of vertices in component of G containing vertex (i, j)—which could be considerably smaller than $O(n^2)$.

Exercises

Graphs

1. Prove that the following definitions are all equivalent.
   - A tree is a connected acyclic graph.
   - A tree is one component of a forest. (A forest is an acyclic graph.)
   - A tree is a connected graph with *at most* $V - 1$ edges.
   - A tree is a minimally connected graph; removing any edge makes the graph disconnected.
   - A tree is an acyclic graph with *at least* $V - 1$ edges.
   - A tree is a maximally acyclic graph; adding an edge between any two vertices creates a cycle.

2. Prove that any connected acyclic graph with $n \geq 2$ vertices has at least two vertices with degree 1. Do not use the words "tree" or "leaf", or any well-known properties of trees; your proof should follow entirely from the definitions of "connected" and "acyclic".

3. A graph (V, E) is *bipartite* if the vertices V can be partitioned into two subsets L and R, such that every edge has one vertex in L and the other in R.
   (a) Prove that every tree is a bipartite graph.
   (b) Describe and analyze an efficient algorithm that determines whether a given undirected graph is bipartite.

4. Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. Surprisingly, the overall pecking order can contain cycles—for example, pigeon i pecks pigeon j, which pecks pigeon k, which pecks pigeon ℓ, which pecks pigeon i.
   (a) Prove that any finite population of pigeons can be placed in a procession (perhaps a parade?) so that each pigeon pecks the preceding pigeon's posterior. Pretty please.

(b) Suppose you are given a directed graph representing the pecking relationships among a set of n pigeons. The graph contains one vertex per pigeon, and it contains an edge $i \to j$ if and only if pigeon i pecks pigeon j. Describe and analyze an algorithm to compute a pecking order for the pigeons, as guaranteed by part (a).

(c) Prove that for any set of at least three pigeons, either the pecking order described in part (a) is unique, or there are three pigeons i, j, and k, such that pigeon i pecks pigeon j, which pecks pigeon k, which pecks pigeon i.

5. An ***Euler tour*** of a graph G is a closed walk through G that traverses every edge of G exactly once.

   (a) Prove that if a connected graph G has an Euler tour, then every vertex in G has even degree. (Euler proved this.)

   (b) Prove that if every vertex in a connected graph G has even degree, then G has an Euler tour. (Euler did *not* prove this.)

   (c) Describe and analyze an algorithm to compute an Euler tour in a given graph, or correctly report that no such tour exists. (Euler vaguely waved his hands at this.)

6. The d-dimensional hypercube is the graph defined as follows. There are $2d$ vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if their labels differ in exactly one bit.

   (a) A Hamiltonian cycle in a graph G is a cycle of edges in G that visits every vertex of G exactly once. Prove that for all $d \geq 2$, the d-dimensional hypercube has a Hamiltonian cycle.

   (b) Which hypercubes have an Euler tour (a closed walk that traverses every edge exactly once)? *[Hint: This is very easy.]*

Traversal Algorithms

7. Recall that a directed graph G is *strongly connected* if, for any two vertices u and v, there is a path in G from u to v and a path in G from v to u.

   Describe an algorithm to determine, given an *undirected* graph G as input, whether it is possible to direct each edge of G so that the resulting directed graph is strongly connected.

8. Let G be a connected graph, and let T be a depth-first spanning tree of G rooted at some node v. Prove that if T is also a breadth-first spanning tree of G rooted at v, then $G = T$.

9. Professors Epprich and Goodstein propose the following optimization of the generic whatever-first search algorithm. Instead of checking whether the vertices we take out of the bag are marked, their algorithm checks before it even puts the vertex into the bag, thereby ensuring that each vertex is put into the bag at most once. Their algorithm also assigns the parent of each vertex when that vertex is marked.

```
EagerWFS(s):
  mark s
  put s into the bag
  while the bag is not empty
      take v from the bag
      for each edge vw
          if w is unmarked
              mark w
              parent(w) ← v
              put w into the bag
```

   (a) Prove that EagerWFS(s) marks every node reachable from s and nothing else. Equivalently, prove that the parent edges $v \to parent(v)$ computed by EagerWFS(s) define a spanning tree of the component containing s.

   (b) Prove that if the bag is implemented as a queue, EagerWFS is equivalent to breadth-first search, meaning the two algorithms mark the same vertices in the same order and construct the same spanning tree. *[Hint: What is the definition of a queue?]*

   (c) Prove that EagerWFS is *never* equivalent to depth-first search, no matter what data structure is used as the bag (and thus, in particular, when the bag is a stack).

   Neither EagerWFS nor RecursiveDFS specify the order that edges vw at each vertex v are considered, and different edge orders may lead to different spanning trees. Thus, you need to argue, for some explicit graph G, that no spanning tree of G produced by RecursiveDFS can be constructed by EagerWFS (using *any* bag data structure), or vice versa.

10. One of the earliest published descriptions of whatever-first search as a generic class of algorithms was by Edsger Dijkstra, Leslie Lamport, Alain J. Martin, Carel S. Scholten, and Elisabeth F. M. Steffens in 1975, as part of the design of an automatic garbage collector. Instead of maintaining marked and unmarked vertices, their algorithm maintains a color for each vertex, which is either white, gray, or black. As usual, in the following algorithm, we imagine a fixed underlying graph G.

```
ThreeColorSearch(s):
  color all nodes white
  color s gray
  while at least one vertex is gray
      ThreeColorStep( )
```

```
ThreeColorStep( ):
  v ← any gray vertex
  if v has no white neighbors
      color v black
  else
      w ← any white neighbor of v
      parent(w) ← v
      color w gray
```

(a) Prove that ThreeColorSearch maintains the following invariant at all times: No black vertex is a neighbor of a white vertex. *[Hint: This should be easy.]*

(b) Prove that after ThreeColorSearch(s) terminates, all vertices reachable from s are black, all vertices not reachable from s are white, and that the parent edges $v \to parent(v)$ define a rooted spanning tree of the component containing s.

[Hint: Intuitively, black nodes are "marked" and gray nodes are "in the bag". Unlike our formulation of WhateverFirstSearch, however, the three-color algorithm is not required to process all edges out of a node at the same time.]

(c) Prove that the following variant of ThreeColorSearch, which maintains the set of gray vertices in a standard stack, is equivalent to depth-first search. *[Hint: The order of the last two lines of ThreeColorStackStep matters!]*

```
ThreeColorStackSearch(s):
  color all nodes white
  color s gray
  push s onto the stack
  while at least one vertex is gray
      ThreeColorStackStep( )
```

```
ThreeColorStackStep( ):
  pop v from the stack
  if v has no white neighbors
      color v black
  else
      w ← any white neighbor of v
      parent(w) ← v
      color w gray
      push v onto the stack
      push w onto the stack
```

(d) Prove that the following variant of ThreeColorSearch, which maintains the set of gray vertices in a standard queue, is ***not*** equivalent to breadth-first search. *[Hint: The order of the last two lines of ThreeColorQueueStep doesn't matter!]*

```
ThreeColorQueueSearch(s):
  color all nodes white
  color s gray
  push s into the queue
  while at least one vertex is gray
      ThreeColorQueueStep( )
```

```
ThreeColorQueueStep( ):
  pull v from the queue
  if v has no white neighbors
      color v black
  else
      w ← any white neighbor of v
      parent(w) ← v
      color w gray
      push v into the queue
      push w into the queue
```

♥(e) Now suppose that another process is adding edges to G while ThreeColorSearch is running. These new edges could violate the color invariant described in part (a) and therefore destroy the correctness of the algorithm—in particular, when ThreeColorSearch terminates, some vertices reachable from s could be white. This would be disastrous if we are relying on "white" to mean "unreachable and therefore safe to delete".

However, if the other process explicitly preserves the color invariant, we can still use the three-color algorithm to safely identify unreachable vertices. We model the two concurrent algorithms as follows; the either/or choice in GarbageCollect and the choice of which vertices u and w to Mutate are entirely out of the main algorithm's control.[14]

```
GarbageCollect(s):
  color all vertices white
  color s gray
  while at least one vertex is gray
      either
          CollectStep( )
      or
          Mutate( )
```

```
CollectStep( ):
  v ← any gray vertex
  if v has no white neighbors
      color v black
  else
      w ← any white neighbor of v
      color w gray
```

```
Mutate( ):
  u ← any vertex
  w ← any vertex
  if uw is not an edge
      add edge uw
      if u is black and w is white
          color u gray
      if u is white and w is black
          color w gray
```

[14]This is a *dramatic* oversimplification of the "mark and sweep" garbage-collection algorithms actually used in multi-threaded languages like Lua and Go. A more thorough discussion of multi-threaded dynamic memory management is unfortunately beyond the scope of this book, except for the First Commandment: **Thou Shalt Not Roll Thine Own Garbage Collector.**

Prove that GARBAGECOLLECT eventually terminates with every vertex reachable from s colored black and every vertex not reachable from s colored white.

♥(f) Suppose instead of recoloring black vertices gray, MUTATE maintains the color invariant by coloring some *white* vertices gray:

```
MUTATE():
  u ← any vertex
  w ← any vertex
  if uw is not an edge
      add edge uw
      if u is black and w is white
          color w gray
      if u is white and w is black
          color u gray
```

Prove that GARBAGECOLLECT eventually terminates with s colored black, every vertex reachable from a black vertex colored black, and every vertex not reachable from a black vertex colored white.

Reductions

11. A ***number maze*** is an $n \times n$ grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner. On each turn, you are allowed to move the token up, down, left, or right; the distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three steps down, three steps left, or three steps right. However, you are never allowed to move the token off the edge of the board.

    Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution.

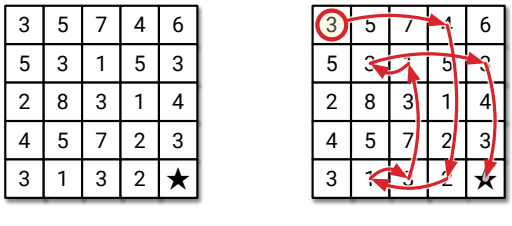


Figure 5.14. A 5 × 5 number maze that can be solved in eight moves.

12. ***Snakes and Ladders*** is a classic board game, originating in India no later than the 16th century. The board consists of an $n \times n$ grid of squares,

numbered consecutively from 1 to n^2, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares in this grid, always in different rows, are connected by either "snakes" (leading down) or "ladders" (leading up). Each square can be an endpoint of at most one snake or ladder.

You start with a token in cell 1, in the bottom left corner. In each move, you advance your token up to k positions, for some fixed constant k. If the token ends the move at the *top* end of a snake, it slides down to the bottom of that snake. Similarly, if the token ends the move at the *bottom* end of a ladder, it climbs up to the top of that ladder.

Describe and analyze an algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.

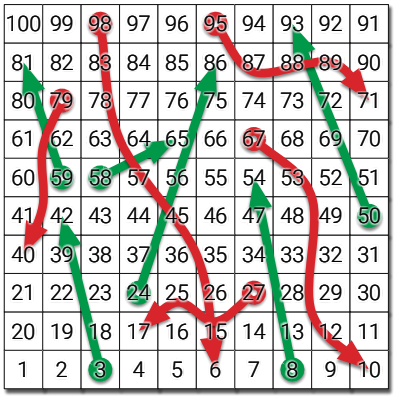


Figure 5.15. A Snakes and Ladders board. Upward straight arrows are ladders; downward wavy arrows are snakes.

13. The infamous Mongolian puzzle-warrior Vidrach Itky Leda invented the following puzzle in the year 1473. The puzzle consists of an $n \times n$ grid of squares, where each square is labeled with a positive integer, and two tokens, one red and the other blue. The tokens always lie on distinct squares of the grid. The tokens start in the top left and bottom right corners of the grid; the goal of the puzzle is to swap the tokens.

    In a single turn, you may move either token up, right, down, or left *by a distance determined by the* ***other*** *token*. For example, if the red token is on a square labeled 3, then you may move the blue token 3 steps up, 3 steps left, 3 steps right, or 3 steps down. However, you may not move either token off the grid, and at the end of a move the two tokens cannot lie on the same square.

    Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given Vidrach Itky Leda

puzzle, or correctly reports that the puzzle has no solution. For example, given the puzzle in Figure 5.16, your algorithm would return the number 5.

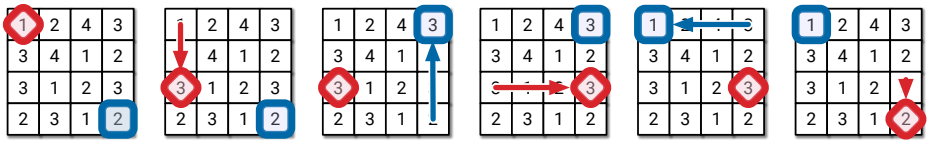
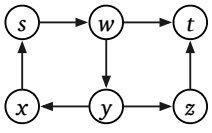


Figure 5.16. A five-move solution for a 4 × 4 Vidrach Itky Leda puzzle.

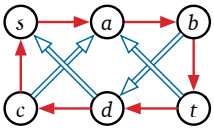
14. Suppose you are given a directed graph $G = (V, E)$ and two vertices s and t. Describe and analyze an algorithm to determine if there is a walk in G from s to t (possibly repeating vertices and/or edges) whose length is divisible by 3.

    For example, given the graph shown below, with the indicated vertices s and t, your algorithm should return True, because the walk $s \to w \to y \to x \to s \to w \to t$ has length 6.



15. Suppose you are given a directed graph G where some edges are red and the remaining edges are blue. Describe an algorithm to find the shortest walk in G from one vertex s to another vertex t in which no three consecutive edges have the same color. That is, if the walk contains two red edges in a row, the next edge must be blue, and if the walk contains two blue edges in a row, the next edge must be red.

    For example, given the following graph as input, your algorithm should return the integer 7, because $s \to a \to b \Rightarrow d \to c \Rightarrow a \to b \to t$ is the shortest legal walk from s to t.



16. Consider a directed graph G, where each edge is colored either red, white, or blue. A walk in G is called a *French flag walk* if its sequence of edge colors is red, white, blue, red, white, blue, and so on. More formally, a walk $v_0 \to v_1 \to \cdots \to v_k$ is a French flag walk if, for every integer i, the edge $v_i \to v_{i+1}$ is red if $i \bmod 3 = 0$, white if $i \bmod 3 = 1$, and blue if $i \bmod 3 = 2$.

    Describe an algorithm to find all vertices in G that can be reached from a given vertex v through a French flag walk.

17. There are n galaxies connected by m intergalactic teleport-ways. Each teleport-way joins two galaxies and can be traversed in both directions. Also, each teleport-way e has an associated cost of $c(e)$ dollars, where $c(e)$ is a positive integer. A teleport-way can be used multiple times, but the toll must be paid every time it is used.

    Judy wants to travel from galaxy s to galaxy t, but teleportation is not very pleasant and she would like to minimize the number of times she needs to teleport. However, she wants the total cost to be a multiple of five dollars, because carrying small change is not pleasant either.

    (a) Describe and analyze an algorithm to compute the smallest number of times Judy needs to teleport to travel from galaxy s to galaxy t so that the total cost is a multiple of five dollars.

    (b) Solve part (a), but now assume that Judy has a coupon that allows her to use exactly one teleport-way for free.

18. Three Sea Shells is a solitaire game, played on a connected undirected graph G. Initially, three tokens are placed on distinct start vertices a, b, c. In each turn, you *must* move *all three* tokens, by moving each token along an edge from its current vertex to an adjacent vertex. At the end of each turn, the three tokens *must* lie on three different vertices. Your goal is to move the tokens onto three goal vertices x, y, z; it does not matter which token ends up on which goal vertex.

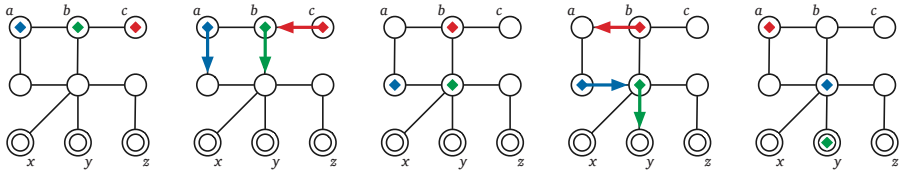


Figure 5.17. The initial configuration of the Three Sea Shells puzzle and the first two turns of a solution.

    Describe and analyze an algorithm to determine whether a given Three Sea Shells puzzle is solvable. Your input consists of the graph G, the start vertices a, b, c, and the goal vertices x, y, z. Your output is a single bit: TRUE or FALSE.

19. Let G be a connected undirected graph. Suppose we start with two coins on two arbitrarily chosen vertices of G, and we want to move the coins so that they lie on the same vertex using as few moves as possible. At every step, each coin *must* move to an adjacent vertex.

    (a) Describe and analyze an algorithm to compute the minimum number of steps to reach a configuration where both coins are on the same vertex,

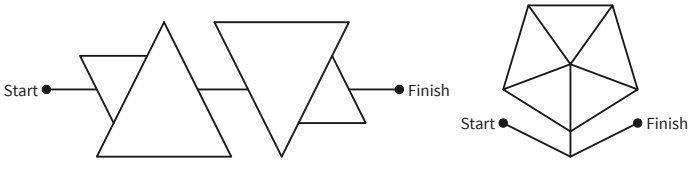
or to report correctly that no such configuration is reachable. The input to your algorithm consists of a graph $G = (V, E)$ and two vertices $u, v \in V$ (which may or may not be distinct).

(b) Now suppose there are three coins. Describe and analyze an algorithm to compute the minimum number of steps to reach a configuration where both coins are on the same vertex, or to report correctly that no such configuration is reachable.

(c) Finally, suppose there are *forty-two* coins. Describe and analyze an algorithm to determine whether it is possible to move all 42 coins to the same vertex. Again, *every* coin must move at *every* step. For full credit, your algorithm should run in $O(V + E)$ time.

20. One of my daughter's elementary-school math workbooks[15] contains several puzzles of the following type:

> Complete each angle maze below by tracing a path from start to finish that has only acute angles.



Describe and analyze an algorithm to solve arbitrary acute-angle mazes.

You are given a connected undirected graph G, whose vertices are points in the plane and whose edges are line segments. Edges do not intersect, except at their endpoints. For example, a drawing of the letter X would have five vertices and four edges, and the first maze above has 18 vertices and 21 edges. You are also given two vertices Start and Finish.

Your algorithm should return True if G contains a walk from Start to Finish that has only acute angles, and False otherwise. Formally, a walk through G is valid if, for any two consecutive edges $u \to v \to w$ in the walk, either $\angle uvw = \pi$ or $0 < \angle uvw < \pi/2$. Assume you have a subroutine that can determine in $O(1)$ time whether the angle between two given segments is straight, obtuse, right, or acute.

21. Suppose you are given a set of n horizontal and vertical line segments and two points s and t in the plane. Describe an efficient algorithm to determine if there is a path from s to t that does not intersect any of the given line segments.

[15] Jason Batterson and Shannon Rogers, *Beast Academy Math: Practice 3A*, 2012. See https://www.beastacademy.com/resources/printables.php for several more examples.

Each horizontal line segment is specified by its left and right x-coordinates and its unique y-coordinate; similarly, each vertical line segment is specified by its unique x-coordinate and its top and bottom y-coordinates. Finally, the points s and t are each specified by their x- and y-coordinates.

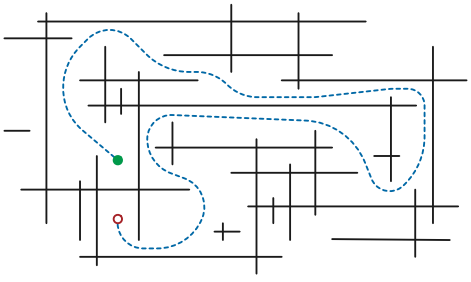


Figure 5.18. A path between two points in a maze of horizontal and vertical line segments.

22. Every cheesy romance movie has a scene where the romantic couple, after a long and frustrating separation, suddenly see each other across a long distance, and then slowly approach one another with unwavering eye contact as the music rolls in and the rain lifts and the sun shines through the clouds and the music swells and everyone starts dancing with rainbows and kittens and chocolate unicorns and. . . .[16]

    Suppose a romantic couple—in grand computer science tradition, named Alice and Bob—enters their favorite park at the east and west entrances and immediately establish eye-contact. They can't just run directly to each other; instead, they must stay on the path that zig-zags through the park between the east and west entrances. To maintain the proper dramatic tension, Alice and Bob must traverse the path so that they always lie on a direct east-west line.

    We can describe the zigzag path as two arrays $X[0..n]$ and $Y[0..n]$, containing the x- and y-coordinates of the corners of the path, in order from the southwest endpoint to the southeast endpoint. The X array is sorted in increasing order, and $Y[0] = Y[n]$. The path is a sequence of straight line segments connecting these corners.

    (a) Suppose $Y[0] = Y[n] = 0$ and $Y[i] > 0$ for every other index i; that is, the endpoints of the path are strictly below every other point on the path. Prove that for any path P meeting these conditions, Alice and Bob can *always* meet. *[Hint: Describe a graph that models all possible locations of the couple along the path. What are the vertices of this graph? What*

[16] Fun fact: Damien Chazelle, the director of *Whiplash* and *La La Land*, is the son of Princeton computer science professor and electric guitarist Bernard Chazelle.

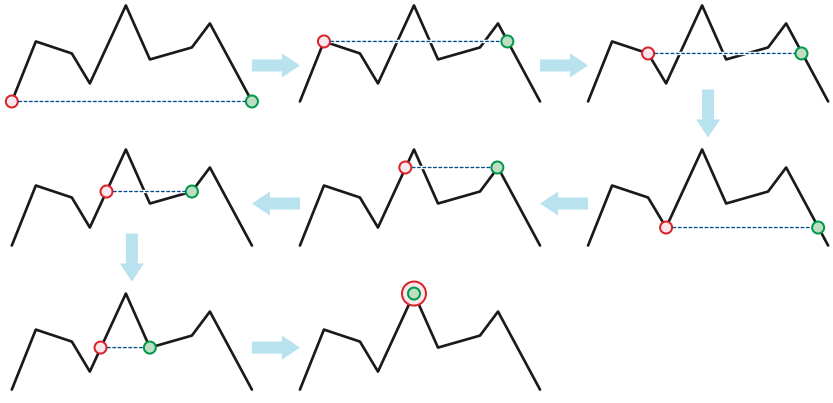


Figure 5.19. Alice and Bob meet. Alice walks backward in step 2, and Bob walks backward in steps 5 and 6.

are the edges? Use the ***Handshake Lemma****: Every graph has an even number of vertices with odd degree.]*

(b) If the endpoints of the path are *not* below every other vertex, Alice and Bob might still be able to meet, or they might not. Describe an algorithm to decide whether Alice and Bob can meet, without either breaking east-west eye contact or stepping off the path, given the arrays $X[0..n]$ and $Y[0..n]$ as input.

♥(c) Describe an algorithm for part (b) that runs in $O(n)$ time.

23. The famous puzzle-maker Kaniel the Dane invented a solitaire game played with two tokens on an $n \times n$ square grid. Some squares of the grid are marked as *obstacles*, and one grid square is marked as the *target*. In each turn, the player must move one of the tokens from is current position *as far as possible* upward, downward, right, or left, stopping just before the token hits (1) the edge of the board, (2) an obstacle square, or (3) the other token. The goal is to move either of the tokens onto the target square.

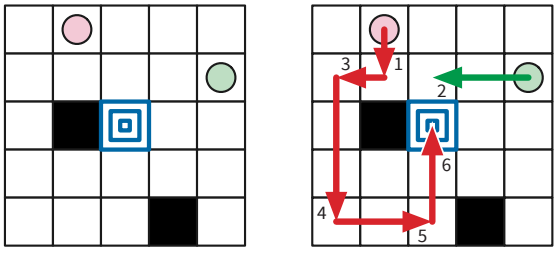


Figure 5.20. An instance of Kaniel the Dane's puzzle that can be solved in six moves. Circles indicate initial token positions; black squares are obstacles; the center square is the target.

For example, we can solve the puzzle shown in Figure 5.20 by moving the red token down until it hits the obstacle, then moving the green token

left until it hits the red token, and then moving the red token left, down, right, and up. The red token stops at the target on the 6th move *because* the green token is just above the target square.

Describe and analyze an algorithm to determine whether an instance of this puzzle is solvable. Your input consist of the integer n, a list of obstacle locations, the target location, and the initial locations of the tokens. The output of your algorithm is a single boolean: TRUE if the given puzzle is solvable and FALSE otherwise. *[Hint: Don't forget about the time required to construct the graph.]*

♥24. ***Rectangle Walk*** is a new abstract puzzle game, available for only 99¢ on Steam, iOS, Android, Xbox One, Playstation 5, Nintendo Wii U, Atari 2600, Palm Pilot, Commodore 64, TRS-80, Sinclair ZX-1, DEC PDP-8, PLATO, Zuse Z3, Duramesc, Odhner Arithmometer, Analytical Engine, Jacquard Loom, Horologium Mirabile Lundense, Leibniz Stepped Reckoner, Al-Jazari's Robot Band, Yan Shi's Automaton, Antikythera Mechanism, Knotted Rope, Ishango Bone, and Pile of Rocks.

The game is played on an $n \times n$ grid of black and white squares. The player moves a rectangle through this grid, subject to the following conditions:

- The rectangle must be aligned with the grid; that is, the top, bottom, left, and right coordinates must be integers.
- The rectangle must fit within the $n \times n$ grid, and it must contain at least one grid cell.
- The rectangle must not contain a black square.
- In a single move, the player can replace the current rectangle r with any rectangle r' that either contains r or is contained in r.

Initially, the player's rectangle is a 1×1 square in the upper right corner. The player's goal is to reach a 1×1 square in the bottom left corner using as few moves as possible.

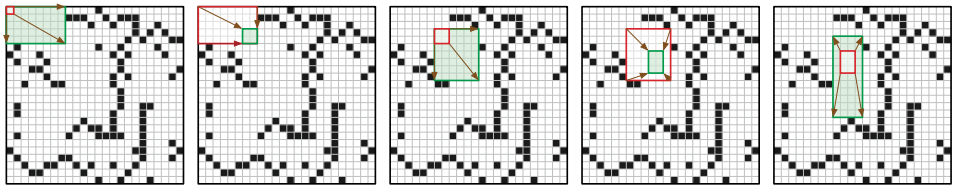


Figure 5.21. The first five steps of a Rectangle Walk.

Describe and analyze an algorithm to compute the length of the shortest Rectangle Walk in a given bitmap. Your input is an array $M[1..n, 1..n]$, where $M[i, j] = 1$ indicates a black square and $M[i, j] = 0$ indicates a white square. Assume that a valid rectangle walk exists; in particular,

$M[1,1] = M[n,n] = 0$. For example, given the bitmap shown above, your algorithm should return the integer 18. *[Hint: Don't forget about the time required to construct the graph!!]*

25. ***Racetrack*** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.[17] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

velocity	position
(0,0)	(1,5)
(1,0)	(2,5)
(2,−1)	(4,4)
(3,0)	(7,4)
(2,1)	(9,5)
(1,2)	(10,7)
(0,3)	(10,10)
(−1,4)	(9,14)
(0,3)	(9,17)
(1,2)	(10,19)
(2,2)	(12,21)
(2,1)	(14,22)
(2,0)	(16,22)
(1,−1)	(17,21)
(2,−1)	(19,20)
(3,0)	(22,20)
(3,1)	(25,21)

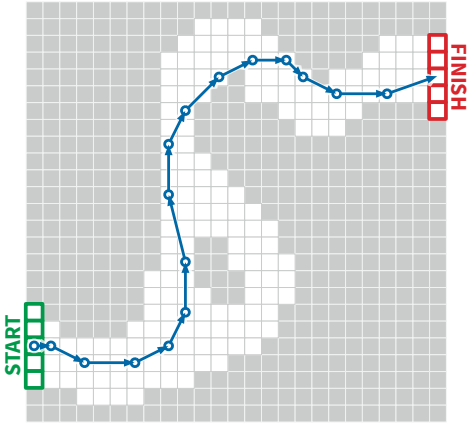


Figure 5.22. A 16-step Racetrack run, on a 25 × 25 track. This is *not* the shortest run on this track.

Each car has a *position* and a *velocity*, both with integer x- and y-coordinates. A subset of grid squares is marked as the *starting area*, and another subset is marked as the *finishing area*. The initial position of each car is chosen by the player somewhere in the starting area; the initial velocity of each car is always $(0,0)$. At each step, the player optionally increments or decrements either or both coordinates of the car's velocity; in other words, each component of the velocity can change by at most 1 in a single step. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race. The race ends when the first car reaches a position inside the finishing area.

Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the "starting area" is the first column, and the "finishing area" is the last column.

[17] The actual game is a bit more complicated than the version described here. See http://harmmade.com/vectorracer/ for an excellent online version.

Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting line to the finish line of a given racetrack.

26. A *rolling die maze* is a puzzle involving a standard six-sided die (a cube with numbers on each side) and a grid of squares. You should imagine the grid lying on a table; the die always rests on and exactly covers one square of the grid. In a single step, you can *roll* the die 90 degrees around one of its bottom edges, moving it to an adjacent square one step north, south, east, or west.

    Some squares in the grid may be *blocked*; the die can never rest on a blocked square. Other squares may be *labeled* with a number; whenever the die rests on a labeled square, the number on the *top* face of the die must equal the label. Squares that are neither labeled nor marked are *free*. You may not roll the die off the edges of the grid. A rolling die maze is *solvable* if it is possible to place a die on the lower left square and roll it to the upper right square under these constraints.

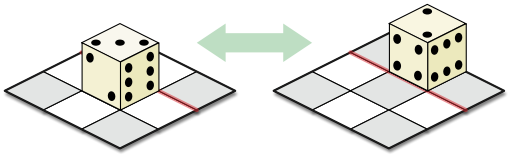


Figure 5.23. Rolling a die

Figure 5.24 shows four rolling die mazes. Assuming we use a standard die with 1 and 6 on opposite sides, only the first two mazes are solvable. For example, the first maze is solvable by by placing the die on the lower left square with 1 on the top face, and then rolling the die east, then north, then north, then east.

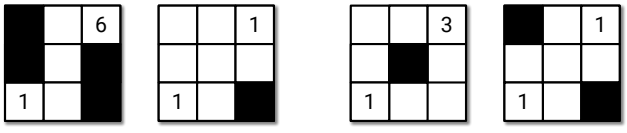


Figure 5.24. Four rolling die mazes; only the first two are solvable.

(a) Suppose the input is a two-dimensional array $L[1..n, 1..n]$, where each entry $L[i, j]$ stores the label of the square in the ith row and jth column, where 0 means the square is free and -1 means the square is blocked. Describe and analyze a polynomial-time algorithm to determine whether the given rolling die maze is solvable.

♥(b) Now suppose the maze is specified *implicitly* by a list of labeled and blocked squares. Specifically, suppose the input consists of an integer M,

specifying the height and width of the maze, and an array $S[1..n]$, where each entry $S[i]$ is a triple (x, y, L) indicating that square (x, y) has label L. As in the explicit encoding, label -1 indicates that the square is blocked; free squares are not listed in S at all. Describe and analyze an efficient algorithm to determine whether the given rolling die maze is solvable. For full credit, the running time of your algorithm should be polynomial in the input size n.

[Hint: You have some freedom in how to place the initial die. There are rolling die mazes that can be solved only if the initial position is chosen correctly.]

♥27. Suppose you are given an arbitrary directed graph G in which each edge is colored either red or blue, along with two special vertices s and t.

(a) Describe an algorithm that either computes a walk from s to t such that the pattern of red and blue edges along the walk is a palindrome, or correctly reports that no such walk exists.

(b) Describe an algorithm that either computes the *shortest* walk from s to t such that the pattern of red and blue edges along the walk is a palindrome, or correctly reports that no such walk exists.

[Hint: Where did we last see palindromes?]

♠♥28. Draughts, also known in the US as "checkers", is a game played on an $m \times m$ grid of squares, alternately colored light and dark.[18] The game is usually played on an 8×8 or 10×10 board, but the rules easily generalize to any board size. Each dark square is occupied by at most one game piece (usually called a *checker* in the U.S.), which is either black or white; light squares are always empty. One player ("White") moves the white pieces; the other ("Black") moves the black pieces. A player loses when her last piece is taken off the board.

Consider the following simple version of the game, essentially American checkers or British draughts, but where every piece is a king.[19] Pieces can be moved in any of the four diagonal directions. On each turn, a player either *moves* one of her pieces one step diagonally into an empty square,

[18] The counting tables used by medieval English government accountants were covered by a green cloth with black squares in a checker pattern; disk-shaped counters were placed in these squares to represent values. For this reason, the British government's accountants have been collectively known since the 10th century as the *Exchequer*. The actual counting tables were used by the Exchequer to tally tax payments well into the 19th century.

[19] Most other variants of draughts have "flying kings", which behave *very* differently than kings in the British/American game, and which make this problem *much* more difficult, as we will see in Chapter 12.

or makes a series of *jumps* with one of her pieces. In each jump, the piece moves to an empty square two steps away in any diagonal direction, but only if the intermediate square is occupied by a piece of the opposite color; this enemy piece is *captured* and immediately removed from the board. All jumps in the same turn must be made with the same piece.

Describe an algorithm to decide whether White can capture every black piece, thereby winning the game, *in a single turn*. The input consists of the width of the board (m), a list of positions of white pieces, and a list of positions of black pieces. For full credit, your algorithm should run in $O(n)$ time, where n is the total number of pieces. *[Hint: The greedy strategy—make arbitrary jumps until you get stuck—does* ***not*** *always find a winning sequence of jumps even when one exists. See problem 5. Parity, parity, parity.]*

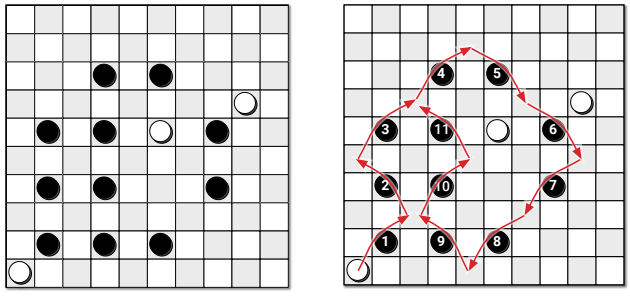


Figure 5.25. White wins in one turn.

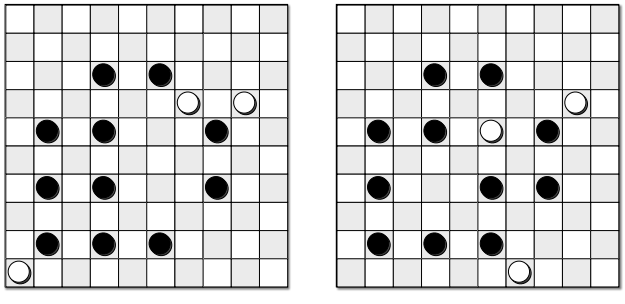


Figure 5.26. White cannot win in one turn from either of these positions.

And, for the hous is crinkled to and fro,
And hath so queinte weyes for to go—
For hit is shapen as the mase is wroght—
Therto have I a remedie in my thoght,
That, by a clewe of twyne, as he hath goon,
The same wey he may returne anoon,
Folwing alwey the threed, as he hath come.

— Geoffrey Chaucer, *The Legend of Good Women* (c. 1385)

"Com'è bello il mondo e come sono brutti i labirinti!" dissi sollevato.
"Come sarebbe bello il mondo se ci fosse una regola per girare nei labirinti,"
rispose il mio maestro.
["How beautiful the world is, and how ugly labyrinths are," I said, relieved.
"How beautiful the world would be if there were a procedure for moving through
labyrinths," my master replied.]

— Umberto Eco, *Il nome della rosa* (1980)
English translation (*The Name of the Rose*) by William Weaver (1983)

6

Depth-First Search

In the previous chapter, we considered a generic algorithm—whatever-first search—for traversing arbitrary graphs, both undirected and directed. In this chapter, we focus on a particular instantiation of this algorithm called *depth-first search*, and primarily on the behavior of this algorithm in directed graphs. Rather than using an explicit stack, depth-first search is normally implemented recursively as follows:

```
DFS(v):
  if v is unmarked
      mark v
      for each edge v→w
          DFS(w)
```

We can make this algorithm slightly faster (in practice) by checking whether a node is marked *before* we recursively explore it. This modification ensures

that we call DFS(v) only once for each vertex v. We can further modify the algorithm to compute other useful information about the vertices and edges, by introducing two black-box subroutines, PREVISIT and POSTVISIT, which we leave unspecified for now.

```
DFS(v):
  mark v
  PREVISIT(v)
  for each edge vw
      if w is unmarked
          parent(w) ← v
          DFS(w)
  POSTVISIT(v)
```

Recall that a node w is *reachable* from another node v in a directed graph G—or more simply, v can reach w—if and only if G contains a directed path from v to w. Let ***reach(v)*** denote the set of vertices reachable from v (including v itself). If we unmark all vertices in G, and then call DFS(v), the set of marked vertices is precisely *reach*(v).

Reachability in undirected graphs is symmetric: v can reach w if and only if w can reach v. As a result, after unmarking all vertices of an undirected graph G, calling DFS(v) traverses the entire component of v, and the parent pointers define a spanning tree of that component.

The situation is more subtle with directed graphs, as shown in the figure below. Even though the graph is "connected", different vertices can reach different, and potentially overlapping, portions of the graph. The parent pointers assigned by DFS(v) define a tree rooted at v whose vertices are precisely *reach*(v), but this is not necessarily a spanning tree of the graph.

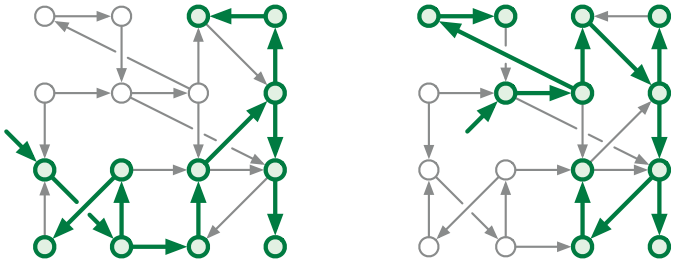


Figure 6.1. Depth-first trees rooted at different vertices in the same directed graph.

As usual, we can extend our reachability algorithm to traverse the *entire* input graph, even if it is disconnected, using the standard wrapper function shown on the left in Figure 6.2. Here we add a generic black-box subroutine PREPROCESS to perform any necessary preprocessing for the PREVISIT and POSTVISIT functions.