, U, W, R, O), and the squares represent the constraints (C1, C2, C3, etc.).

In this graph: Each constraint (C1, C2, C3, etc.) is connected to the variables it involves.

- The connections indicate how each digit influences the other digits through the carries and the final result.

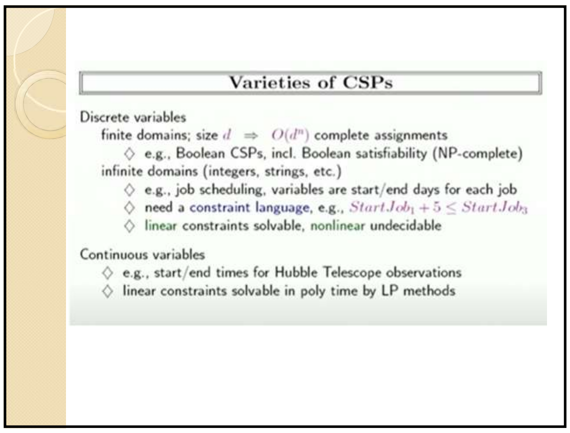
CSPs
Cryptarithmetic Puzzles

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$



Constraints:

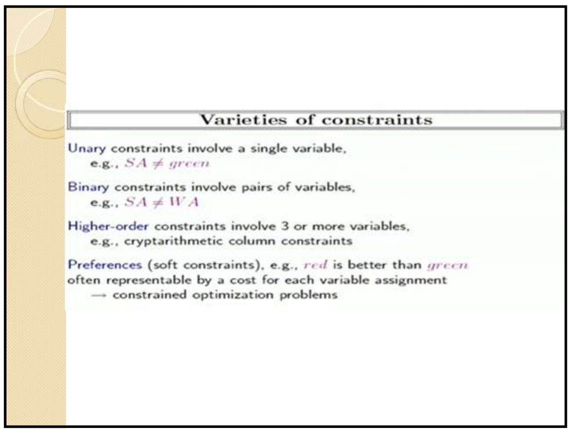
- $AllDiff(F, T, U, W, R, O)$
- $O + O = R + 10 \times C_{10}$
- $C_{10} + W + W = U + 10 \times C_{100}$
- $C_{100} + T + T = O + 10 \times C_{1000}$
- $C_{1000} = F$



Varieties of CSPs

Discrete variables
- finite domains; size $d \Rightarrow O(d^n)$ complete assignments
  - ◇ e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)
- infinite domains (integers, strings, etc.)
  - ◇ e.g., job scheduling, variables are start/end days for each job
  - ◇ need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
  - ◇ linear constraints solvable, nonlinear undecidable

Continuous variables
- ◇ e.g., start/end times for Hubble Telescope observations
- ◇ linear constraints solvable in poly time by LP methods



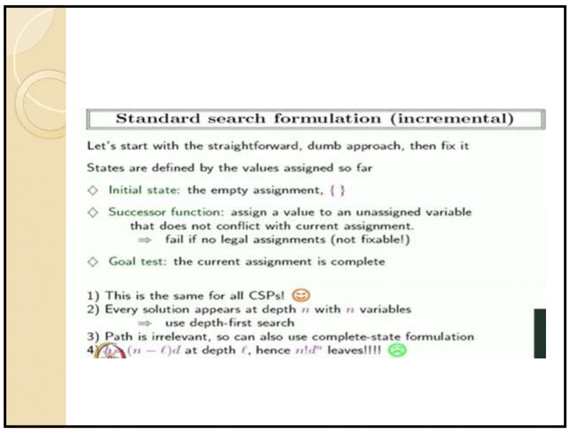
Varieties of constraints

Unary constraints involve a single variable,
e.g., $SA \neq green$

Binary constraints involve pairs of variables,
e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables,
e.g., cryptarithmetic column constraints

Preferences (soft constraints), e.g., red is better than $green$
often representable by a cost for each variable assignment
→ constrained optimization problems



Standard search formulation (incremental)

Let's start with the straightforward, dumb approach, then fix it

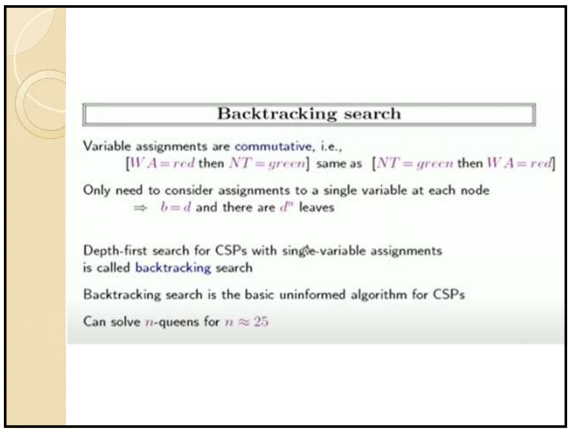
States are defined by the values assigned so far

◇ Initial state: the empty assignment, { }

◇ Successor function: assign a value to an unassigned variable
that does not conflict with current assignment.
⇒ fail if no legal assignments (not fixable!)

◇ Goal test: the current assignment is complete

1) This is the same for all CSPs! 😊
2) Every solution appears at depth n with n variables
⇒ use depth-first search
3) Path is irrelevant, so can also use complete-state formulation
4) $b = (n - \ell)d$ at depth ℓ, hence $n!d^n$ leaves!!!! ☹



Backtracking search

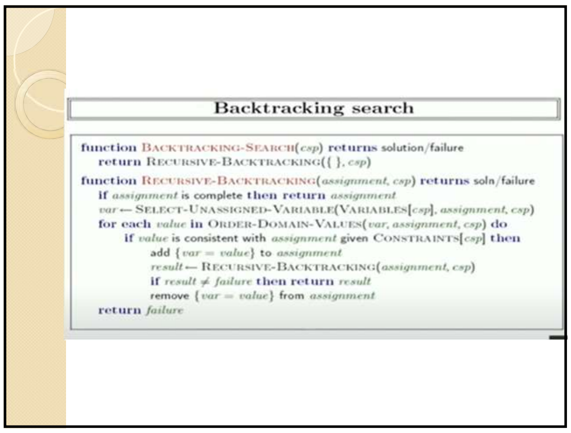
Variable assignments are commutative, i.e.,
$[WA = red$ then $NT = green]$ same as $[NT = green$ then $WA = red]$

Only need to consider assignments to a single variable at each node
⇒ $b = d$ and there are d^n leaves

Depth-first search for CSPs with single-variable assignments
is called backtracking search

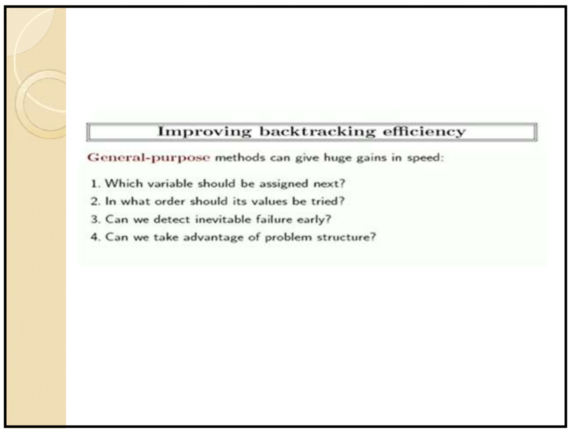
Backtracking search is the basic uninformed algorithm for CSPs

Can solve n-queens for $n \approx 25$



Backtracking search

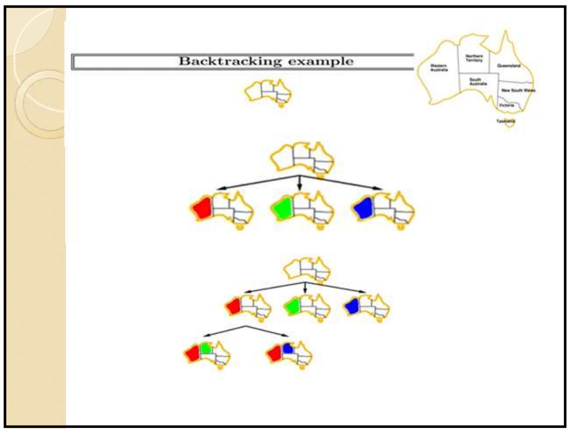
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
       if value is consistent with assignment given CONSTRAINTS[csp] then
           add {var = value} to assignment
           result ← RECURSIVE-BACKTRACKING(assignment, csp)
           if result ≠ failure then return result
           remove {var = value} from assignment
   return failure
```



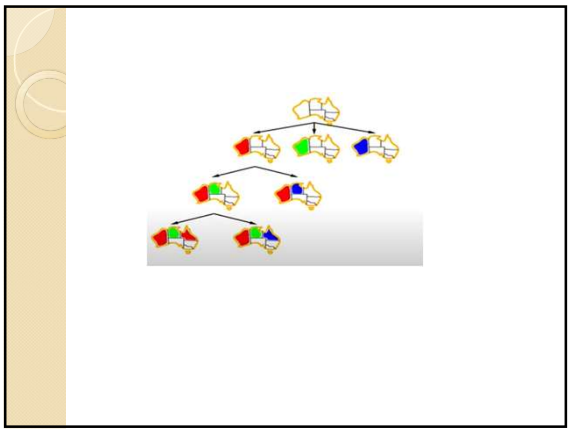
Improving backtracking efficiency

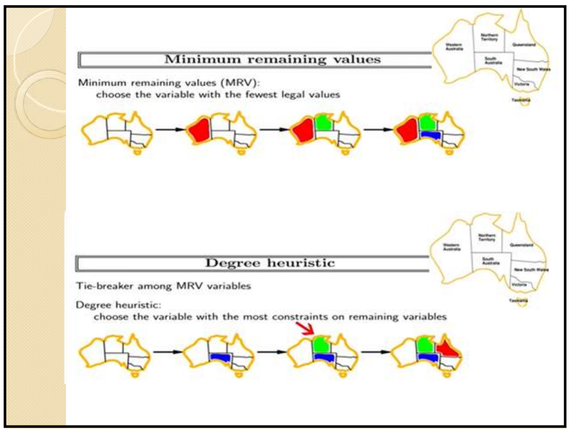
General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?



Backtracking example





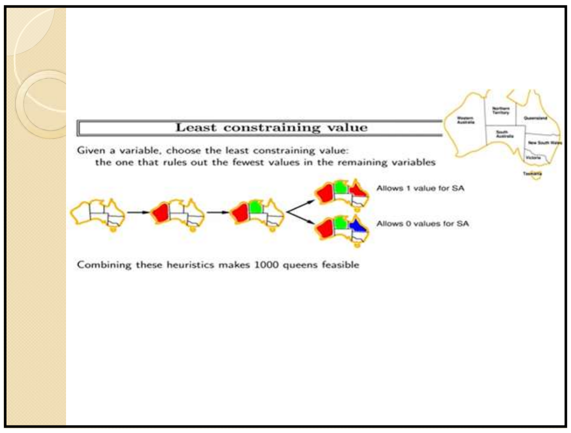
Minimum remaining values

Minimum remaining values (MRV):
choose the variable with the fewest legal values

Degree heuristic

Tie-breaker among MRV variables

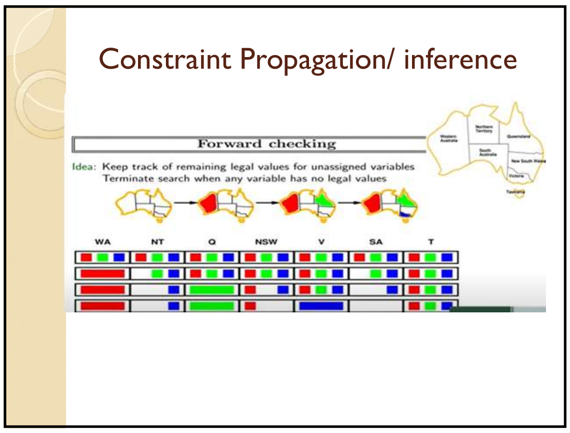
Degree heuristic:
choose the variable with the most constraints on remaining variables



Least constraining value

Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables

Combining these heuristics makes 1000 queens feasible

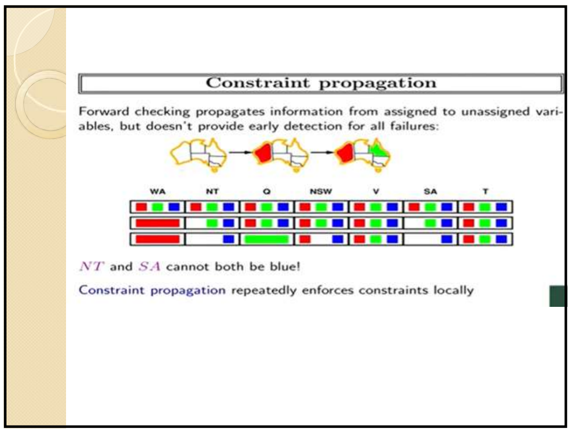


Constraint Propagation/ inference

Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦
🟥	🟦	🟩	🟥	🟦		🟥🟩🟦



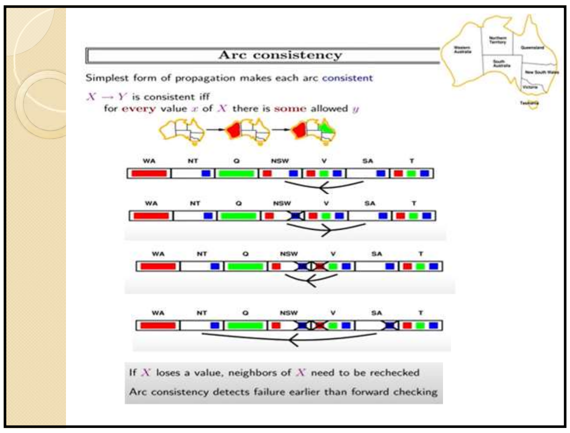
Constraint propagation

Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

NT and SA cannot both be blue!

Constraint propagation repeatedly enforces constraints locally



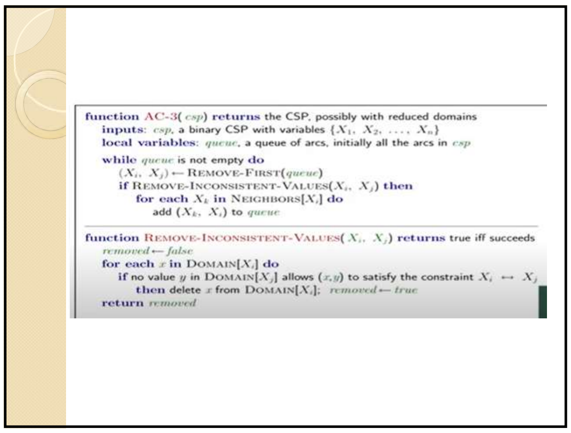
Arc consistency

Simplest form of propagation makes each arc consistent

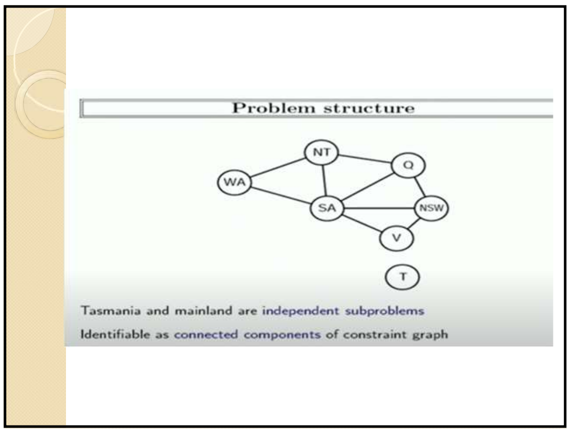
$X \to Y$ is consistent iff
for every value x of X there is some allowed y

If X loses a value, neighbors of X need to be rechecked

Arc consistency detects failure earlier than forward checking



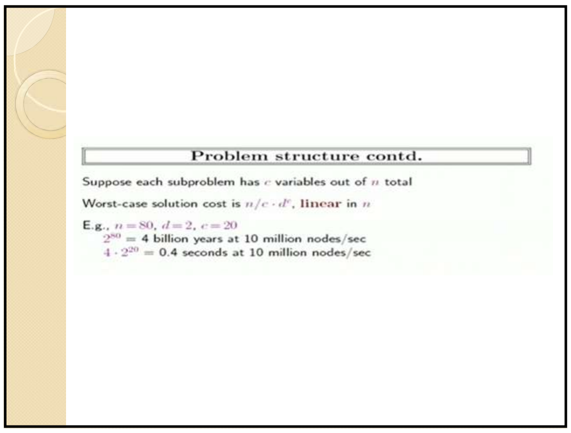
```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (X_i, X_j) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
         for each X_k in NEIGHBORS[X_i] do
            add (X_k, X_i) to queue

function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[X_i] do
      if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
         then delete x from DOMAIN[X_i]; removed ← true
   return removed
```



Problem structure

Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

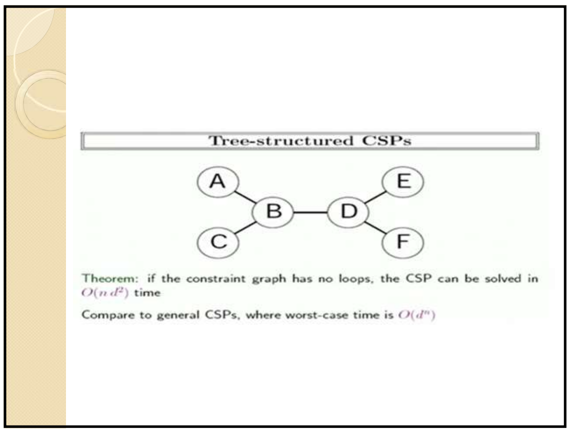


Problem structure contd.

Suppose each subproblem has c variables out of n total

Worst-case solution cost is $n/c \cdot d^c$, **linear** in n

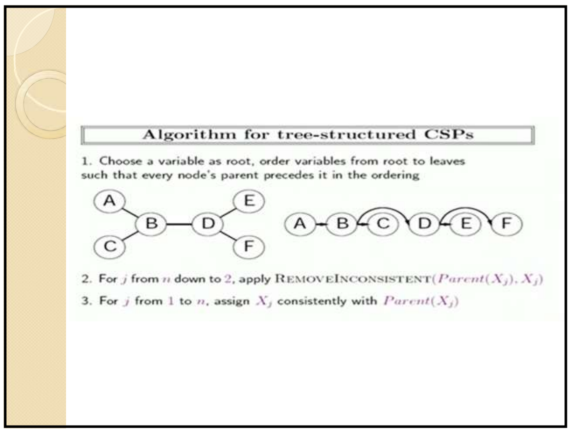
E.g., $n = 80$, $d = 2$, $c = 20$
2^{80} = 4 billion years at 10 million nodes/sec
$4 \cdot 2^{20}$ = 0.4 seconds at 10 million nodes/sec



Tree-structured CSPs

Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\,d^2)$ time

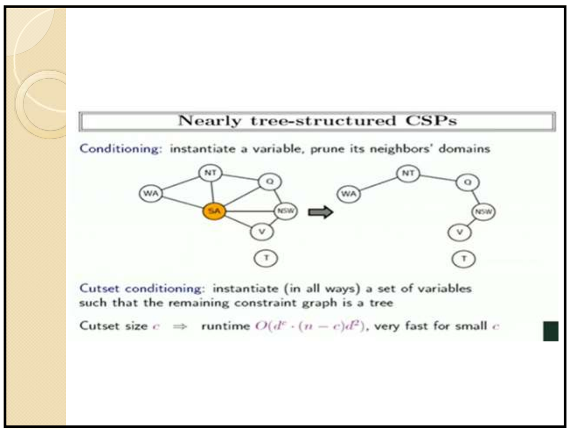
Compare to general CSPs, where worst-case time is $O(d^n)$



Algorithm for tree-structured CSPs

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering

2. For j from n down to 2, apply REMOVEINCONSISTENT$(Parent(X_j), X_j)$
3. For j from 1 to n, assign X_j consistently with $Parent(X_j)$



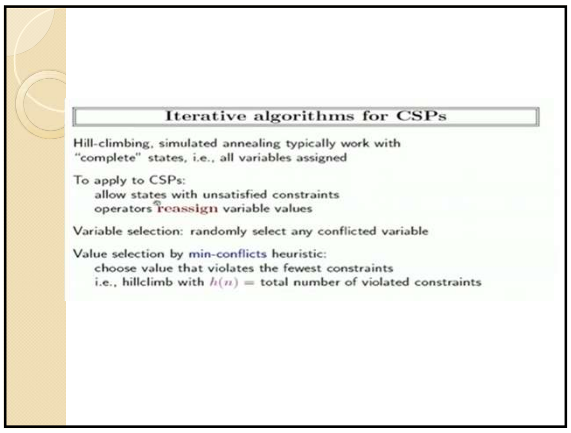
Nearly tree-structured CSPs

Conditioning: instantiate a variable, prune its neighbors' domains

WA
NT
Q
SA
NSW
V
T

Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size c $\Rightarrow$ runtime $O(d^c \cdot (n-c)d^2)$, very fast for small c



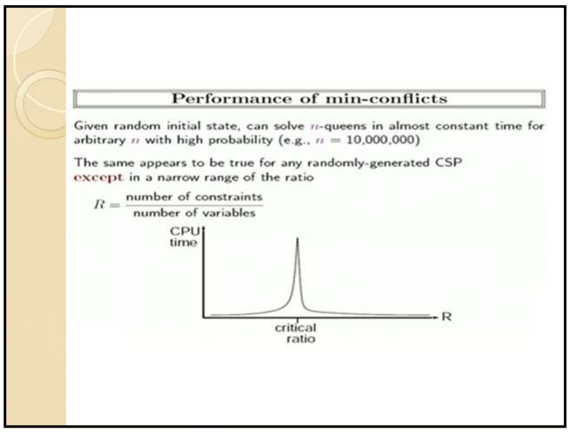
Iterative algorithms for CSPs

Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned

To apply to CSPs:
  allow states with unsatisfied constraints
  operators **reassign** variable values

Variable selection: randomly select any conflicted variable

Value selection by min-conflicts heuristic:
  choose value that violates the fewest constraints
  i.e., hillclimb with $h(n)$ = total number of violated constraints



Performance of min-conflicts

Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)

The same appears to be true for any randomly-generated CSP **except** in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$

CPU time
R
critical ratio