CHAPTER 3

SOLVING PROBLEMS BY SEARCHING

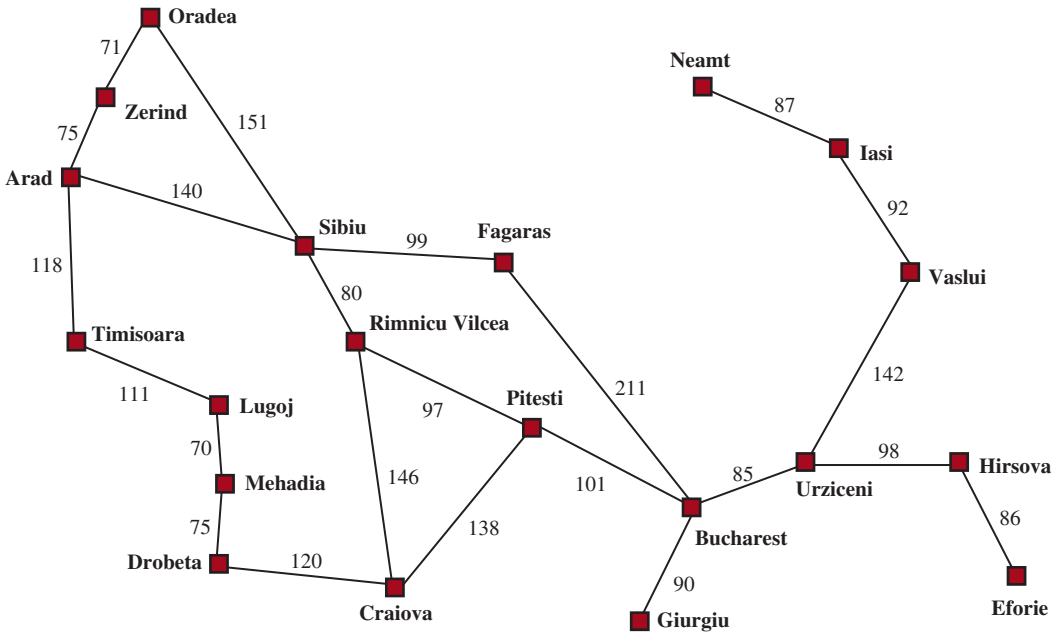


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

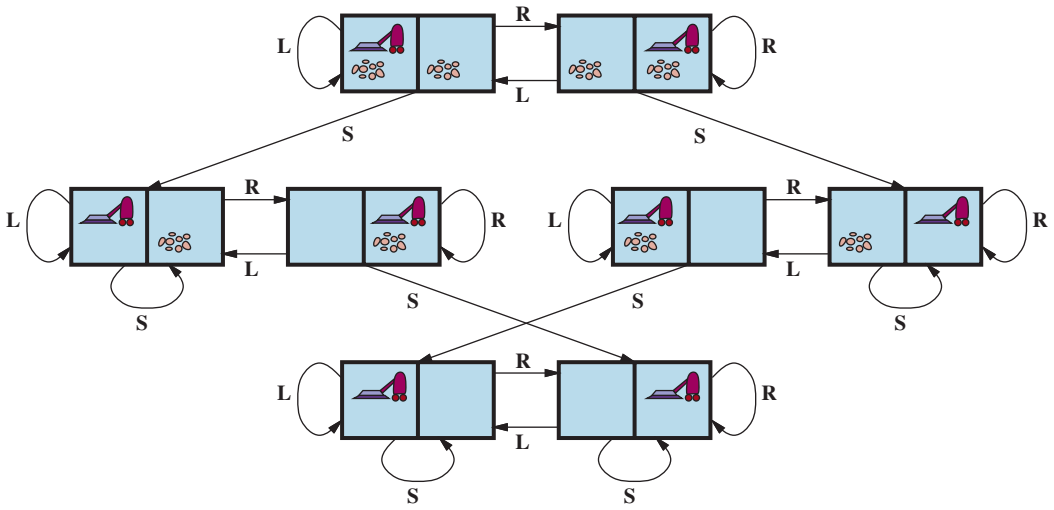


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

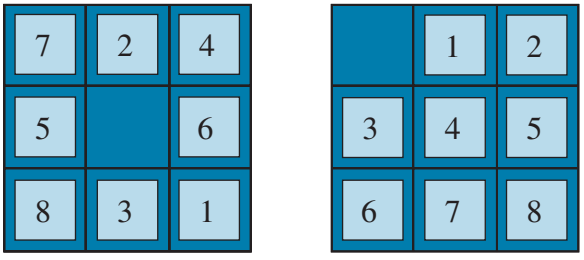


Figure 3.3 A typical instance of the 8-puzzle.

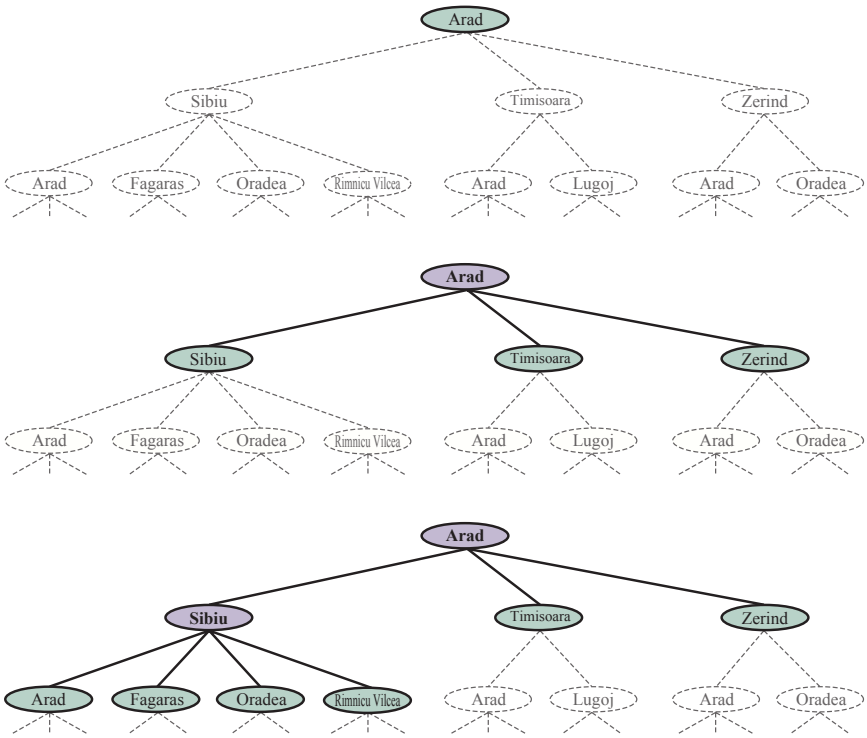


Figure 3.4 Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to 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.

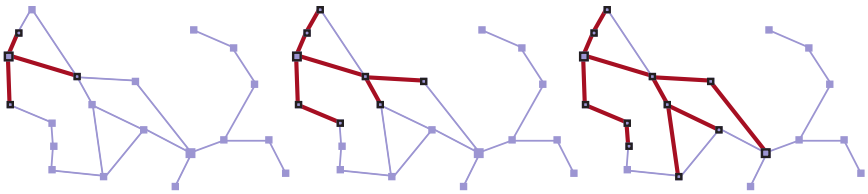
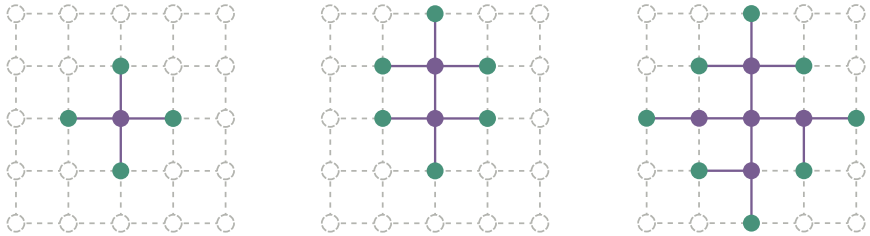


Figure 3.5 A sequence of search trees generated by a graph search on the Romania problem of Figure **??**. At each stage, we have expanded every node on the frontier, extending every path with all applicable actions that don't result in a state that has already been reached. Notice that at the third stage, the topmost city (Oradea) has two successors, both of which have already been reached by other paths, so no paths are extended from Oradea.



(a) (b) (c)

Figure 3.6 The separation property of graph search, illustrated on a rectangular-grid problem. The frontier (green) separates the interior (lavender) from the exterior (faint dashed). The frontier is the set of nodes (and corresponding states) that have been reached but not yet expanded; the interior is the set of nodes (and corresponding states) that have been expanded; and the exterior is the set of states that have not been reached. In (a), just the root has been expanded. In (b), the top frontier node is expanded. In (c), the remaining successors of the root are expanded in clockwise order.

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

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

Figure 3.7 The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section **??**. See Appendix B for **yield**.

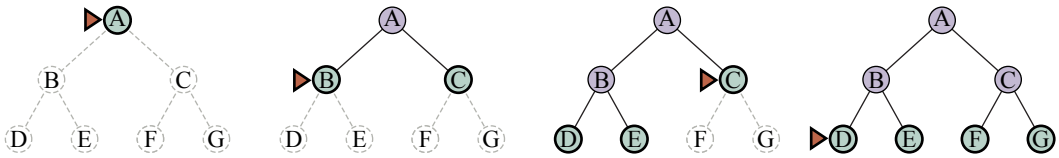


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

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*

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

Figure 3.9 Breadth-first search and uniform-cost search algorithms.

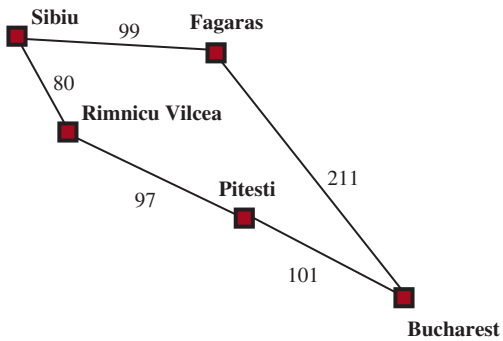


Figure 3.10 Part of the Romania state space, selected to illustrate uniform-cost search.

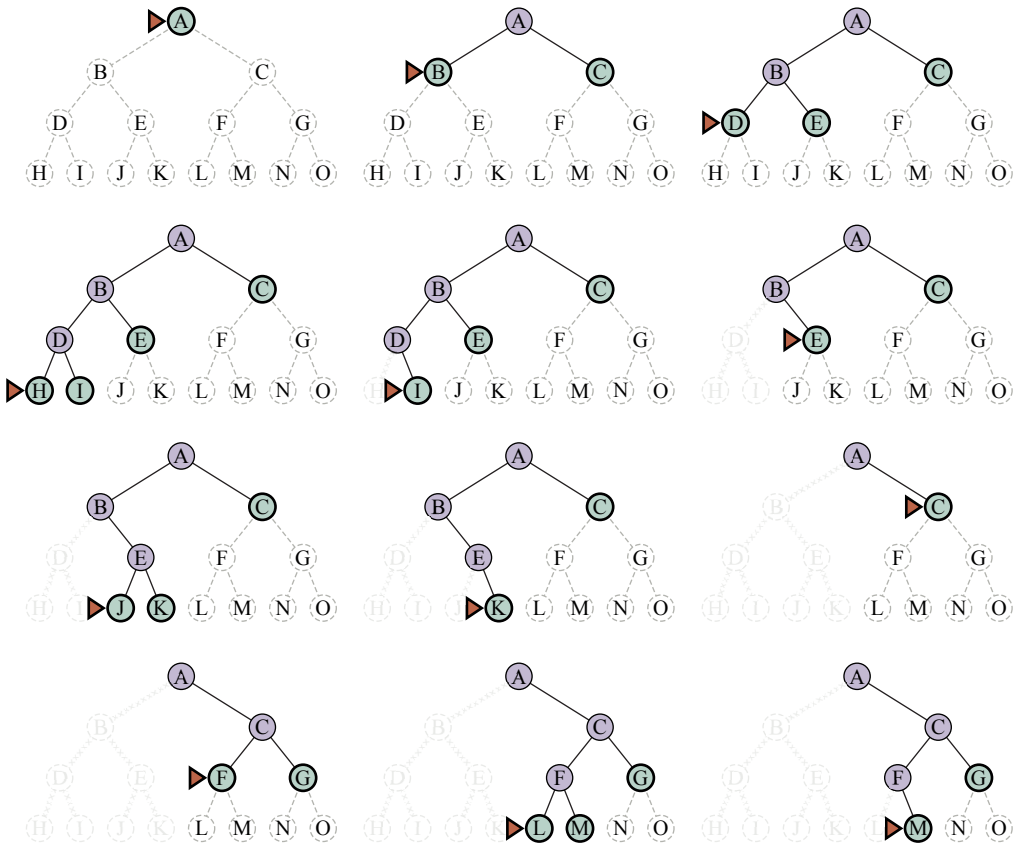


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

```
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
```

Figure 3.12 Iterative deepening and depth-limited tree-like search. Iterative deepening repeatedly applies depth-limited search with increasing limits. It returns one of three different types of values: either a solution node; or *failure*, when it has exhausted all nodes and proved there is no solution at any depth; or *cutoff*, to mean there might be a solution at a deeper depth than ℓ. This is a tree-like search algorithm that does not keep track of *reached* states, and thus uses much less memory than best-first search, but runs the risk of visiting the same state multiple times on different paths. Also, if the IS-CYCLE check does not check *all* cycles, then the algorithm may get caught in a loop.

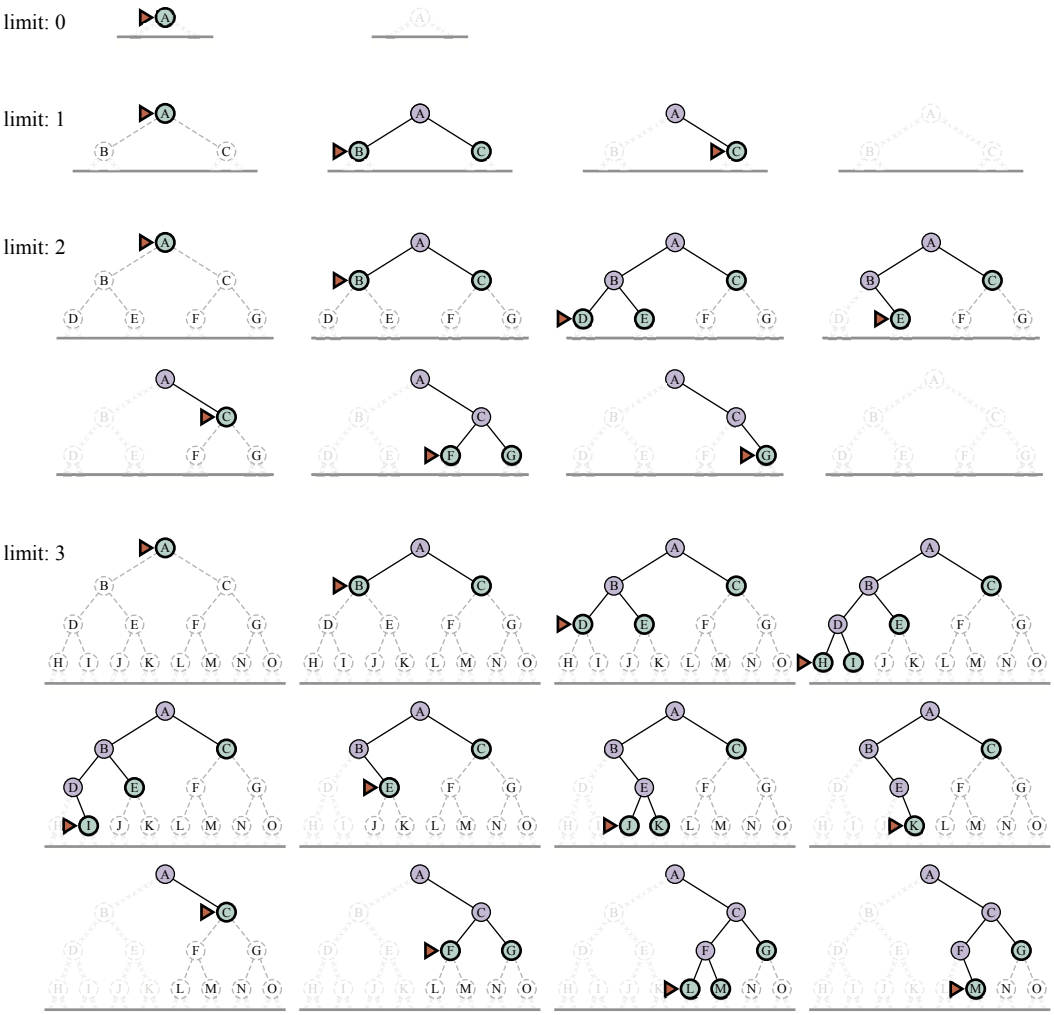


Figure 3.13 Four iterations of iterative deepening search for goal M on a binary tree, with the depth limit varying from 0 to 3. Note the interior nodes form a single path. The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can't be part of a solution with this depth limit.

```
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
```

Figure 3.14 Bidirectional best-first search keeps two frontiers and two tables of reached states. 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.

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

Figure 3.15 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 $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

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

Figure 3.16 Values of h_{SLD}—straight-line distances to Bucharest.

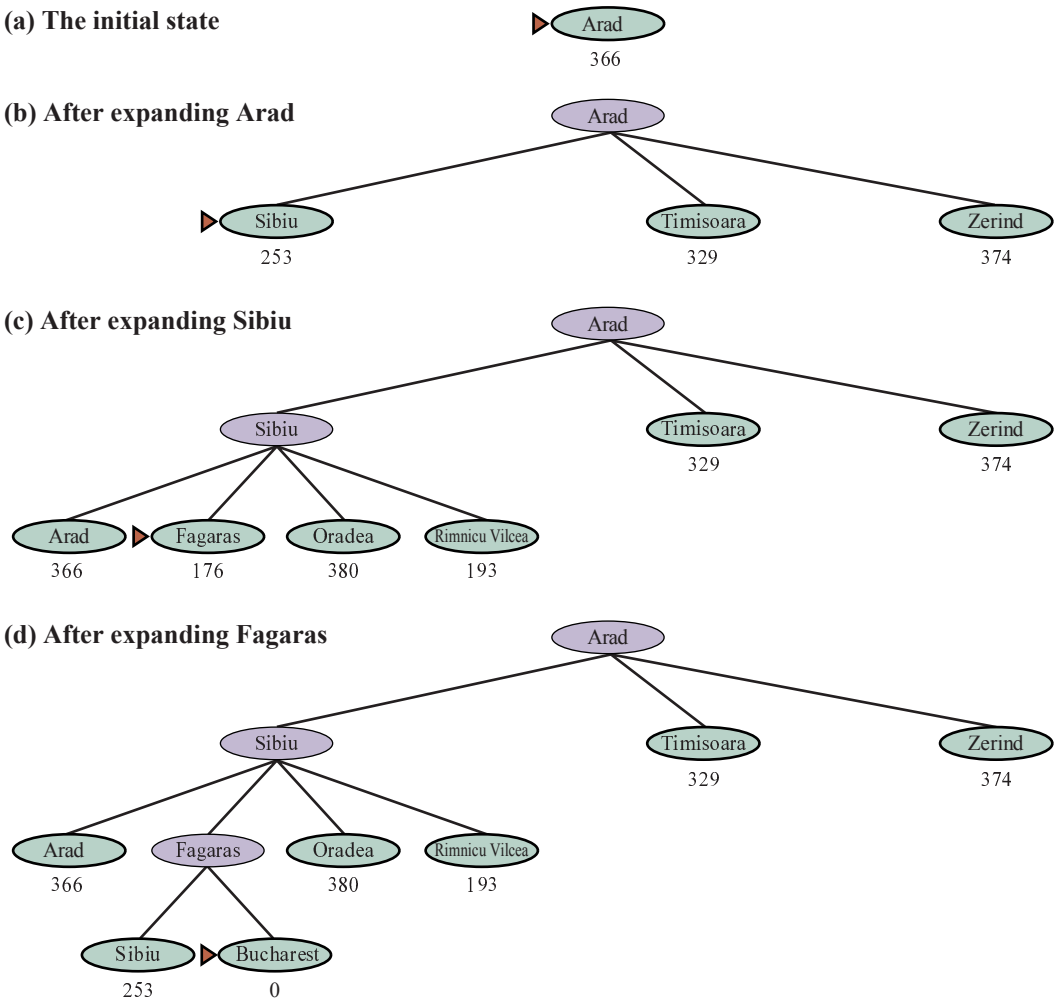


Figure 3.17 Stages in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

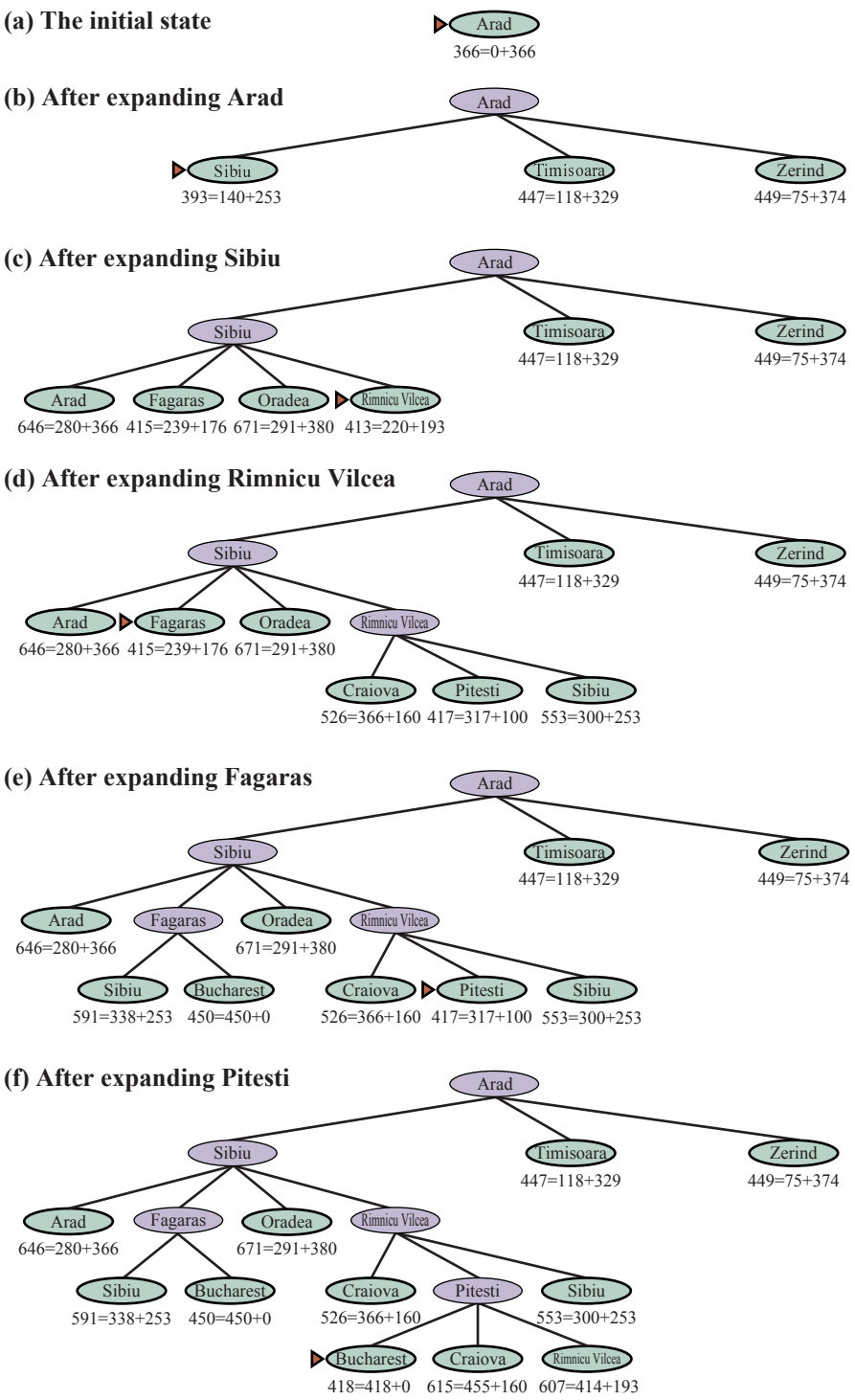


Figure 3.18 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure **??**.

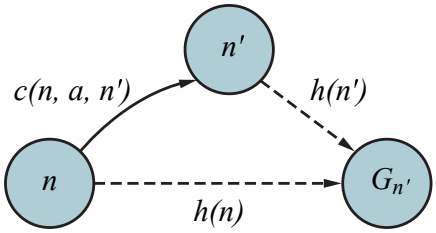


Figure 3.19 Triangle inequality: If the heuristic h is **consistent**, then the single number $h(n)$ will be less than the sum of the cost $c(n, a, a')$ of the action from n to n' plus the heuristic estimate $h(n')$.

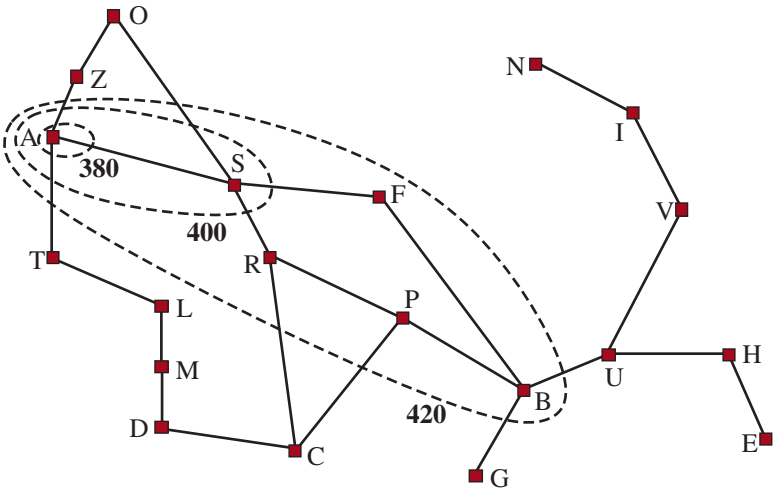
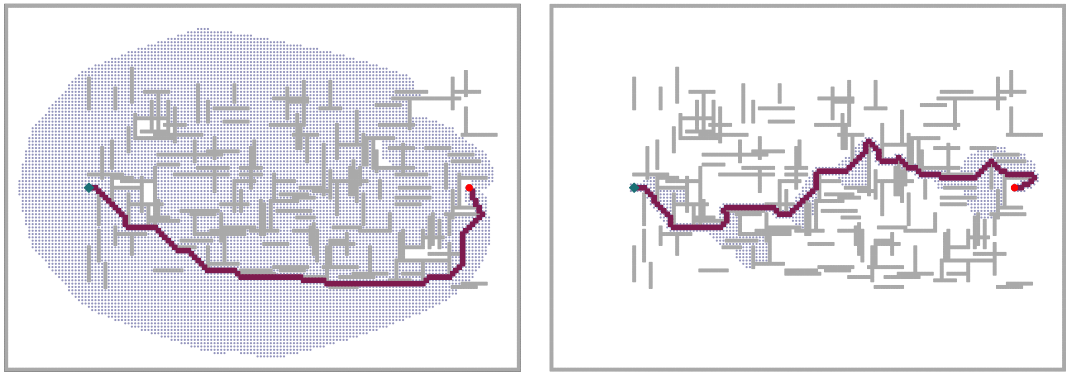


Figure 3.20 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have $f = g + h$ costs less than or equal to the contour value.



(a) (b)

Figure 3.21 Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight $W = 2$. The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly.

function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** a solution or *failure*
  solution, *fvalue* ← RBFS(*problem*, NODE(*problem*.INITIAL), ∞)
  return *solution*

function RBFS(*problem*, *node*, *f_limit*) **returns** a solution or *failure*, and a new f-cost limit
  if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
  successors ← LIST(EXPAND(*node*))
  if *successors* is empty **then return** *failure*, ∞
  for each *s* **in** *successors* **do** // *update f with value from previous search*
    $s.f \leftarrow \max(s.\text{PATH-COST} + h(s), node.f)$
  while *true* **do**
    best ← the node in *successors* with lowest f-value
    if $best.f > f\_limit$ **then return** *failure*, $best.f$
    alternative ← the second-lowest f-value among *successors*
    result, $best.f$ ← RBFS(*problem*, *best*, $\min(f\_limit, alternative)$)
    if $result \neq failure$ **then return** *result*, $best.f$

Figure 3.22 The algorithm for recursive best-first search.

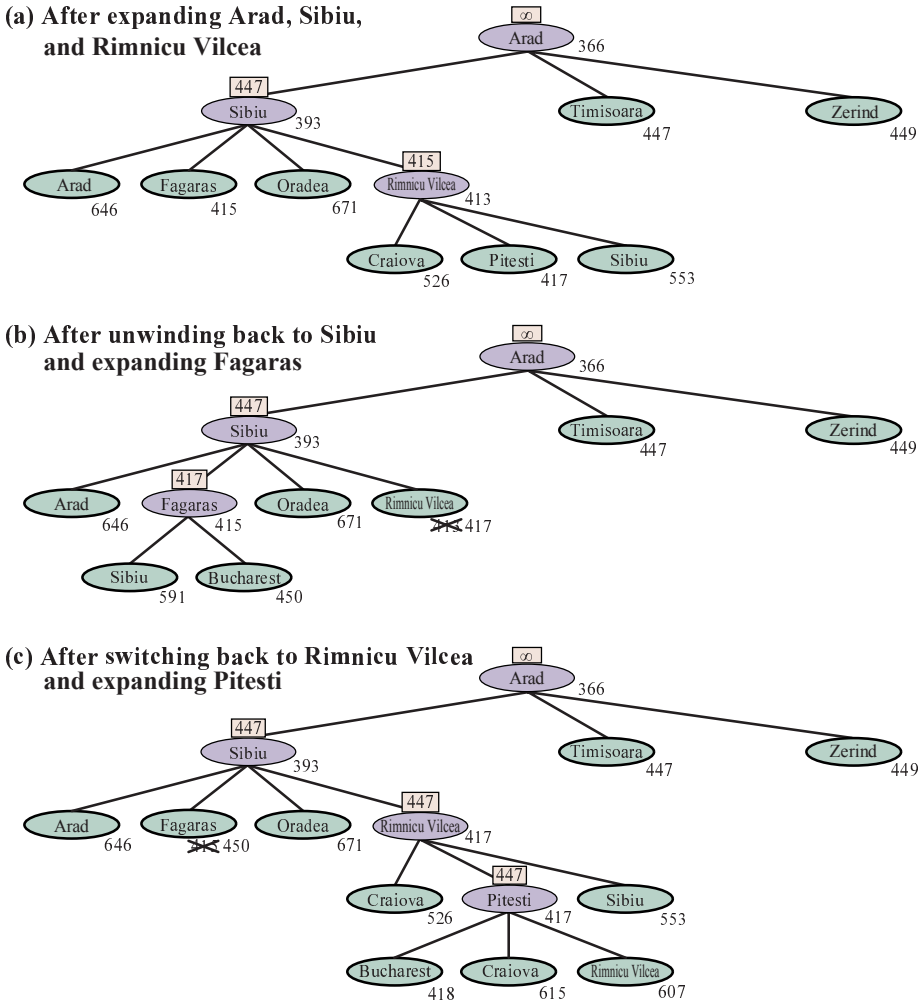


Figure 3.23 Stages in an RBFS search for the shortest route to Bucharest. The *f-limit* value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

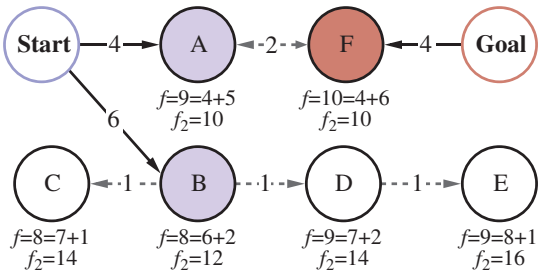


Figure 3.24 Bidirectional search maintains two frontiers: on the left, nodes A and B are successors of Start; on the right, node F is an inverse successor of Goal. Each node is labeled with $f = g + h$ values and the $f_2 = \max(2g, g + h)$ value. (The g values are the sum of the action costs as shown on each arrow; the h values are arbitrary and cannot be derived from anything in the figure.) The optimal solution, Start-A-F-Goal, has cost $C^* = 4 + 2 + 4 = 10$, so that means that a meet-in-the-middle bidirectional algorithm should not expand any node with $g > \frac{C^*}{2} = 5$; and indeed the next node to be expanded would be A or F (each with $g = 4$), leading us to an optimal solution. If we expanded the node with lowest f cost first, then B and C would come next, and D and E would be tied with A, but they all have $g > \frac{C^*}{2}$ and thus are never expanded when f_2 is the evaluation function.

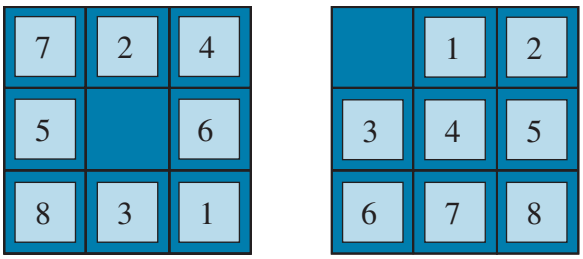


Figure 3.25 A typical instance of the 8-puzzle. The shortest solution is 26 actions long.

	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	A*(h_1)	A*(h_2)	BFS	A*(h_1)	A*(h_2)
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Figure 3.26 Comparison of the search costs and effective branching factors for 8-puzzle problems using breadth-first search, A* with h_1 (misplaced tiles), and A* with h_2 (Manhattan distance). Data are averaged over 100 puzzles for each solution length d from 6 to 28.

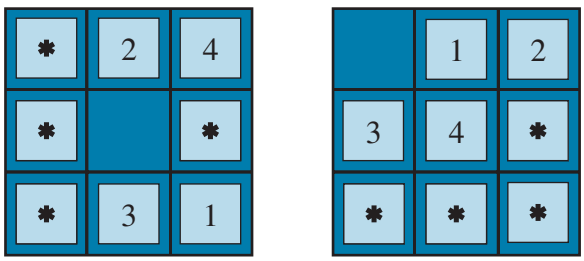


Figure 3.27 A subproblem of the 8-puzzle instance given in Figure **??**. The task is to get tiles 1, 2, 3, 4, and the blank into their correct positions, without worrying about what happens to the other tiles.

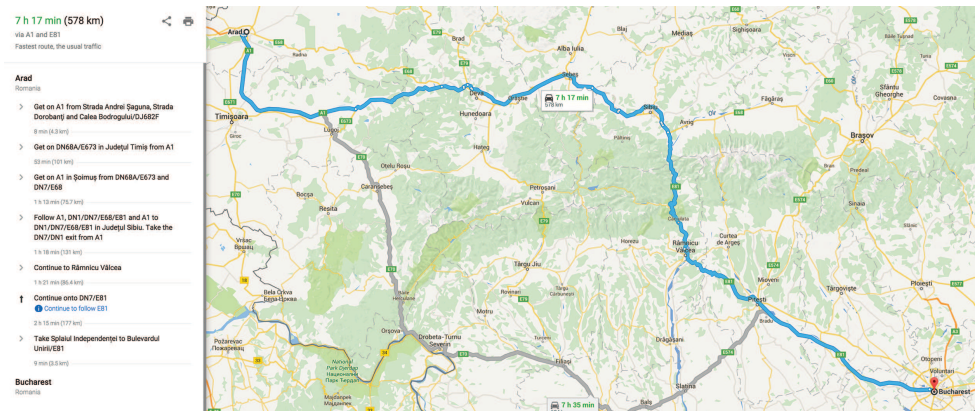


Figure 3.28 A Web service providing driving directions, computed by a search algorithm.