# EXERCISES 3

# SOLVING PROBLEMS BY SEARCHING

## 3.1 Problem-Solving Agents

**Exercise 3.**FORM

Explain why problem formulation must follow goal formulation.

In goal formulation, we decide which aspects of the world we are interested in, and which can be ignored or abstracted away. Then in problem formulation we decide how to manipulate the important aspects (and ignore the others). If we did problem formulation first we would not know what to include and what to leave out. That said, it can happen that there is a cycle of iterations between goal formulation, problem formulation, and problem solving until one arrives at a sufficiently useful and efficient solution.

## 3.2 Example Problems

**Exercise 3.**PRFO

Give a complete problem formulation for each of the following problems. Choose a formulation that is precise enough to be implemented.

**a**. There are six glass boxes in a row, each with a lock. Each of the first five boxes holds a key unlocking the next box in line; the last box holds a banana. You have the key to the first box, and you want the banana.

**b**. You start with the sequence ABABAECCEC, or in general any sequence made from A, B, C, and E. You can transform this sequence using the following equalities: AC = E, AB = BC, BB = E, and E$x$ = $x$ for any $x$. For example, ABBC can be transformed into AEC (using BB = E), and then AC (using E$x$ = $x$), and then E (using AC = E). Your goal is to produce the sequence E.

**c**. There is an $n \times n$ grid of squares, each square initially being either unpainted floor or a bottomless pit. You start standing on an unpainted floor square, and can either paint the square under you or move onto an adjacent unpainted floor square. You want the whole floor painted.

**d**. A container ship is in port, loaded high with containers. There 13 rows of containers, each 13 containers wide and 5 containers tall. You control a crane that can move to any

location above the ship, pick up the container under it, and move it onto the dock. You want the ship unloaded.

**a**. Initial state: as described in the question.
Goal test: you have banana.
Successor function: open any box you have the key for, get the contents of any open box.
Cost function: number of actions.

**b**. Initial state: ABABAECCEC.
Goal test: is the state E
Successor function: apply an equality substituting one subsequence for the other.
Cost function: number of transformations.

**c**. Initial state: all floor squares unpainted, you start standing on one square unpainted floor square.
Goal test: all floor squares painted.
Successor function: paint current tile, move to adjacent unpainted floor tile.
Cost function: number of moves.

**d**. Initial state: all containers stacked on the ship.
Goal test: all containers unloaded.
Successor function: move crane to a certain location, pick up a container, put down container.
Cost function: time taken to unload ship.

**Exercise 3.**RMAZ

Your goal is to navigate a robot out of a maze. The robot starts in the center of the maze facing north. You can turn the robot to face north, east, south, or west. You can direct the robot to move forward a certain distance, although it will stop before hitting a wall.

**a**. Formulate this problem. How large is the state space?

**b**. In navigating a maze, the only place we need to turn is at the intersection of two or more corridors. Reformulate this problem using this observation. How large is the state space now?

**c**. From each point in the maze, we can move in any of the four directions until we reach a turning point, and this is the only action we need to do. Reformulate the problem using these actions. Do we need to keep track of the robot's orientation now?

**d**. In our initial description of the problem we already abstracted from the real world, restricting actions and removing details. List three such simplifications we made.

a. We'll define the coordinate system so that the center of the maze is at $(0, 0)$, and the maze itself is a square from $(-1, -1)$ to $(1, 1)$.

Initial state: robot at coordinate $(0, 0)$, facing North.
Goal test: either $|x| > 1$ or $|y| > 1$ where $(x, y)$ is the current location.
Successor function: move forwards any distance $d$; change direction robot it facing.
Cost function: total distance moved.

The state space is infinitely large, since the robot's position is continuous.

b. The state will record the intersection the robot is currently at, along with the direction it's facing. At the end of each corridor leaving the maze we will have an exit node. We'll assume some node corresponds to the center of the maze.

Initial state: at the center of the maze facing North.
Goal test: at an exit node.
Successor function: move to the next intersection in front of us, if there is one; turn to face a new direction.
Cost function: total distance moved.

There are $4n$ states, where $n$ is the number of intersections.

c. Initial state: at the center of the maze.
Goal test: at an exit node.
Successor function: move to next intersection to the North, South, East, or West.
Cost function: total distance moved.

We no longer need to keep track of the robot's orientation since it is irrelevant to predicting the outcome of our actions, and not part of the goal test. The motor system that executes this plan will need to keep track of the robot's current orientation, to know when to rotate the robot.

d. State abstractions:

(i) Ignoring the height of the robot off the ground, whether it is tilted off the vertical.
(ii) The robot can face in only four directions.
(iii) Other parts of the world ignored: possibility of other robots in the maze, the weather in the Caribbean.

Action abstractions:

(i) We assumed all positions we safely accessible: the robot couldn't get stuck or damaged.
(ii) The robot can move as far as it wants, without having to recharge its batteries.
(iii) Simplified movement system: moving forwards a certain distance, rather than controlled each individual motor and watching the sensors to detect collisions.

## 3.3 Search Algorithms

## 3.4 Uninformed Search Strategies

## 3.5 Informed (Heuristic) Search Strategies

## 3.6 Heuristic Functions

**Exercise 3.**GRRB

You have a $9 \times 9$ grid of squares, each of which can be colored red or blue. The grid is initially colored all blue, but you can change the color of any square any number of times. Imagining the grid divided into nine $3 \times 3$ sub-squares, you want each sub-square to be all one color but neighboring sub-squares to be different colors.

**a**. Formulate this problem in the straightforward way. Compute the size of the state space.

**b**. You need color a square only once. Reformulate, and compute the size of the state space. Would breadth-first graph search perform faster on this problem than on the one in (a)? How about iterative deepening tree search?

**c**. Given the goal, we need consider only colorings where each sub-square is uniformly colored. Reformulate the problem and compute the size of the state space.

**d**. How many solutions does this problem have?

**e**. Parts (b) and (c) successively abstracted the original problem (a). Can you give a translation from solutions in problem (c) into solutions in problem (b), and from solutions in problem (b) into solutions for problem (a)?

**a**. Initial state: all squares colored blue.
Goal test: each sub-square colored the same, and adjacent subsquares colored different.
Successor function: color a square red or blue.
Cost function: number of times squares are colored.

There are $2^{81}$ states.

**b**. To avoid recoloring squares we could either record squares that have been colored, and not allow them to be colored again, or color all the squares in a fixed order. We describe the latter:

Initial state: all squares colored blue, the top-left square is next to be colored.
Goal test: each sub-square colored the same, and adjacent subsquares colored different.
Successor function: if we haven't colored all squares yet, color the current square red or blue (this automatically updates the next square to be colored).
Cost function: number of squares colored.

There are $82 \cdot 2^{81}$ states, since we need to record whether there are squares left to be colored, and if so which square is next.

**c**. Initial state: all sub-squares colored blue, the top-left sub-square is next to be colored.
Goal test: adjacent sub-squares colored differently.
Successor function: if we haven't colored all sub-squares yet, color the current sub-square red or blue.
Cost function: number of sub-squares colored.

The state space has $10 \cdot 2^9$ nodes.

**d**. This problem has 2 solutions: as soon as you choose the color of any square, the color of any other square is determined.

**e**. Given a solution for problem (c) first determine final coloring of squares, after all actions are taken. Then color each of the squares the required color in order.

Given a solution for problem (b) first determine final coloring of squares. We know that each sub-square will be colored a single color, so we can color them in order to reach the same goal state.

**Exercise 3.**ROMF

Suppose two friends live in different cities on a map, such as the Romania map shown in Figure 3.1. On every turn, we can simultaneously move each friend to a neighboring city on the map. The amount of time needed to move from city $i$ to neighbor $j$ is equal to the road distance $d(i, j)$ between the cities, but on each turn the friend that arrives first must wait until the other one arrives (and calls the first on his/her cell phone) before the next turn can begin. We want the two friends to meet as quickly as possible.

**a**. Write a detailed formulation for this search problem. (You will find it helpful to define some formal notation here.)

**b**. Let $D(i, j)$ be the straight-line distance between cities $i$ and $j$. Which of the following heuristic functions are admissible? (i) $D(i, j)$; (ii) $2 \cdot D(i, j)$; (iii) $D(i, j)/2$.

**c**. Are there completely connected maps for which no solution exists?

**d**. Are there maps in which all solutions require one friend to visit the same city twice?

**a**. State space: States are all possible city pairs $(i, j)$. The map is *not* the state space.

Successor function: The successors of $(i, j)$ are all pairs $(x, y)$ such that $Adjacent(x, i)$ and $Adjacent(y, j)$.

Goal: Be at $(i, i)$ for some $i$.

Step cost function: The cost to go from $(i, j)$ to $(x, y)$ is $max(d(i, x), d(j, y))$.

**b**. In the best case, the friends head straight for each other in steps of equal size, reducing their separation by twice the time cost on each step. Hence (iii) is admissible.

**c**. Yes: e.g., a map with two nodes connected by one link. The two friends will swap places forever. The same will happen on any chain if they start an odd number of steps apart. (One can see this best on the graph that represents the state space, which has two disjoint sets of nodes.) The same even holds for a grid of any size or shape, because every move changes the Manhattan distance between the two friends by 0 or 2.

**d**. Yes: take any of the unsolvable maps from part (c) and add a self-loop to any one of the nodes. If the friends start an odd number of steps apart, a move in which one of the friends takes the self-loop changes the distance by 1, rendering the problem solvable. If the self-loop is not taken, the argument from (c) applies and no solution is possible.

**Exercise 3.**PART

Show that the 8-puzzle states are divided into two disjoint sets, such that any state is reachable from any other state in the same set, while no state is reachable from any state in the other set. (*Hint:* See Berlekamp *et al.*, (1982).) Devise a procedure to decide which set a given state is in, and explain why this is useful for generating random states.

From `http://www.cut-the-knot.com/pythagoras/fifteen.shtml`, this proof applies to the fifteen puzzle, but the same argument works for the eight puzzle:

**Definition**: The goal state has the numbers in a certain order, which we will measure as starting at the upper left corner, then proceeding left to right, and when we reach the end of a row, going down to the leftmost square in the row below. For any other configuration besides the goal, whenever a tile with a greater number on it precedes a tile with a smaller number, the two tiles are said to be **inverted**.

**Proposition**: For a given puzzle configuration, let $N$ denote the sum of the total number of inversions and the row number of the empty square. Then $(N \mod 2)$ is invariant under any legal move. In other words, after a legal move an odd $N$ remains odd whereas an even $N$ remains even. Therefore the goal state in Figure 3.3, with no inversions and empty square in the first row, has $N = 1$, and can only be reached from starting states with odd $N$, not from starting states with even $N$.

**Proof**: First of all, sliding a tile horizontally changes neither the total number of inversions nor the row number of the empty square. Therefore let us consider sliding a tile vertically.

Let's assume, for example, that the tile $A$ is located directly over the empty square. Sliding it down changes the parity of the row number of the empty square. Now consider the total number of inversions. The move only affects relative positions of tiles $A$, $B$, $C$, and $D$. If none of the $B$, $C$, $D$ caused an inversion relative to $A$ (i.e., all three are larger than $A$) then after sliding one gets three (an odd number) of additional inversions. If one of the three is smaller than $A$, then before the move $B$, $C$, and $D$ contributed a single inversion (relative to $A$) whereas after the move they'll be contributing two inversions - a change of 1, also an odd number. Two additional cases obviously lead to the same result. Thus the change in the sum $N$ is always even. This is precisely what we have set out to show.

So before we solve a puzzle, we should compute the $N$ value of the start and goal state and make sure they have the same parity, otherwise no solution is possible.

**Exercise 3.**NQUE

Consider the $n$-queens problem using an efficient incremental formulation where a state is represented as a an $n$-element vector of row numbers for queens (one for each column). Explain why the state space has at least $\sqrt[3]{n!}$ states and estimate the largest $n$ for which exhaustive exploration is feasible. (*Hint*: Derive a lower bound on the branching factor by considering the maximum number of squares that a queen can attack in any column.)

The formulation puts one queen per column, with a new queen placed only in a square

that is not attacked by any other queen. To simplify matters, we'll first consider the $n$–rooks problem. The first rook can be placed in any square in column 1 ($n$ choices), the second in any square in column 2 except the same row that as the rook in column 1 ($n-1$ choices), and so on. This gives $n!$ elements of the search space.

For $n$ queens, notice that a queen attacks at most three squares in any given column, so in column 2 there are at least $(n-3)$ choices, in column at least $(n-6)$ choices, and so on. Thus the state space size $S \geq n \cdot (n-3) \cdot (n-6) \cdots$. Hence we have

$$\begin{aligned} S^3 &\geq n \cdot n \cdot n \cdot (n-3) \cdot (n-3) \cdot (n-3) \cdot (n-6) \cdot (n-6) \cdot (n-6) \cdot \cdots \\ &\geq n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot (n-5) \cdot (n-6) \cdot (n-7) \cdot (n-8) \cdot \cdots \\ &= n! \end{aligned}$$

or $S \geq \sqrt[3]{n!}$.

**Exercise 3.**COMP

Give a complete problem formulation for each of the following. Choose a formulation that is precise enough to be implemented.

**a**. Using only four colors, you have to color a planar map in such a way that no two adjacent regions have the same color.

**b**. A 3-foot-tall monkey is in a room where some bananas are suspended from the 8-foot ceiling. The monkey would like to get the bananas. The room contains two stackable, movable, climbable 3-foot-high crates.

**c**. You have a program that outputs the message "illegal input record" when fed a certain file of input records. You know that processing of each record is independent of the other records. You want to discover what record is illegal.

**d**. You have three jugs, measuring 12 gallons, 8 gallons, and 3 gallons, and a water faucet. You can fill the jugs up or empty them out from one to another or onto the ground. You need to measure out exactly one gallon.

**a**. Initial state: No regions colored.
Goal test: All regions colored, and no two adjacent regions have the same color.
Successor function: Assign a color to a region.
Cost function: Number of assignments.

**b**. Initial state: As described in the text.
Goal test: Monkey has bananas.
Successor function: Hop on crate; Hop off crate; Push crate from one spot to another; Walk from one spot to another; grab bananas (if standing on crate).
Cost function: Number of actions.

**c**. Initial state: considering all input records.
Goal test: considering a single record, and it gives "illegal input" message.
Successor function: run again on the first half of the records; run again on the second half of the records.
Cost function: Number of runs.

Note: This is a **contingency problem**; you need to see whether a run gives an error message or not to decide what to do next.

**d**. Initial state: jugs have values $[0, 0, 0]$.
Successor function: given values $[x, y, z]$, generate $[12, y, z]$, $[x, 8, z]$, $[x, y, 3]$ (by filling); $[0, y, z]$, $[x, 0, z]$, $[x, y, 0]$ (by emptying); or for any two jugs with current values $x$ and $y$, pour $y$ into $x$; this changes the jug with $x$ to the minimum of $x + y$ and the capacity of the jug, and decrements the jug with $y$ by by the amount gained by the first jug.
Cost function: Number of actions.

**Exercise 3.**PPPO

Consider the problem of finding the shortest path between two points on a plane that has convex polygonal obstacles as shown in Figure **??**. This is an idealization of the problem that a robot has to solve to navigate in a crowded environment.

**a**. Suppose the state space consists of all positions $(x, y)$ in the plane. How many states are there? How many paths are there to the goal?

**b**. Explain briefly why the shortest path from one polygon vertex to any other in the scene must consist of straight-line segments joining some of the vertices of the polygons. Define a good state space now. How large is this state space?

**c**. Define the necessary functions to implement the search problem, including an ACTIONS function that takes a vertex as input and returns a set of vectors, each of which maps the current vertex to one of the vertices that can be reached in a straight line. (Do not forget the neighbors on the same polygon.) Use the straight-line distance for the heuristic function.

**d**. Apply one or more of the algorithms in this chapter to solve a range of problems in the domain, and comment on their performance.

**a**. If we consider all $(x, y)$ points, then there are an infinite number of states, and of paths.

**b**. (For this problem, we consider the start and goal points to be vertices.) The shortest distance between two points is a straight line, and if it is not possible to travel in a straight line because some obstacle is in the way, then the next shortest distance is a sequence of line segments, end-to-end, that deviate from the straight line by as little as possible. So the first segment of this sequence must go from the start point to a tangent point on an obstacle – any path that gave the obstacle a wider girth would be longer. Because the obstacles are polygonal, the tangent points must be at vertices of the obstacles, and hence the entire path must go from vertex to vertex. So now the state space is the set of vertices, of which there are 35 in Figure **??**.

**c**. Code not shown.

**d**. Implementations and analysis not shown.

**Exercise 3.**NNEG

We said in the chapter that we would not consider problems with negative path costs. In this exercise, we explore this decision in more depth.

**a**. Suppose that actions can have arbitrarily large negative costs; explain why this possibility would force any optimal algorithm to explore the entire state space.

**b**. Does it help if we insist that step costs must be greater than or equal to some negative constant $c$? Consider both trees and graphs.

**c**. Suppose that a set of actions forms a loop in the state space such that executing the set in some order results in no net change to the state. If all of these actions have negative cost, what does this imply about the optimal behavior for an agent in such an environment?

**d**. One can easily imagine actions with high negative cost, even in domains such as route finding. For example, some stretches of road might have such beautiful scenery as to far outweigh the normal costs in terms of time and fuel. Explain, in precise terms, within the context of state-space search, why humans do not drive around scenic loops indefinitely, and explain how to define the state space and actions for route finding so that artificial agents can also avoid looping.

**e**. Can you think of a real domain in which step costs are such as to cause looping?

**a**. Any path, no matter how bad it appears, might lead to an arbitrarily large reward (negative cost). Therefore, one would need to exhaust all possible paths to be sure of finding the best one.

**b**. Suppose the greatest possible reward is $c$. Then if we also know the maximum depth of the state space (e.g. when the state space is a tree), then any path with $d$ levels remaining can be improved by at most $cd$, so any paths worse than $cd$ less than the best path can be pruned. For state spaces with loops, this guarantee doesn't help, because it is possible to go around a loop any number of times, picking up $c$ reward each time.

**c**. The agent should plan to go around this loop forever (unless it can find another loop with even better reward).

**d**. The value of a scenic loop is lessened each time one revisits it; a novel scenic sight is a great reward, but seeing the same one for the tenth time in an hour is tedious, not rewarding. To accommodate this, we would have to expand the state space to include a memory—a state is now represented not just by the current location, but by a current location and a bag of already-visited locations. The reward for visiting a new location is now a (diminishing) function of the number of times it has been seen before.

**e**. Real domains with looping behavior include eating junk food and going to class.

**Exercise 3.**MICA

The **missionaries and cannibals** problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of mis-

sionaries in one place outnumbered by the cannibals in that place. This problem is famous in AI because it was the subject of the first paper that approached problem formulation from an analytical viewpoint (Amarel, 1968).

**a**. Formulate the problem precisely, making only those distinctions necessary to ensure a valid solution. Draw a diagram of the complete state space.

**b**. Implement and solve the problem optimally using an appropriate search algorithm. Is it a good idea to check for repeated states?

**c**. Why do you think people have a hard time solving this puzzle, given that the state space is so simple?

**a**. Here is one possible representation: A state is a six-tuple of integers listing the number of missionaries, cannibals, and boats on the first side, and then the second side of the river. The goal is a state with 3 missionaries and 3 cannibals on the second side. The cost function is one per action, and the successors of a state are all the states that move 1 or 2 people and 1 boat from one side to another.

**b**. The search space is small, so any optimal algorithm works. For an example, see the file `aima-lisp/search/domains/cannibals.lisp`. It suffices to eliminate moves that circle back to the state just visited. From all but the first and last states, there is only one other choice.

**c**. It is not obvious that almost all moves are either illegal or revert to the previous state. There is a feeling of a large branching factor, and no clear way to proceed.

**Exercise 3.**DEFS

Define in your own words the following terms: state, state space, search tree, search node, goal, action, transition model, and branching factor.

A **state** is a situation that an agent can find itself in. We distinguish two types of states: world states (the actual concrete situations in the real world) and representational states (the abstract descriptions of the real world that are used by the agent in deliberating about what to do).

A **state space** is a graph whose nodes are the set of all states, and whose links are actions that transform one state into another.

A **search tree** is a tree (a graph with no undirected loops) in which the root node is the start state and the set of children for each node consists of the states reachable by taking any action.

A **search node** is a node in the search tree.

A **goal** is a state that the agent is trying to reach.

An **action** is something that the agent can choose to do.

A **successor function** described the agent's options: given a state, it returns a set of (action, state) pairs, where each state is the state reachable by taking the action.

The **branching factor** in a search tree is the number of actions available to the agent.

**Exercise 3.**STAT

What's the difference between a world state, a state description, and a search node? Why is this distinction useful?

A world state is how reality is or could be. In one world state we're in Arad, in another we're in Bucharest. The world state also includes which street we're on, what's currently on the radio, and the price of tea in China. A state description is an agent's internal description of a world state. Examples are $In(Arad)$ and $In(Bucharest)$. These descriptions are necessarily approximate, recording only some aspect of the state.

We need to distinguish between world states and state descriptions because state description are lossy abstractions of the world state, because the agent could be mistaken about how the world is, because the agent might want to imagine things that aren't true but it could make true, and because the agent cares about the world not its internal representation of it.

Search nodes are generated during search, representing a state the search process knows how to reach. They contain additional information aside from the state description, such as the sequence of actions used to reach this state. This distinction is useful because we may generate different search nodes which have the same state, and because search nodes contain more information than a state representation.

**Exercise 3.**AABS

An action such as going to Sibiu really consists of a long sequence of finer-grained actions: turn on the car, release the brake, accelerate forward, etc. Having composite actions of this kind reduces the number of steps in a solution sequence, thereby reducing the search time. Suppose we take this to the logical extreme, by making super-composite actions out of every possible sequence of *Go* actions. Then every problem instance is solved by a single super-composite action, such as *Go(Sibiu) Go(Rimnicu Vilcea) Go(Pitesti) Go(Bucharest)*. Explain how search would work in this formulation. Is this a practical approach for speeding up problem solving?

The state space is a tree of depth one, with all states successors of the initial state. There is no distinction between depth-first search and breadth-first search on such a tree. If the sequence length is unbounded the root node will have infinitely many successors, so only algorithms which test for goal nodes as we generate successors can work.

What happens next depends on how the composite actions are sorted. If there is no particular ordering, then a random but systematic search of potential solutions occurs. If they are sorted by dictionary order, then this implements depth-first search. If they are sorted by length first, then dictionary ordering, this implements breadth-first search.

A significant disadvantage of collapsing the search space like this is if we discover that a plan starting with the action "unplug your battery" can't be a solution, there is no easy way to ignore all other composite actions that start with this action. This is a problem in particular for informed search algorithms.

Discarding sequence structure is not a particularly practical approach to search.

**Exercise 3.**FINS

Does a finite state space always lead to a finite search tree? How about a finite state space that is a tree? Can you be more precise about what types of state spaces always lead to finite search trees? (Adapted from Bender (1996).)

No, a finite state space does not always lead to a finite search tree. Consider a state space with two states, both of which have actions that lead to the other. This yields an infinite search tree, because we can go back and forth any number of times. However, if the state space is a finite tree, or in general, a finite DAG (directed acyclic graph), then there can be no loops, and the search tree is finite.

**Exercise 3.**GRAS

Prove that GRAPH-SEARCH satisfies the graph separation property illustrated in Figure 3.6. (*Hint*: Begin by showing that the property holds at the start, then show that if it holds before an iteration of the algorithm, it holds afterwards.) Describe a search algorithm that violates the property.

The graph separation property states that "every path from the initial state to an unexplored state has to pass through a state in the frontier."

At the start of the search, the frontier holds the initial state; hence, trivially, every path from the initial state to an unexplored state includes a node in the frontier (the initial state itself).

Now, we assume that the property holds at the beginning of an arbitrary iteration of the GRAPH-SEARCH algorithm in Figure **??**. We assume that the iteration completes, i.e., the frontier is not empty and the selected leaf node $n$ is not a goal state. At the end of the iteration, $n$ has been removed from the frontier and its successors (if not already explored or in the frontier) placed in the frontier. Consider any path from the initial state to an unexplored state; by the induction hypothesis such a path (at the beginning of the iteration) includes at least one frontier node; except when $n$ is the only such node, the separation property automatically holds. Hence, we focus on paths passing through $n$ (and no other frontier node). By definition, the next node $n'$ along the path from $n$ must be a successor of $n$ that (by the preceding sentence) is already not in the frontier. Furthermore, $n'$ cannot be in the explored set, since by assumption there is a path from $n'$ to an unexplored node not passing through the frontier, which would violate the separation property as every explored node is connected to the initial state by explored nodes (see lemma below for proof this is always possible). Hence, $n'$ is not in the explored set, hence it will be added to the frontier; then the path will include a frontier node and the separation property is restored.

The property is violated by algorithms that move nodes from the frontier into the explored set before all of their successors have been generated, as well as by those that fail to add some of the successors to the frontier. Note that it is not necessary to generate *all* successors of a node at once before expanding another node, as long as partially expanded nodes remain in the frontier.

Lemma: Every explored node is connected to the initial state by a path of explored nodes.

Proof: This is true initially, since the initial state is connected to itself. Since we never remove nodes from the explored region, we only need to check new nodes we add to the explored list on an expansion. Let $n$ be such a new explored node. This is previously on the frontier, so it is a neighbor of a node $n'$ previously explored (i.e., its parent). $n'$ is, by hypothesis is connected to the initial state by a path of explored nodes. This path with $n$ appended is a path of explored nodes connecting $n'$ to the initial state.

**Exercise 3.**TFSA

Which of the following are true and which are false? Explain your answers.

**a**. Depth-first search always expands at least as many nodes as A* search with an admissible heuristic.

**b**. $h(n) = 0$ is an admissible heuristic for the 8-puzzle.

**c**. A* is of no use in robotics because percepts, states, and actions are continuous.

**d**. Breadth-first search is complete even if zero step costs are allowed.

**e**. Assume that a rook can move on a chessboard any number of squares in a straight line, vertically or horizontally, but cannot jump over other pieces. Manhattan distance is an admissible heuristic for the problem of moving the rook from square A to square B in the smallest number of moves.

**a**. *False*: a lucky DFS might expand exactly $d$ nodes to reach the goal. A* largely dominates any graph-search algorithm that is *guaranteed to find optimal solutions*.

**b**. *True*: $h(n) = 0$ is always an admissible heuristic, since costs are nonnegative.

**c**. *True*: A* search is often used in robotics; the space can be discretized or skeletonized.

**d**. *True*: depth of the solution matters for breadth-first search, not cost.

**e**. *False*: a rook can move across the board in move one, although the Manhattan distance from start to finish is 8.

**Exercise 3.**KTWO

Consider a state space where the start state is number 1 and each state $k$ has two successors: numbers $2k$ and $2k + 1$.

**a**. Draw the portion of the state space for states 1 to 15.

**b**. Suppose the goal state is 11. List the order in which nodes will be visited for breadth-first search, depth-limited search with limit 3, and iterative deepening search.

**c**. How well would bidirectional search work on this problem? What is the branching factor in each direction of the bidirectional search?

**d**. Does the answer to (c) suggest a reformulation of the problem that would allow you to solve the problem of getting from state 1 to a given goal state with almost no search?

**e**. Call the action going from $k$ to $2k$ Left, and the action going to $2k + 1$ Right. Can you find an algorithm that outputs the solution to this problem without any search at all?

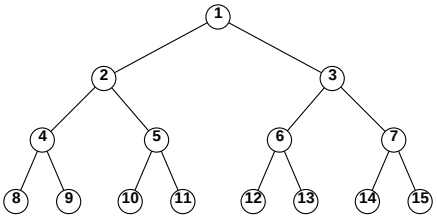


**Figure S3.1** The state space for the problem defined in Ex. 3.15.

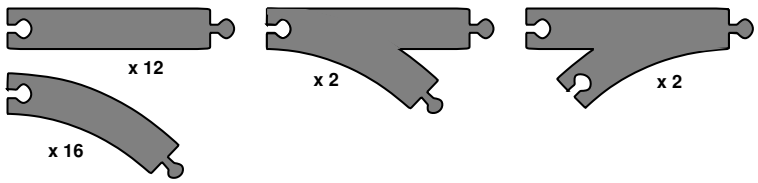


**Figure S3.2** Basic wooden railway pieces.

**a**. See Figure S3.1.

**b**. Breadth-first: 1 2 3 4 5 6 7 8 9 10 11
Depth-limited: 1 2 4 8 9 5 10 11
Iterative deepening: 1; 1 2 3; 1 2 4 5 3 6 7; 1 2 4 8 9 5 10 11

**c**. Bidirectional search is very useful, because the only successor of $n$ in the reverse direction is $\lfloor (n/2) \rfloor$. This helps focus the search. The branching factor is 2 in the forward direction; 1 in the reverse direction.

**d**. Yes; start at the goal, and apply the single reverse successor action until you reach 1.

**e**. The solution can be read off the binary numeral for the goal number. Write the goal number in binary. Since we can only reach positive integers, this binary expansion beings with a 1. From most- to least- significant bit, skipping the initial 1, go Left to the node $2n$ if this bit is 0 and go Right to node $2n + 1$ if it is 1. For example, suppose the goal is 11, which is 1011 in binary. The solution is therefore Left, Right, Right.

**Exercise 3.**BRIW

A basic wooden railway set contains the pieces shown in Figure S3.2. The task is to connect these pieces into a railway that has no overlapping tracks and no loose ends where a train could run off onto the floor.

**a**. Suppose that the pieces fit together *exactly* with no slack. Give a precise formulation of the task as a search problem.

**b**. Identify a suitable uninformed search algorithm for this task and explain your choice.

**c**. Explain why removing any one of the "fork" pieces makes the problem unsolvable.

**d**. Give an upper bound on the total size of the state space defined by your formulation. (*Hint*: think about the maximum branching factor for the construction process and the maximum depth, ignoring the problem of overlapping pieces and loose ends. Begin by pretending that every piece is unique.)

**a**. **Initial state**: one arbitrarily selected piece (say a straight piece).
**Successor function**: for any open peg, add any piece type from remaining types. (You can add to open holes as well, but that isn't necessary as all complete tracks can be made by adding to pegs.) For a curved piece, add *in either orientation*; for a fork, add *in either orientation* and (if there are two holes) connecting *at either hole*. It's a good idea to disallow any overlapping configuration, as this terminates hopeless configurations early. (Note: there is no need to consider open holes, because in any solution these will be filled by pieces added to open pegs.)
**Goal test**: all pieces used in a single connected track, no open pegs or holes, no overlapping tracks.
**Step cost**: one per piece (actually, doesn't really matter).

**b**. All solutions are at the same depth, so depth-first search would be appropriate. (One could also use depth-limited search with limit $n-1$, but strictly speaking it's not necessary to do the work of checking the limit because states at depth $n-1$ have no successors.) The space is very large, so uniform-cost and breadth-first would fail, and iterative deepening simply does unnecessary extra work. There are many repeated states, so it might be good to use a closed list.

**c**. A solution has no open pegs or holes, so every peg is in a hole, so there must be equal numbers of pegs and holes. Removing a fork violates this property. There are two other "proofs" that are acceptable: 1) a similar argument to the effect that there must be an even number of "ends"; 2) each fork creates two tracks, and only a fork can rejoin those tracks into one, so if a fork is missing it won't work. The argument using pegs and holes is actually more general, because it also applies to the case of a three-way fork that has one hole and three pegs or one peg and three holes. The "ends" argument fails here, as does the fork/rejoin argument (which is a bit handwavy anyway).

**d**. The maximum possible number of open pegs is 3 (starts at 1, adding a two-peg fork increases it by one). Pretending each piece is unique, any piece can be added to a peg, giving at most $12 + (2 \cdot 16) + (2 \cdot 2) + (2 \cdot 2 \cdot 2) = 56$ choices per peg. The total depth is 32 (there are 32 pieces), so an upper bound is $168^{32}/(12! \cdot 16! \cdot 2! \cdot 2!)$ where the factorials deal with permutations of identical pieces. One could do a more refined analysis to handle the fact that the branching factor shrinks as we go down the tree, but it is not pretty.

**Exercise 3.**RESE

Implement two versions of the RESULT$(s, a)$ function for the 8-puzzle: one that copies

and edits the data structure for the parent node $s$ and one that modifies the parent state directly (undoing the modifications as needed). Write versions of iterative deepening depth-first search that use these functions and compare their performance.

For the 8 puzzle, there shouldn't be much difference in performance. Indeed, the file `"aima-lisp/search/domains/puzzle8.lisp"` shows that you can represent an 8 puzzle state as a single 32-bit integer, so the question of modifying or copying data is moot. But for the $n \times n$ puzzle, as $n$ increases, the advantage of modifying rather than copying grows. The disadvantage of a modifying successor function is that it only works with depth-first search (or with a variant such as iterative deepening).

**Exercise 3.ITLE**

**Iterative lengthening search** is an iterative analog of uniform cost search. The idea is to use increasing limits on path cost. If a node is generated whose path cost exceeds the current limit, it is immediately discarded. For each new iteration, the limit is set to the lowest path cost of any node discarded in the previous iteration.

**a**. Show that this algorithm is optimal for general path costs.

**b**. Consider a uniform tree with branching factor $b$, solution depth $d$, and unit step costs. How many iterations will iterative lengthening require?

**c**. Now consider step costs drawn from the continuous range $[\epsilon, 1]$, where $0 < \epsilon < 1$. How many iterations are required in the worst case?

**d**. Implement the algorithm and apply it to instances of the 8-puzzle and traveling salesperson problems. Compare the algorithm's performance to that of uniform-cost search, and comment on your results.

**a.** The algorithm expands nodes in order of increasing path cost; therefore the first goal it encounters will be the goal with the cheapest cost.

**b.** It will be the same as iterative deepening, $d$ iterations, in which $O(b^d)$ nodes are generated.

**c.** $d/\epsilon$

**d.** Implementation not shown.

**Exercise 3.ITDW**

Describe a state space in which iterative deepening search performs much worse than depth-first search (for example, $O(n^2)$ vs. $O(n)$).

Consider a domain in which every state has a single successor, and there is a single goal at depth $n$. Then depth-first search will find the goal in $n$ steps, whereas iterative deepening search will take $1 + 2 + 3 + \cdots + n = O(n^2)$ steps.

**Exercise 3.**WWWL

Write a program that will take as input two web page URLs and find a path of links from one to the other. What is an appropriate search strategy? Is bidirectional search a good idea? Could a search engine be used to implement a predecessor function?

As an ordinary person (or agent) browsing the web, we can only generate the successors of a page by visiting it. We can then do breadth-first search, or perhaps best-search search where the heuristic is some function of the number of words in common between the start and goal pages; this may help keep the links on target. Search engines keep the complete graph of the web, and may provide the user access to all (or at least some) of the pages that link to a page; this would allow us to do bidirectional search.

**Exercise 3.**VACG

Consider the vacuum-world problem defined in Figure 2.2.

**a**. Which of the algorithms defined in this chapter would be appropriate for this problem? Should the algorithm use tree search or graph search?

**b**. Apply your chosen algorithm to compute an optimal sequence of actions for a $3 \times 3$ world whose initial state has dirt in the three top squares and the agent in the center.

**c**. Construct a search agent for the vacuum world, and evaluate its performance in a set of $3 \times 3$ worlds with probability 0.2 of dirt in each square. Include the search cost as well as path cost in the performance measure, using a reasonable exchange rate.

**d**. Compare your best search agent with a simple randomized reflex agent that sucks if there is dirt and otherwise moves randomly.

**e**. Consider what would happen if the world were enlarged to $n \times n$. How does the performance of the search agent and of the reflex agent vary with $n$?

Code not shown, but a good start is in the code repository. Clearly, graph search must be used—this is a classic grid world with many alternate paths to each state. Students will quickly find that computing the optimal solution sequence is prohibitively expensive for moderately large worlds, because the state space for an $n \times n$ world has $n^2 \cdot 2^n$ states. The completion time of the random agent grows less than exponentially in $n$, so for any reasonable exchange rate between search cost ad path cost the random agent will eventually win.

**Exercise 3.**SESC

Prove each of the following statements, or give a counterexample:

**a**. Breadth-first search is a special case of uniform-cost search.

**b**. Depth-first search is a special case of best-first tree search.

**c**. Uniform-cost search is a special case of A* search.

**a**. When all step costs are equal, $g(n) \propto depth(n)$, so uniform-cost search reproduces breadth-first search.

**b**. Breadth-first search is best-first search with $f(n) = depth(n)$; depth-first search is best-first search with $f(n) = -depth(n)$; uniform-cost search is best-first search with $f(n) = g(n)$.

**c**. Uniform-cost search is A* search with $h(n) = 0$.

**Exercise 3.**RBFS

Compare the performance of A* and RBFS on a set of randomly generated problems in the 8-puzzle (with Manhattan distance) and TSP (with MST—see Exercise 3.MSTR) domains. Discuss your results. What happens to the performance of RBFS when a small random number is added to the heuristic values in the 8-puzzle domain?

The student should find that on the 8-puzzle, RBFS expands more nodes (because it does not detect repeated states) but has lower cost per node because it does not need to maintain a queue. The number of RBFS node re-expansions is not too high because the presence of many tied values means that the best path changes seldom. When the heuristic is slightly perturbed, this advantage disappears and RBFS's performance is much worse.

For TSP, the state space is a tree, so repeated states are not an issue. On the other hand, the heuristic is real-valued and there are essentially no tied values, so RBFS incurs a heavy penalty for frequent re-expansions.

**Exercise 3.**BULU

Trace the operation of A* search applied to the problem of getting to Bucharest from Lugoj using the straight-line distance heuristic. That is, show the sequence of nodes that the algorithm will consider and the $f$, $g$, and $h$ score for each node.

The sequence of queues is as follows:

L[0+244=244]

M[70+241=311], T[111+329=440]

L[140+244=384], D[145+242=387], T[111+329=440]

D[145+242=387], T[111+329=440], M[210+241=451], T[251+329=580]

C[265+160=425], T[111+329=440], M[210+241=451], M[220+241=461], T[251+329=580]

T[111+329=440], M[210+241=451], M[220+241=461], P[403+100=503], T[251+329=580], R[411+193=604], D[385+242=627]

M[210+241=451], M[220+241=461], L[222+244=466], P[403+100=503], T[251+329=580], A[229+366=595], R[411+193=604], D[385+242=627]

M[220+241=461], L[222+244=466], P[403+100=503], L[280+244=524], D[285+242=527], T[251+329=580], A[229+366=595], R[411+193=604], D[385+242=627]

L[222+244=466], P[403+100=503], L[280+244=524], D[285+242=527], L[290+244=534], D[295+242=537], T[251+329=580], A[229+366=595], R[411+193=604], D[385+242=627]

P[403+100=503], L[280+244=524], D[285+242=527], M[292+241=533], L[290+244=534], D[295+242=537], T[251+329=580], A[229+366=595], R[411+193=604], D[385+242=627], T[333+329=662]

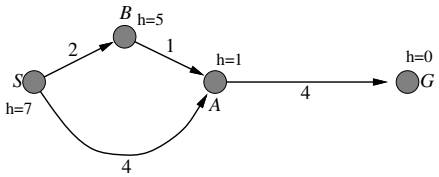


**Figure S3.3** A graph with an inconsistent heuristic on which GRAPH-SEARCH fails to return the optimal solution. The successors of $S$ are $A$ with $f = 5$ and $B$ with $f = 7$. $A$ is expanded first, so the path via $B$ will be discarded because $A$ will already be in the closed list.

B[504+0=504], L[280+244=524], D[285+242=527], M[292+241=533], L[290+244=534], D[295+242=537], T[251+329=580], A[229+366=595], R[411+193=604], D[385+242=627], T[333+329=662], R[500+193=693], C[541+160=701]

**Exercise 3.**EVCO

Sometimes there is no good evaluation function for a problem but there is a good comparison method: a way to tell whether one node is better than another without assigning numerical values to either. Show that this is enough to do a best-first search. Is there an analog of A* for this setting?

If we assume the comparison function is transitive, then we can always sort the nodes using it, and choose the node that is at the top of the sort. Efficient priority queue data structures rely only on comparison operations, so we lose nothing in efficiency—except for the fact that the comparison operation on states may be much more expensive than comparing two numbers, each of which can be computed just once.

A* relies on the division of the total cost estimate $f(n)$ into the cost-so-far and the cost-to-go. If we have comparison operators for each of these, then we can prefer to expand a node that is better than other nodes on both comparisons. Unfortunately, there will usually be no such node. The tradeoff between $g(n)$ and $h(n)$ cannot be realized without numerical values.

**Exercise 3.**AFAL

Devise a state space in which A* using GRAPH-SEARCH returns a suboptimal solution with an $h(n)$ function that is admissible but inconsistent.

See Figure S3.3.

**Exercise 3.**HEUR

Accurate heuristics don't necessarily reduce search time in the worst case. Given any depth $d$, define a search problem with a goal node at depth $d$, and write a heuristic function such that $|h(n) - h^*(n)| \leq O(\log h^*(n))$ but A* expands all nodes of depth less than $d$.

Let each state be a binary string, with the initial state the empty string, where the successors of a string $x$ are $\{x0, x1\}$ the string extended by one bit, and where each action has unit cost. Suppose all strings with length greater than $d$ are goals, along with the string of $d$ zeros. Define $h(x) = d - |x|$ if $x$ is of length at most $d$, and $h(x) = 0$ otherwise. Note that $|h(n) - h^*(n)| \leq 1$. Since A* will expand all nodes with $f$-value less than $d$, it will expand all nodes of depth less than $d$.

**Exercise 3.**HEPA

The **heuristic path algorithm** (Pohl, 1977) is a best-first search in which the evaluation function is $f(n) = (2 - w)g(n) + wh(n)$. For what values of $w$ is this complete? For what values is it optimal, assuming that $h$ is admissible? What kind of search does this perform for $w = 0$, $w = 1$, and $w = 2$?

It is complete whenever $0 \leq w < 2$. $w = 0$ gives $f(n) = 2g(n)$. This behaves exactly like uniform-cost search—the factor of two makes no difference in the *ordering* of the nodes. $w = 1$ gives A* search. $w = 2$ gives $f(n) = 2h(n)$, i.e., greedy best-first search. We also have

$$f(n) = (2 - w)[g(n) + \frac{w}{2 - w}h(n)]$$

which behaves exactly like A* search with a heuristic $\frac{w}{2-w}h(n)$. For $w \leq 1$, this is always less than $h(n)$ and hence admissible, provided $h(n)$ is itself admissible.

**Exercise 3.**UNGR

Consider the unbounded version of the regular 2D grid shown in Figure 3.6. The start state is at the origin, (0,0), and the goal state is at $(x, y)$.

**a**. What is the branching factor $b$ in this state space?
**b**. How many distinct states are there at depth $k$ (for $k > 0$)?
**c**. What is the maximum number of nodes expanded by breadth-first tree search?
**d**. What is the maximum number of nodes expanded by breadth-first graph search?
**e**. Is $h = |u - x| + |v - y|$ an admissible heuristic for a state at $(u, v)$? Explain.
**f**. How many nodes are expanded by A* graph search using $h$?
**g**. Does $h$ remain admissible if some links are removed?
**h**. Does $h$ remain admissible if some links are added between nonadjacent states?

**a**. The branching factor is 4 (number of neighbors of each location).
**b**. The states at depth $k$ form a square rotated at 45 degrees to the grid. Obviously there are a linear number of states along the boundary of the square, so the answer is $4k$.
**c**. Without repeated state checking, BFS expends exponentially many nodes: counting precisely, we get $((4^{x+y+1} - 1)/3) - 1$.

**d**. There are quadratically many states within the square for depth $x + y$, so the answer is $2(x + y)(x + y + 1) - 1$.

**e**. True; this is the Manhattan distance metric.

**f**. False; all nodes in the rectangle defined by $(0, 0)$ and $(x, y)$ are candidates for the optimal path, and there are quadratically many of them, all of which may be expended in the worst case.

**g**. True; removing links may induce detours, which require more steps, so $h$ is an underestimate.

**h**. False; nonlocal links can reduce the actual path length below the Manhattan distance.

**Exercise 3.**VEGR

$n$ vehicles occupy squares $(1, 1)$ through $(n, 1)$ (i.e., the bottom row) of an $n \times n$ grid. The vehicles must be moved to the top row but in reverse order; so the vehicle $i$ that starts in $(i, 1)$ must end up in $(n - i + 1, n)$. On each time step, every one of the $n$ vehicles can move one square up, down, left, or right, or stay put; but if a vehicle stays put, one other adjacent vehicle (but not more than one) can hop over it. Two vehicles cannot occupy the same square.

**a**. Calculate the size of the state space as a function of $n$.

**b**. Calculate the branching factor as a function of $n$.

**c**. Suppose that vehicle $i$ is at $(x_i, y_i)$; write a nontrivial admissible heuristic $h_i$ for the number of moves it will require to get to its goal location $(n - i + 1, n)$, assuming no other vehicles are on the grid.

**d**. Which of the following heuristics are admissible for the problem of moving all $n$ vehicles to their destinations? Explain.

(i) $\sum_{i=1}^{n} h_i$.
(ii) $\max\{h_1, \ldots, h_n\}$.
(iii) $\min\{h_1, \ldots, h_n\}$.

**a**. $n^{2n}$. There are $n$ vehicles in $n^2$ locations, so roughly (ignoring the one-per-square constraint) $(n^2)^n = n^{2n}$ states.

**b**. $5^n$.

**c**. Manhattan distance, i.e., $|(n - i + 1) - x_i| + |n - y_i|$. This is exact for a lone vehicle.

**d**. Only (iii) $\min\{h_1, \ldots, h_n\}$. The explanation is nontrivial as it requires two observations. First, let the *work* $W$ in a given solution be the total *distance* moved by all vehicles over their joint trajectories; that is, for each vehicle, add the lengths of all the steps taken. We have $W \geq \sum_i h_i \geq\geq n \cdot min\{h_1, ..., h_n\}$. Second, the total work we can get done per step is $\leq n$. (Note that for every car that jumps 2, another car has to stay put (move 0), so the total work per step is bounded by $n$.) Hence, completing all the work requires at least $n \cdot min\{h_1, ..., h_n\}/n = min\{h_1, ..., h_n\}$ steps.

**Exercise 3.**KNIG

Consider the problem of moving $k$ knights from $k$ starting squares $s_1, \ldots, s_k$ to $k$ goal squares $g_1, \ldots, g_k$, on an unbounded chessboard, subject to the rule that no two knights can land on the same square at the same time. Each action consists of moving *up to* $k$ knights simultaneously. We would like to complete the maneuver in the smallest number of actions.

**a**. What is the maximum branching factor in this state space, expressed as a function of $k$?

**b**. Suppose $h_i$ is an admissible heuristic for the problem of moving knight $i$ to goal $g_i$ by itself. Which of the following heuristics are admissible for the $k$-knight problem? Of those, which is the best?

(i) $\min\{h_1, \ldots, h_k\}$.
(ii) $\max\{h_1, \ldots, h_k\}$.
(iii) $\sum_{i=1}^{k} h_i$.

**c**. Repeat (b) for the case where you are allowed to move only one knight at a time.

**a**. $9^k$. For each of $k$ knights, we have one of 8 moves in addition to the possibility of not moving at all; each action executes one of these 9 choices for each knight (unless some choices conflict by landing on the same square), so there are $9^k$ actions.

**b**. (i) and (ii) are admissible. If the $h_i$ were exact, (ii) would be exact for the relaxed problem where knights can land on the same square. The $h_i$ are admissible and hence no larger than the exact values, so (ii) is admissible. (i) is no larger than (ii), so (i) is admissible. (iii) is not admissible. For example, if each $g_i$ is one move from its $s_i$, then (iii) returns $k$ whereas the optimal solution cost is 1.

(ii) dominates (i) so it must be as good or better. (iii) is probably of little value since it isn't admissible and completely ignores the capability for parallel moves.

**c**. In this case all are admissible. (i) and (ii) as the problem in part (b) is a relaxation of the problem in this part, and (i) and (ii) are admissible for the relaxed problem. (iii) is admissible because it exact for the relaxed problem where knights can land on the same square.

(iii) is best since it dominates the rest and is now admissible.

**Exercise 3.**IAFA

We saw on page **??** that the straight-line distance heuristic leads greedy best-first search astray on the problem of going from Iasi to Fagaras. However, the heuristic is perfect on the opposite problem: going from Fagaras to Iasi. Are there problems for which the heuristic is misleading in both directions?

Going between Rimnicu Vilcea and Lugoj is one example. The shortest path is the southern one, through Mehadia, Dobreta and Craiova. But a greedy search using the straight-line heuristic starting in Rimnicu Vilcea will start the wrong way, heading to Sibiu. Starting at Lugoj, the heuristic will correctly lead us to Mehadia, but then a greedy search will return to Lugoj, and oscillate forever between these two cities.

**Exercise 3.**HEUE

Invent a heuristic function for the 8-puzzle that sometimes overestimates, and show how it can lead to a suboptimal solution on a particular problem. (You can use a computer to help if you want.) Prove that if $h$ never overestimates by more than $c$, A* using $h$ returns a solution whose cost exceeds that of the optimal solution by no more than $c$.

The heuristic $h = h_1 + h_2$ (adding misplaced tiles and Manhattan distance) sometimes overestimates. Now, suppose $h(n) \leq h^*(n) + c$ (as given) and let $G_2$ be a goal that is suboptimal by more than $c$, i.e., $g(G_2) > C^* + c$. Now consider any node $n$ on a path to an optimal goal. We have

$$\begin{aligned} f(n) &= g(n) + h(n) \\ &\leq g(n) + h^*(n) + c \\ &\leq C^* + c \\ &\leq g(G_2) \end{aligned}$$

so $G_2$ will never be expanded before an optimal goal is expanded.

**Exercise 3.**CONH

Prove that if a heuristic is consistent, it must be admissible. Construct an admissible heuristic that is not consistent.

A heuristic is consistent iff, for every node $n$ and every successor $n'$ of $n$ generated by any action $a$,

$$h(n) \leq c(n, a, n') + h(n')$$

One simple proof is by induction on the number $k$ of nodes on the shortest path to any goal from $n$. For $k = 1$, let $n'$ be the goal node; then $h(n) \leq c(n, a, n')$. For the inductive case, assume $n'$ is on the shortest path $k$ steps from the goal and that $h(n')$ is admissible by hypothesis; then

$$h(n) \leq c(n, a, n') + h(n') \leq c(n, a, n') + h^*(n') = h^*(n)$$

so $h(n)$ at $k + 1$ steps from the goal is also admissible.

**Exercise 3.**MSTR

The traveling salesperson problem (TSP) can be solved with the minimum-spanning-tree (MST) heuristic, which estimates the cost of completing a tour, given that a partial tour has already been constructed. The MST cost of a set of cities is the smallest sum of the link costs of any tree that connects all the cities.

**a**. Show how this heuristic can be derived from a relaxed version of the TSP.

**b**. Show that the MST heuristic dominates straight-line distance.

**c**. Write a problem generator for instances of the TSP where cities are represented by random points in the unit square.

**d**. Find an efficient algorithm in the literature for constructing the MST, and use it with A* graph search to solve instances of the TSP.

This exercise reiterates a small portion of the classic work of Held and Karp (1970).

a. The TSP problem is to find a minimal (total length) path through the cities that forms a closed loop. MST is a relaxed version of that because it asks for a minimal (total length) graph that need not be a closed loop—it can be any fully-connected graph. As a heuristic, MST is admissible—it is always shorter than or equal to a closed loop.

b. The straight-line distance back to the start city is a rather weak heuristic—it vastly underestimates when there are many cities. In the later stage of a search when there are only a few cities left it is not so bad. To say that MST dominates straight-line distance is to say that MST always gives a higher value. This is obviously true because a MST that includes the goal node and the current node must either be the straight line between them, or it must include two or more lines that add up to more. (This all assumes the triangle inequality.)

c. See `aima-list/search/domains/tsp.lisp` for a start at this. The file includes a heuristic based on connecting each unvisited city to its nearest neighbor, a close relative to the MST approach.

d. See Cormen *et al.*, 1990, page 505 for an algorithm that runs in $O(E \log E)$ time, where $E$ is the number of edges. The code repository currently contains a somewhat less efficient algorithm.

**Exercise 3.**GASC

On page 118, we defined the relaxation of the 8-puzzle in which a tile can move from square A to square B if B is blank. The exact solution of this problem defines **Gaschnig's heuristic** (Gaschnig, 1979). Explain why Gaschnig's heuristic is at least as accurate as $h_1$ (misplaced tiles), and show cases where it is more accurate than both $h_1$ and $h_2$ (Manhattan distance). Explain how to calculate Gaschnig's heuristic efficiently.

The misplaced-tiles heuristic is exact for the problem where a tile can move from square A to square B. As this is a relaxation of the condition that a tile can move from square A to square B if B is blank, Gaschnig's heuristic cannot be less than the misplaced-tiles heuristic. As it is also admissible (being exact for a relaxation of the original problem), Gaschnig's heuristic is therefore more accurate.

If we permute two adjacent tiles in the goal state, we have a state where misplaced-tiles and Manhattan both return 2, but Gaschnig's heuristic returns 3.

To compute Gaschnig's heuristic, repeat the following until the goal state is reached: let B be the current location of the blank; if B is occupied by tile X (not the blank) in the goal

state, move X to B; otherwise, move any misplaced tile to B. Students could be asked to prove that this is the optimal solution to the relaxed problem.

**Exercise 3.**MANH

We gave two simple heuristics for the 8-puzzle: Manhattan distance and misplaced tiles. Several heuristics in the literature purport to improve on this—see, for example, Nilsson (1971), Mostow and Prieditis (1989), and Hansson *et al.*(1992). Test these claims by implementing the heuristics and comparing the performance of the resulting algorithms.

Students should provide results in the form of graphs and/or tables showing both runtime and number of nodes generated. (Different heuristics have different computation costs.) Runtimes may be very small for 8-puzzles, so you may want to assign the 15-puzzle or 24-puzzle instead. The use of pattern databases is also worth exploring experimentally.