3. Solving Problems by Searching

The Romania problem

- An agent is in the city of Arad, Romania, enjoying a touring holiday
- The agent wants to
  - improve its suntan
  - improve its Romanian
  - enjoy nightlife, and
  - avoid hangovers
- The agent has a nonrefundable ticket to fly out of Bucharest the following day

Goal: Reach Bucharest on time.

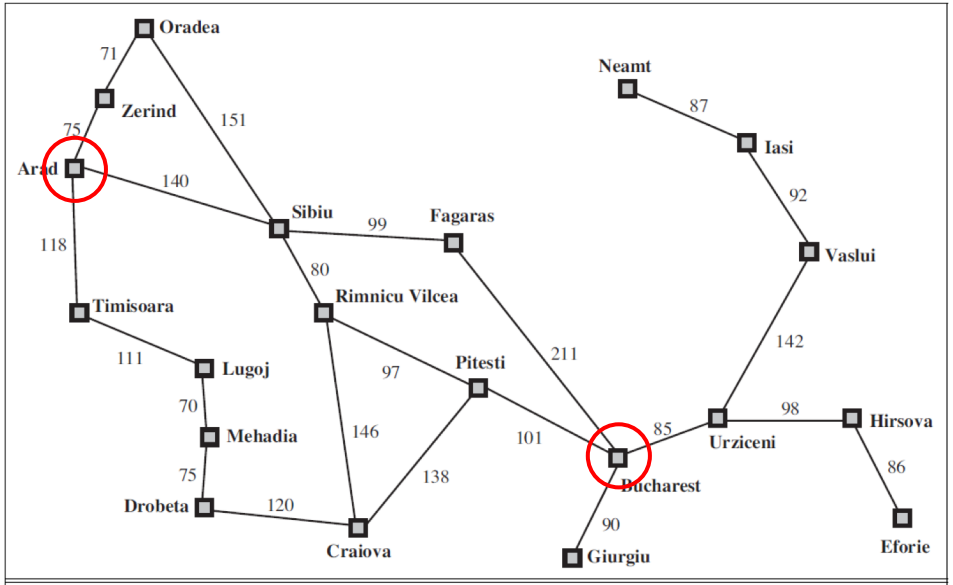


Figure 3.2 A simplified road map of part of Romania.

3.1.1 Well-defined problems and solutions

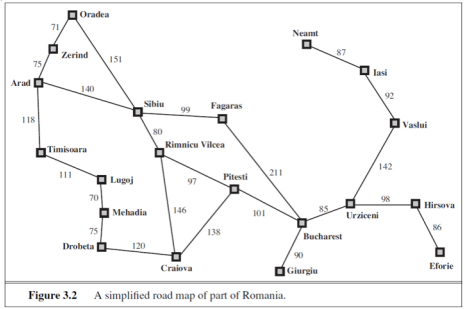


Figure 3.2 A simplified road map of part of Romania.

A problem can be defined formally by five components:

- The **initial state** that the agent starts in
  - Initial state for our agent = `In(Arad)`
- A description of possible **actions** available to the agent
  - From the state `In(Arad)` the applicable actions are `{Go(Sibiu), Go(Timisoara), Go(Zerind)}`
- A description of what each action does (**transition model**)
  - Successor = any state reachable from a given state by a single action
  - `RESULT(In(Arad), Go(Zerind)) = In(Zerind)`
  - 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
  - The state space forms a directed network (graph) in which the nodes are states and the links are actions
- The **goal test**, which determines whether a given state is a goal state
  - Goal states = `{In(Bucharest)}`
- A **path cost function** that assigns numeric cost to each path

3.1.1 Optimal solution

- We have defined the problem (including goals)
- How do we know that we achieved the goal/s?

3.1.1 Optimal solution

- A **solution** to a problem is:
  - an action sequence that leads from the initial state to goal state.
- Solution quality is measured by the **path cost** function.
- An **optimal solution** has the lowest path cost among all solutions.

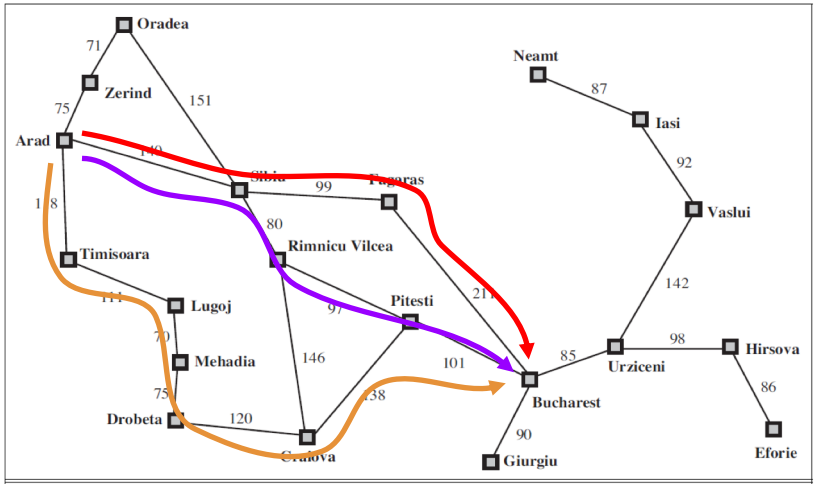


Figure 3.2 A simplified road map of part of Romania.

3.1.2 Abstraction when formulating problems

- There are so many details in the actual world!
- **Actual World State** = the travelling companions, the scenery out of the window, the condition of the road, the weather, etc.
- **Abstract Mathematical State** = `In(Arad)`
- We left out all other considerations in the state description because they are irrelevant to the problem of finding a route to Bucharest.

  Is weather irrelevant? Why?

- This process of removing details from a representation is called **abstraction**.
- Choosing the level of abstraction
  - How many details do we want to consider? Do we want to consider everything such as stopping in red light, looking out of the window?

Sample Problems

3.2.1 The Vacuum World Problem

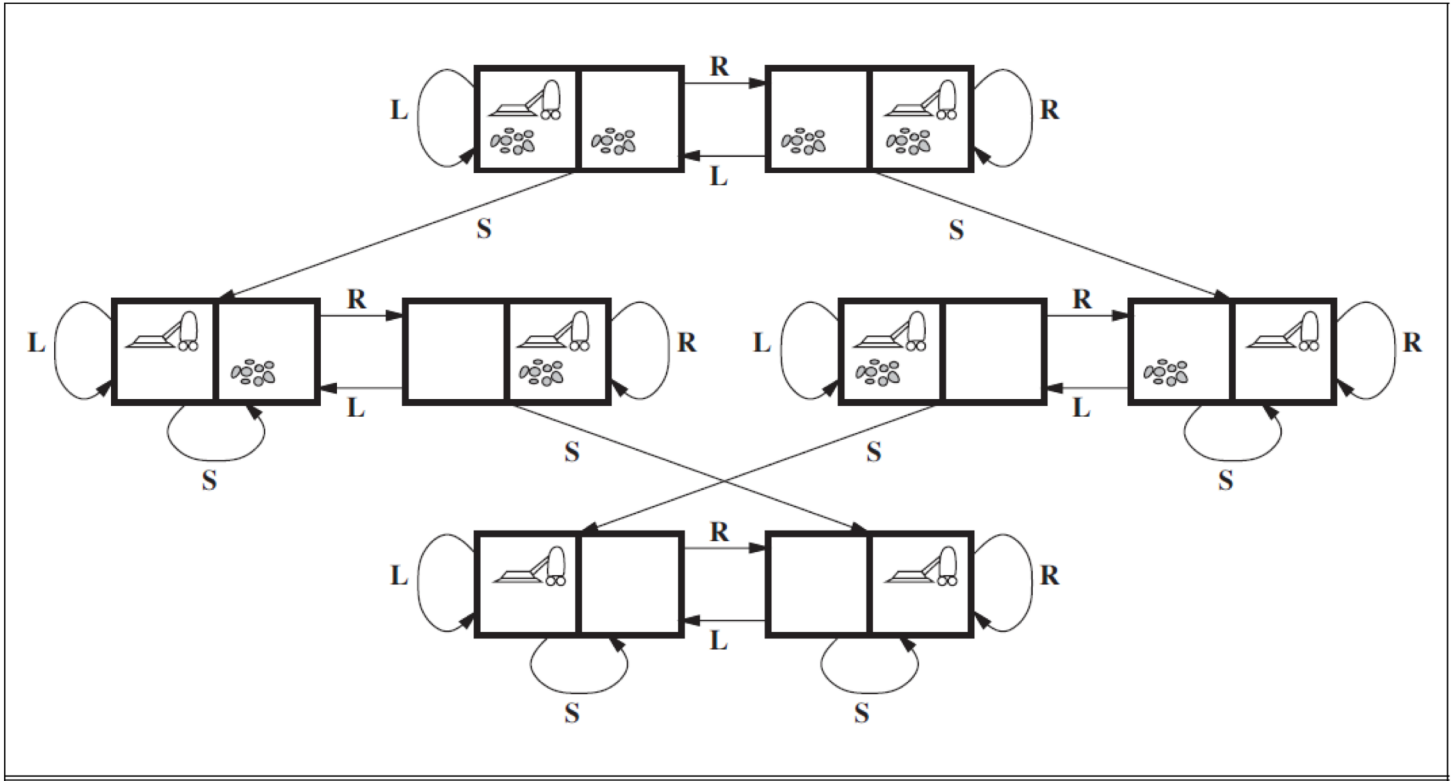


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

3.2.1 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 = $2 * 2^2 = 8$

    How many states would we have if we had four rooms instead of two?

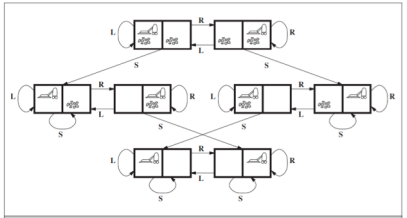


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

- **(1) Initial State:** Any state may be designated as initial state
- **(2) Actions:** Three actions: Left, Right, and Suck
- **(3) 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
- **(4) Goal test:** Check whether all the squares are clean
- **(5) Path cost:** Each step costs 1, so the path cost is the number of steps in the path

Compared with the real world, this toy problem has discrete locations, discrete dirt, reliable cleaning, and it never gets any dirtier.

3.2.1 8-puzzle

- 8-puzzle online
- 3×3 board with eight numbered tiles and a blank space.
- A tile adjacent to the blank space can slide into the space.
- The objective is to reach a specified goal state.

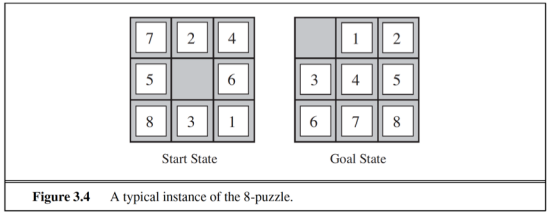


Figure 3.4 A typical instance of the 8-puzzle.

- States:
- Initial State:
- Actions:
- Transition Model:
- Goal Test:
- Path Cost:

3.2.1 8-puzzle

- 8-puzzle online
- 3×3 board with eight numbered tiles and a blank space.
- A tile adjacent to the blank space can slide into the space.
- The objective is to reach a specified goal state.

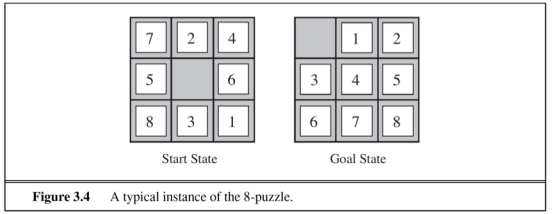
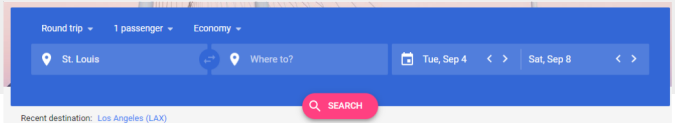


Figure 3.4 A typical instance of the 8-puzzle.

- **States:** Location of each of the eight tiles & the blank in one of the nine squares. N!
- **Initial State:** Any state
- **Actions:** Movement of the blank space Up, Down, Left, or Right
- **Transition Model:** Given a state and action, it results in the resulting state
- **Goal Test:** Checks whether the state matches the goal configuration
- **Path Cost:** Each step costs 1. Path cost is the number of steps in the path.

3.2.2 Real-world problems

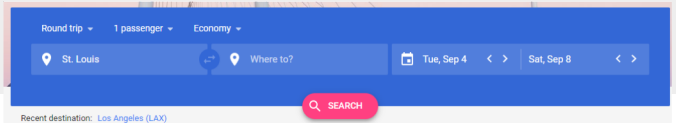


Route-finding problems are common in real world. Consider the airline travel problem that must be solved by a travel-planning Website:

- **States:** Each state includes a location (e.g. an airport) and the current time
- **Initial state:** Specified by the user's query
- **Actions:** Take any flight from current location, in any seat class, leaving after current time, leaving enough time for within-airport transfer if needed
- **Transition model:**

- **Goal test:**
- **Path cost:**

3.2.2 Real-world problems



Route-finding problems are common in real world. Consider the airline travel problem that must be solved by a travel-planning Website:

- **States:** Each state includes a location (e.g. an airport) and the current time
- **Initial state:** Specified by the user's query
- **Actions:** Take any flight from current location, in any seat class, leaving after current time, leaving enough time for within-airport transfer if needed
- **Transition model:** The state resulting from taking a flight will have flight's destination as the current location and the flight's arrival time as the current time
- **Goal test:** Are we at the final destination specified by the user?
- **Path cost:** Depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, etc.

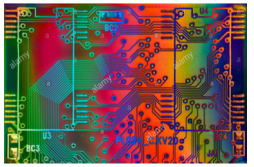
3.2.2 Real-world problems

- **Touring problems** are slightly different from route-finding problems
  - Example, "An agent wants to visit every species of animals at the St. Louis zoo at least once, starting and ending at the Snake House near the South Entrance."
- The **state space is different** - each state must include not just the current location but also the set of cities the agent has visited
  - Agent's initial state would be `In(Snake-house), Visited({Snake-house})`
  - A typical intermediate state would be `In(Giraffe-house), Visited({Snake-house, Primate-house, Giraffe-house})`
- The goal test is also different
  - Check whether the agent is in Snake-house and whether all animals have been visited

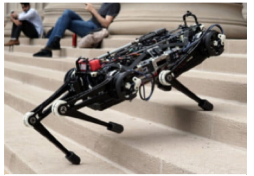
3.2.2 Additional real-world problems

- The traveling salesman problem (TSP)
  - Each city must be visited exactly once. The aim is to find the shortest tour.

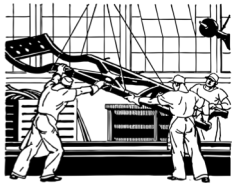
- VLSI layout problem
  - Positioning millions of components and connections on a chip to minimize area, minimize circuit delays, and maximize manufacturing yield



- Robot navigation
  - General version of the route-finding problem (a robot can move in continuous space)
  - When the robot has arms and legs or wheels, the search space becomes many-dimensional



- Automatic assembly sequencing
  - The aim is to find the order in which to assemble the parts of some object
  - If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done



Which are touring and which are route-finding problems?

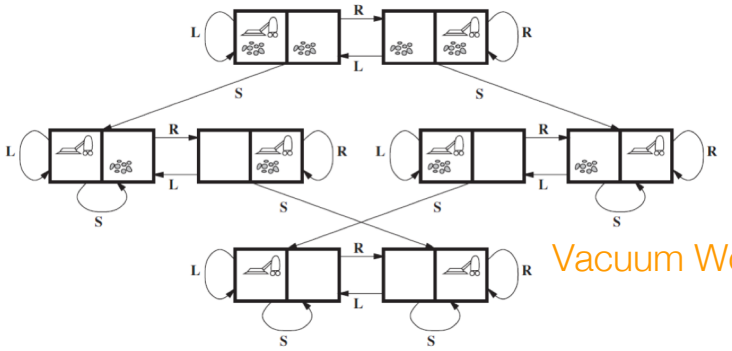
Concepts for searching algorithms

Before we discuss specific search algorithms:

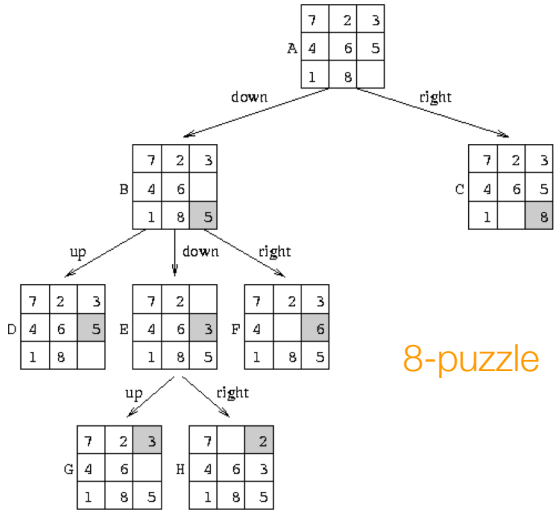
- State space
- What Exactly is ‘Searching for Solution’?
- A General Tree-Search Algorithm
- Tree Search vs Graph Search
- Graph Search Algorithm (General Tree-Search)
- Nodes vs States
- The queue data structure
- Measuring Problem-solving Performance

3.3 The concept of state space

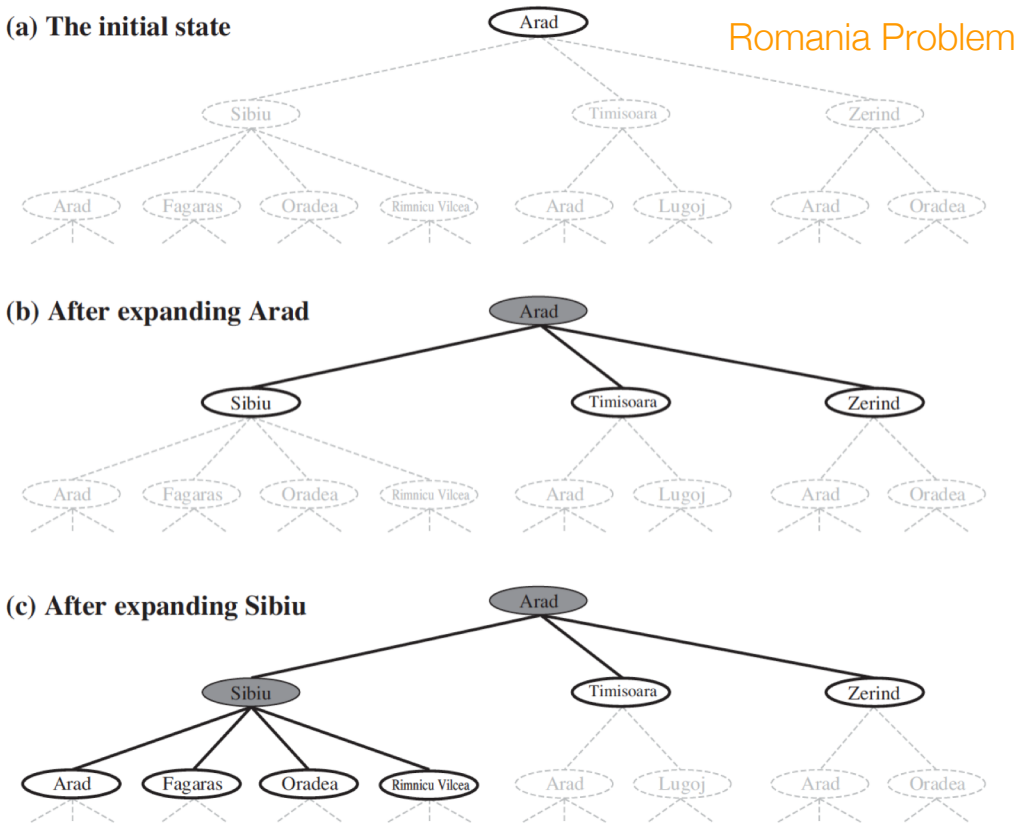
State Space = { "All Possible Configurations" } (notice the curly braces)



Vacuum World



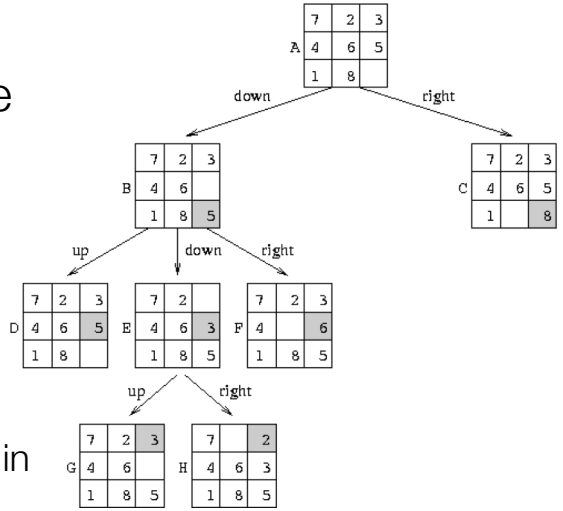
8-puzzle



Romania Problem

3.3 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 essence of search:
  - Following up one option now and putting the others aside for later, in case the first choice does not lead to a solution



3.3 What exactly is ‘Searching for Solution’? Example

- The root node of the tree corresponds to the initial state, In(Arad).
- The first step is to test whether this is a goal state
- Then we need to consider taking various actions
  - We do this by expanding the current state
  - i.e., applying each legal action to the current state, thereby generating a new set of states (Transition model)
- We add three branches from the parent node In(Arad) leading to three new child nodes
  - In(Sibiu),
  - In(Timisoara), and
  - In(Zerind)
- Now we must choose which of these three possibilities to consider further

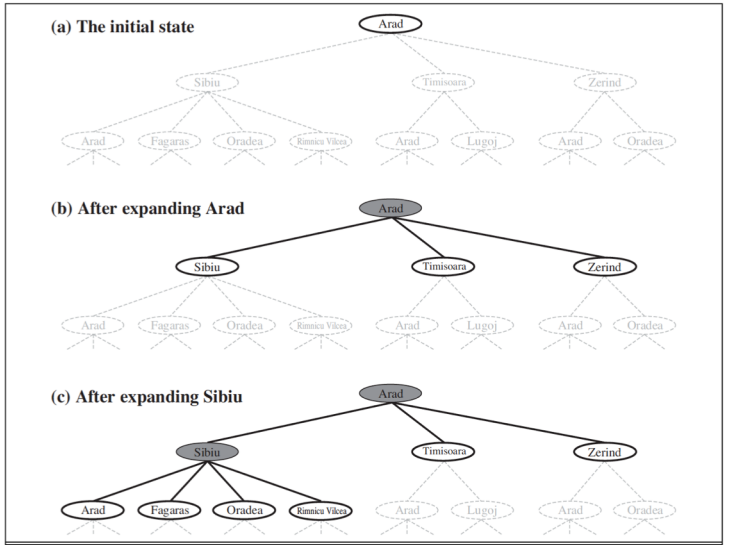


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

3.3 What exactly is ‘Searching for Solution’? Example

- Suppose we choose Sibiu first.
  - We check to see whether it is a goal state (it is not) and then expand it to get In(Arad), In(Fagaras), In(Oradea), and In(RimnicuVilcea)
  - We can then choose any of these four or go back and choose Timisoara or Zerind

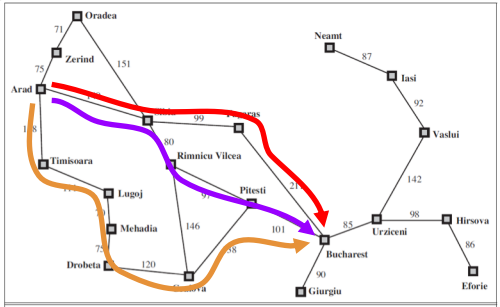


Figure 3.2 A simplified road map of part of Romania.

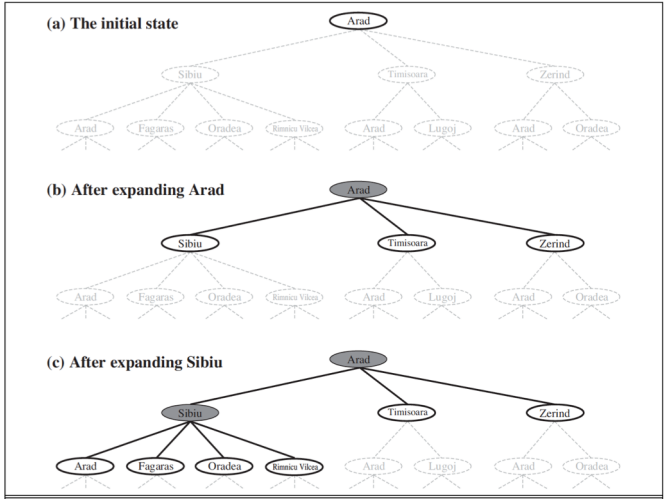


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

3.3 What exactly is 'Searching for Solution'? Example

- Suppose we choose Sibiu first.
  - We check to see whether it is a goal state (it is not) and then expand it to get In(Arad), In(Fagaras), In(Oradea), and In(RimnicuVilcea)
  - We can then choose any of these four or go back and choose Timisoara or Zerind
- Each of these six node is a **leaf node**
  - a node with no children in the tree
- The set of all leaf nodes available for expansion at any given point is called the **frontier**
  - tree consists of those nodes with bold outlines

What is the difference between a leaf node and frontier? THINK PAIR SHARE

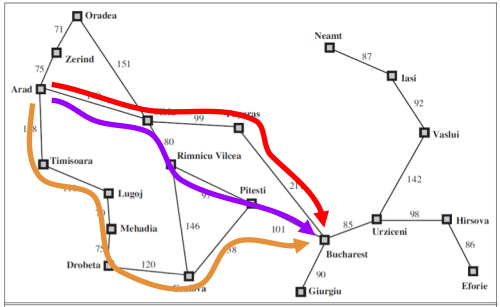


Figure 3.2 A simplified road map of part of Romania.

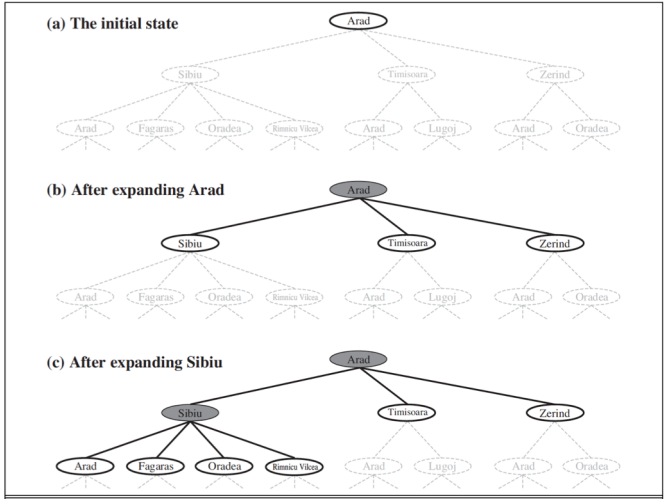
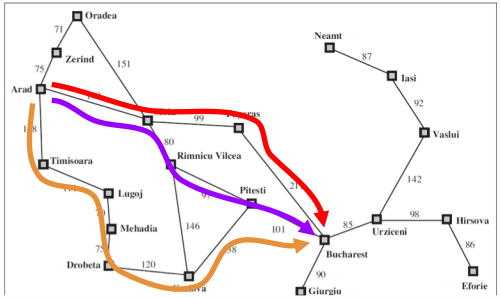


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

3.3 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



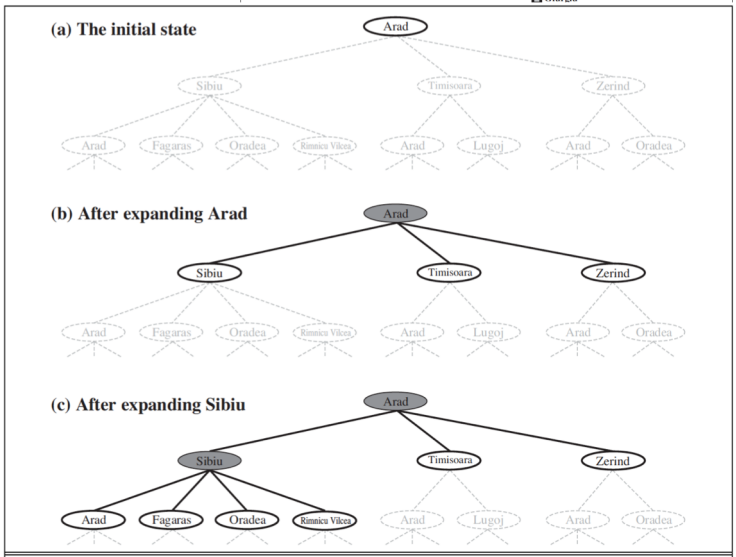


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

What is wrong with the Tree Search idea?

THINK PAIR SHARE

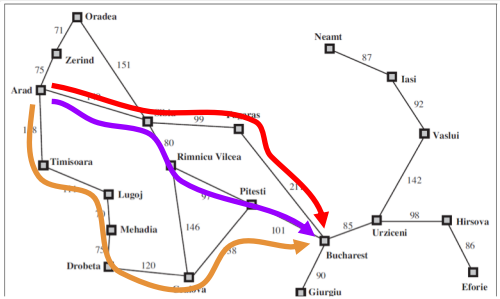
3.3 Problem with the general tree search algorithm

"algorithms that forget their history are doomed to repeat it"

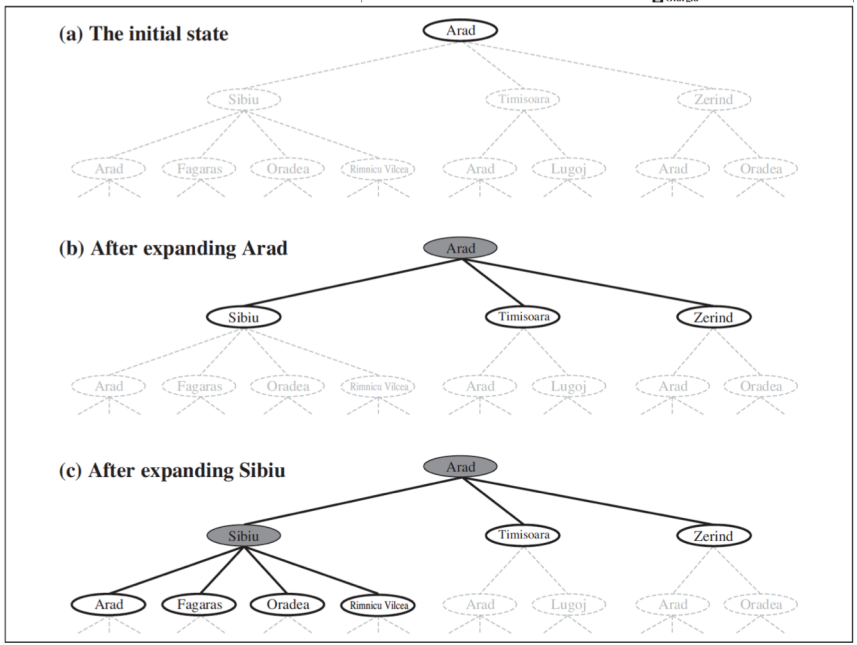
- The way to avoid exploring redundant paths is to remember where one has been
- To do this, we augment the TREE-SEARCH algorithm with a data structure called the **explored set**
  - which remembers every expanded node
- Newly generated node, that match previously generated nodes—ones in the explored set or the frontier—can be discarded instead of being added to the frontier

3.3 Graph 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



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



3.3.1 Data structures to keep track of the search tree

- Search algorithms require a **data structure** to keep track of the search tree that is being constructed

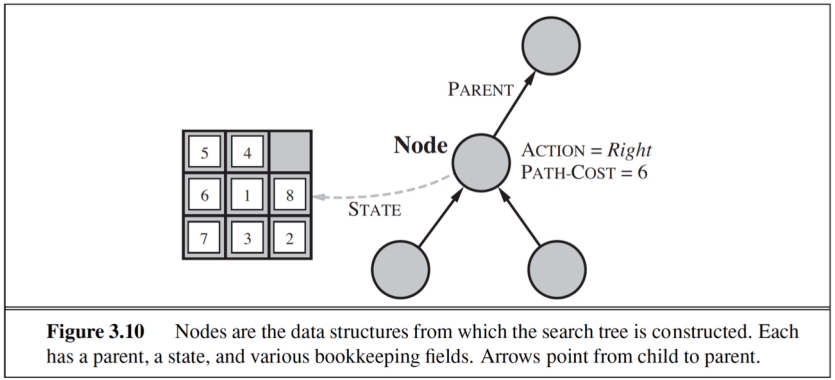


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

- For each node n of the tree, we have a structure that contains four components:
  - **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 cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

3.3.1 Data structures to keep track of the search tree

Nodes vs States

- A node is a bookkeeping data structure used to represent the search tree
- A state corresponds to a configuration of the world.

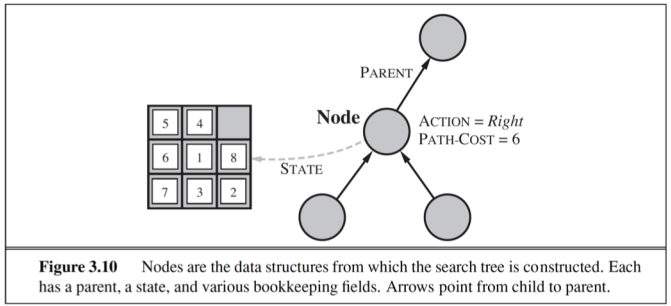


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Nodes are on particular paths, as defined by PARENT pointers, whereas states are not

Two different nodes can contain the same world state if that state is generated via two different search paths

3.3.1 The queue data structure

- The operations on a queue data structure are as follows:
  - **EMPTY?(queue)** returns true only if there are no more elements in the queue
  - **POP(queue)** removes the first element of the queue and returns it
  - **INSERT(element, queue)** inserts an element and returns the resulting queue
- Queues are characterized by the order in which they store the inserted nodes
- Three common variants are:
  - **The first-in, first-out** or FIFO queue, which pops the oldest element of the queue;
  - **The last-in, first-out** or LIFO queue (also known as a stack), which pops the newest element
  - **The priority queue**, which pops the element of the queue with the highest priority according to some ordering function.

3.3.1 The queue data structure

"The first-in, first-out" / "The last-in, first-out" / "The priority queue"

3.3.2 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?
- For S&T complexity, the typical measure is size of the state space graph, |V| + |E|
  - where V is the set of vertices (nodes) of the graph and E is the set of edges (links)

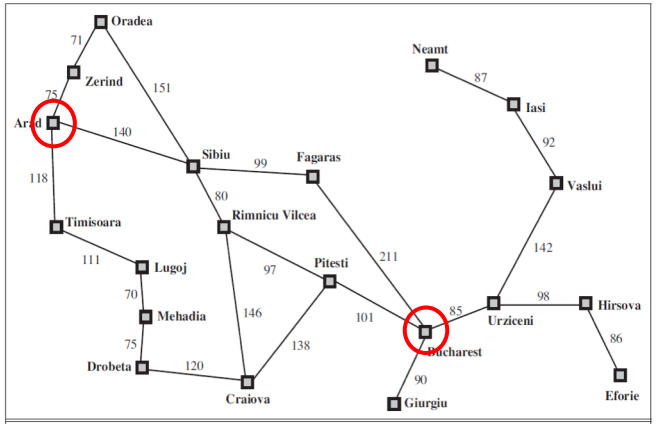


Figure 3.2 A simplified road map of part of Romania.

Search algorithms

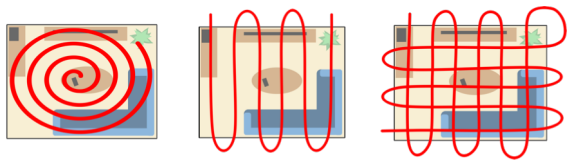
- Uninformed:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
- Heuristic-based:
  - Greedy best-first search
  - A* search

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

3.4 Uninformed search strategies

- Also called ‘blind search’
- The strategies 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**
- All search strategies are distinguished by the order in which nodes are expanded

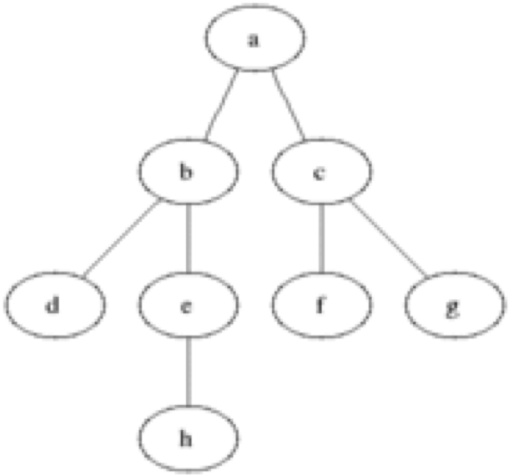
Airplane which just lost its engine...



- Strategies that know whether one non-goal state is “more promising” than another are called **informed search** or **heuristic search strategies** (we will cover later)

3.4.1 Breadth-first search

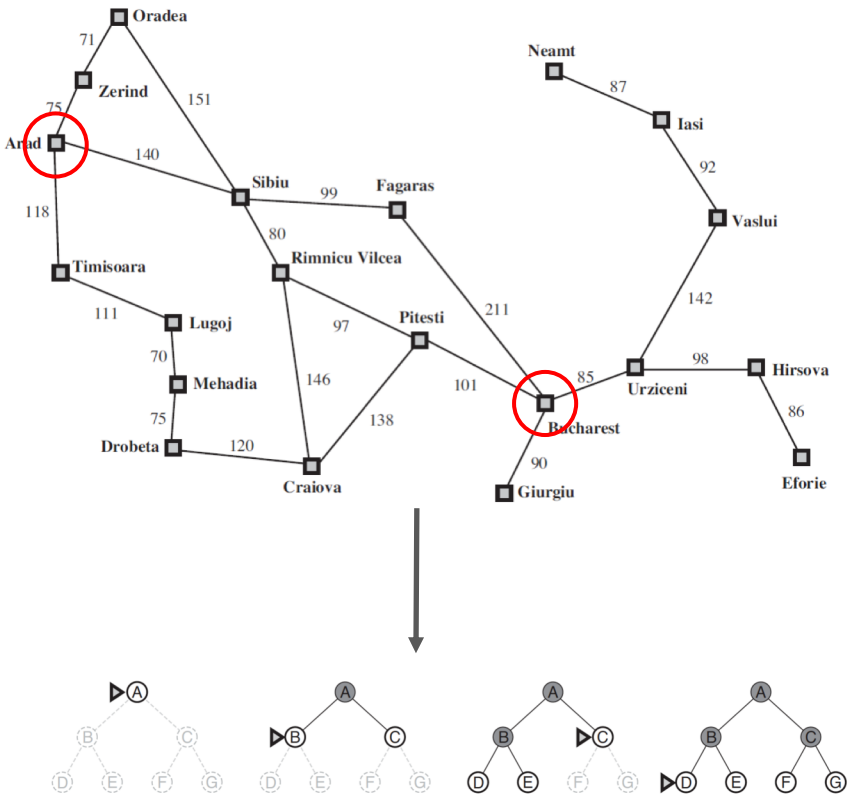
- Breadth-first search is a simple strategy in which the root node is expanded first
  - then all the successors of the root node are expanded next, then their successors, and so on



- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- This is achieved very simply by using a FIFO queue for the frontier.

3.4.1 Breadth-first search

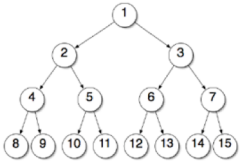
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
1   node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
2   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
3   frontier ← a FIFO queue with node as the only element
4   explored ← an empty set
5   loop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



BFS Trees

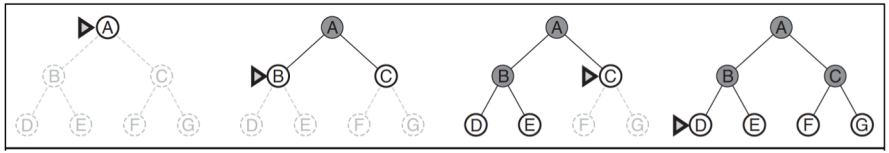
http://umsl.edu/~adhikarib/cs4130-fall2017/student-projects/mazeatron/

3.4.1 Space & time complexity of BFS



- Imagine searching a uniform tree where every state has *b* successors.
- The root of the search tree generates b nodes at the first level
  - each of which generate b more nodes, for a total of b^2 at the second level
  - each of these generate b more nodes, yielding b^3 nodes at the third level, and so on.
- Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is

$$b + b^2 + b^3 + \cdots + b^d = O(b^d)$$



- There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier, so the space complexity is $O(b^d)$, i.e., it is dominated by the size of the frontier.

3.4.1 Space & time complexity of BFS

An exponential complexity bound such as $O(b^d)$ is scary!

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

(1) Memory requirements are a bigger problem than execution time

(2) Time is still a major factor

- If your problem has a solution at depth 16, then (given our assumptions) it will take about 350 years for breadth-first search (or indeed any uninformed search) to find it.

Limitation of BFS

- When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node
- But...

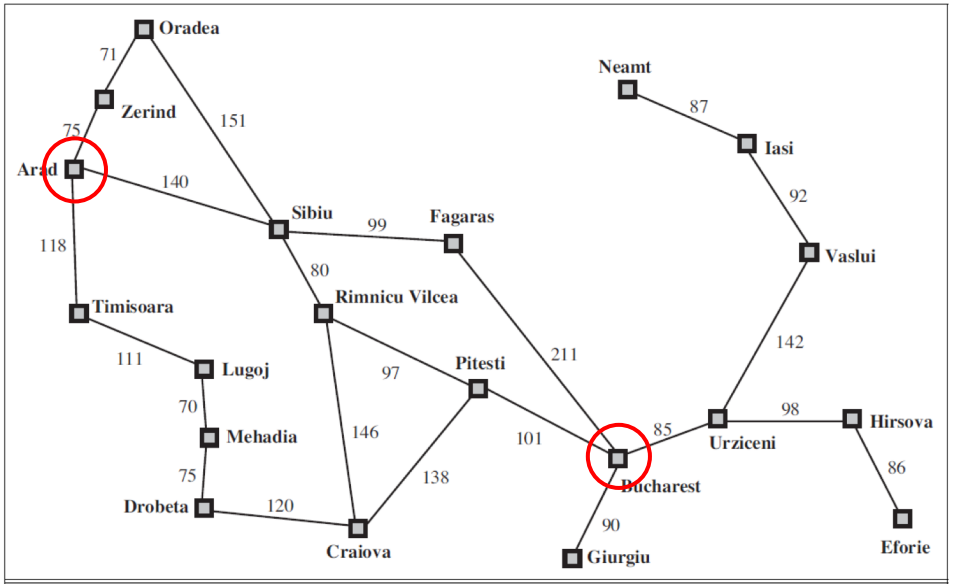


Figure 3.2 A simplified road map of part of Romania.

3.4.2 Uniform-cost search

- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost g(n).
- This is done by storing the frontier as a **priority queue** ordered by g.
- Uniform-cost search does not care about the number of steps a path has, but only about their **total cost**

Animation of the Uniform Cost Algorithm

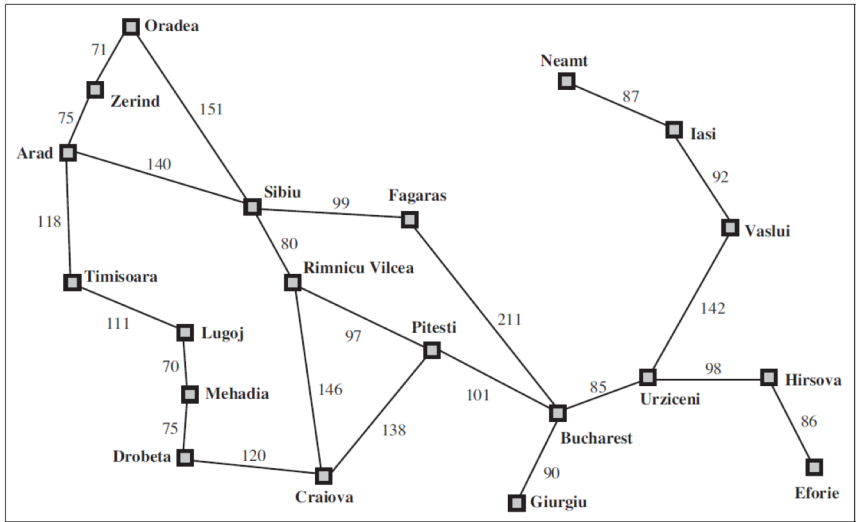
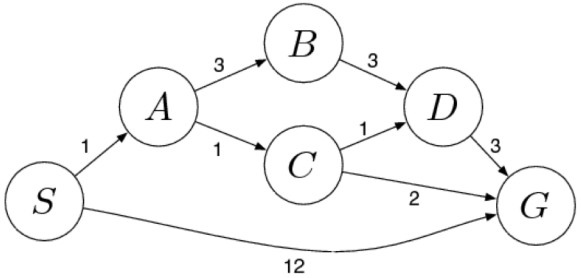


Figure 3.2 A simplified road map of part of Romania.

3.4.2 Uniform-cost search

Problem. What will be the contents of the priority queue, as the UCS algorithm proceeds?



Contents of the Priority Queue (Frontier)

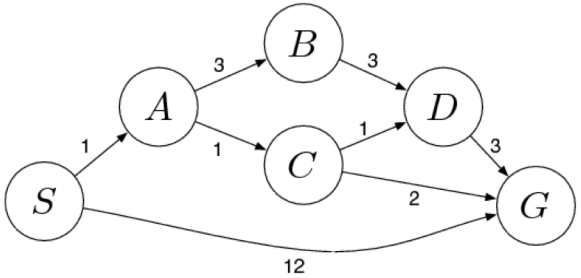
1. ϕ

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

```
1  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
2  frontier ← a priority queue ordered by PATH-COST, with node as the only element
3  explored ← an empty set
4  loop do
5      if EMPTY?(frontier) then return failure
6      node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
7      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
8      add node.STATE to explored
       // Print the contents of the Priority Queue (frontier)
9      for each action in problem.ACTIONS(node.STATE) do
10         child ← CHILD-NODE(problem, node, action)
11         if child.STATE is not in explored or frontier then
12             frontier ← INSERT(child, frontier)
13         else if child.STATE is in frontier with higher PATH-COST then
14             replace that frontier node with child
```

3.4.2 Uniform-cost search

Problem. What will be the contents of the priority queue, as the UCS algorithm proceeds?



Contents of the Priority Queue (Frontier)

1. ϕ
2. G-12
3. B-4, G-12
4. B-4, G-4
5. G-4

If G-4 is selected before B-4, B-4 will never be processed!

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

```
1  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
2  frontier ← a priority queue ordered by PATH-COST, with node as the only element
3  explored ← an empty set
4  loop do
5      if EMPTY?(frontier) then return failure
6      node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
7      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
8      add node.STATE to explored
       // Print the contents of the Priority Queue (frontier)
9      for each action in problem.ACTIONS(node.STATE) do
10         child ← CHILD-NODE(problem, node, action)
11         if child.STATE is not in explored or frontier then
12             frontier ← INSERT(child, frontier)
13         else if child.STATE is in frontier with higher PATH-COST then
14             replace that frontier node with child
```

How will your results change if we check the PQ contents right after the "loop do"? THINK PAIR SHARE

3.4.3 Depth-first search

- Depth-first search always expands the deepest node in the current frontier of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.

Breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue (stack).

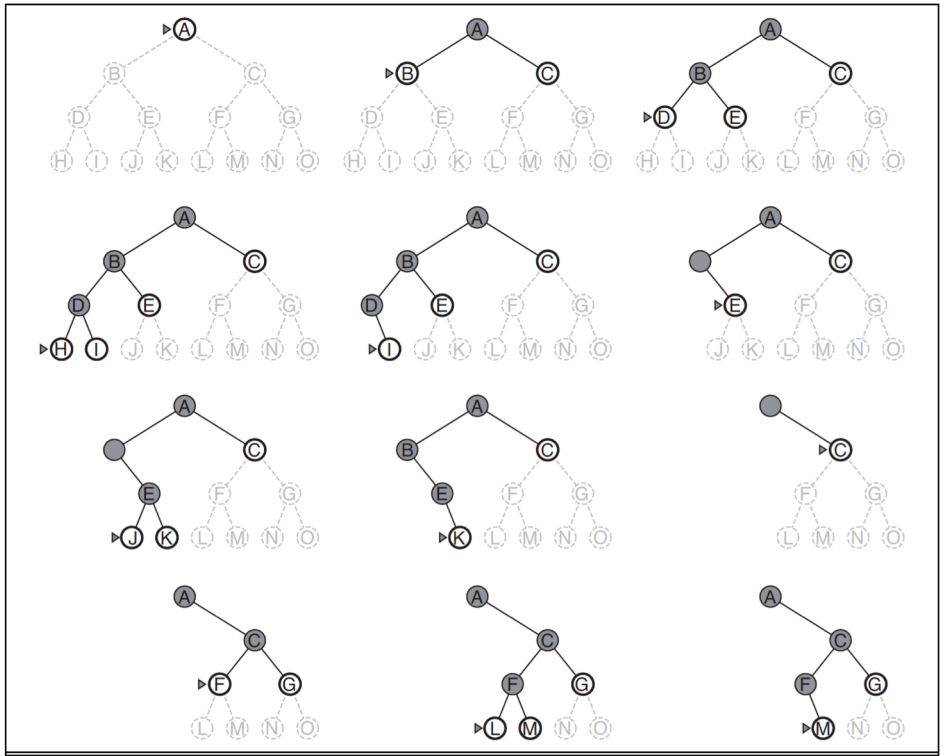


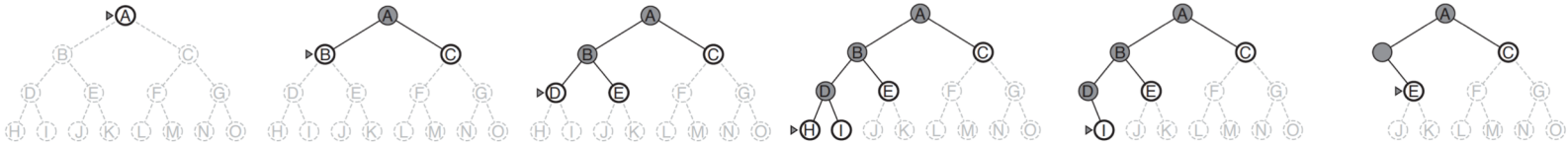
Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

3.4.3 Depth-first search

DFS uses LIFO data structure, i.e. stack.

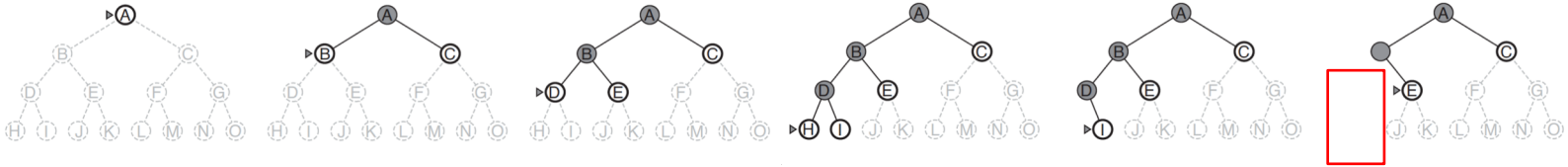
What changes will you make to this algorithm to make it Depth-first Search?

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



3.4.3 Advantage of DFS over BFS - space complexity

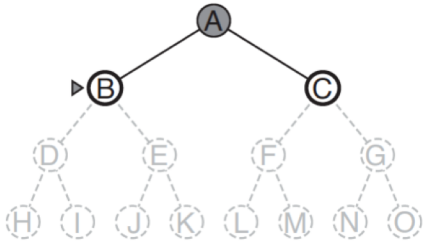
- DFS has advantages over BFS when searching in a tree (**not in a graph**)
- In case of tree search, DFS needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
- Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.



- This has led to the adoption of depth-first tree search as the basic workhorse of many areas of AI, including **constraint satisfaction**.

3.5 Informed (heuristic) search strategies

- Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm.
- A node is selected for expansion based on an evaluation function, **f(n)**
- The evaluation function is construed as a **cost estimate**
  - so the node with the lowest evaluation is expanded first
- The implementation is identical to that for uniform-cost search
  - except for the use of **f instead of g** to order the priority queue.
- The choice of **f** determines the search strategy
- Most best-first algorithms include as a component of **f** a heuristic function, **h(n)**
  - **h(n) = estimated cost of the cheapest path from the state at node n to a goal state**
- We can consider h(n) to be arbitrary, nonnegative, problem-specific functions, with one constraint: if n is a goal node, then h(n)=0



3.5.1 Greedy best-first search

- Greedy best-first search tries to expand the node that is closest to the goal
  - on the grounds that this is likely to lead to a solution quickly
- Route-finding problems in Romania
  - Heuristic = straight line distance heuristic (h_{SLD})
  - If the goal is Bucharest, we need to know the straight-line distances to Bucharest
  - The values of h_{SLD} cannot be computed from the problem description itself
  - It takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Values of h_{SLD}—straight-line distances to Bucharest.

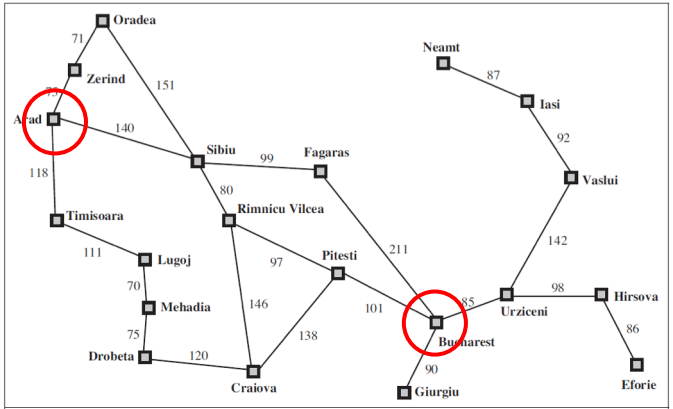
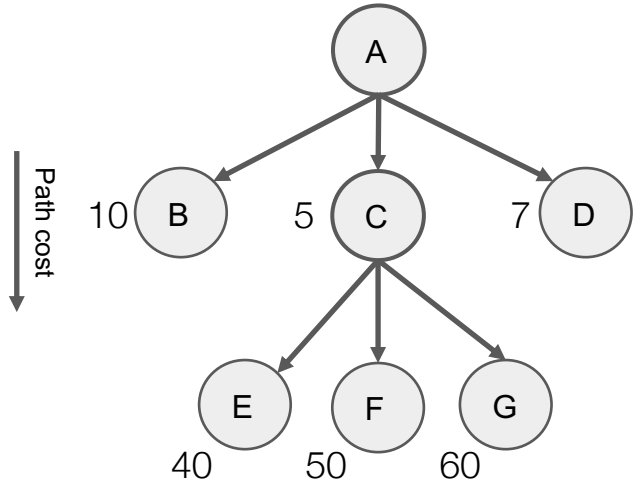


Figure 3.2 A simplified road map of part of Romania.

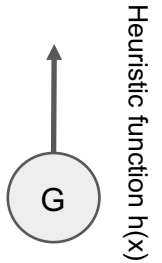
Uniform-cost vs Greedy best-first

Example: Given node A, having child nodes B, C, D with associated costs of (10, 5, 7). After expanding C, we see nodes E, F, G with costs of (40, 50, 60). Which node will be chosen by UCS and GBS? Why?



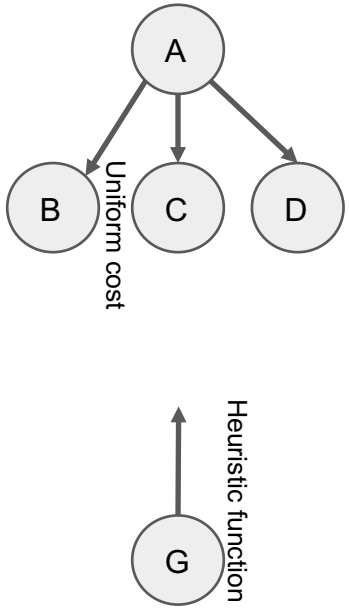
Uniform-cost-search:

Greedy Best-first Search:



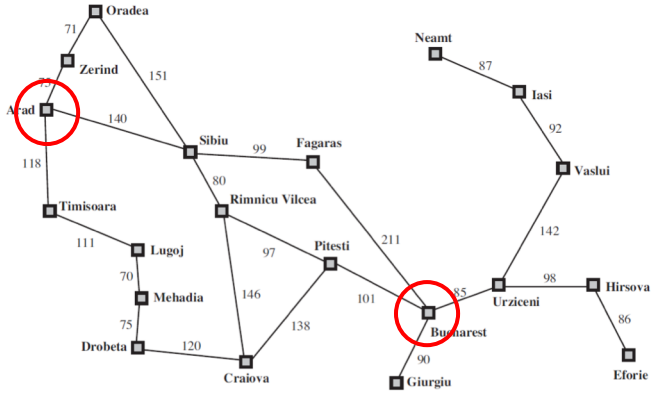
3.5.2 A* search

- The most widely known form of best-first search
- It evaluates nodes by combining
  - g(n), the cost to reach the node, and
  - h(n), the cost to get from the node to the goal
  - f(n) = g(n) + h(n)
- Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal
  - f(n) = estimated cost of the cheapest solution through n
- The algorithm is identical to UNIFORM-COST-SEARCH except that A* uses g + h instead of g



3.5.2 A* search

Problem. Given the following map (with the path costs shown) and the table with the heuristic function, show the stages of the A* search algorithm.

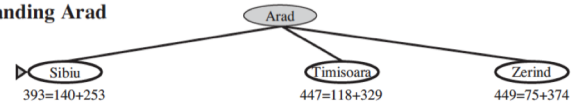


Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Values of h_{SLD}—straight-line distances to Bucharest.

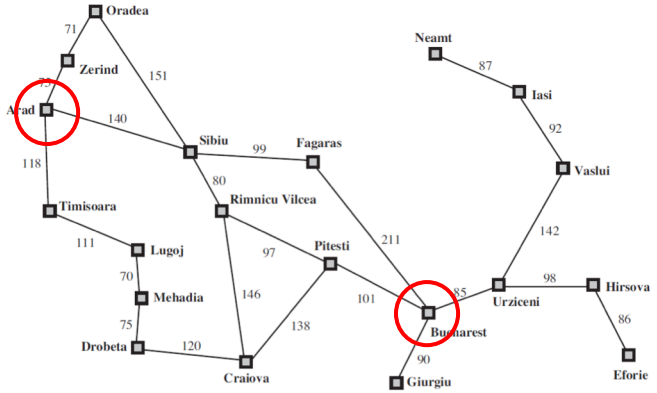
(a) The initial state

(b) After expanding Arad



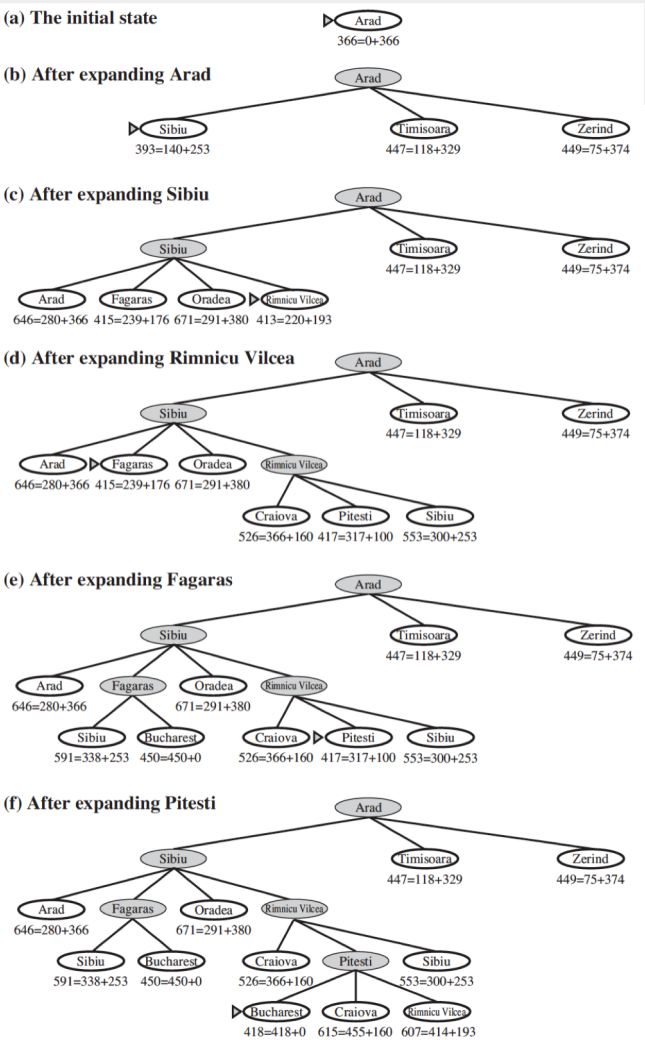
3.5.2 A* search

Problem. Given the following map (with the path costs shown) and the table with the heuristic function, show the stages of the A* search algorithm.



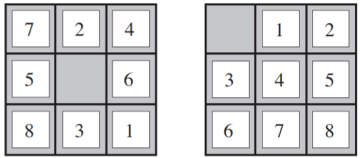
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Values of h_{SLD}—straight-line distances to Bucharest.



3.6 Another example of heuristic function (8-puzzle)

- The average solution cost for a randomly generated 8-puzzle instance is about 22 steps
- The branching factor is about 3
  - When the empty tile is in the middle, four moves are possible; when it is in a corner, two; and when it is along an edge, three

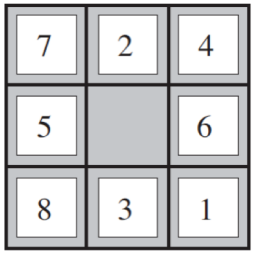


- This means that an exhaustive tree search to depth 22 would look at about $3^{22} \approx 3.1 \times 10^{10}$ states
  - This is reduced to 181, 440 distinct states (which is still very large)
  - Corresponding number for the 15-puzzle is roughly 10^{13}
- We need a heuristic function that never overestimates the number of steps to the goal

3.6 Another example of heuristic function (8-puzzle)

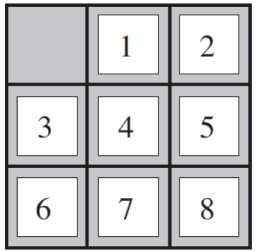
Two commonly used heuristics for the 8-puzzle:

- h_1 = the number of misplaced tiles
  - all of the eight tiles are out of position
  - h_1 is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once
  - h_1 for the start state = 8



Start State

- h_2 = the sum of the distances of the tiles from their goal positions
  - h_2 for the start state = 3 + 1 + 2 + 2+ 2 + 3+ 3 + 2 = 18
  - the distance we count is the sum of the horizontal and vertical distances
    - because tiles cannot move along diagonals
  - h_2 is admissible because all any move can do is move one tile one step closer to the goal



Goal State

- The "h_2" heuristic is sometimes called the **city block distance** or **Manhattan distance**.

True solution cost = 26 steps

Summary

- A problem consists of five parts: the initial state, a set of actions, a transition model describing the results of those actions, a goal test function, and a path cost function.
- Uninformed search methods have access only to the problem definition
  - Breadth-first search expands the shallowest nodes first
  - Uniform-cost search expands the node with lowest path cost, g(n)
  - Depth-first search expands the deepest unexpanded node first.
- Informed search methods may have access to a heuristic function h(n) that estimates the cost of a solution from n
  - Greedy best-first search expands nodes with minimal h(n)
  - A∗ search expands nodes with minimal f(n) = g(n) + h(n)
- The performance of heuristic search algorithms depends on the quality of the heuristic function.