In [3]:

```
1
```

In [10]:

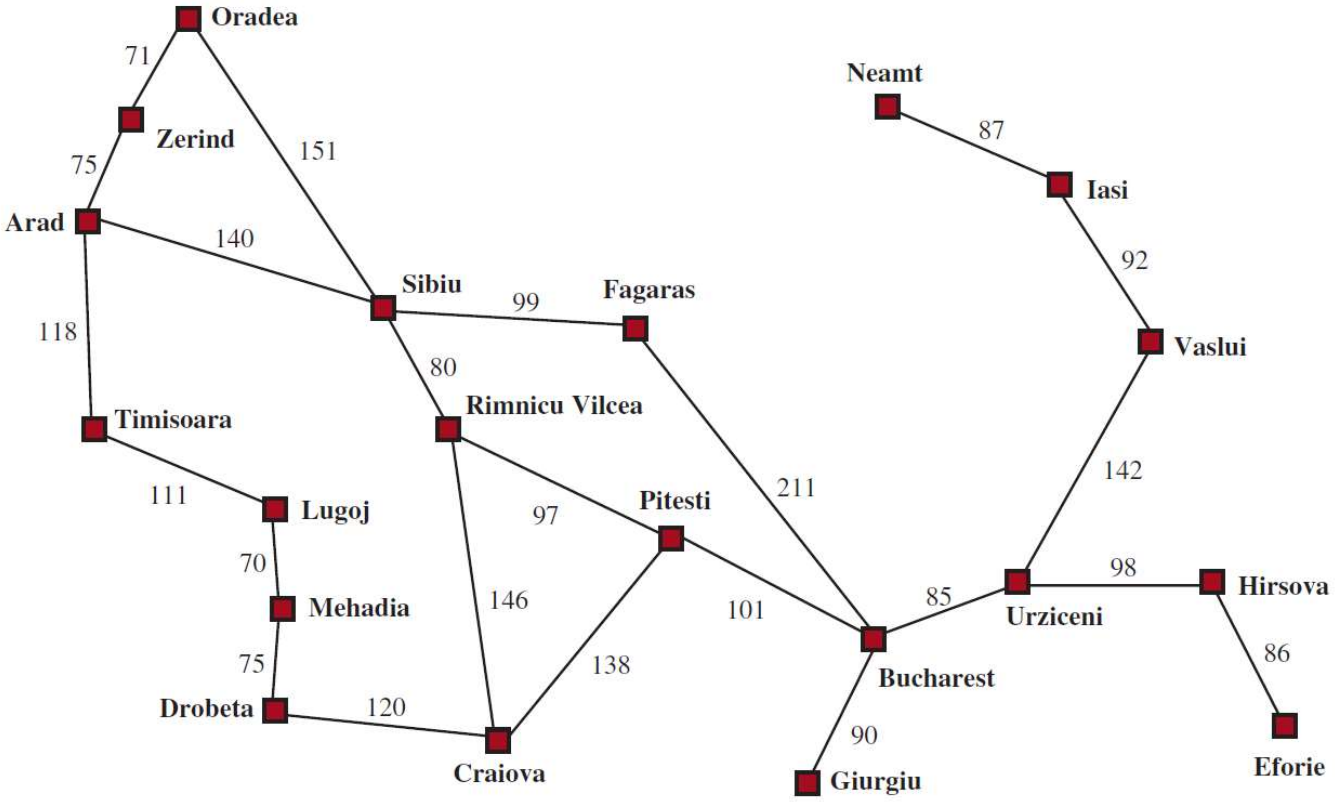
```
1 from IPython.display import Video
```

> **Solving Problems by Searching**

Introduction

- When the correct action to take is not immediately obvious, an agent may need 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**.
- In case of **informed algorithms**, the agent can estimate how far it is from the goal, and in case of uninformed algorithms, no such estimate is available.

Suppose the agent is currently in the city of Arad and adopts the goal of reaching Bucharest the following day. The agent observes street signs and sees that there are three roads leading out of Arad: one toward Sibiu, one to Timisoara, and one to Zerind. None of these are the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow. Let us assume our agents always have access to information about the world, such as the map in the following figure. With that information, the agent can follow a four-phase problem-solving process:



Four-phases of problem-solving process:

Goal formulation: The agent adopts the goal of reaching Bucharest. Goals organize behavior by limiting the objectives and hence the actions to be considered.

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 travelling 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**. The agent might have to simulate multiple sequences that do not reach the goal, but eventually it will find a solution (such as going from Arad to Sibiu to Fagaras to Bucharest), or it will find that no solution is possible.

Execution: The agent can now execute the actions in the solution, one at a time.

In a fully observable, deterministic, known environment, the solution to any problem is a

fixed sequence of actions.

- Drive to Sibiu, then Fagaras, then Bucharest.

If the model is correct, then once the agent has found a solution, it can ignore its percepts while it is executing the actions—closing its eyes, so to speak—because the solution is guaranteed to lead to the goal. Control theorists call this an **open-loop system**: ignoring the percepts breaks the loop between agent and environment. If there is a chance that the model is incorrect, or the environment is nondeterministic, then the agent would be safer using a **closed-loop approach** that monitors the percepts.

In partially observable or nondeterministic environments, a solution would be a **branching strategy** that recommends different future actions depending on what percepts arrive.

- For example, the agent might plan to drive from Arad to Sibiu but might need a contingency plan in case it arrives in Zerind by accident or finds a sign saying "Road Closed".

Search problems and solutions

A search problem can be defined formally as follows:

- A set of possible states that the environment can be in. We call this the **state space**.
- The **initial state** that the agent starts in.
    - For example: Arad. State space
- A set of one or more **goal states**. Sometimes there is one goal state (e.g., Bucharest), sometimes there is a small set of alternative goal states, and sometimes the goal is defined by a property that applies to many states (potentially an infinite number).
    - For example, in a vacuum-cleaner world, the goal might be to have no dirt in any location, regardless of any other facts about the state.
    - We can account for all three of these possibilities by specifying an IS-GOAL method for a problem.
- The **actions** available to the agent. Given a state s, ACTIONS(s) returns a finite set of actions that can be executed in s. We say that each of these actions is applicable in s.

    - ACTIONS(Arad) = {ToSibiu, ToTimisoara, ToZerindg}.
- A **transition model**, which describes what each action does. RESULT(s, a) returns the state that results from doing action a in state s.
    - RESULT(Arad, ToZerind) = Zerind.
- An **action cost function**, denoted by ACTION-COST(s, a, s') when we are programming or c(s, a, s') when we are doing math, that gives the numeric cost of applying action a in state s to reach state s'. A problem-solving agent should use a cost function that reflects its own performance measure.
    - for example, for route-finding agents, the cost of an action might be the length in miles, or it might be the time it takes to complete the action.
- A sequence of actions forms a **path**, and a **solution** is a path from the initial state to a goal state. We assume that action costs are additive; that is, the total cost of a path is the sum of the individual action costs.
- An **optimal solution** has the lowest path cost among all solutions.
- We assume that all action costs will be positive, to avoid certain complications.
- The state space can be represented as a graph in which the vertices are states and the directed edges between them are actions.
- The map of Romania shown in the previous figure is such a graph, where each road indicates two actions, one in each direction.

Formulating problems

- Our formulation of the problem of getting to Bucharest is a model—an abstract mathematical description—and not the real thing.
- Compare the simple atomic state description Arad to an actual trip, where the state of the world includes so many things: the traveling companions, the scenery out of the window, the condition of the road, the weather, the traffic, and so on. All these considerations are left out of our model because they are irrelevant to the problem of finding a route to Bucharest.
- **Abstraction** The process of removing detail from a representation is called abstraction.
- A good problem formulation has the right level of detail.

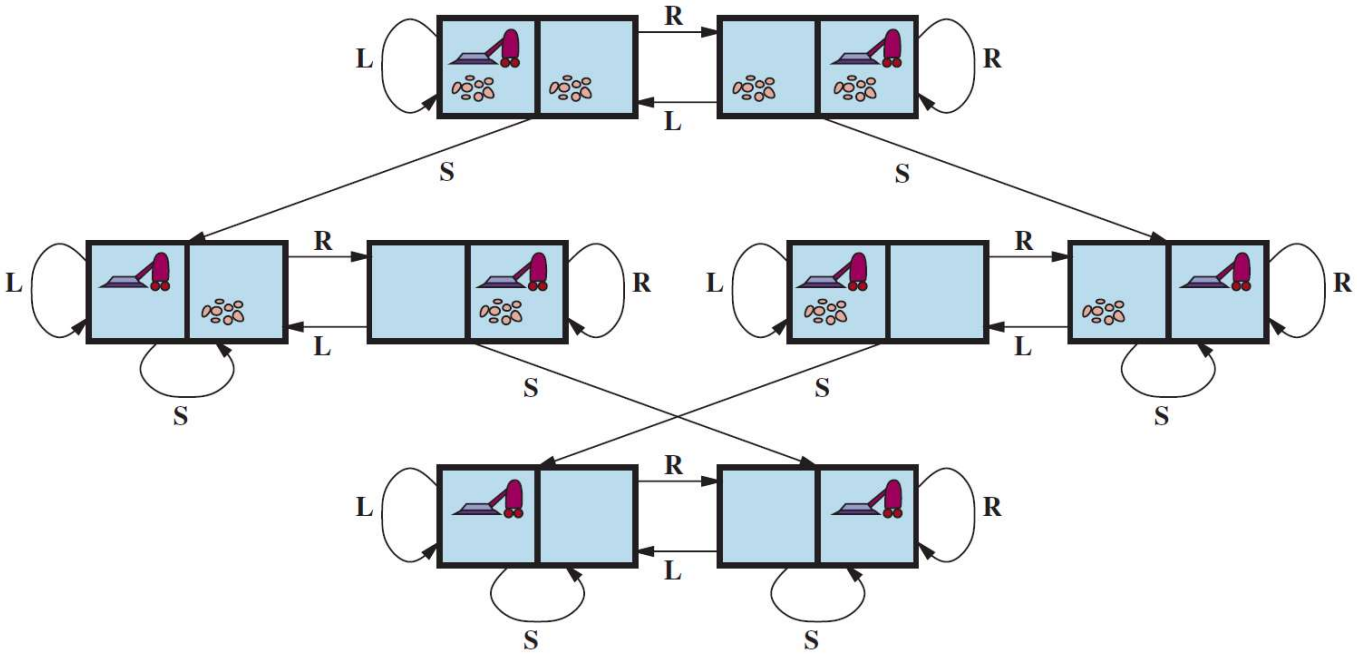
- The abstraction is **valid** if we can elaborate any abstract solution into a solution in the more detailed world.
- The abstraction is useful if carrying out each of the actions in the solution is easier than the original problem;
    - in our case, the action “drive from Arad to Sibiu” can be carried out without further search or planning by a driver with average skill.

Standardized Problem (Grid World Problem)

A **standardized problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is suitable as a benchmark for researchers to compare the performance of algorithms.

A **grid world problem** is a two-dimensional rectangular array of square cells in which agents can move from cell to cell. Typically the agent can move to any obstacle-free adjacent cell horizontally or vertically and in some problems diagonally. Cells can contain objects, which the agent can pick up, push, or otherwise act upon; a wall or other impassible obstacle in a cell prevents an agent from moving into that cell.

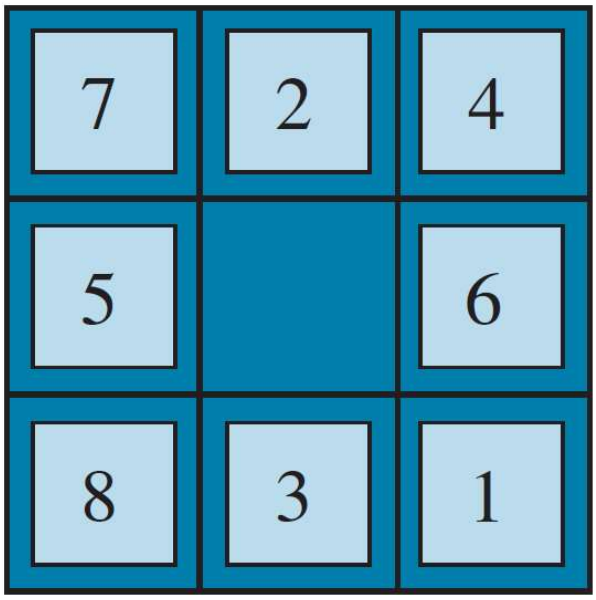
The **vacuum cleaner problem** can be formulated as a grid world problem as follows:



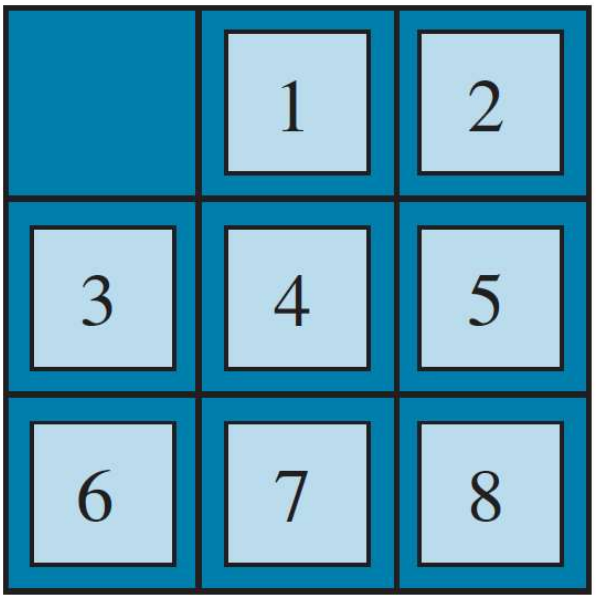
- States: A state of the world says which objects are in which cells. For the vacuum world, the objects are the agent and any dirt. In the simple two-cell version, the agent can be in either of the two cells, and each cell can either contain dirt or not, so there are $2 \times 2 \times 2 = 8$ states. In general, a vacuum environment with n cells has $n \times 2^n$ states.
- 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.

Standardized Problem (Sliding-tile puzzle)

In a sliding-tile puzzle, a number of tiles (sometimes called blocks or pieces) are arranged in a grid with one or more blank spaces so that some of the tiles can slide into the blank space. Perhaps the best-known variant is the 8-puzzle, which consists of a 3×3 grid with eight numbered tiles and 15-puzzle one blank space, and the 15-puzzle on a 4×4 grid. The object is to reach a specified goal state, such as the one shown on the right of the figure.



Start State



Goal State

The standard formulation of the 8 puzzle is as follows:

- 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.
- 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.
- Goal state: Although any state could be the goal, we typically specify a state with the numbers in order.
- Action cost: Each action costs 1.

Key Terminology

- A search algorithm takes a search problem as input and returns a solution, or an indication of Search algorithm failure.

- In this chapter we consider algorithms that superimpose a search tree over the statespace graph, forming various paths from the initial state, trying to find a path that reaches a goal state.
- Each node in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the tree corresponds to the initial state of the problem.
- It is important to understand the distinction between the state space and the search tree.
    - The **state space** describes the (possibly infinite) set of states in the world, and the actions that allow transitions from one state to another.
    - The **search tree** describes paths between these states, reaching towards the goal. The search tree may have multiple paths to (and thus multiple nodes for) any given state, but each node in the tree has a unique path back to the root (as in all trees).
- The root node of the search tree is at the initial state. We can **expand** the node, by considering the available ACTIONS for that state, using the RESULT function to see where those actions lead to, and generating a new node (called a child node or successor node) for each of the resulting states.
- A set of unexpanded nodes are called the **frontier** of the search tree.
- Any state that has had a node generated for it has been **reached** (whether or not that node has been expanded)
- The **frontier** separates two regions of the state-space graph: an interior region where every state has been expanded, and an exterior region of states that have not yet been reached.

Search data structure

Search algorithms require a data structure to keep track of the search tree. A node in the tree is represented by a data structure with four components:

- **node.STATE**: the state to which the node corresponds;
- **node.PARENT**: the node in the tree that generated this node;
- **node.ACTION**: the action that was applied to the parent's state to generate this node;

- **node.PATH-COST**: the total cost of the path from the initial state to this node. In mathematical formulas, we use g(node) as a synonym for PATH-COST.

Measuring search algorithm's performance

An algorithm's performance can be measured in four ways:

- **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 Time complexity seconds, or more abstractly by the number of states and actions considered.
- **Space complexity**: How much memory is needed to perform the search?

Uninformed Search Strategies

Best First Search

In Best first search, we choose a node, n, with minimum value of some evaluation function, f (n).

On each iteration we choose a node on the frontier with minimum f (n) value, return it if its state is a goal state, and otherwise apply EXPAND to generate child nodes. Each child node is added to the frontier if it has not been reached before, or is re-added if it is now being reached with a path that has a lower path cost than any previous path.

The algorithm returns either an indication of failure, or a node that represents a path to a goal.

By employing different f(n) functions, we get different specific algorithms

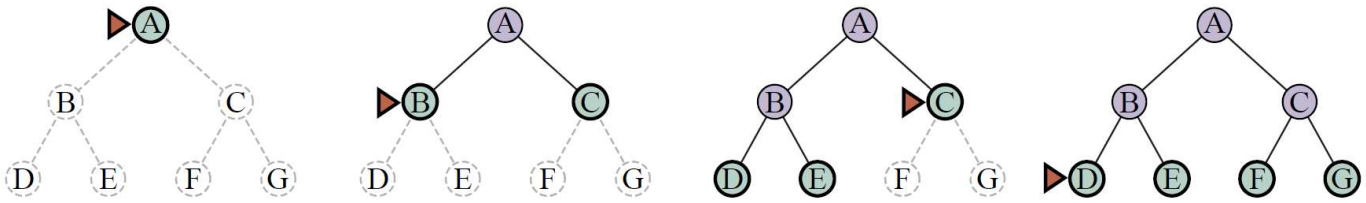
```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Breadth First search

When all actions have the same cost, an appropriate strategy is breadth first search, 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. This is a systematic search strategy that is therefore complete even on infinite state spaces.

Breadth-first search always finds a solution with a minimal number of actions, because when it is generating nodes at depth d, it has already generated all the nodes at depth d-1, so if one of them were a solution, it would have been found. That means it is cost-optimal



```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Dijkstra's algorithm or uniform-cost search

When actions have different costs, an obvious choice is to use best-first search where the evaluation function is the cost of the path from the root to the current node. This is called Dijkstra's algorithm by the theoretical computer science community, and uniform-cost search by the AI community.

The algorithm can be implemented as a call to BEST-FIRST-SEARCH with PATH-COST as the evaluation function.

```
function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

When all action costs are equal, uniform-cost search is similar to breadth first search.

Uniform-cost search is complete and is cost-optimal, because the first solution it finds will have a cost that is at least as low as the cost of any other node in the frontier.

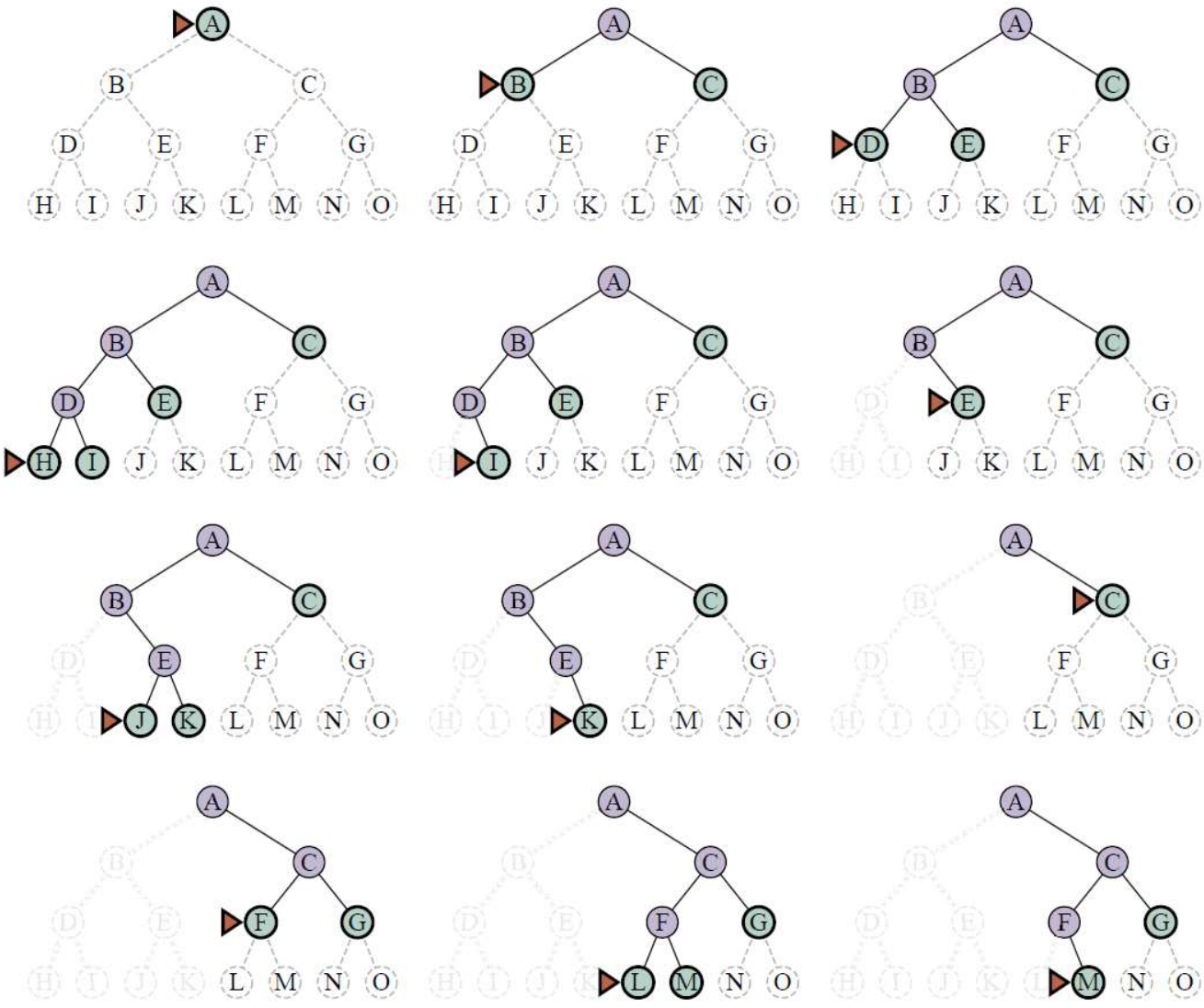
Uniformcost search considers all paths systematically in order of increasing cost, never getting caught going down a single infinite path (assuming that all action costs are $> \epsilon > 0$).

Depth-first search

Depth-first search always expands the deepest node in the frontier first.

It could be implemented as a call to BEST-FIRST-SEARCH where the evaluation function f is the negative of the depth.

However, it is usually implemented not as a graph search but as a tree-like search that does not keep a table of reached states.



Depth-first search is not cost-optimal; it returns the first solution it finds, even if it is not cheapest.

For finite state spaces that are trees it is efficient and complete; for acyclic state spaces it may end up expanding the same state many times via different paths, but will (eventually) systematically explore the entire space.

In cyclic state spaces it can get stuck in an infinite loop; therefore some implementations of depth-first search check each new node for cycles.

Finally, in infinite state spaces, depthfirst search is not systematic: it can get stuck going down an infinite path, even if there are no cycles. Thus, **depth-first search is incomplete**.

With all these demerits, why would anyone consider using depth-first search rather than breadth-first or best-first?

For problems where a tree-like search is feasible, depth-first search has much smaller needs for memory. We don't keep a reached table at all, and the frontier is very small: think of the frontier in breadth-first search as the surface of an ever-expanding sphere, while the frontier in depth-first search is just a radius of the sphere.

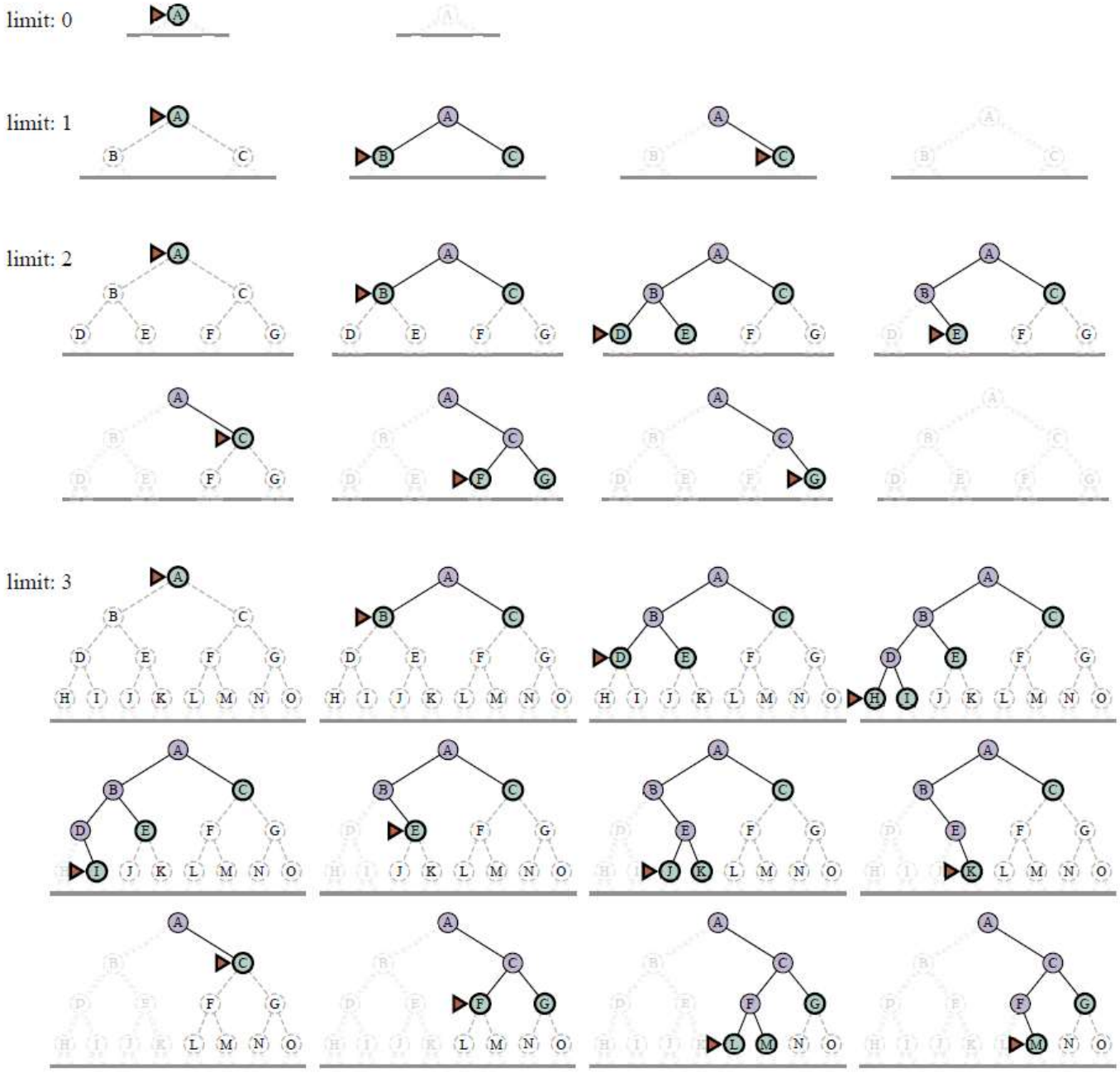
Depth-limited and iterative deepening search

To keep depth-first search from wandering down an infinite path, we can use Depth-limited search, a version of depth-first search in which we supply a depth limit, l, and treat all nodes at depth l as if they had no successors. The time complexity is $O(b^l)$ and the space complexity is $O(bl)$. Unfortunately, if we make a poor choice for l the algorithm will fail to reach the solution, making it incomplete again.

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.

```
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
```



In general, iterative deepening is the preferred uninformed search method when the search state space is larger than can fit in memory and the depth of the solution is not known.

Bidirectional Search

Bidirectional search simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet.

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                    // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                    // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F, frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

We pass in two versions of the problem and the evaluation function, one in the forward direction (subscript F) and one in the backward direction (subscript B).

Bidirectional best-first search keeps two frontiers and two tables of reached states.

Although there are two separate frontiers, the node to be expanded next is always one with a minimum value of the evaluation function, across either frontier.

When a path in one frontier reaches a state that was also reached in the other half of the search, the two paths are joined (by the function JOIN-NODES) to form a solution.

The first solution we get is not guaranteed to be the best; the function TERMINATED determines when to stop looking for new solutions.

Comparing uninformed search algorithms

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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Where 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; C^*, the cost of the optimal solution, and ϵ, a lower bound on the cost of each action.

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 $\geq \epsilon > 0$;
- 3 cost-optimal if action costs are all identical;
- 4 if both directions are breadth-first or uniform-cost.

Informed (Heuristic) Search Strategies

Informed (Heuristic) Search Strategy

An informed search strategy uses domain-specific hints about the location of goals to find solutions more efficiently than an uninformed strategy.

The hints come in the form of a heuristic function, denoted h(n)

h(n) = estimated cost of the cheapest path from the state at node n to a goal state.

Greedy best first search

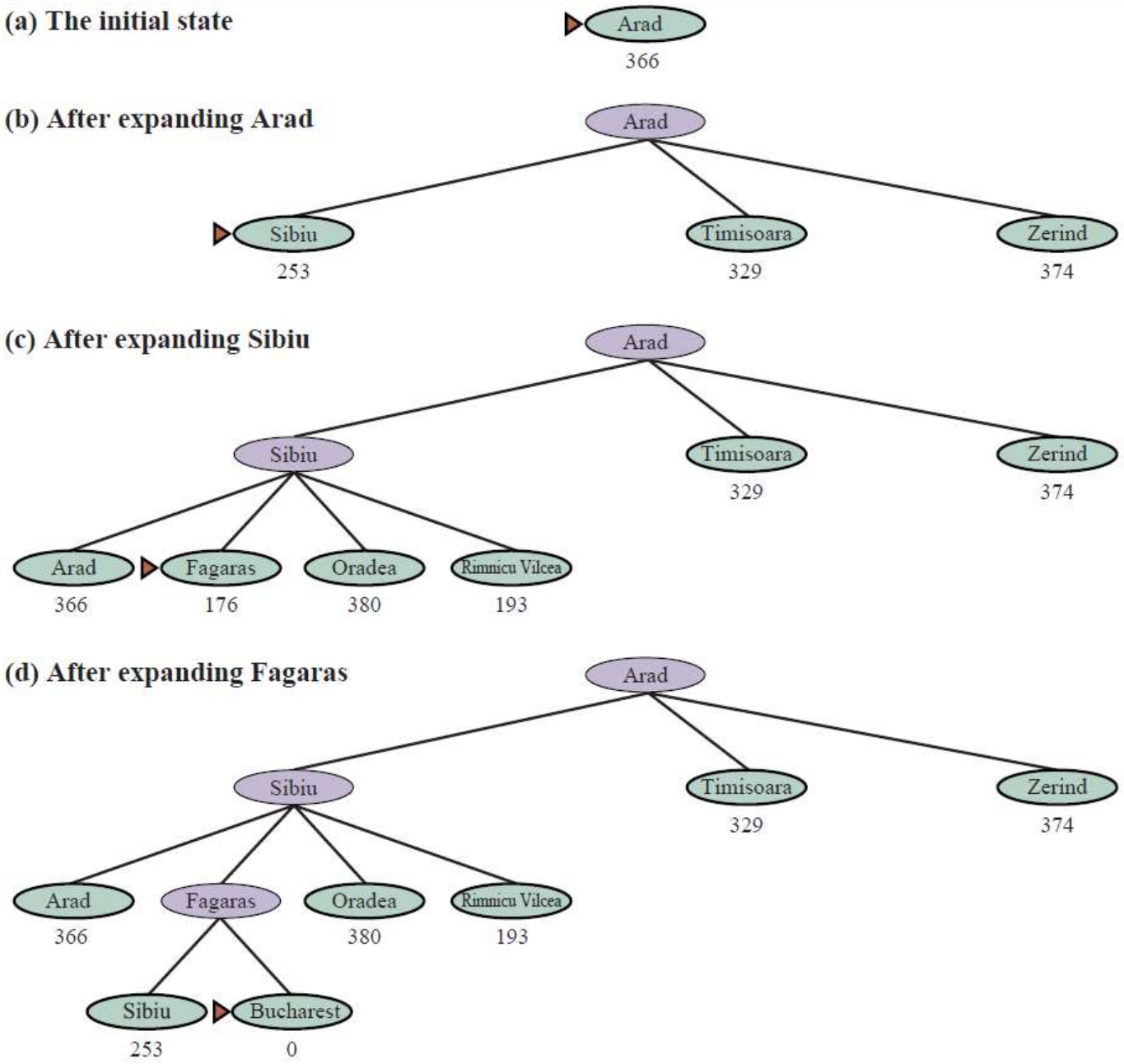
Greedy best-first search is a form of best-first search that expands first the node with the lowest h(n) value—the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly. So the evaluation function f(n) = h(n).

Let us see how this works for route-finding problems in Romania; we use the straightline distance heuristic, which we will call h_{SLD}. If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown below.

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

It takes a certain amount of world knowledge to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

The following figure shows the progress of a greedy best-first search using h_{SLD} to find a path from Arad to Bucharest.



Greedy best-first graph search is complete in finite state spaces, but not in infinite ones.

The worst-case time and space complexity is O(|V|). With a good heuristic function, however, the complexity can be reduced substantially, on certain problems reaching O(bm).

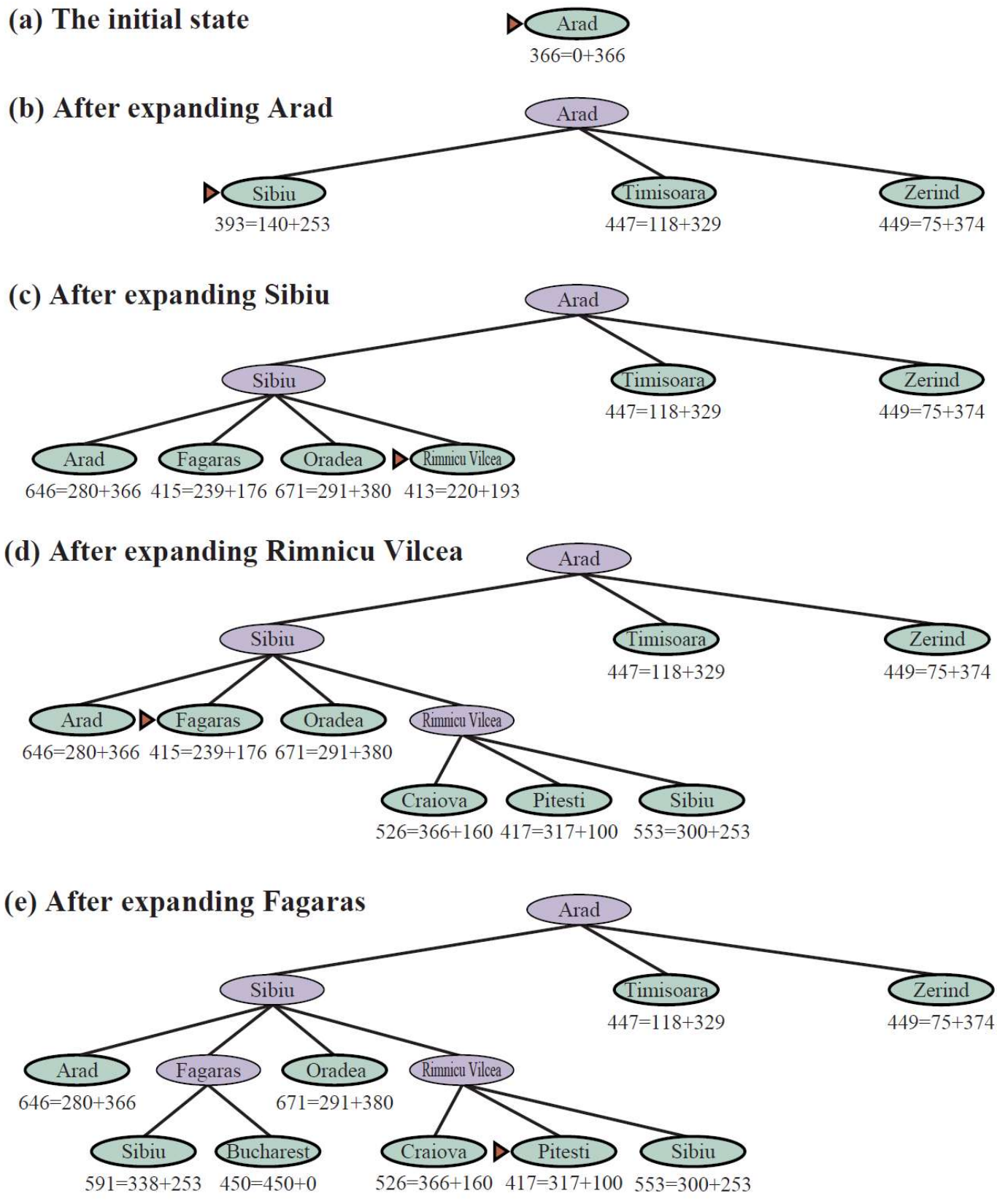
A^* Search

The most common informed search algorithm is A^* search (pronounced “A-star search”), a best-first search that uses the evaluation function

$$f(n) = g(n) + h(n)$$

where g(n) is the path cost from the initial state to node n, and h(n) is the estimated cost of the shortest path from n to a goal state.

In the following figure we show the progress of an A^* search with the goal of reaching Bucharest. The values of g are computed from the action costs



(f) After expanding Pitesti