Obie looked at the seein' eye dog. Then at the twenty-seven 8 by 10 color glossy pictures with the circles and arrows and a paragraph on the back of each one. . . and then he looked at the seein' eye dog. And then at the twenty-seven 8 by 10 color glossy pictures with the circles and arrows and a paragraph on the back of each one and began to cry.

Because Obie came to the realization that it was a typical case of American blind justice, and there wasn't nothin' he could do about it, and the judge wasn't gonna look at the twenty-seven 8 by 10 color glossy pictures with the circles and arrows and a paragraph on the back of each one explainin' what each one was, to be used as evidence against us.

And we was fined fifty dollars and had to pick up the garbage. In the snow.

But that's not what I'm here to tell you about.

— Arlo Guthrie, "Alice's Restaurant" (1966)

I study my Bible as I gather apples.
First I shake the whole tree, that the ripest might fall.
Then I climb the tree and shake each limb,
and then each branch and then each twig,
and then I look under each leaf.

— Martin Luther

17 Basic Graph Properties

17.1 Definitions

A ***graph*** G is a pair of sets (V, E). V is a set of arbitrary objects that we call ***vertices***[1] or ***nodes***. E is a set of pairs of vertices, which we call ***edges*** or (more rarely) ***arcs***. In an *undirected* graph, the edges are unordered pairs, or just sets of two vertices; I will usually write $\boldsymbol{uv}$ instead of $\{u, v\}$ to denote the undirected edge between u and v. In a *directed* graph, the edges are ordered pairs of vertices; I will usually write $\boldsymbol{u \to v}$ instead of (u, v) to denote the directed edge from u to v. We will usually be concerned only with ***simple*** graphs, which do not have loops (edges from a vertex to itself) or parallel edges (multiple edges with the same endpoints).

Following standard (but admittedly confusing) practice, I'll also use V to denote the *number* of vertices in a graph, and E to denote the *number* of edges. Thus, in an undirected graph, we have $0 \le E \le \binom{V}{2}$, and in a directed graph, $0 \le E \le V(V-1)$.

For any edge uv in an undirected graph, we call u a ***neighbor*** of v and vice versa. The ***degree*** of a node is its number of neighbors. In directed graphs, we have two kinds of neighbors. For any directed edge $u \to v$, we call u a ***predecessor*** of v and v a ***successor*** of u. The ***in-degree*** of a node is the number of predecessors, which is the same as the number of edges going into the node. The ***out-degree*** is the number of successors, or the number of edges going out of the node.

A graph $G' = (V', E')$ is a ***subgraph*** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

A ***path*** is a sequence of edges, where each successive pair of edges shares a vertex, and all other edges are disjoint. A graph is ***connected*** if there is a path from any vertex to any other vertex. A disconnected graph consists of several ***components***, which are its maximal connected subgraphs. Two vertices are in the same component if and only if there is a path between them.

[1] The singular of 'vertices' is **vertex**. The singular of 'matrices' is **matrix**. Unless you're speaking Italian, there is no such thing as a vertice, a matrice, an indice, an appendice, a helice, an apice, a vortice, a radice, a simplice, a codice, a directrice, a dominatrice, a Unice, a Kleenice, an Asterice, an Obelice, a Dogmatice, a Getafice, a Cacofonice, a Vitalstatistice, a Geriatrice, or Jimi Hendrice! You *will* lose points for using any of these so-called words.

A ***cycle*** is a path that starts and ends at the same vertex, and has at least one edge. An undirected graph is ***acyclic*** if no subgraph is a cycle; acyclic graphs are also called ***forests***. ***Trees*** are special graphs that can be defined in several different ways. You can easily prove by induction (hint, hint, hint) that the following definitions are equivalent.

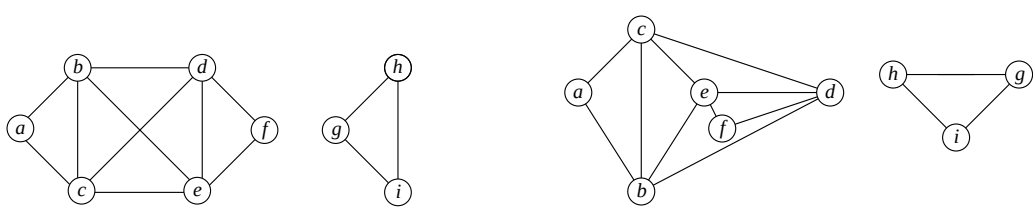
- A tree is a connected acyclic graph.
- A tree is a connected component of a forest.
- A tree is a connected graph with *at most* $V - 1$ edges.
- A tree is a minimal connected graph; removing any edge makes the graph disconnected.
- A tree is an acyclic graph with *at least* $V - 1$ edges.
- A tree is a maximal acyclic graph; adding an edge between any two vertices creates a cycle.

A ***spanning tree*** of a graph G is a subgraph that is a tree and contains every vertex of G. Of course, a graph can only have a spanning tree if it's connected. A ***spanning forest*** of G is a collection of spanning trees, one for each connected component of G.

Directed graphs can contain directed paths and directed cycles. A directed graph is ***strongly connected*** if there is a directed path from any vertex to any other. A directed graph is ***acyclic*** if it does not contain a directed cycle; directed acyclic graphs are often called ***dags***.

17.2 Abstract Representations and Examples

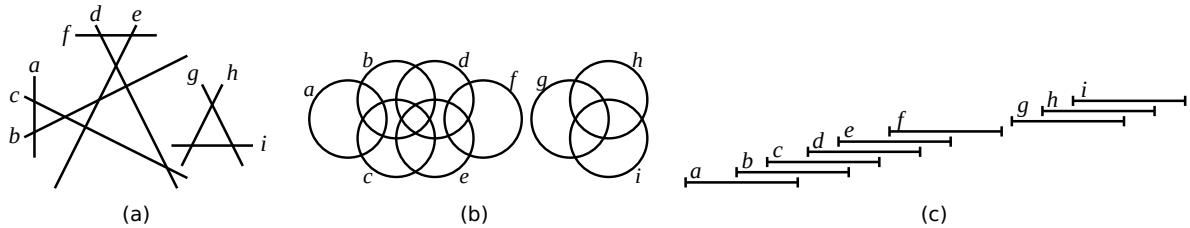
The most common way to visually represent graphs is by looking at an ***embedding***. An embedding of a graph maps each vertex to a point in the plane and each edge to a curve or straight line segment between the two vertices. A graph is ***planar*** if it has an embedding where no two edges cross. The same graph can have many different embeddings, so it is important not to confuse a particular embedding with the graph itself. In particular, planar graphs can have non-planar embeddings!



A non-planar embedding of a planar graph with nine vertices, thirteen edges, and two connected components, and a planar embedding of the same graph.

However, embeddings are not the only useful representation of graphs. For example, the ***intersection graph*** of a collection of objects has a node for every object and an edge for every intersecting pair. Whether a particular graph can be represented as an intersection graph depends on what kind of object you want to use for the vertices. Different types of objects—line segments, rectangles, circles, etc.—define different classes of graphs. One particularly useful type of intersection graph is an *interval graph*, whose vertices are intervals on the real line, with an edge between any two intervals that overlap.

Another good example is the ***dependency graph*** of a recursive algorithm. Dependency graphs are directed acyclic graphs. The vertices are all the distinct recursive subproblems that arise when executing the algorithm on a particular input. There is an edge from one subproblem to another if evaluating the



The example graph is also the intersection graph of (a) a set of line segments, (b) a set of circles, and (c) a set of intervals on the real line (stacked for visibility).

second subproblem requires a recursive evaluation of the first. For example, for the Fibonacci recurrence

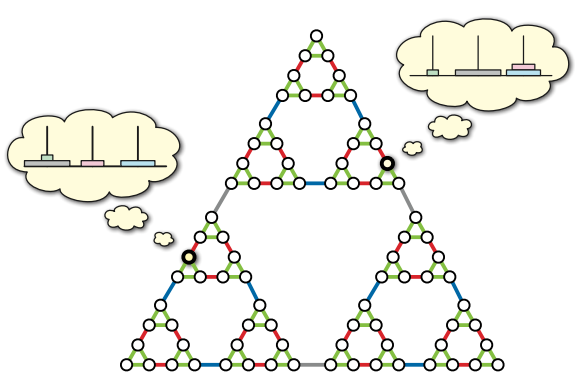
$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{otherwise,} \end{cases}$$

the vertices of the dependency graph are the integers $0, 1, 2, \ldots, n$, and the edges are the pairs $(i-1)\to i$ and $(i-2)\to i$ for every integer i between 2 and n. For the edit distance recurrence

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{l} Edit(i-1,j)+1, \\ Edit(i,j-1)+1, \\ Edit(i-1,j-1)+\left[A[i] \neq B[j]\right] \end{array} \right\} & \text{otherwise} \end{cases}$$

the dependency graph is an $m \times n$ grid with diagonals. Dynamic programming works efficiently for any recurrence that has a reasonably small dependency graph; a proper evaluation order ensures that each subproblem is visited *after* its predecessors.

Another interesting example is the ***configuration graph*** of a game, puzzle, or mechanism like tic-tac-toe, checkers, the Rubik's Cube, the Towers of Hanoi, or a Turing machine. The vertices of the configuration graph are all the valid configurations of the puzzle; there is an edge from one configuration to another if it is possible to transform one configuration into the other with a simple move. (Obviously, the precise definition depends on what moves are allowed.) Even for reasonably simple mechanisms, the configuration graph can be extremely complex, and we typically only have access to local information about the graph.



The configuration graph of the 4-disk Tower of Hanoi.

Finally, the ***finite-state automata*** used in formal language theory are just labeled directed graphs. A deterministic finite-state automaton is usually formally defined as a 5-tuple $M = (Q, \Sigma, \delta, q_0, A)$, where Q is a finite set of *states*, Σ is a finite set called the *alphabet*, $\delta : Q \times \Sigma \to Q$ is a *transition function*, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q$ is the set of *accepting states*. But it is often more useful to think of M as a directed graph G_M whose vertices are the states Q, and whose edges have the form $q \to \delta(q, x)$ for every state $q \in Q$ and character $x \in \Sigma$. Then basic questions about the language accepted by M can be phrased as questions about the graph G_M. For example, the language accepted by M is empty if and only if there is no path in in the graph G_M from the start state/vertex q_0 to an accepting state/vertex.

It's important not to confuse these examples/representations of graphs with the actual formal *definition*: A graph is always a pair of sets (V, E), where V is an arbitrary finite set, and E is a set of pairs (either ordered or unordered) of elements of V.

17.3 Graph Data Structures

In practice, graphs are represented by two data structures: *adjacency matrices*[2] and *adjacency lists*.

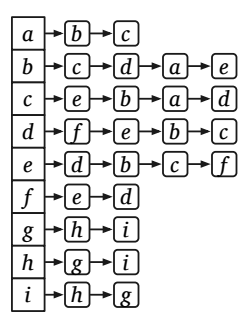
The ***adjacency matrix*** of a graph G is a $V \times V$ matrix, in which each entry indicates whether a particular edge is or is not in the graph:

$$A[i, j] := \big[(i, j) \in E\big].$$

For undirected graphs, the adjacency matrix is always *symmetric*: $A[i, j] = A[j, i]$. Since we don't allow edges from a vertex to itself, the diagonal elements $A[i, i]$ are all zeros.

Given an adjacency matrix, we can decide in $\Theta(1)$ time whether two vertices are connected by an edge just by looking in the appropriate slot in the matrix. We can also list all the neighbors of a vertex in $\Theta(V)$ time by scanning the corresponding row (or column). This is optimal in the worst case, since a vertex can have up to $V - 1$ neighbors; however, if a vertex has few neighbors, we may still have to examine every entry in the row to see them all. Similarly, adjacency matrices require $\Theta(V^2)$ space, regardless of how many edges the graph actually has, so it is only space-efficient for very *dense* graphs.

	a	b	c	d	e	f	g	h	i
a	0	1	1	0	0	0	0	0	0
b	1	0	1	1	1	0	0	0	0
c	1	1	0	1	1	0	0	0	0
d	0	1	1	0	1	1	0	0	0
e	0	1	1	1	0	1	0	0	0
f	0	0	0	1	1	0	0	0	0
g	0	0	0	0	0	0	0	1	0
h	0	0	0	0	0	0	1	0	1
i	0	0	0	0	0	0	1	1	0



Adjacency matrix and adjacency list representations for the example graph.

For *sparse* graphs—graphs with relatively few edges—adjacency lists are usually a better choice. An ***adjacency list*** is an array of linked lists, one list per vertex. Each linked list stores the neighbors of the corresponding vertex. For undirected graphs, each edge (u, v) is stored twice, once in u's neighbor list and once in v's neighbor list; for directed graphs, each edge is stored only once. Either way, the overall space required for an adjacency list is $O(V + E)$. Listing the neighbors of a node v takes $O(1 + \deg(v))$ time; just scan the neighbor list. Similarly, we can determine whether (u, v) is an edge in $O(1 + \deg(u))$ time by scanning the neighbor list of u. For undirected graphs, we can improve the time

[2] See footnote 1.

to $O(1 + \min\{\deg(u), \deg(v)\})$ by simultaneously scanning the neighbor lists of both u and v, stopping either we locate the edge or when we fall of the end of a list. This faster search strategy takes .

The adjacency list data structure should immediately remind you of hash tables with chaining; the two data structures are identical.[3] Just as with chained hash tables, we can make adjacency lists more efficient by using something other than a linked list to store the neighbors of each vertex. For example, if we use a hash table with constant load factor, when we can detect edges in $O(1)$ time, just as with an adjacency matrix. (Most hash give us only $O(1)$ *expected* time, but we can get $O(1)$ *worst-case* time using cuckoo hashing.)

The following table summarizes the performance of the various standard graph data structures. Stars* indicate expected amortized time bounds for maintaining dynamic hash tables.

	Adjacency matrix	Standard adjacency list (linked lists)	Adjacency list (hash tables)
Space	$\Theta(V^2)$	$\Theta(V+E)$	$\Theta(V+E)$
Time to test if $uv \in E$	$O(1)$	$O(1+\min\{\deg(u)+\deg(v)\}) = O(V)$	$O(1)$
Time to test if $u \to v \in E$	$O(1)$	$O(1+\deg(u)) = O(V)$	$O(1)$
Time to list the neighbors of v	$O(V)$	$O(1+\deg(v))$	$O(1+\deg(v))$
Time to list all edges	$\Theta(V^2)$	$\Theta(V+E)$	$\Theta(V+E)$
Time to add edge uv	$O(1)$	$O(1)$	$O(1)^*$
Time to delete edge uv	$O(1)$	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)^*$

At this point, one might reasonably wonder why anyone would ever use an adjacency matrix; after all, adjacency lists with hash tables support the same operations in the same time, using less space. Similarly, why would anyone use linked lists in an adjacency list structure to store neighbors, instead of hash tables? Although the main reason in practice is almost surely ***tradition***—If it was good enough for your grandfather's code, it should be good enough for yours!—there are some more principled arguments. One reason is that the standard adjacency lists are usually good enough; most graph algorithms never actually ask whether a given edge is present or absent! Another reason is that for sufficiently dense graphs, adjacency matrices are simpler and more efficient in practice, because they avoid the overhead of chasing pointers or computing hash functions.

But perhaps the most compelling reason is that many graphs are *implicitly* represented by adjacency matrices and standard adjacency lists. For example, intersection graphs are usually represented as a list of the underlying geometric objects. As long as we can test whether two objects intersect in constant time, we can apply any graph algorithm to an intersection graph by *pretending* that it is stored explicitly as an adjacency matrix.

Similarly, any data structure composed from records with pointers between them can be seen as a directed graph; graph algorithms can be applied to these data structures by *pretending* that the graph is stored in a standard adjacency list. Similarly, we can apply any graph algorithm to a configuration graph *as though* it were given to us as a standard adjacency list, provided we can enumerate all possible moves from a given configuration in constant time each. In both of these contexts, we can enumerate the edges leaving any vertex in time proportional to its degree, but we *cannot* necessarily determine in constant time if two vertices are connected. (Is there a pointer from this record to that record? Can we get from this configuration to that configuration in one move?) Thus, a standard adjacency list, with neighbors stored in linked lists, is the appropriate model data structure.

[3] For some reason, adjacency lists are always drawn with *horizontal* lists, while chained hash tables are always drawn with *vertical* lists. Don't ask me; I just work here.

17.4 Traversing Connected Graphs

To keep things simple, we'll consider only undirected graphs for the rest of this lecture, although the algorithms I'll describe also work for directed graphs.

Suppose we want to visit every node in a connected graph (represented either explicitly or implicitly). The simplest method to do this is an algorithm called *depth-first search*, which can be written either recursively or iteratively. It's exactly the same algorithm either way; the only difference is that we can actually see the 'recursion' stack in the non-recursive version. Both versions are initially passed a *source* vertex s.

```
RecursiveDFS(v):
  if v is unmarked
      mark v
      for each edge vw
          RecursiveDFS(w)
```

```
IterativeDFS(s):
  Push(s)
  while the stack is not empty
      v ← Pop
      if v is unmarked
          mark v
          for each edge vw
              Push(w)
```

Depth-first search is one (perhaps the most common) instance of a general family of graph traversal algorithms. The generic graph traversal algorithm stores a set of candidate edges in some data structure that I'll call a 'bag'. The only important properties of a 'bag' are that we can put stuff into it and then later take stuff back out. (In C++ terms, think of the 'bag' as a template for a real data structure.) A stack is a particular type of bag, but certainly not the only one. Here is the generic traversal algorithm:

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

This traversal algorithm clearly marks each vertex in the graph *at most* once. In order to show that it visits every node in a connected graph *at least* once, we modify the algorithm slightly; the modifications are highlighted in red. Instead of keeping vertices in the bag, the modified algorithm stores pairs of vertices. This modification allows us to remember, whenever we visit a vertex v for the first time, which previously-visited neighbor vertex put v into the bag. We call this earlier vertex the *parent* of v.

```
Traverse(s):
  put (∅,s) in bag
  while the bag is not empty
      take (p,v) from the bag          (⋆)
      if v is unmarked
          mark v
          parent(v) ← p
          for each edge vw             (†)
              put (v,w) into the bag   (⋆⋆)
```

Lemma 1. Traverse(s) *marks every vertex in any connected graph exactly once, and the set of pairs* $(v, parent(v))$ *with* $parent(v) \neq \varnothing$ *defines a spanning tree of the graph.*

Proof: The algorithm marks s. Let v be any vertex other than s, and let $(s, \ldots, u, v)$ be the path from s to v with the minimum number of edges. Since the graph is connected, such a path always exists. (If s and v are neighbors, then $u = s$, and the path has just one edge.) If the algorithm marks u, then it must put (u, v) into the bag, so it must later take (u, v) out of the bag, at which point v must be marked. Thus, by induction on the shortest-path distance from s, the algorithm marks every vertex in the graph, which implies that $parent(v)$ is well-defined for every vertex v.

The algorithm clearly marks every vertex at most once, so it must mark every vertex *exactly* once.

Call any pair $(v, parent(v))$ with $parent(v) \neq \varnothing$ a *parent edge*. For any node v, the path of parent edges $(v, parent(v), parent(parent(v)), \ldots)$ eventually leads back to s, so the set of parent edges form a connected graph. Clearly, both endpoints of every parent edge are marked, and the number of parent edges is exactly one less than the number of vertices. Thus, the parent edges form a spanning tree. □

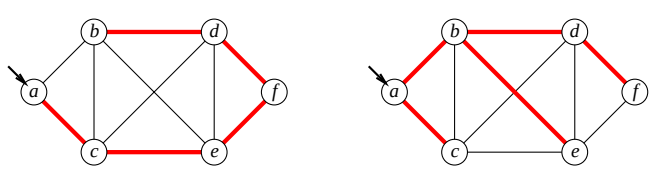
The exact running time of the traversal algorithm depends on how the graph is represented and what data structure is used as the 'bag', but we can make a few general observations. Because each vertex is marked at most once, the for loop (†) is executed at most V times. Each edge uv is put into the bag exactly twice; once as the pair (u, v) and once as the pair (v, u), so line ($\star\star$) is executed at most $2E$ times. Finally, we can't take more things out of the bag than we put in, so line ($\star$) is executed at most $2E + 1$ times.

17.5 Examples

Let's first assume that the graph is represented by a standard adjacency list, so that the overhead of the for loop (†) is only constant time per edge.

- If we implement the 'bag' using a *stack*, we recover our original depth-first search algorithm. Each execution of ($\star$) or ($\star\star$) takes constant time, so the algorithms runs in $\boldsymbol{O(V + E)}$ ***time*** . If the graph is connected, we have $V \leq E + 1$, and so we can simplify the running time to $O(E)$. The spanning tree formed by the parent edges is called a ***depth-first spanning tree***. The exact shape of the tree depends on the start vertex and on the order that neighbors are visited in the for loop (†), but in general, depth-first spanning trees are long and skinny.

- If we use a *queue* instead of a stack, we get ***breadth-first search***. Again, each execution of ($\star$) or ($\star\star$) takes constant time, so the overall running time for connected graphs is still $\boldsymbol{O(E)}$. In this case, the ***breadth-first spanning tree*** formed by the parent edges contains *shortest paths* from the start vertex s to every other vertex in its connected component. We'll see shortest paths again in a future lecture. Again, exact shape of a breadth-first spanning tree depends on the start vertex and on the order that neighbors are visited in the for loop (†), but in general, breadth-first spanning trees are short and bushy.



A depth-first spanning tree and a breadth-first spanning tree of one component of the example graph, with start vertex a.

- Now suppose the edges of the graph are weighted. If we implement the 'bag' using a *priority queue*, always extracting the minimum-weight edge in line ($\star$), the resulting algorithm is reasonably

called ***shortest-first search***. In this case, each execution of ($\star$) or ($\star\star$) takes $O(\log E)$ time, so the overall running time is $\boldsymbol{O(V + E \log E)}$, which simplifies to $\boldsymbol{O(E \log E)}$ if the graph is connected. For this algorithm, the set of parent edges form the ***minimum spanning tree*** of the connected component of s. Surprisingly, as long as all the edge weights are distinct, the resulting tree does *not* depend on the start vertex or the order that neighbors are visited; in this case, there is only one minimum spanning tree. We'll see minimum spanning trees again in the next lecture.

If the graph is represented using an adjacency matrix instead of an adjacency list, finding all the neighbors of each vertex in line (†) takes $O(V)$ time. Thus, depth- and breadth-first search each run in $O(V^2)$ time, and 'shortest-first search' runs in $O(V^2 + E \log E) = O(V^2 \log V)$ time.

17.6 Searching Disconnected Graphs

If the graph is disconnected, then TRAVERSE(s) only visits the nodes in the connected component of the start vertex s. If we want to visit all the nodes in every component, we can use the following 'wrapper' around our generic traversal algorithm. Since TRAVERSE computes a spanning tree of one component, TRAVERSEALL computes a spanning *forest* of the entire graph.

```
TRAVERSEALL(s):
  for all vertices v
      if v is unmarked
          TRAVERSE(v)
```

Some textbooks claim that this wrapper can only be used with depth-first search; they're wrong.

Exercises

1. Prove that the following definitions are all equivalent.
   - A tree is a connected acyclic graph.
   - A tree is a connected component of a forest.
   - A tree is a connected graph with *at most* $V - 1$ edges.
   - A tree is a minimal 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 maximal 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' of 'leaf', or any well-known properties of trees; your proof should follow entirely from the definitions.

3. 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 $G = T$.

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 A pecks pigeon B, which pecks pigeon C, which pecks pigeon A.

(a) Prove that any finite set of pigeons can be arranged in a row from left to right so that every pigeon pecks the pigeon immediately to its left. 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).

5. You are helping a group of ethnographers analyze some oral history data they have collected by interviewing members of a village to learn about the lives of people lived there over the last two hundred years. From the interviews, you have learned about a set of people, all now deceased, whom we will denote $P_1, P_2, \ldots, P_n$. The ethnographers have collected several facts about the lifespans of these people. Specifically, for some pairs (P_i, P_j), the ethnographers have learned one of the following facts:

   (a) P_i died before P_j was born.

   (b) P_i and P_j were both alive at some moment.

   Naturally, the ethnographers are not sure that their facts are correct; memories are not so good, and all this information was passed down by word of mouth. So they'd like you to determine whether the data they have collected is at least internally consistent, in the sense that there could have existed a set of people for which all the facts they have learned simultaneously hold.

   Describe and analyze and algorithm to answer the ethnographers' problem. Your algorithm should either output possible dates of birth and death that are consistent with all the stated facts, or it should report correctly that no such dates exist.

6. Let $G = (V, E)$ be a given directed graph.

   (a) The *transitive closure* G^T is a directed graph with the same vertices as G, that contains any edge $u \to v$ if and only if there is a directed path from u to v in G. Describe an efficient algorithm to compute the transitive closure of G.

   (b) The *transitive reduction* G^{TR} is the smallest graph (meaning fewest edges) whose transitive closure is G^T. Describe an efficient algorithm to compute the transitive reduction of G.

7. 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.

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

   (a) Prove that a connected graph G has an Euler tour if and only if every vertex has even degree.

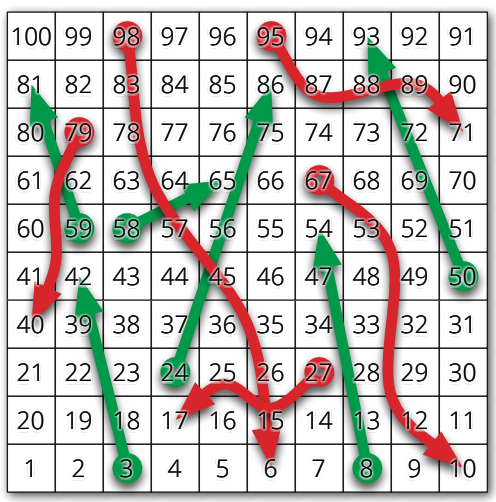
   (b) Describe and analyze an algorithm to compute an Euler tour in a given graph, or correctly report that no such graph exists.

9. 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.]*

10. ***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.



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

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.

11. ***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.[4] 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.

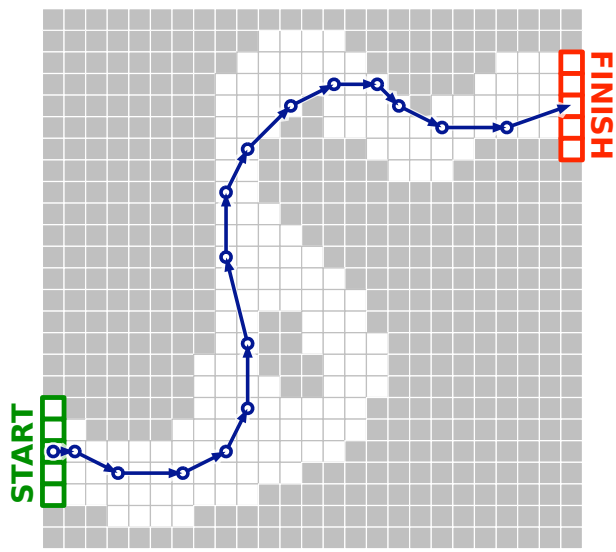
[4]The actual game is a bit more complicated than the version described here. See http://harmmade.com/vectorracer/ for an excellent online version.

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.

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. *[Hint: Build a graph. What are the vertices? What are the edges? What problem is this?]*

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



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

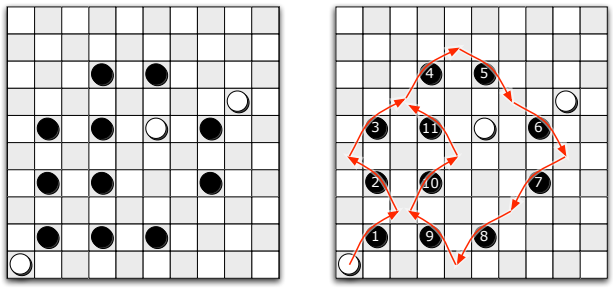
*12. Draughts/checkers is a game played on an $m \times m$ grid of squares, alternately colored light and dark. (The game is usually played on an 8×8 or 10×10 board, but the rules easily generalize to any board size.) Each dark square is occupied by at most one game piece (usually called a *checker* 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.

Consider the following simple version of the game, essentially American checkers or British draughts, but where every piece is a king.[5] Pieces can be moved in any of the four diagonal directions, either one or two steps at a time. On each turn, a player either *moves* one of her pieces

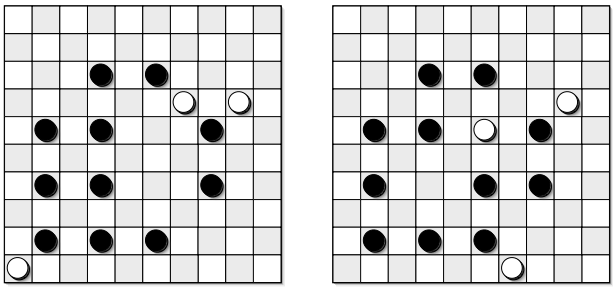
[5]Most other variants of draughts have 'flying kings', which behave *very* differently than what's described here. In particular, if we allow flying kings, it is actually NP-hard to determine which move captures the most enemy pieces. The most common international version of draughts also has a forced-capture rule, which *requires* each player to capture the maximum possible number of enemy pieces in each move. Thus, just following the rules is NP-hard.

one step diagonally into an empty square, or makes a series of *jumps* with one of her checkers. In a single jump, a 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. Multiple jumps are allowed in a single turn as long as they are made by the same piece. A player wins if her opponent has no pieces left on the board.

Describe an algorithm that correctly determines 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 8. Parity, parity, parity.]*

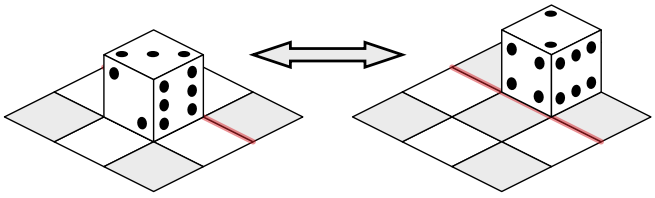


White wins in one turn.



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

13. 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 top of a table; the die always rests on and exactly covers one square. 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.

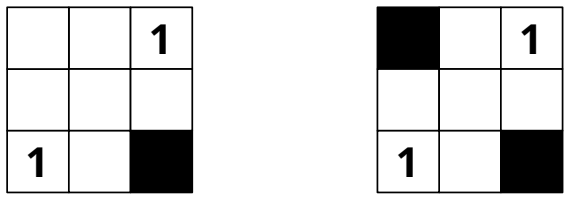


Rolling a die.

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 of pips 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.

For example, here are two rolling die mazes. Black squares are blocked. The maze on the left can be solved by placing the die on the lower left square with 1 pip on the top face, and then rolling it north, then north, then east, then east. The maze on the right is not solvable.



Two rolling die mazes. Only the maze on the left is 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 only be solved if the initial position is chosen correctly.]