Solving Problem by Searching

Dr. Sumanto Dutta
School of Computer Applications
Kalinga Institute of Industrial Technology (KIIT)

January 23, 2024

Contents

Problem-Solving Agents

The problem-solving agent performs precisely by defining problems and their several solutions.

- According to psychology, *"problem-solving refers to a state where we wish to reach a definite goal from a present state or condition."*
- According to computer science, *"problem-solving is a part of artificial intelligence which encompasses several techniques such as algorithms, heuristics to solve a problem."*

Therefore, a problem-solving agent is goal-driven and focuses on satisfying the goal.

To build a system to solve a particular problem, we need to do four things:

- **Define** the problem precisely. This definition must include specification of the initial and final situations that constitute (i.e.) acceptable solutions to the problem.
- **Analyze** the problem (i.e) important features have an immense (i.e) huge impact on the appropriateness of various techniques for solving the problems.
- **Isolate and represent** the knowledge to solve the problem.
- **Choose the best problem** – solving techniques and apply it to the particular problem.

Steps performed by Problem-solving agent:

1. **Goal Formulation** → It is the first and simplest step in problem-solving. It organizes the steps/sequence required to formulate one goal out of multiple goals as well as actions to achieve that goal.
2. **Problem Formulation** → It is the most important step of problem-solving, which decides what actions should be taken to achieve the formulated goal.
3. **Search for Solution** → It identifies all the best possible sequences of actions to reach the goal state from the current state. It takes a problem as an input and returns a solution as its output.
4. **Execution** → It executes the best optimal solution from the searching algorithms to reach the goal state from the current state.

Example - Road Map of Part of Romania

On holiday in Romania; currently in Arad.
The flight leaves tomorrow from Bucharest.

- **Formulate goal** → be in Bucharest.
- **Formulate problem**
  - **states** → various cities.
  - **actions** → drive between cities.
- **Find solution** → sequence of cities,e.g., Arad, Sibiu, Fagaras, Bucharest.

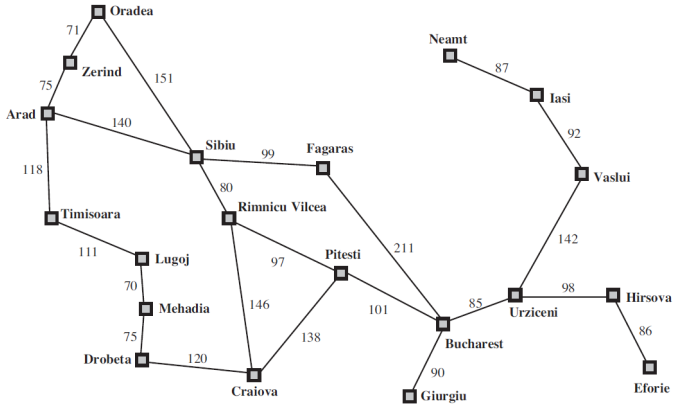


Figure 1: Road Map of Part of Romania.[1]

[a] Russell, S. J., Norvig, P. (2021). Artificial Intelligence: A Modern Approach. United Kingdom: Pearson.

Restricted form of general agent:

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← RECOMMENDATION(seq, state)
  seq ← REMAINDER(seq, state)
  return action
```

Figure 2: A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.[1]

[1] Russell, S. J., Norvig, P. (2021). Artificial Intelligence: A Modern Approach. United Kingdom: Pearson.

Problem Types

Deterministic, Fully Observable → Single-State Problem

An agent knows exactly which state it will be in; the solution is a sequence.

Non-observable → Conformant Problem

An agent may not know where it is; it reasons in terms of belief states; the solution (if any) is a sequence.

Nondeterministic and/or Partially Observable → Contingency Problem

The percepts provide new information about the current state solution is a contingent plan or a policy often interleave search, and execution.

Unknown State Space → Exploration Problem

Example - Vacuum World

- **Single-state** → start in state number 5.
  - **Solution ???**

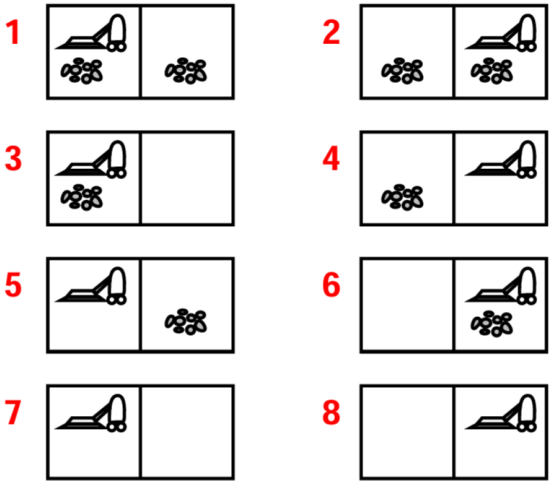


Figure 3: Vacuum World

- **Single-state** $\rightarrow$ start in state number 5.
  - **Solution** $\rightarrow$ [Right, Clean]
- **Conformant** $\rightarrow$ start in state number $\{1, 2, 3, 4, 5, 6, 7, 8\}$.
  - Right goes to $\{2, 4, 6, 8\}$. **Solution ???**

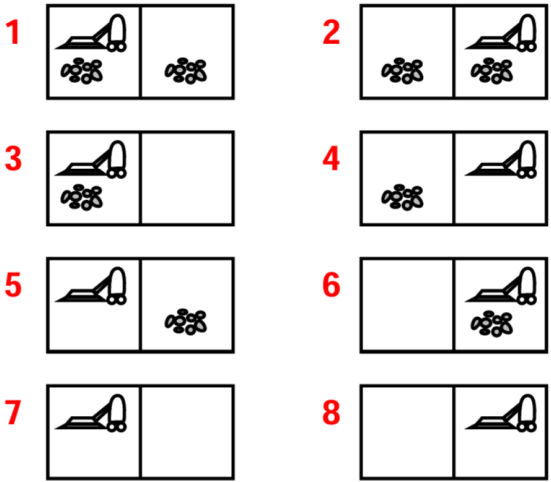


Figure 4: Vacuum World

- **Single-state** → start in state number 5.
  - **Solution** → [Right, Clean]
- **Conformant** → start in state number $\{1, 2, 3, 4, 5, 6, 7, 8\}$.
  - Right goes to $\{2, 4, 6, 8\}$.
    Solution → [Right, Clean, Left, Clean]
- **Contingency** → start in state number 5.
  - Nondeterministic: If clean operation may dirty the already cleaned carpet.
  - Partially Observable: Location, Dirt at the current location.
  - Precept : [Left, Clean]
  - **Solution ???**

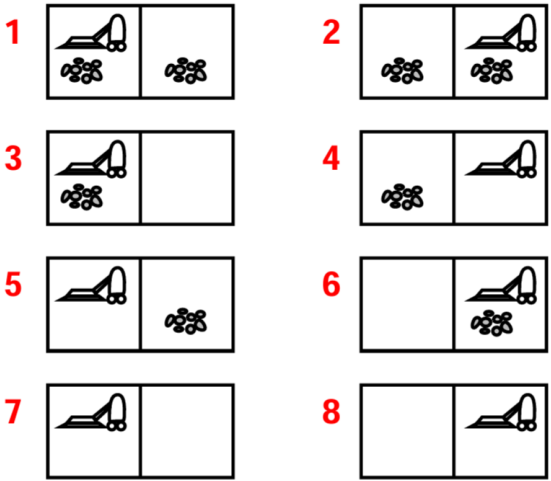


Figure 5: Vacuum World

- **Single-state** → start in state number 5.
  - **Solution** → [Right, Clean]
- **Conformant** → start in state number $\{1, 2, 3, 4, 5, 6, 7, 8\}$.
  - Right goes to $\{2, 4, 6, 8\}$.
    Solution → [Right, Clean, Left, Clean]
- **Contingency** → start in state number 5.
  - Nondeterministic: If clean operation may dirty the already cleaned carpet.
  - Partially Observable: Location, Dirt at the current location.
  - Precept : [Left, Clean]
  - **Solution**→ [Right, If dirt then Clean]

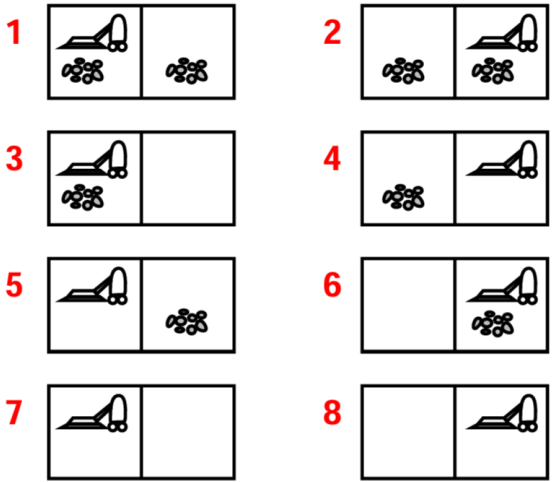


Figure 6: Vacuum World

Well-defined Problems and Solutions

A **problem** can be defined formally by **five** components:

1. **Initial State :** The initial state that the agent starts in.
   For example, the initial state of our agent in Romania might be described as In(Arad).
2. **Actions :** A description of the possible actions available to the agent. Given a particular state s, $ACTIONS(s)$ returns the set of actions that can be executed in s.
   For example, from the state $In(Arad)$, the applicable actions are $\{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$.
3. **Transition Model :** A description of what each action does. It is specified by a function $RESULT(s, a)$ that returns the state that results from doing action a in state$s$. For example, we have $RESULT(In(Arad), Go(Zerind)) = In(Zerind)$.
4. **Goal Test :** which determines whether a given state is a goal state or has an explicit set of possible goal states, and the test simply checks whether the given state is one of them.
   The agent's goal in Romania is the singleton set $\{In(Bucharest)\}$
5. **Path Cost :** which assigns a numeric cost to each path, reflecting the performance measure of an agent. The cost of a path can be described as the sum of the costs of the individual actions along the path. The step cost of taking action a in state s to reach state s' is denoted by $c(s, a, s')$, assumed to be ≥ 0.

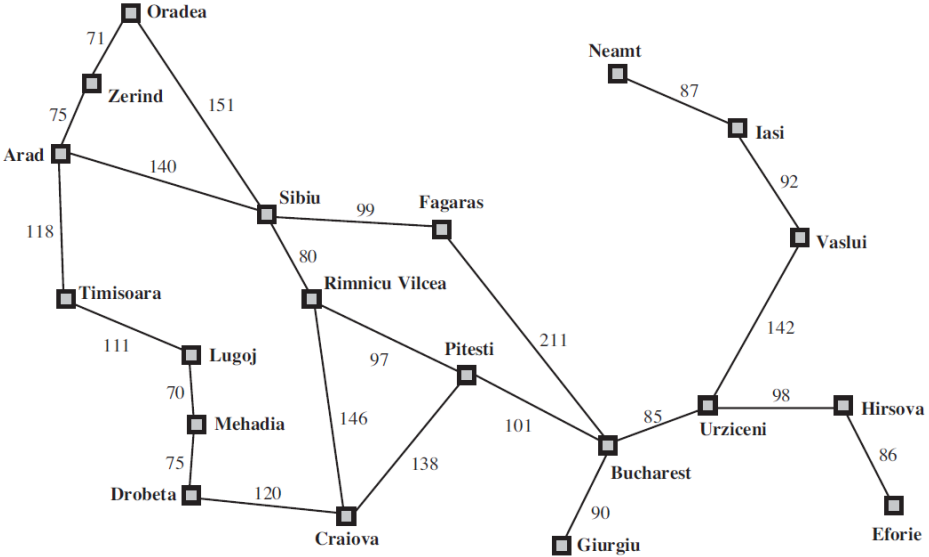


Figure 7: Road Map of Part of Romania.[1]

[1] Russell, S. J., Norvig, P. (2021). Artificial Intelligence: A Modern Approach. United Kingdom: Pearson.

State Space

Together, the **initial state, actions, and transition model** implicitly define the **state space** of the problem—the set of all states reachable from the initial state by any sequence of actions.

Path in State Space

A path in the state space is a sequence of states connected by a sequence of actions.

Solution to the Problem

A solution to a problem is an action sequence that leads from the initial state to a goal state.

Optimal Solution to the Problem

An optimal solution has the lowest path cost among all solutions.

Example

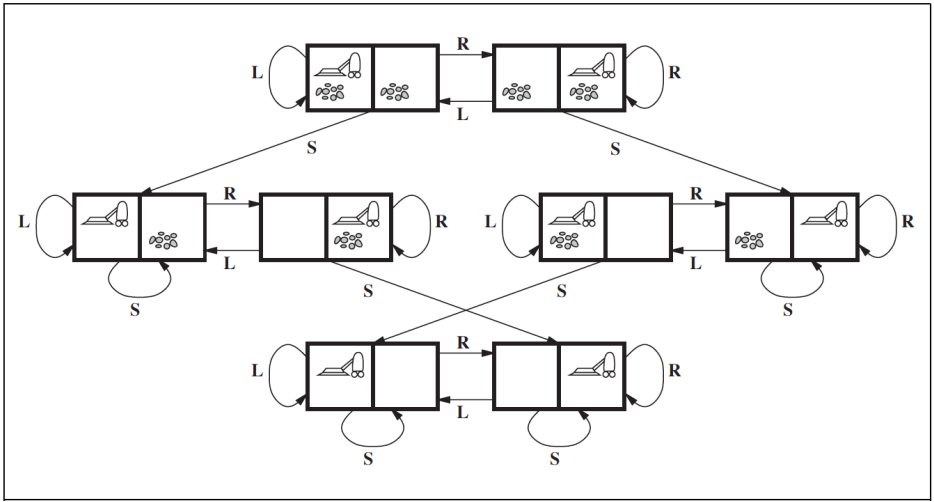


Figure 8: The state space for the vacuum world. Links denote actions: L = Left, R = Right, S = Clean.[1]

[1] Russell, S. J., Norvig, P. (2021). Artificial Intelligence: A Modern Approach. United Kingdom: Pearson.

- **States :** The state is determined by both the agent location and the dirt location. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n \cdot 2^n$ states.
- **Initial state :** Any state can be designated as the initial state.
- **Actions :** Each state has just three actions in this simple environment: Left, Right, and Clean. Larger environments might also include Up and Down.
- **Transition model :** The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Cleaning in a clean square has no effect.
- **Goal test :** This checks whether all the squares are clean.
- **Path cost :** Each step costs 1, so the path cost is the number of steps in the path.

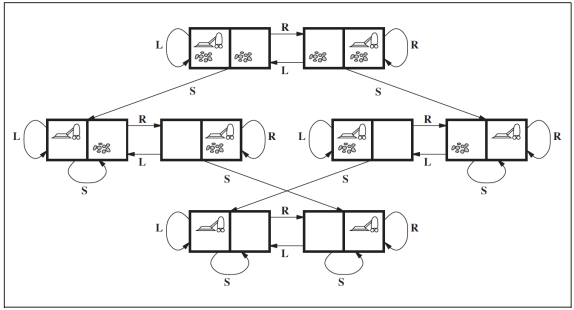


Figure 9: Vacuum World State Space

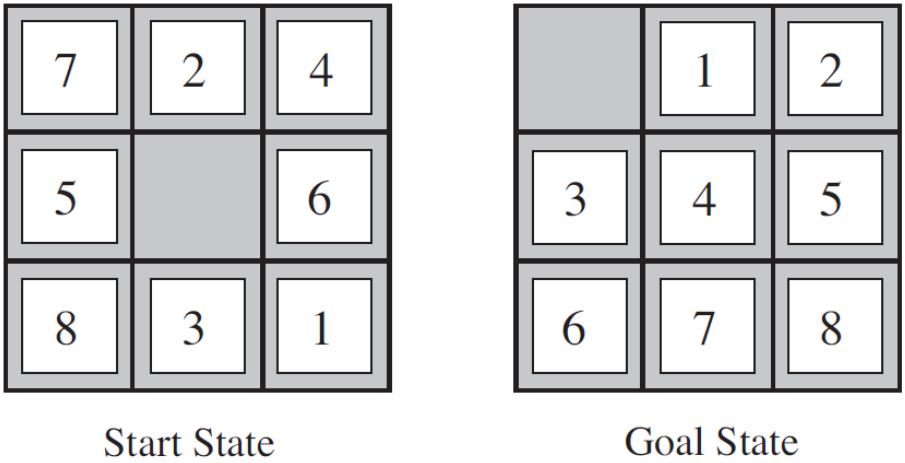


Figure 10: A typical instance of the 8-puzzle.[1]

[1] Russell, S. J., Norvig, P. (2021). Artificial Intelligence: A Modern Approach. United Kingdom: Pearson.

- **States :** A state description specifies the location of each of the eight tiles, and the blank in one of the nine squares.
- **Initial state :** Any state can be designated as the initial state.
- **Actions :** The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down.
- **Transition model :** Given a state and action, this returns the resulting state; for example, if we apply Left to the start state in Figure, the resulting state has the 5 and the blank switched.
- **Goal test :** This checks whether the state matches the goal configuration shown in Figure. Other goal configurations are possible.
- **Path cost :** Each step costs 1, so the path cost is the number of steps in the path.

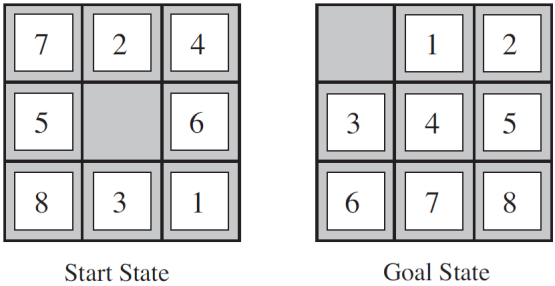
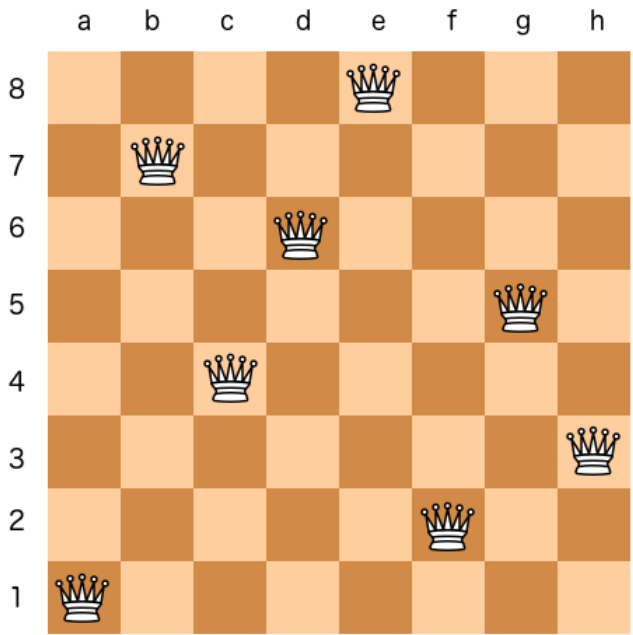


Figure 11: A typical instance of the 8-puzzle.


a b c d e f g h
8
7
6
5
4
3
2
1

- **States :** Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state :**No queens on the board.
- **Actions :** Add a queen to any empty square.
- **Transition model :** Returns the board with a queen added to the specified square.
- **Goal test :** 8 queens are on the board, none attacked.
- **Path cost :** Each step costs 1, so the path cost is the number of steps in the path.

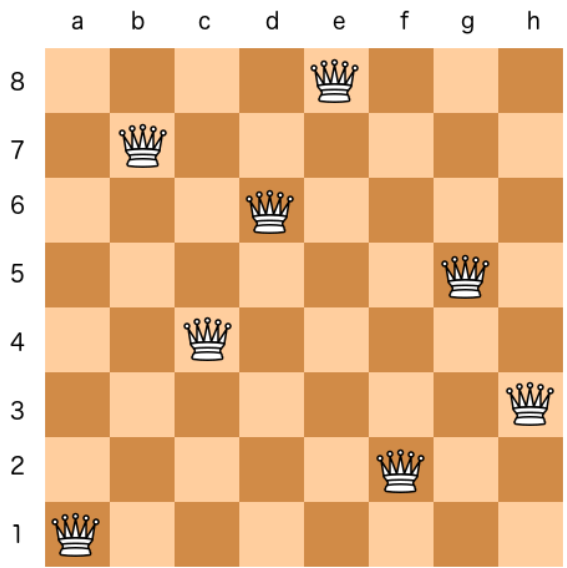
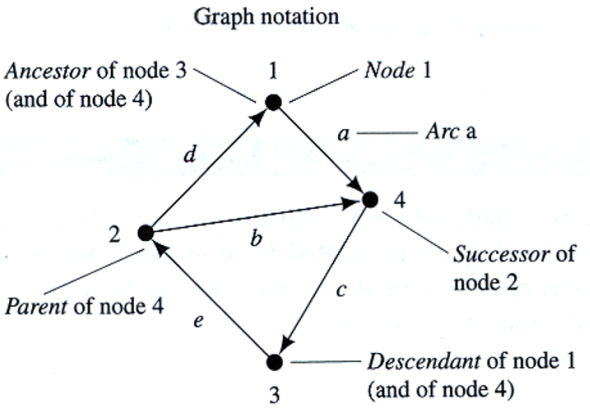


Figure 13: A solution to the 8 queens problem

Uninformed Search (Blind Search) : Graph and Search Tree Terminologies



$c(a)$, alternatively $c(1, 4)$, is the *cost* of arc a
(d, a), alternatively $(2, 1, 4)$, is a *path* from node 2 to node 4

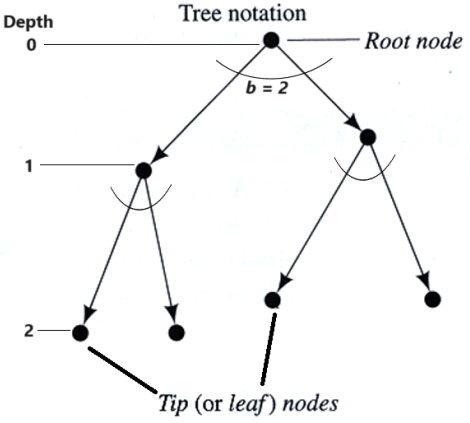


Figure 14: Graph and Search Tree Terminologies.

Uninformed Search Strategies or Blind Searches

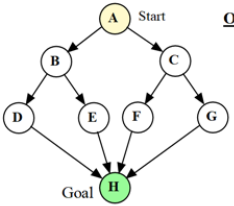
These are search strategies that have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state.

The Uninformed Search Strategies are as follows:

- **Breadth-first search** expands the shallowest nodes first; it is complete, optimal for unit step costs, but has exponential space complexity.
- **Uniform-cost search** expands the node with the lowest path cost, g(n), and is optimal for general step costs.
- **Depth-first search** expands the deepest unexpanded node first. It is neither complete nor optimal but has linear space complexity. Depth-limited search adds a depth bound.
- **Iterative deepening search** calls depth-first search with increasing depth limits until a goal is found. It is complete, optimal for unit step costs, has time complexity comparable to breadth-first search, and has linear space complexity.
- **Bidirectional search** can enormously reduce time complexity, but it is not always applicable and may require too much space.

Breadth First Search (BFS)

Step 1



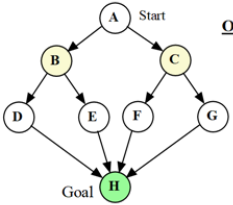
Open List (Queue)

[A]

Visited List

{}

Step 2



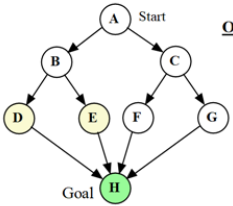
Open List (Queue)

[B, C]

Visited List

{ A }

Step 3



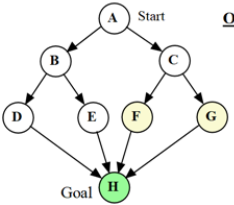
Open List (Queue)

[C, D, E]

Visited List

{ A, B }

Step 4



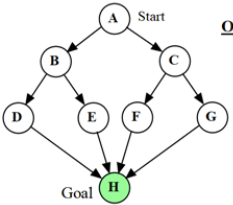
Open List (Queue)

[D, E, F, G]

Visited List

{ A, B, C}

Step 5



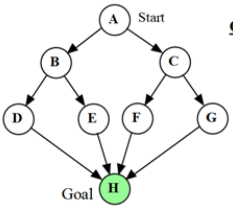
Open List (Queue)

[E, F, G, H]

Visited List

{ A, B, C, D }

Step 6



Open List (Queue)

[F, G, H]

Visited List

{ A, B, C, D, E }

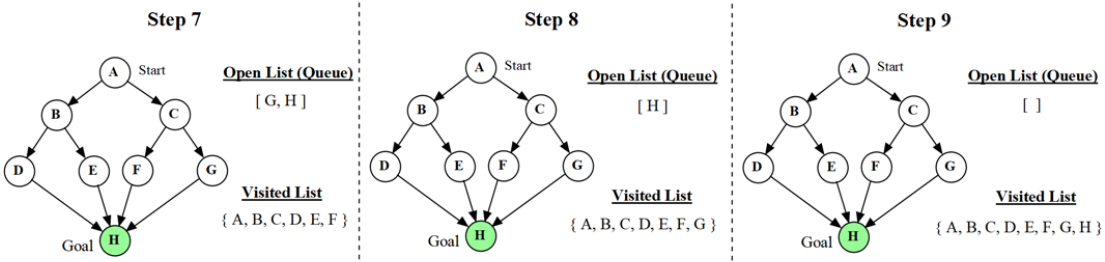


Figure 16: BFS.

Algorithm for BFS

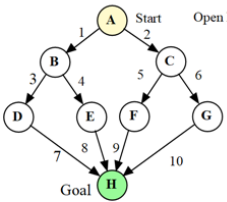
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop 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 17: Algorithm for BFS.[1]

[1] Russell, S. J., Norvig, P. (2021). Artificial Intelligence: A Modern Approach. United Kingdom: Pearson.

Uniform-Cost Search (UCS)

Step 1



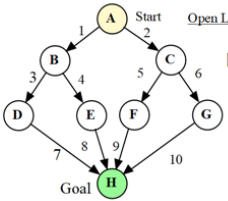
Open List (Priority Queue)

[A]

Visited List

{}

Step 2



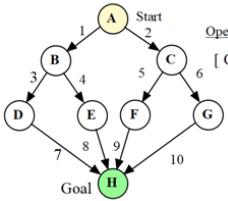
Open List (Priority Queue)

[B (1), C (2)]

Visited List

{ A }

Step 3



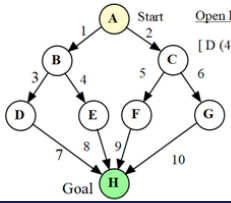
Open List (Priority Queue)

[C (2), D (4), E (5)]

Visited List

{ A, B }

Step 4



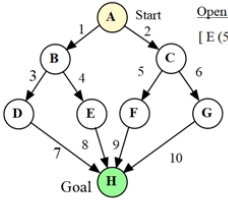
Open List (Priority Queue)

[D (4), E (5), F (7), G (8)]

Visited List

{ A, B, C }

Step 5



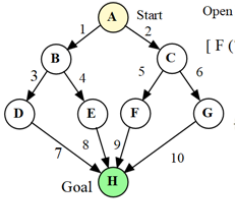
Open List (Priority Queue)

[E (5), F (7), G(8), H (11)]

Visited List

{ A, B, C, D }

Step 6



Open List (Priority Queue)

[F (7), G (8), H (11)]

Visited List

{ A, B, C, D, E }

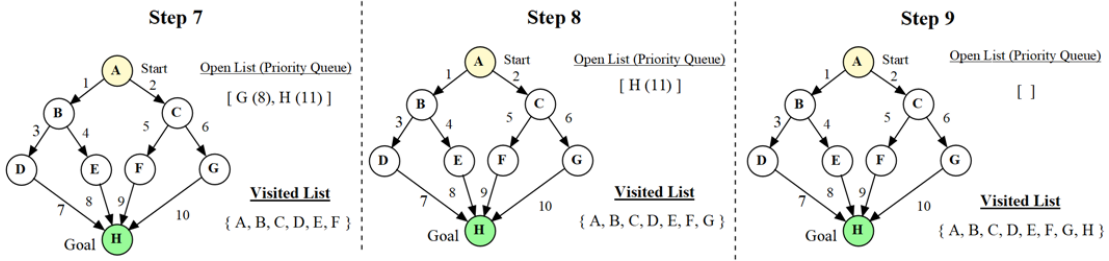


Figure 19: UCS.

Algorithm for UCS

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

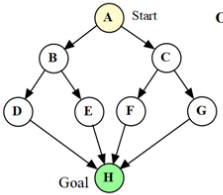
```
node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with node as the only element
explored ← an empty set
loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    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
            frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
            replace that frontier node with child
```

Figure 20: Algorithm for UCS.[1]

[1] Russell, S. J., Norvig, P. (2021). Artificial Intelligence: A Modern Approach. United Kingdom: Pearson.

Depth First Search (DFS)

Step 1



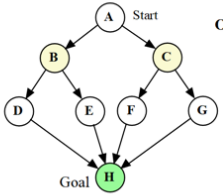
Open List (Stack)

A --- Top

Visited List

{}

Step 2



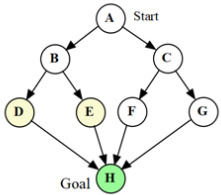
Open List (Stack)

B --- Top
C

Visited List

{ A }

Step 3



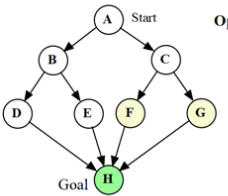
Open List (Stack)

D --- Top
E
C

Visited List

{ A, B }

Step 4



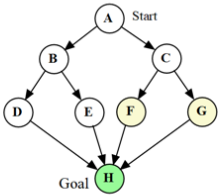
Open List (Stack)

H --- Top
E
C

Visited List

{ A, B, D}

Step 5



Open List (Stack)

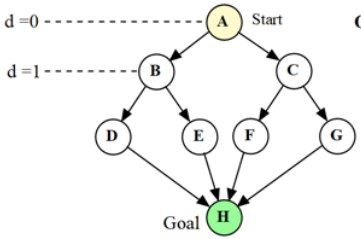
E --- Top
C

Visited List

{ A, B, D, H}

Depth Limited Search (DLS)

Depth Limit = 1

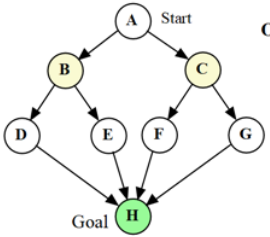


Open List (Stack)

A --- Top

Visited List

{}

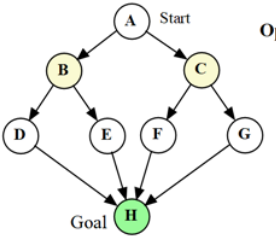


Open List (Stack)

C --- Top

Visited List

{ A, B }

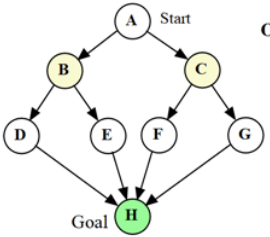


Open List (Stack)

B --- Top
C

Visited List

{ A }



Open List (Stack)

[ ]

Visited List

{ A , B, C}

Algorithm for DLS

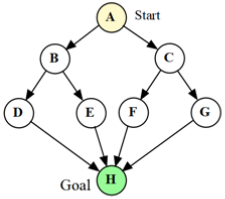
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 23: Algorithm for DLS.[1]

[1] Russell, S. J., Norvig, P. (2021). Artificial Intelligence: A Modern Approach. United Kingdom: Pearson.

Iterative Deepening Depth-First Search (IDDFS)



Iteration 1

Open List (Stack)

A --- Top

Visited List

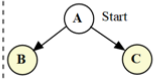
{ }

Open List (Stack)

Visited List

{ A }

Iteration 2



Open List (Stack)

[ ]

Visited List

{ A, B, C }

Open List (Stack)

A --- Top

Visited List

{ }

Open List (Stack)

B --- Top
C

Visited List

{ A }

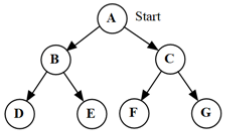
Open List (Stack)

C --- Top

Visited List

{ A, B }

Iteration 3



Open List (Stack)

A --- Top

Visited List

{ }

Open List (Stack)

B --- Top
C

Visited List

{ A }

Open List (Stack)

D --- Top
E
C

Visited List

{ A, B }

Open List (Stack)

E --- Top
C

Visited List

{ A, B, D }

Open List (Stack)

C --- Top

Visited List

{ A, B, D, E }

Open List (Stack)

F --- Top
G

Visited List

{ A, B, D, E, C }

Open List (Stack)

G --- Top

Visited List

{ A, B, D, E, C, F }

Open List (Stack)

[ ]

Visited List

{ A, B, D, E, C, F, G }

Algorithm for IDDFS

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Figure 25: Algorithm for IDDFS.[1]

[1] Russell, S. J., Norvig, P. (2021). Artificial Intelligence: A Modern Approach. United Kingdom: Pearson.

Bidirectional Search

Idea

- Simultaneously searches forward from Start (S) and backwards from Goal (G).
- Stops when both "meet in the middle".
- Needs to keep track of the intersection of 2 open sets of nodes.

What does searching backward from G mean

- Needs a way to specify the predecessors of G.
- What if there are multiple goal states?
- What if there is only a goal test, no explicit list?

Complexity

- Time complexity: Best $O(b^{d/2})$, worst : $O(b^{d+1})$.
- Memory complexity : $O(b^{d/2})$

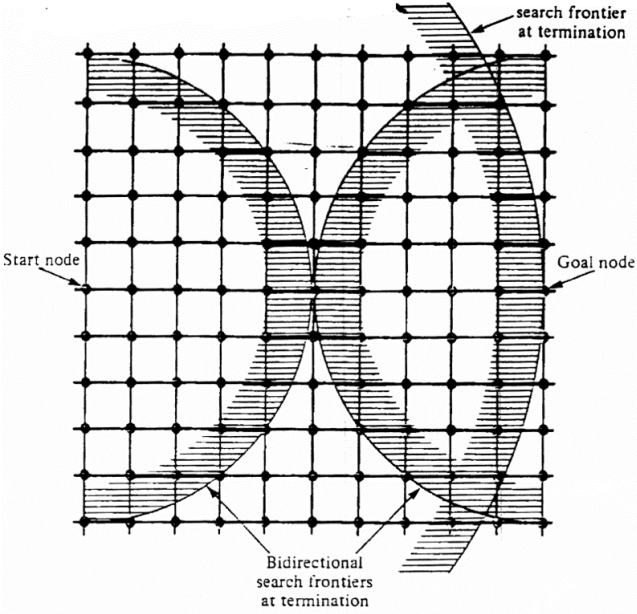


Figure 26: Bidirectional Search.

Comparing Uninformed Search Strategies

An algorithm's performance is evaluated 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

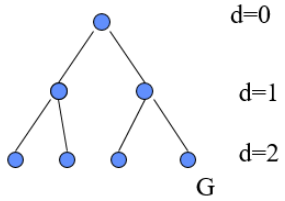
How long does it take to find a solution?

Space complexity

How much memory is needed to perform the search?

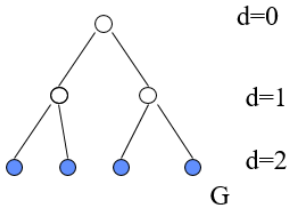
Breadth-First Search (BFS) - Time Complexity

- Assume (worst case) that there is 1 goal leaf at the RHS , so BFS will expand all nodes as $1 + b + b^2 + ... + b^d = \mathbf{O(b^d)}$



Breadth-First Search (BFS) - Space Complexity

- At depth d-1 there are b^d unexpanded nodes in the queue, $\therefore$ complexity is $\mathbf{O(b^d)}$



Breadth-First Search (BFS) - Complete and Optimal

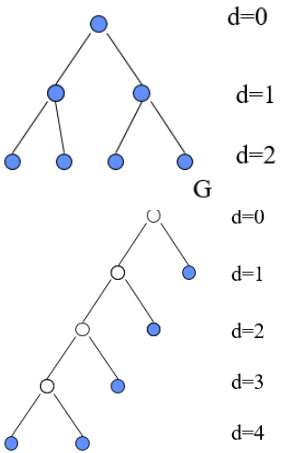
Optimal if step costs are all identical. **Complete** if branching factor is finite.

Depth-First Search (DFS) - Time Complexity

- Assume (worst case) that there is 1 goal leaf at the RHS , so BFS will expand all nodes as $1 + b + b^2 + ... + b^d = \mathbf{O(b^d)}$

Depth-First Search (DFS) - Space Complexity

- At depth $\ell < d$ we have $b - 1$ nodes and at depth d we have b nodes, $\therefore$ total $= (d-1) * (b-1) + b = \mathbf{O(bd)}$



Depth-First Search (DFS) - Complete and Optimal

Optimal : No (Not for infinite spaces). **Complete :** No.

Comparing DFS and BFS Strategies

- **Same worst-case time Complexity, but**
  - In the worst-case, BFS is always better than DFS.
  - Sometime, on average DFS is better if: many goals, no loops, and no infinite paths.
- **BFS is much worse memory-wise**
  - DFS is linear space
  - BFS may store the whole search space.
- **In general**
  - BFS is better if the goal is not deep, if there are infinite paths if there are many loops, and if there is a small search space.
  - DFS is better if there are many goals, not many loops.
  - DFS is much better in terms of memory.

Comments on Iterative Deepening Search

Complexity

- Space complexity is **O(bd)** (since it's like depth-first search runs at different times)
- Time Complexity is $1 + (1 + b) + (1 + b + b^2) + ...(1 + b + ...b^d) = \mathbf{O(bd)}$ (i.e., asymptotically the same as BFS or DFS in the worst case)
- The overhead in repeated searching of the same subtrees is small relative to the overall time, e.g., for $b = 10$, it only takes about 11% more time than DFS.

A Useful Practical Method

- Guarantees finding an optimal solution if one exists (as in BFS).
- Space efficiency, O(bd) of DFS.
- But still has problems with loops like DFS.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Figure 27: Evaluation of Search Strategies.[1]

[1] Russell, S. J., Norvig, P. (2021). Artificial Intelligence: A Modern Approach. United Kingdom: Pearson.

Informed Search Strategies

In Informed Search, the algorithm is aware of where the best chances of finding the element are and the algorithm heads that way! Heuristic search is an informed search technique. A heuristic value tells the algorithm which path will provide the solution as early as possible.

Heuristic Functions $h(n)$

- A heuristic function is a function $f(n)$ that estimates the "cost" of getting from node n to the goal state.
- Heuristics improve the efficiency of search algorithms by reducing the effective branching factor from b to (ideally) a low constant b^* such that $1 < b^* << b$.

Traditional Informed Search Strategies

- Greedy Best first search
  - f measures the estimated cost of reaching the goal from the current state, i.e. $f(n) = h(n)$ where $h(n) =$ an estimate of the cost to get from n to a goal (heuristic Value).
- A* search
  - f measures the estimated cost of getting to the goal state from the current state and the cost of the existing path to it, i.e. $f(n) = g(n) + h(n)$ where $g(n) =$ the cost to get to n from start node.

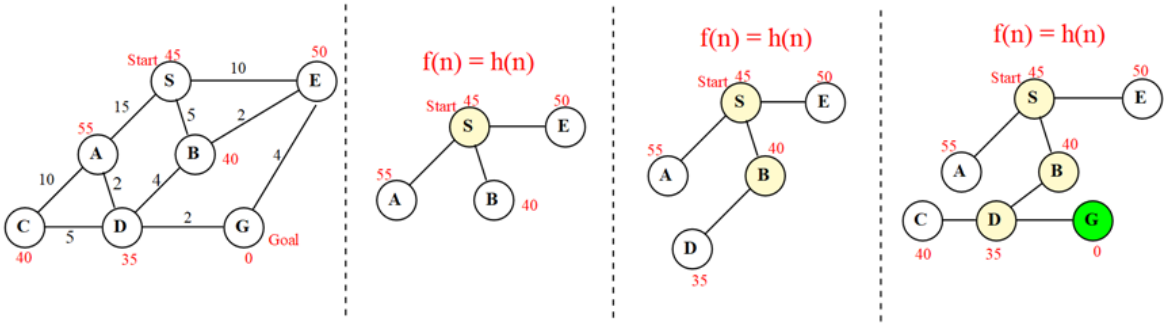


Figure 28: Best-First Search Strategy.

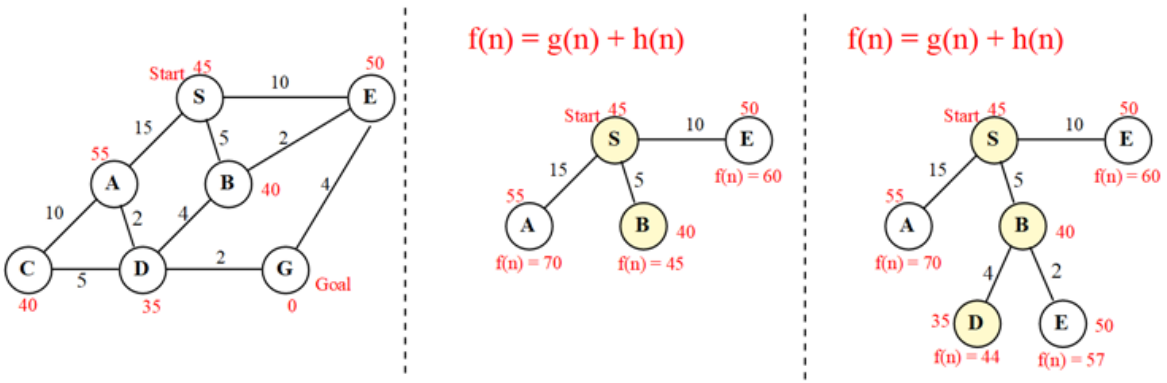


Figure 29: A* Search Strategy.

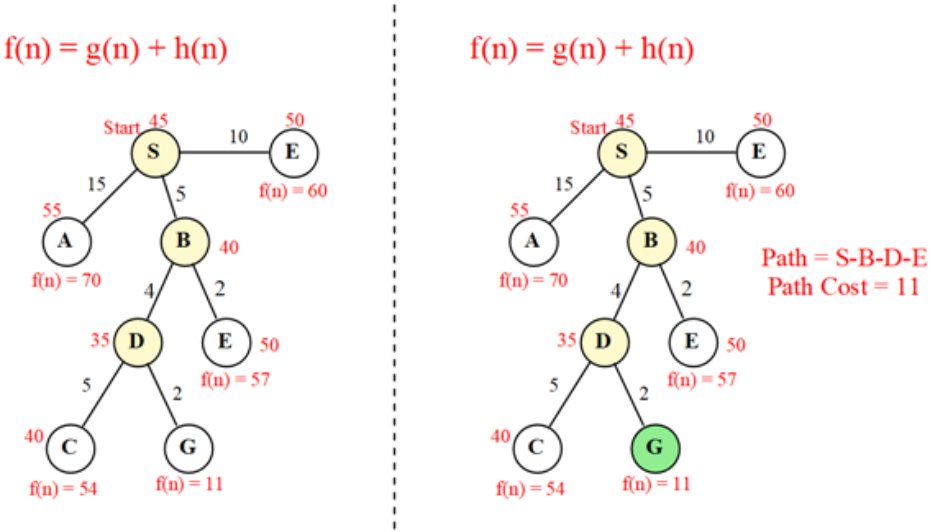


Figure 30: A* Search Strategy.

Heuristics: Admissibility and Consistency

Admissibility

- A heuristic $h(n)$ is admissible if for every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic. (But no info of where the goal is if set to 0.)

Consistency

A heuristic is consistent (or monotone) if for every node n, every successorn' of n generated by any action a, and follows the following equation : $\mid h(n) - h(n') \mid = c(n, n')$ where $c(n, n')$ is the actual cost from node n to successor node n'.

Hill-climbing Algorithm

- Goal: Optimizing an objective function.
- Does not require differentiable functions.
- Can be applied to "goal" predicate type of problems.
- Intuition: Always move to a better state

1. Start = random state or special state.
2. Until (no improvement)
   - if Steepest Ascent: Find the best successor. or if Greedy: Select first improving successor
   - Go to that successor.
3. Repeat the above process some number of times (Restarts).

- Problems :
  - finds local maximum, not global.
  - plateaux: large flat regions.
  - ridges: fast up ridge, slow on ridge
- Not complete, not optimal.
- No memory problems since it stores one information node at a time.

Prepare the solution for these problems.

- Water Jug Problem.
- Monkey Banana Problem.
- Missionaries and Cannibals problem.

Thank You.