Chapter 3: Problem Solving by Searching

3.1 Problem-Solving Agents

When the correct action to take is not immediately obvious, an agent may need to to plan ahead: to consider a sequence of actions that form a path to a goal state. Such an agent is called a problem-solving agent, and the computational process it undertakes is called search.

Uniformed – Unguided – Brute Force (no such estimate is available)
Informed – Guided (the agent can estimate how far it is from the goal)

Problem -→ Search Algorithm → Solution (Sequence of Actions)

• How to structure an "agent-oriented" problem

Defined Problem:

1. Initial State
2. A description of possible actions
3. Transition Model
4. Goal Test
5. Path Cost

GOAL FORMULATION:
Goals organize behavior by limiting the objectives and hence the actions to be considered.

INITIAL STATE:
The state the agents starts in.
ACTIONS:

What the agent is capable of doing?
Each possible action is applicable for a state s, if ACTION(s) function lists them in a sequence of states.
{Leave(garage),Go(Street), Trun(Right)}
TRANSITION MODEL:
A description of each action
Specified by a function, RESULT(s,a)
Return state when a is performed in state s.
RESULT(In(Garage), Go(Street), Turn(Right))= In(Street)
INITIAL STATE, ACTIONS and TRANSTION MODEL define the problem's STATE SPACE.
GOAL TEST:
If the goal is to be at the store, then the goal set could just be {At(Store)}
PATH COST FUNCTION :
Step Cost
PROBLEM FORMULATION:
The agent devises a description of the states and actions necessary to reach the goal—an abstract model of the relevant part of the world. For our agent, one good model is to consider the actions of

traveling from one city to an adjacent city, and therefore the only fact about the state of the world that will change due to an action is the current city.

SEARCH: Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal. Such a sequence is called a solution.

A set of possible states that the environment can be in. We call this the state space.

FROMULATION OF PROBLEMS:

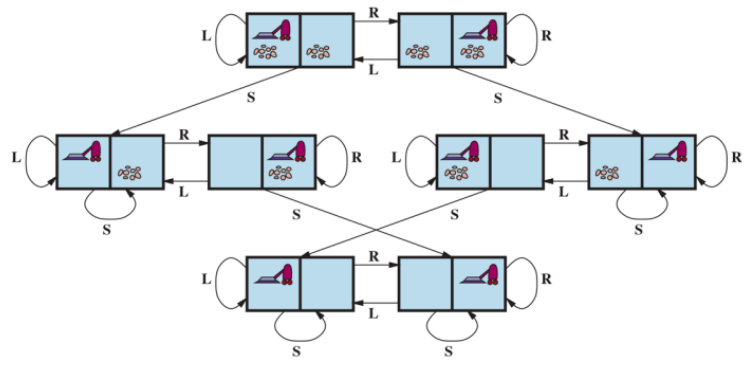
Level of abstraction of a problem.

State description and action description need to be abstracted.

1. Incremental Formulation
2. Complete – state function

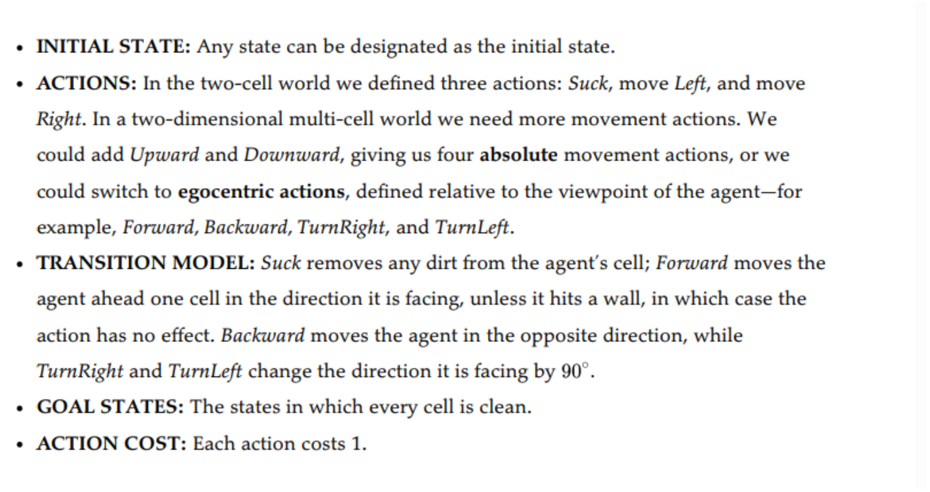
3.2 Example Problems

• Toy problems like puzzles



The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

VACCUM PROBLEM

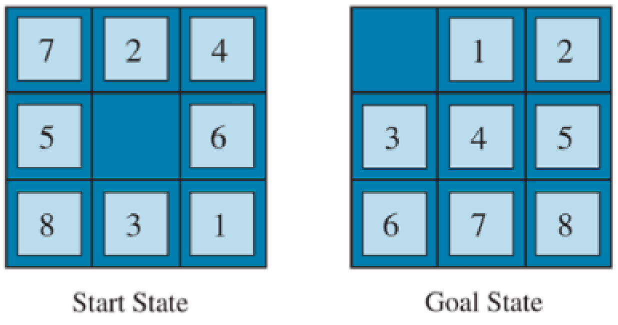


- **INITIAL STATE:** Any state can be designated as the initial state.
- **ACTIONS:** In the two-cell world we defined three actions: *Suck*, move *Left*, and move *Right*. In a two-dimensional multi-cell world we need more movement actions. We could add *Upward* and *Downward*, giving us four **absolute** movement actions, or we could switch to **egocentric actions**, defined relative to the viewpoint of the agent—for example, *Forward*, *Backward*, *TurnRight*, and *TurnLeft*.
- **TRANSITION MODEL:** *Suck* removes any dirt from the agent's cell; *Forward* moves the agent ahead one cell in the direction it is facing, unless it hits a wall, in which case the action has no effect. *Backward* moves the agent in the opposite direction, while *TurnRight* and *TurnLeft* change the direction it is facing by 90°.
- **GOAL STATES:** The states in which every cell is clean.
- **ACTION COST:** Each action costs 1.

The 8 Puzzle:

- **STATES:** A state description specifies the location of each of the tiles.
- **INITIAL STATE:** Any state can be designated as the initial state. Note that a parity property partitions the state space—any given goal can be reached from exactly half of the possible initial states (see Exercise 3.PART).
- **ACTIONS:** While in the physical world it is a tile that slides, the simplest way of describing an action is to think of the blank space moving *Left*, *Right*, *Up*, or *Down*. If the blank is at an edge or corner then not all actions will be applicable.
- **TRANSITION MODEL:** Maps a state and action to a resulting state; for example, if we apply *Left* to the start state in Figure 3.3, the resulting state has the 5 and the blank switched.

Figure 3.3



A typical instance of the 8-puzzle.

- **GOAL STATE:** Although any state could be the goal, we typically specify a state with the numbers in order, as in Figure 3.3.
- **ACTION COST:** Each action costs 1.

- **Route finding**

  Travel Planning Web site:

  State: Each state is maybe an airport and the time. Supplemental information about travel times , fares etc.

  Initial State : Specified by user query

  Actions : take a flight , in any seat class, the fare base...

Transition Model: Updated state will have the destination of the flight become the new current location as well as updated time.

Goal test : Is the destination location to be reached.

Path cost: Depends on the fare, waiting time etc.

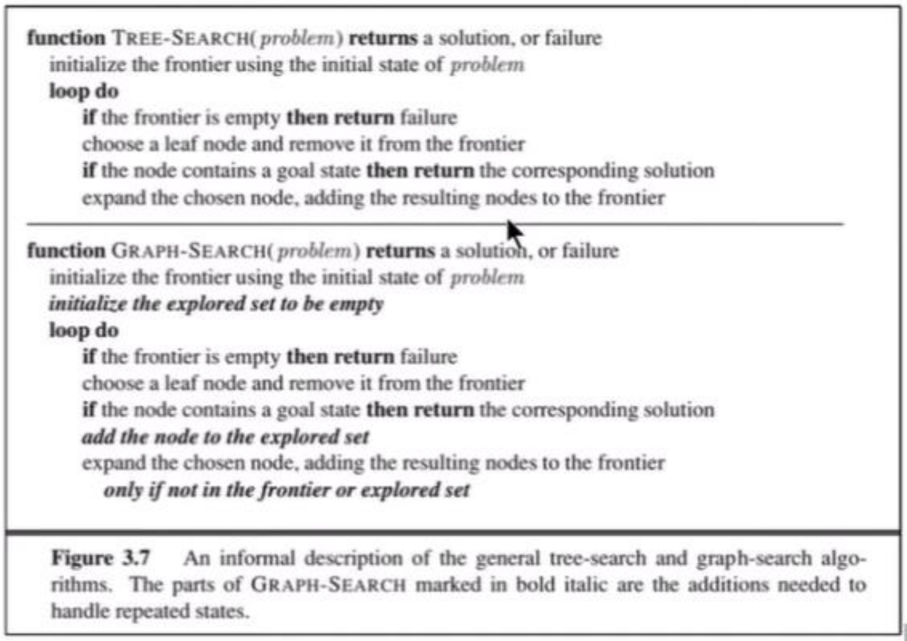
Travel Sales man Problem:
A touring problem where each city is visited and returned to the start state.

3.3 Searching for Solutions
Solutions are action of sequence.
Forms a tree, Branch are actions, Nodes contain the states in the state space.

Frontier – Set of all nodes available for expanding.

TREE SEARCH:
Can end up with loop.

GRAPH SEARCH:
Save the set of already searched nodes.



```
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
```

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

• **Introducing the general state-space search framework and the abstract graph-search algorithm**

The structure needed:

- State : the state in the state space to which the node corresponds
- Parent : the node in the tree that generated this node
- Action : Action applied to the parent node to be generated
- Path-Cost: the cost g(n) of the path from the node

NODE VS STATES:

1. Node is a structure used in representing a tree
2. A state is a configuration of the world.
3. A node will contain a state
4. Nodes are particular paths, states aren't

Two Different nodes can contain the same world state if it gets by two paths.

Measuring performance (completeness, optimality, time/space complexity)
COMPLETENESS: Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?

COST OPTIMALITY: Does it find a solution with the lowest path cost of all solutions?

TIME COMPLEXITY: How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered.

SPACE COMPLEXITY: How much memory is needed to perform the search?

COMPLEXITY CAN BE MESURED IN TERMS OF:

BRACHING FACTOR- MAXIMUM NUMBER OF SUCCESSOR Nodes

DEPTH – The sallowest goal node

Maximum length – Of any path in the state space

3.4 Uninformed Search Strategies

BLIND SEARCH

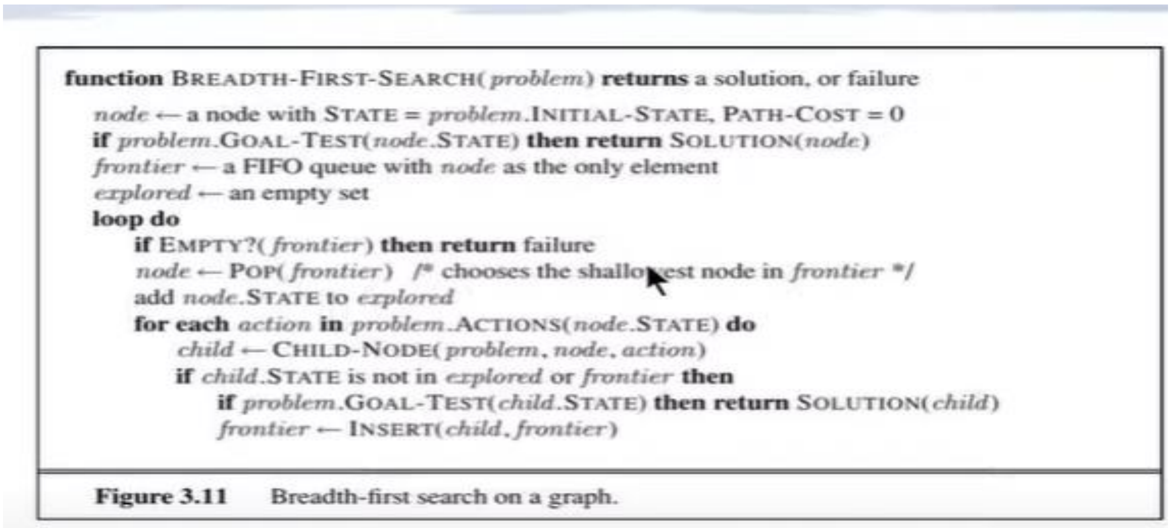
• **BFS, DFS, Uniform Cost Search, Iterative Deepening, Bidirectional Search, etc.**

BREADTH- FIRST SEARCH:

Adjacent nodes get expanded first, before moving on to the further away nodes.

Uses a FIFO queue for the frontier.

New nodes go to the back, older nodes are at the front.



```
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.

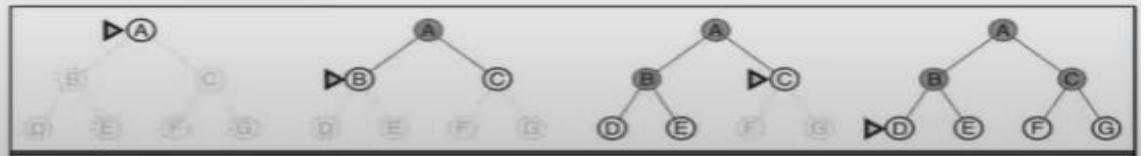


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Complete: Since all the nodes are examined a solution is found, if it exits

Shallowest goal node not necessarily the optimal once.

Could be if the path cost is non decreasing function of the depth of the node

Example: Furthest right, bottom most node vs further lest bottom most node.

Complexity : Time – O(b^d) Space – O(b^ d -1) for explored set and O(b^d) for the frontier

Uniform- cost search

When all step cost are equal , BFS is optimal because it expands the shallowest node.
Instead of doing that, uniform- cost search expands the node with the lowest path cost.
Uses a priority queue, sorted by g where n is a node and g(n) is the path cost of the node.

DFS:
Always expands deepest node in the current froniter first.
As the nodes at the deepest level are expanded, the search backs up to the next deepest node
DFS is a graph search algorithm, using LIFO queue. Most recently generated node is expanded first.

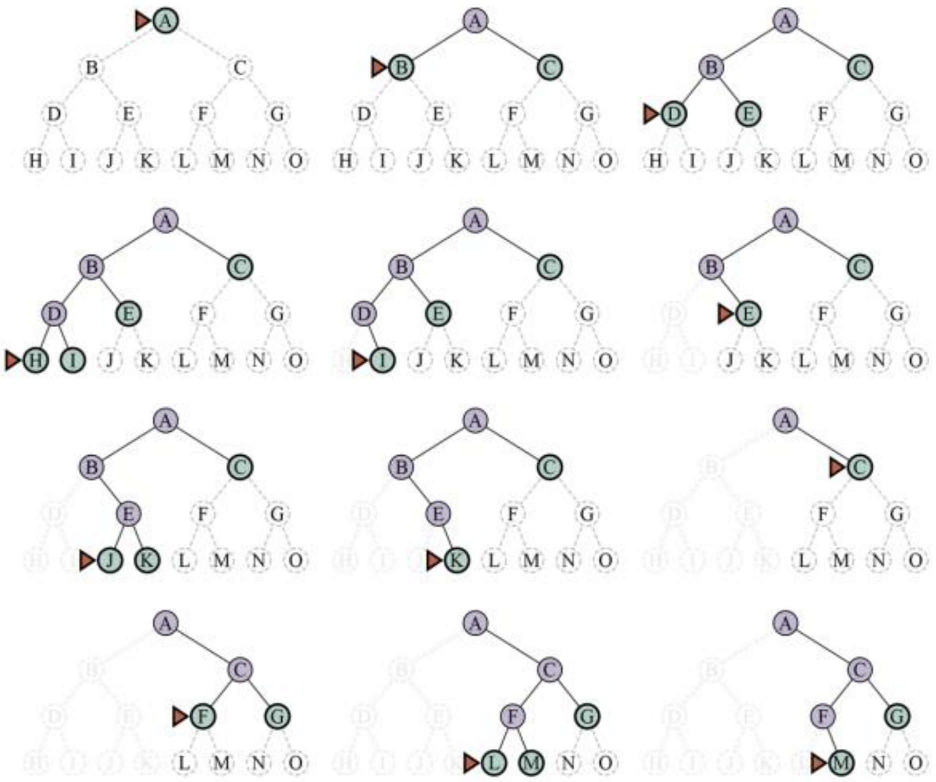


Figure 3.12

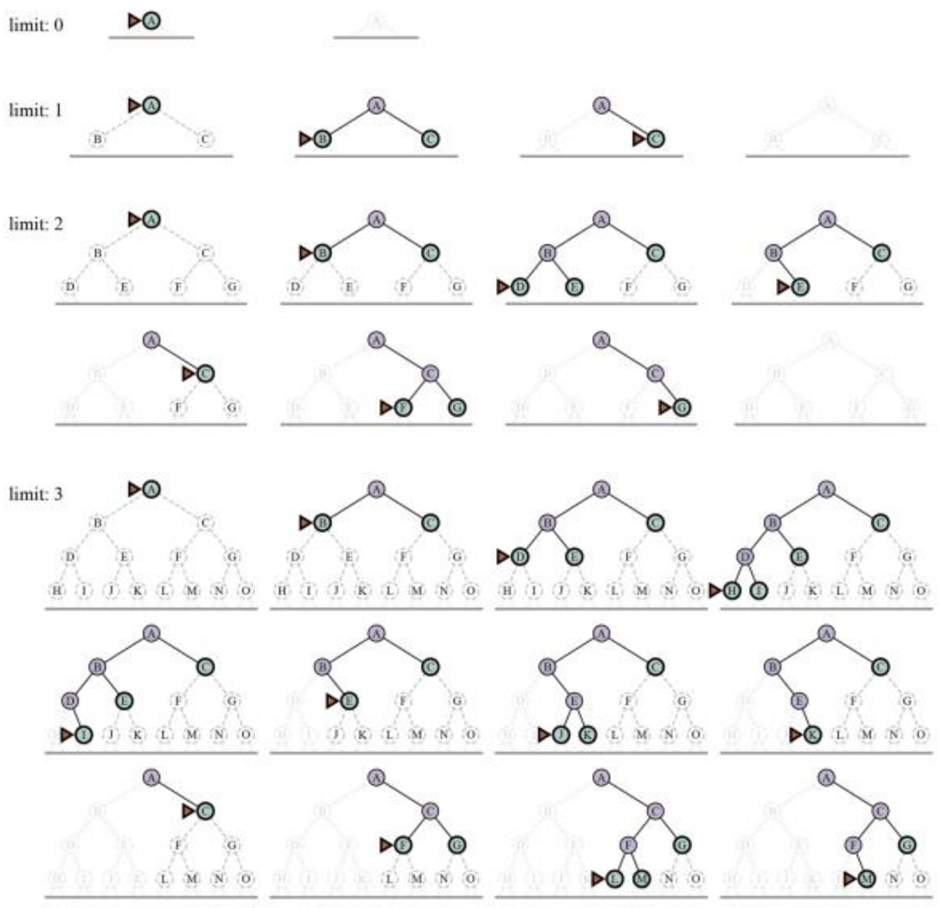
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

The time complexity is O(b ℓ) and the space complexity is O(bℓ).

Iterative deepening search

Iterative deepening search solves the problem of picking a good value for by trying all values: first 0, then 1, then 2, and so on—until either a solution is found, or the depth‐limited search returns the failure value rather than the cutoff value. The algorithm is shown in Figure 3.12 . Iterative deepening combines many of the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are modest: when there is a solution, or on finite state spaces with no solution. Like breadth-first search, iterative deepening is optimal for problems where all actions have the same cost, and is complete on finite acyclic state spaces, or on any finite state space when we check nodes for cycles all the way up the path.



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \varepsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

Iterative Deepening DFS:

A general strategy used with DFS for trees that finds the best depth limit.
Gradually increase the depth limit until the optimal limit is found.
Start at 0, then 1 , then 2,...
A combination of dfs and bfs
Memory required: O(bd)
It is complete when the breaching factor is finite.

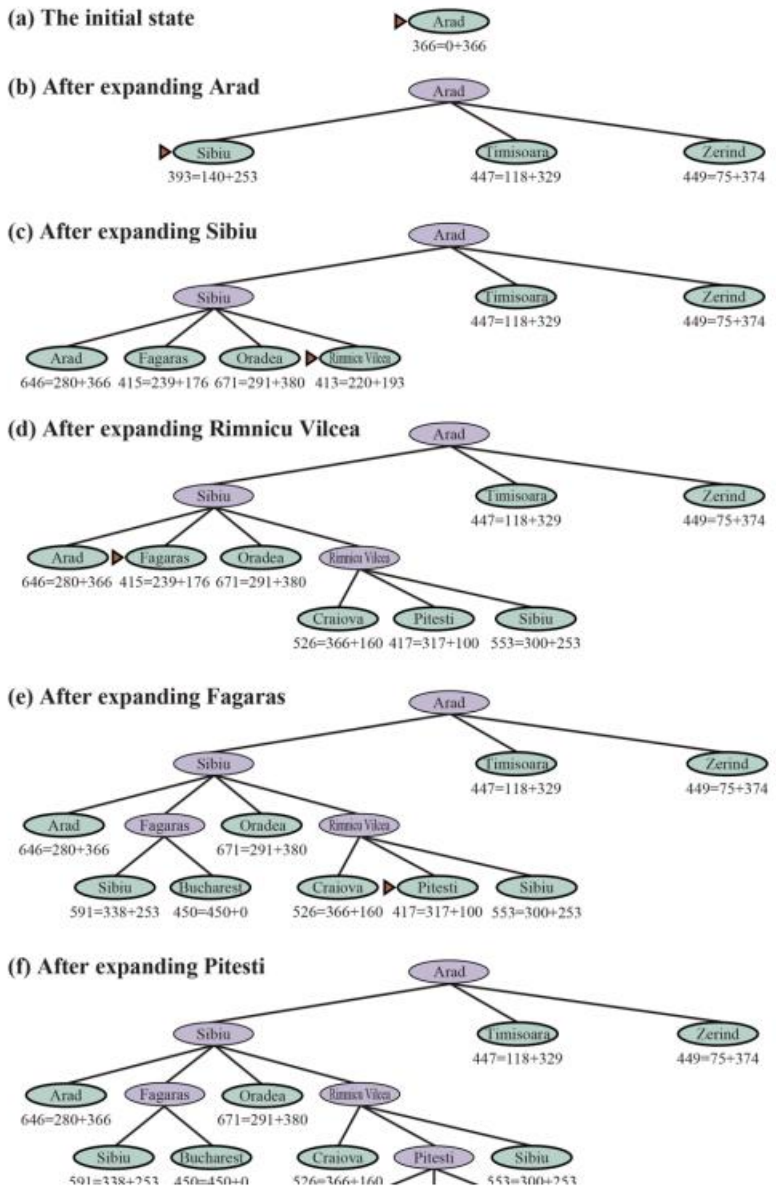
Bidirectional Search
Run two directional search, one from the start state, the other from the goal state
Hope : They meet in the middle

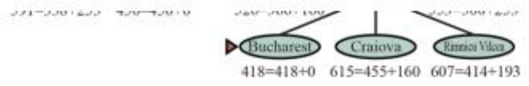
3.5 Informed Search
Uses problem specific knowledge beyond definition of the problem itself.
BFS : A general TREE- SEARCH ore GRAPH-SEARCH algorithm selecting a node for expansion based on an evaluation function.
h(n) = estimated cost of the cheapest path from the state at node n to a goal state.
Lowest evaluation is expanded first.

- **Greedy best-first**

Tries to expand the node closest to the goal.
Just uses the heuristic function f(n)= g(n)
Incomplete, as the node with least cost are only searched.

A*
f(n) = g(n) + h(n)
Optimal And Complete





MEMORY – BOUNDED HEURISTIC SEARCH

1. Iterative deepening A*
2. Use recursion – best first search with linear search

• **Admissability and Monotonicity – be able to define the conditions, but no proofs, for example**

that A* is optimal with an admissable solution
1. Admissibility:

- An admissible heuristic is a function used in A* search that never overestimates the true cost to reach the goal from any given node. In other words, it always underestimates or exactly estimates the cost.

- A* is optimal when using an admissible heuristic. This means that if the heuristic used in A* is admissible, the algorithm will always find the optimal solution, i.e., the solution with the lowest cost.

2. Monotonicity (also known as consistency):

- A heuristic function is said to be monotonic (or consistent) if for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the cost of getting from n to n' plus the estimated cost of reaching the goal from n'.

- In other words, if h(n) represents the estimated cost of reaching the goal from node n, and c(n, a, n') represents the cost of the action a from node n to its successor n', then a heuristic is monotonic if $h(n) \leq c(n, a, n') + h(n')$ for all nodes n and n', and for all actions a.

- A* with a monotonic heuristic is also optimal. This is because monotonicity ensures that the heuristic is consistent throughout the search process, allowing A* to avoid revisiting nodes unnecessarily and thereby guaranteeing optimality.

In summary, admissibility and monotonicity are important properties of heuristic functions used in A* search. Admissibility ensures that the heuristic does not overestimate the cost, while monotonicity ensures consistency in the estimated costs from one node to another, both of which contribute to A* being able to find optimal solutions.

- **Memory-bounded search – we skipped the MA* and SMA* algorithms, so no questions about those**