CS 3011: Artificial Intelligence

Solving Problems by Searching

Instructors: Dr. Durgesh Singh

CSE Discipline, PDPM IIITDM, Jabalpur -482005

Problem-Solving Agents

- This is a kind of goal-based agents that try to find a sequence of actions to achieve a goal.
  - Sometimes an action directly achieves a goal; sometimes a series of actions are required
- Finding this sequence of actions is called search

Example: Traveling in Romania

- Suppose the agent is currently in the Arad and has to reach Bucharest .
- Goal: Reach Bucharest on time

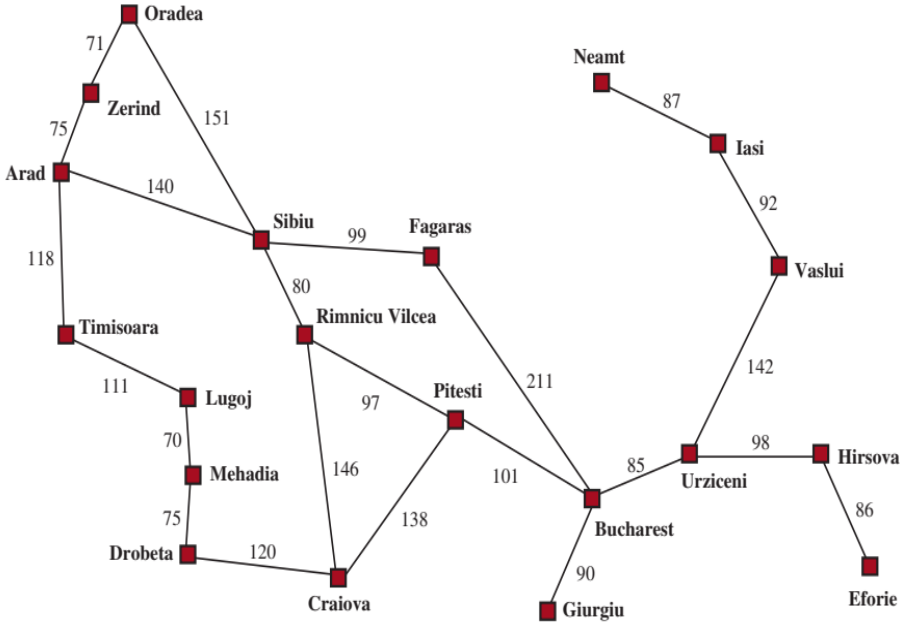
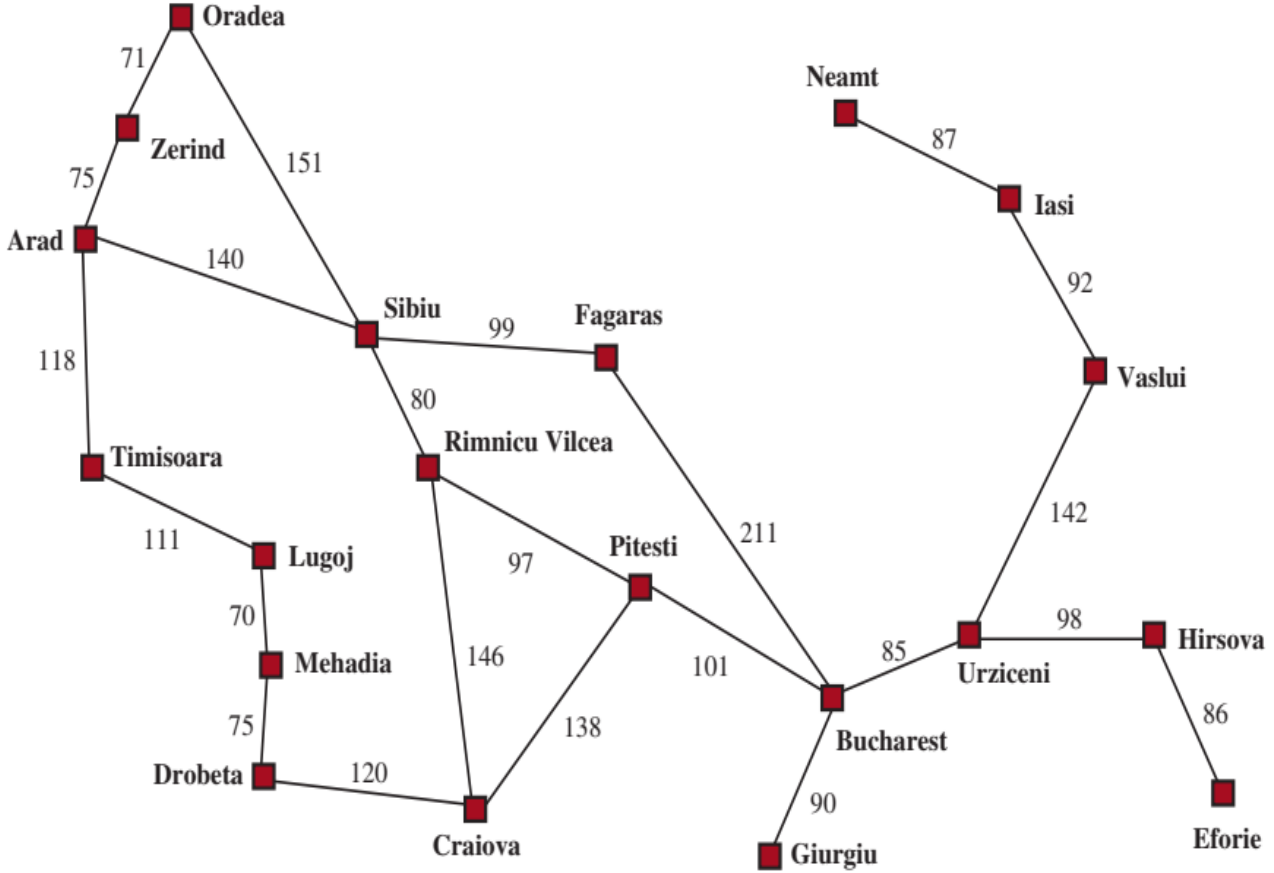


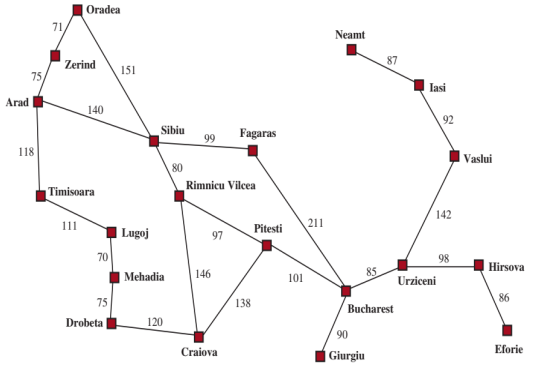
Figure: A simplified road map of part of Romania, with road distances in miles.

Example: Traveling in Romania



Search problems and solutions

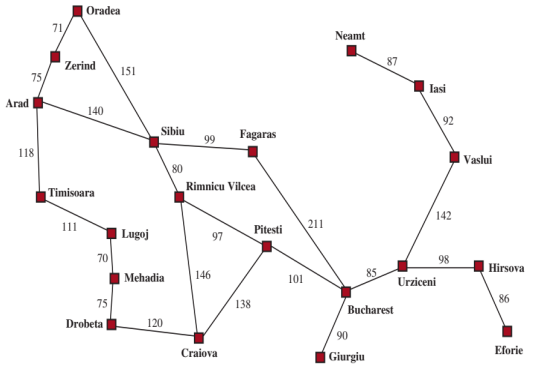
A search **problem** can be defined formally by these components:



- The initial state that the agent starts in. Example: *Arad*.
- A description of possible actions available to the agent. Given a state s, ACTIONS(s) returns a finite set of actions that can be executed in s.
  - Example: ACTIONS(Arad) = {ToSibiu, ToTimisoara, ToZerind}
- A transition model, which describes what each action does. RESULT(s,a) returns the state (successor) that results from doing action a in state s.
  - Example: RESULT(Arad, ToZerind) = Zerind

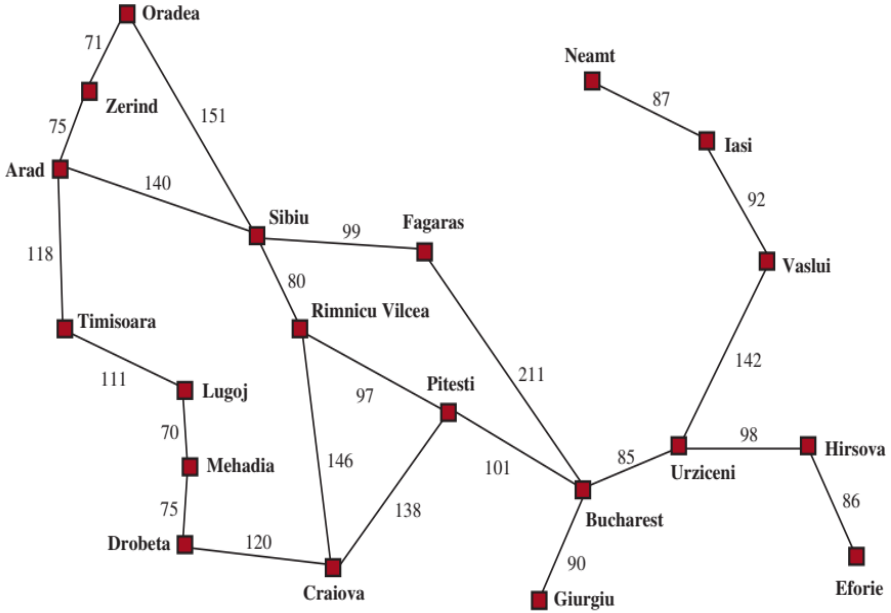
Search problems and solutions…

- The goal test, which determines whether a given state is a goal state
  - Example: Goal state= Bucharest
- path cost: function assigns numeric cost to each path
  - A sequence of actions forms a path.



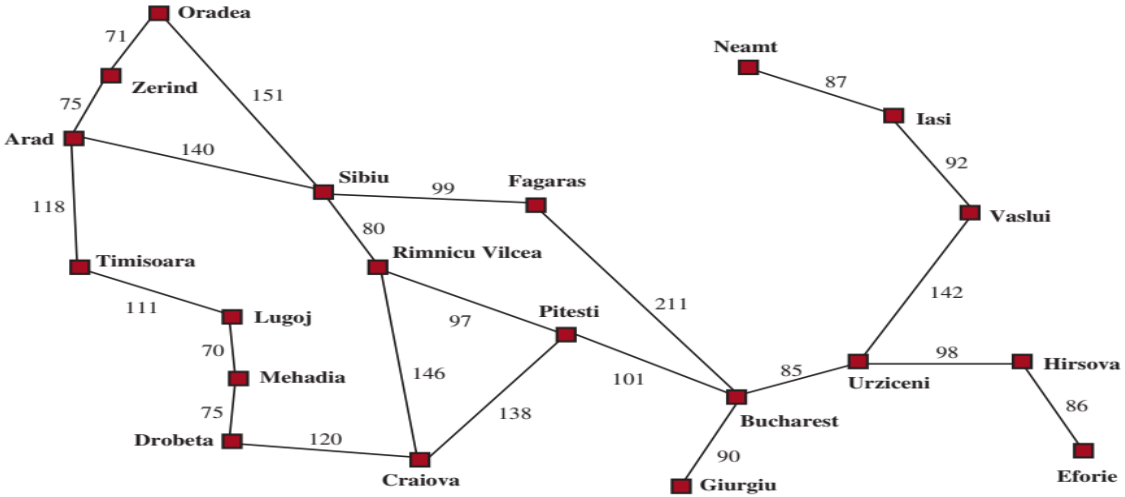
state space in Search problems

- state space: The set of all states reachable from the initial state by any sequence of actions.
  - The initial state, actions, and transition model implicitly define the state space of the problem.
  - The state space forms a directed network (graph) in which the nodes are states, and the links are actions



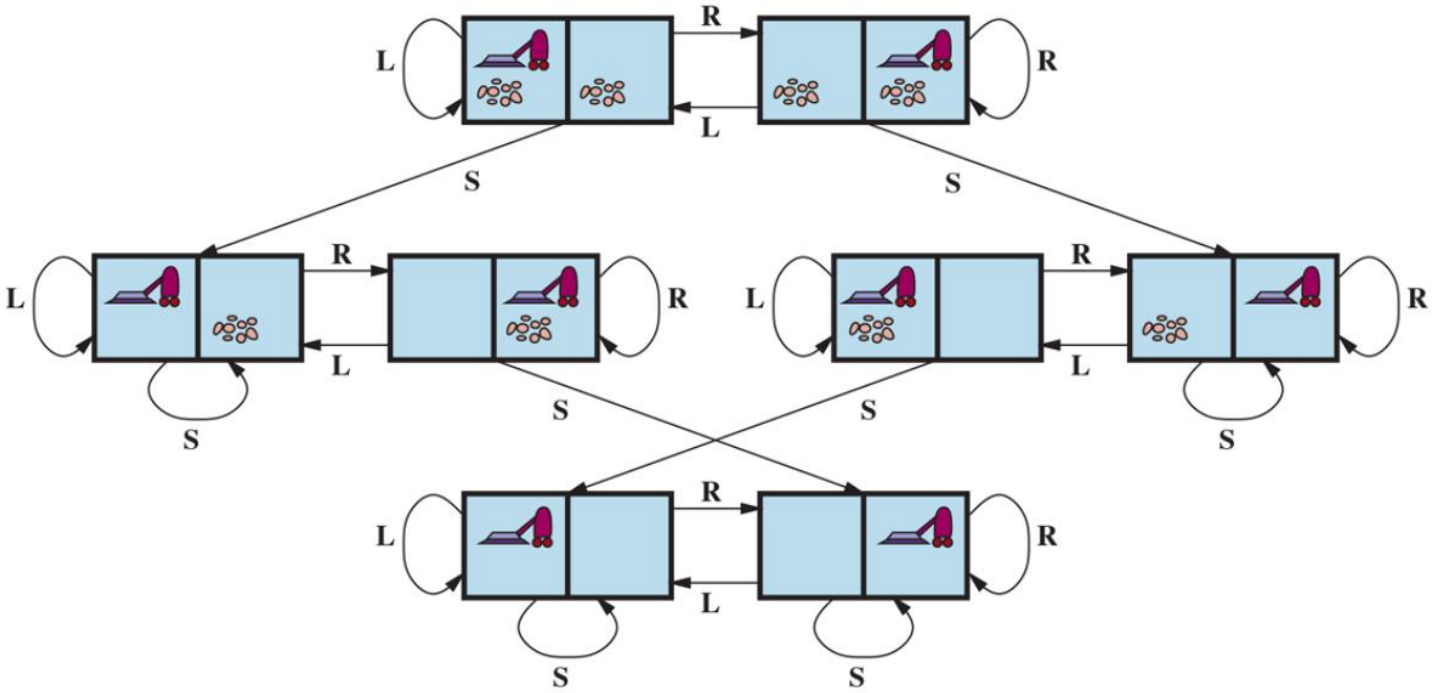
Optimal solution

- A solution to problem is a path from the initial state to a goal state.
  - Solution quality is measured by the path cost function.
- An optimal solution has the lowest path cost among all solutions.



Sample Search Problem

Vacuum World State Space Graph



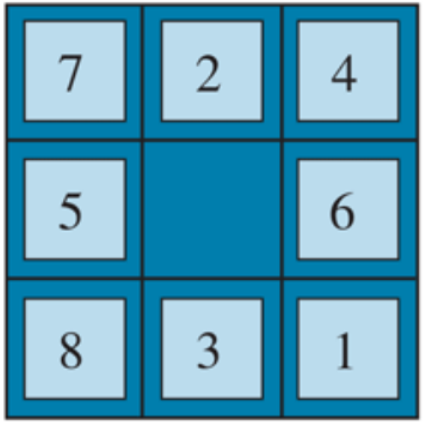
The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

Formulating the Vacuum World Problem

- States: The state is determined by both the agent location and dirt locations. The agent is in one of the two locations, each of which might or might not contain dirt.
  - Possible world states= 2*2^2= 8
- Initial state: Any state may be designated as initial state
- Actions: Left, Right, and Suck
- Transition model: The actions have expected effects, except that moving Left in leftmost square, moving Right in rightmost square, and Sucking in a clean square have no effect.
- Goal State: Check whether all the squares are clean.
- Path Cost: Each cost 1, so the path cost is the number of steps in the path

Example: The 8-puzzle

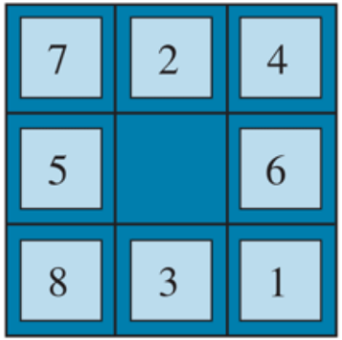
- States?
- Initial state?
- Actions?
- Goal test?
- Path cost?



Start State

Goal State

Example: The 8-puzzle



Start State

Goal State

- States? locations of tiles
- Actions? move blank left, right, up, down
- Goal test? = goal state (given)
- Path cost? 1 per move

[Note: optimal solution of *n*-Puzzle family is NP-hard]

What exactly is “ Searching for Solution”?

- A solution is an action sequence
  - Search algorithms work by considering various action sequences.
- The possible action sequences, starting at the initial state form a search tree with
  - The initial state at the root
  - The branches are the actions, and
  - The nodes correspond to states in the state space of the problem
- The essences for search
  - Following up one option now and putting others aside for later, in case the first choice does not lead to a solution

Nodes in search Tree

- Node n is a data structure that keep track of

n.STATE: the state in the state space to which the node corresponds.

n.PARENT: the node in the search tree that generated this node.

n.ACTION: the action that was applied to the parent to generate the node.

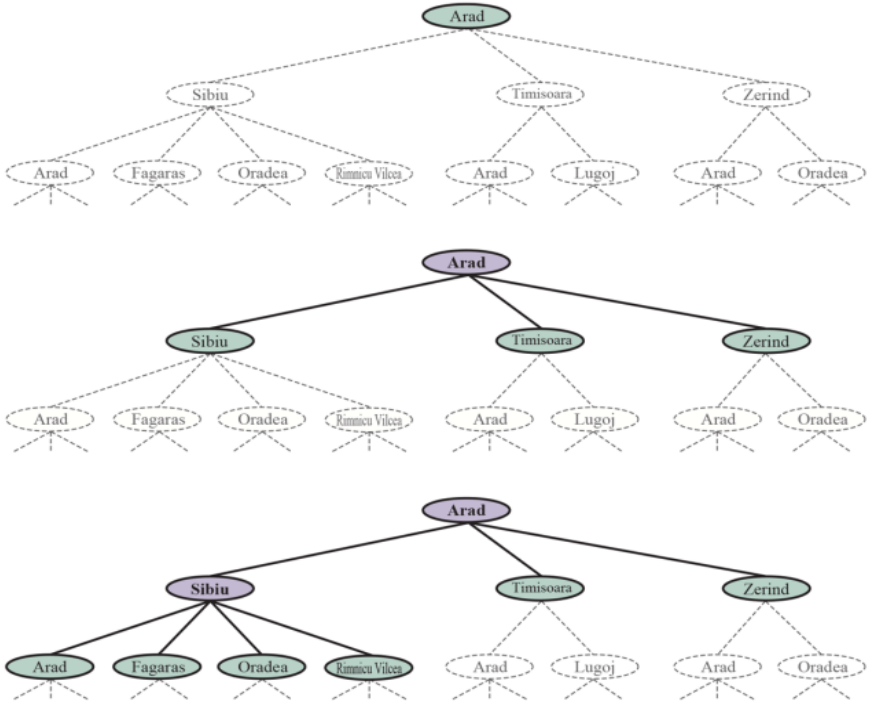
n.PATH-COST: the total cost of the path (i.e., g(n)) from the initial state to this node.

A general Tree search algorithm

```
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
```

A general Tree search algorithm

```
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
```



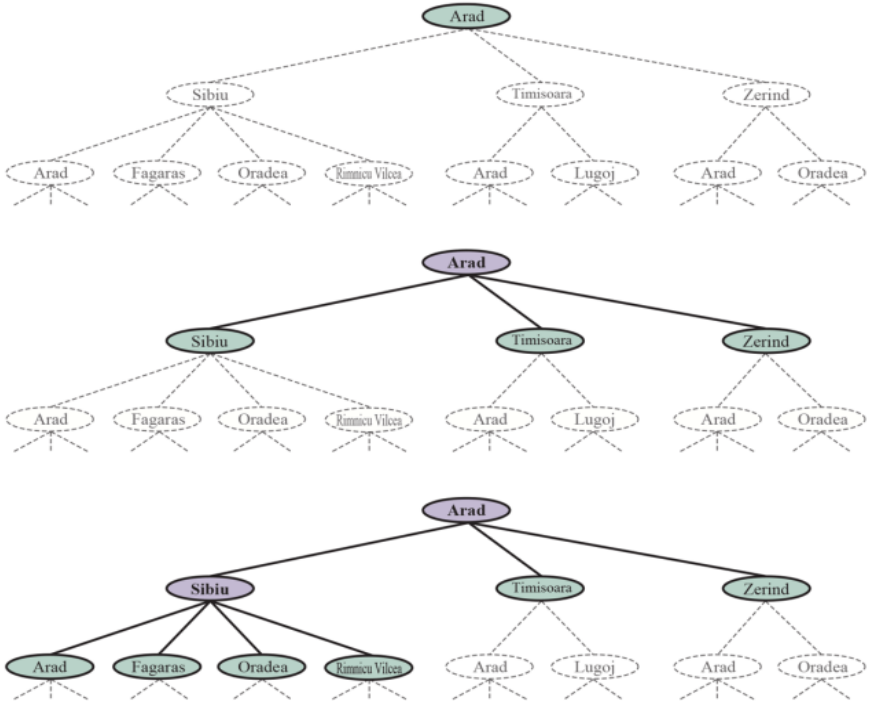
Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been *reached*. Nodes that could be generated next are shown in faint dashed lines. Notice in the bottom tree there is a cycle from Arad to Sibiu to Arad; that can't be an optimal path, so search should not continue from there.

A general Graph search algorithm

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

A general Graph search algorithm

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



Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been *reached*. Nodes that could be generated next are shown in faint dashed lines. Notice in the bottom tree there is a cycle from Arad to Sibiu to Arad; that can't be an optimal path, so search should not continue from there.

Search Strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - Completeness: Does it always find a solution if one exists?
  - Time complexity: Number of nodes generated
  - Space complexity: Maximum number of nodes in memory
  - Optimality: Does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - *b:* maximum no. of successors of any node
  - *d:* depth of the shallowest goal node
  - *m*: maximum length of any path in the state space.

Artificial Intelligence

Uninformed Search

28 August 2023

Uninformed Search Strategies

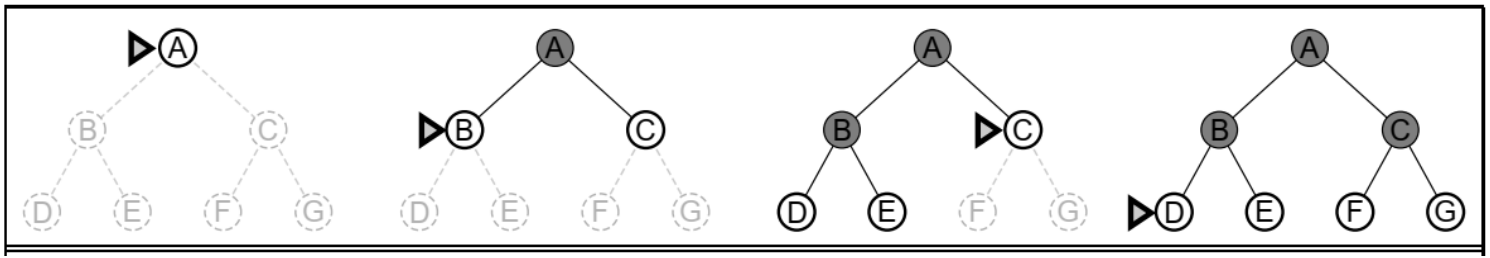
- It is also called **blind search**
- Uninformed search strategies use only the information available in the problem definition
- All they can do is generate successors and distinguish a goal state from a nongoal state.
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

Breadth-first search

- Breadth-first search is a simple strategy in which the root node is expanded first
  - then all the successors of the root node are expanded next, then their successors, and so on
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
  - Expand shallowest unexpanded node
- Implementation:
  - This is achieved very simply by using a FIFO queue for the frontier.

Breadth-first search

```
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 do
              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
              frontier ← INSERT(child, frontier)
```



Breadth-first search

```
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 do
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```