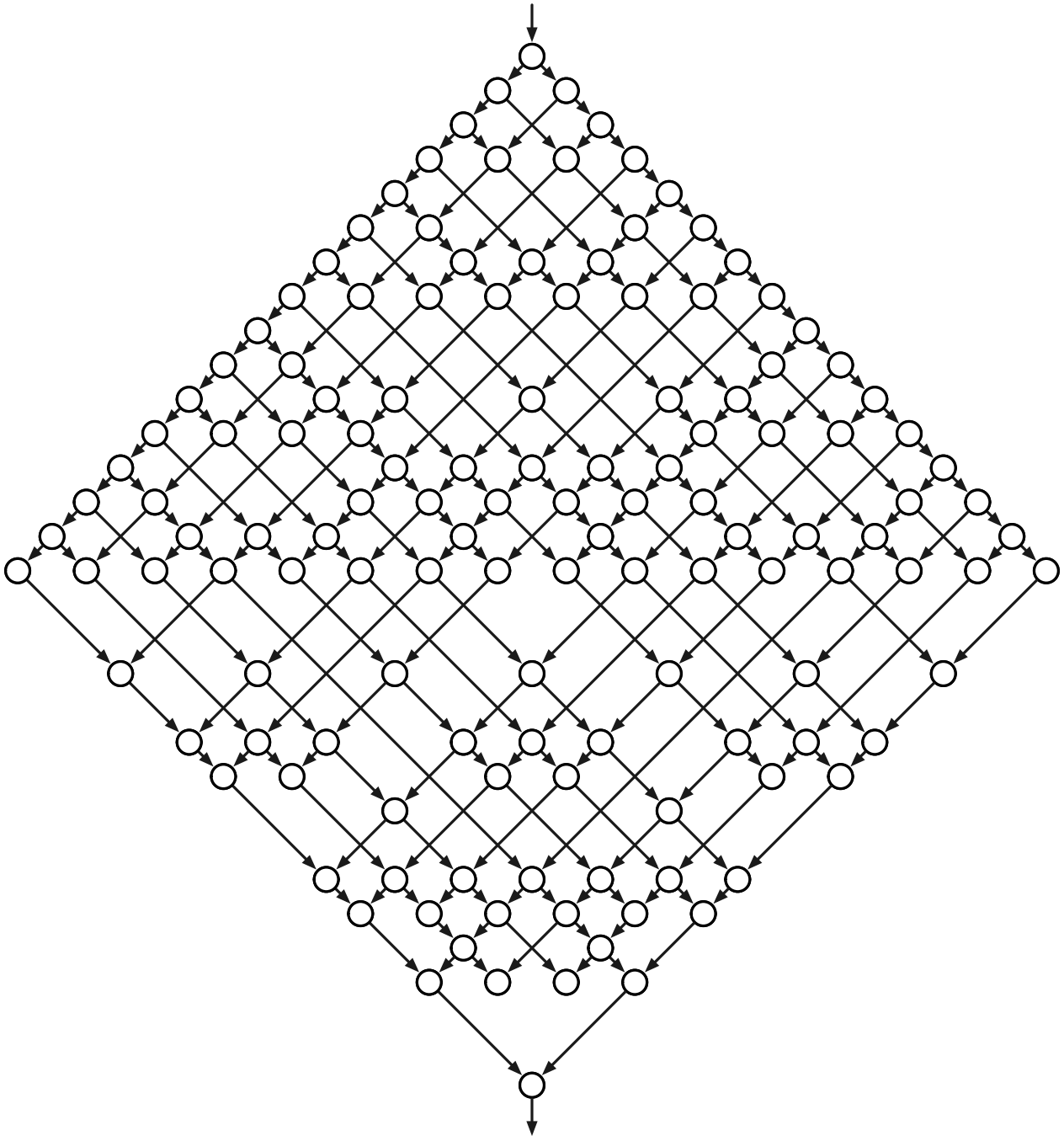
Graphs



Thus you see, most noble Sir, how this type of solution bears little relationship to mathematics, and I do not understand why you expect a mathematician to produce it, rather than anyone else, for the solution is based on reason alone, and its discovery does not depend on any mathematical principle. Because of this, I do not know why even questions which bear so little relationship to mathematics are solved more quickly by mathematicians than by others.

— Leonhard Euler, describing the Königsburg bridge problem
in a letter to Carl Leonhard Gottlieb Ehler (April 3, 1736)

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

18 Basic Graph Algorithms

18.1 Definitions

A ***graph*** is normally defined as a pair of sets (V, E), where V is a set of arbitrary objects called ***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 usually write 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 usually write $u \to v$ instead of (u, v) to denote the directed edge from u to v.

The definition of a graph as a pair of *sets* forbids graphs with loops (edges from a vertex to itself) and/or parallel edges (multiple edges with the same endpoints). Graphs *without* loops and parallel edges are often called ***simple*** graphs; non-simple graphs are sometimes called ***multigraphs***. Despite the formal definitional gap, most algorithms for simple graphs extend to non-simple graphs with little or no modification.

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 any undirected graph we have $0 \le E \le \binom{V}{2}$, and in any directed graph we have $0 \le E \le V(V-1)$.

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 ***walk*** in a graph is a sequence of edges, where each successive pair of edges shares one vertex; a walk is called a ***path*** if it visits each vertex at most once. An undirected graph is ***connected*** if there is a walk (and therefore a path) between any two vertices. A disconnected graph consists of several ***components***, which are its maximal connected subgraphs. Two vertices are in the

[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.

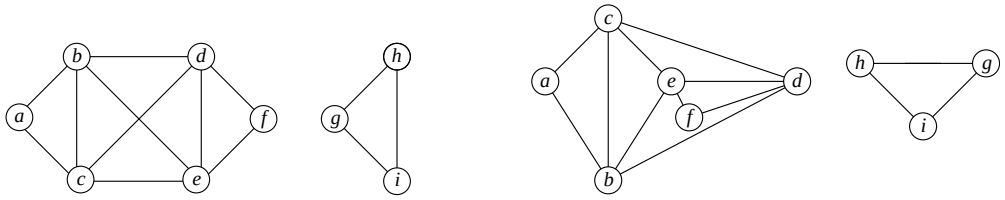
same component if and only if there is a path between them. Components are sometimes called "connected components", but this usage is redundant; components are connected by definition.

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***. A ***tree*** is a connected acyclic graph, or equivalently, one component of a forest. A ***spanning tree*** of a graph G is a subgraph that is a tree and contains every vertex of G. A graph has a spanning tree if and only if it is 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***.

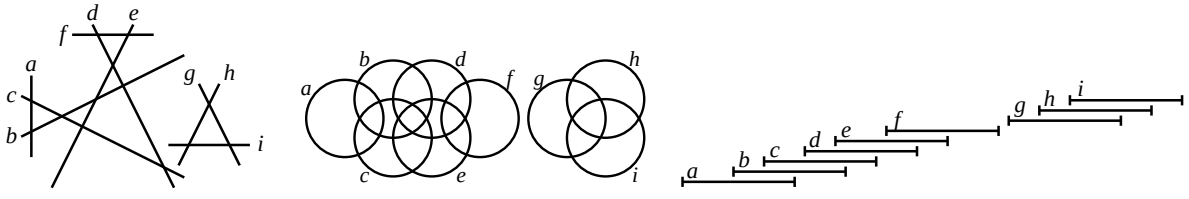
18.2 Abstract Representations and Examples

The most common way to visually represent graphs is with an ***embedding***. An embedding of a graph maps each vertex to a point in the plane (typically drawn as a small circle) 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 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.



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).

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

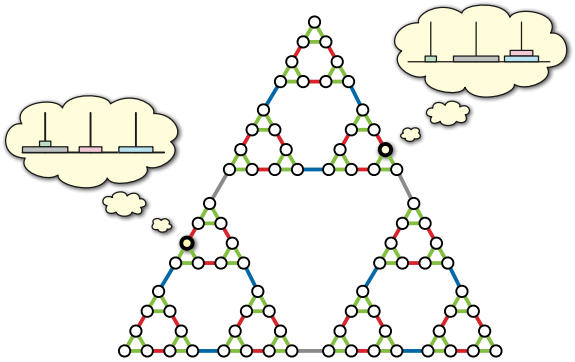
$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{otherwise,} \end{cases}$$

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. As a more complex example, consider the following recurrence, which solves a certain sequence-alignment problem called *edit distance*; see the dynamic programming notes for details:

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

The dependency graph of this recurrence is an $m \times n$ grid of vertices (i, j) connected by vertical edges $(i-1, j)\to(i, j)$, horizontal edges $(i, j-1)\to(i, j)$, and diagonal edges $(i-1, j-1)\to(i, j)$. 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 configuration graph.



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

Finite-state automata used in formal language theory can be modeled as labeled directed graphs. Recall that a deterministic finite-state automaton is formally defined as a 5-tuple $M = (\Sigma, Q, s, A, \delta)$, where Σ is a finite set called the *alphabet*, Q is a finite set of *states*, $s \in Q$ is

the *start state*, $A \subseteq Q$ is the set of *accepting states*, and $\delta : Q \times \Sigma \to Q$ is a *transition function*. 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 $q \to \delta(q, a)$ for every state $q \in Q$ and symbol $a \in \Sigma$. 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 G_M from the start state/vertex q_0 to an accepting state/vertex.

Finally, sometimes one graph can be used to implicitly represent other larger graphs. A good example of this implicit representation is the subset construction used to convert NFAs into DFAs. The subset construction can be generalized to *arbitrary* directed graphs as follows. Given *any* directed graph $G = (V, E)$, we can define a new directed graph $G' = (2^V, E')$ whose vertices are all *subsets* of vertices in V, and whose edges E' are defined as follows:

$$E' := \left\{ A \to B \;\middle|\; u \to v \in E \text{ for some } u \in A \text{ and } v \in B \right\}$$

We can mechanically translate this definition into an algorithm to construct G' from G, but strictly speaking, this construction is unnecessary, because ***G is already an implicit representation of G′***. Viewed in this light, the *incremental* subset construction used to convert NFAs to DFAs without unreachable states is just a breadth-first search of the implicitly-represented DFA.

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

18.3 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:

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

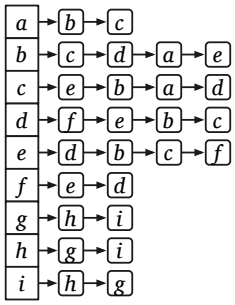
For undirected graphs, the adjacency matrix is always *symmetric*: $A[i, j] = A[j, i]$. Since we don't allow edges from a vertex to itself, the diagonal elements $A[i, i]$ are all zeros.

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.

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

[2] See footnote 1.

	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.

undirected graphs, we can improve the time 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.

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

[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.

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.

18.4 Traversing Connected Graphs

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

Suppose we want to visit every node in a connected graph (represented either explicitly or implicitly). Perhaps the simplest graph-traversal algorithm is *depth-first search*. This algorithm 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 just one (perhaps the most common) species 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 $parent(v) \neq \emptyset$ 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 $parent(v) \neq \emptyset$ 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. □

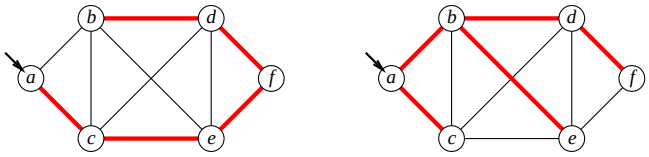
The exact running time of the traversal algorithm depends on how the graph is represented and what data structure is used as the 'bag', but we can make a few general observations. Because each vertex is marked at most once, the for loop (†) is executed at most V times. Each edge uv is put into the bag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

18.5 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 (⋆) or (⋆⋆) takes constant time, so the algorithms runs in ***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 (⋆) or (⋆⋆) 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 (⋆), the resulting algorithm is reasonably called ***shortest-first search***. In this case, each execution of (⋆) or (⋆⋆) 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.

18.6 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)
```

Surprisingly, a few well-known algorithms 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 one 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" or "leaf", or any well-known properties of trees; your proof should follow entirely from the definitions of "connected" and "acyclic".

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. 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.

6. 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.

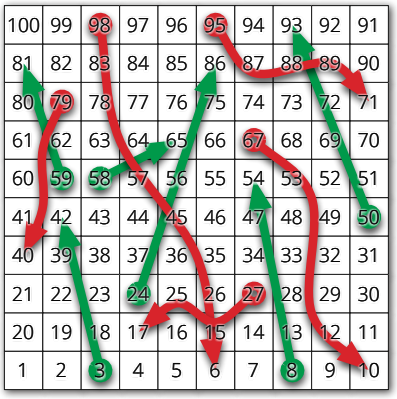
   (b) Describe and analyze an algorithm to compute an Euler tour in a given graph, or correctly report that no such graph exists.

7. 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.]*

8. ***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.



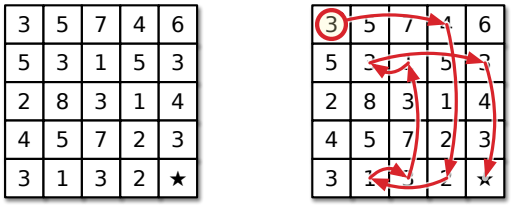
A typical Snakes and Ladders board.
Upward straight arrows are ladders; downward wavy arrows are snakes.

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.

9. 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.

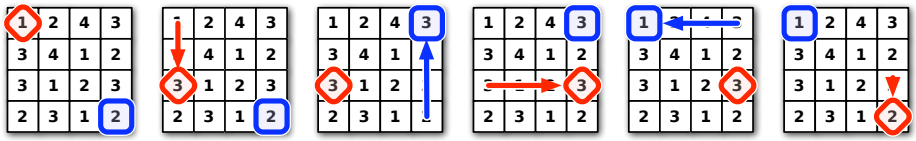
   Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution.



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

10. The following puzzle was invented by the infamous Mongolian puzzle-warrior Vidrach Itky Leda 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.

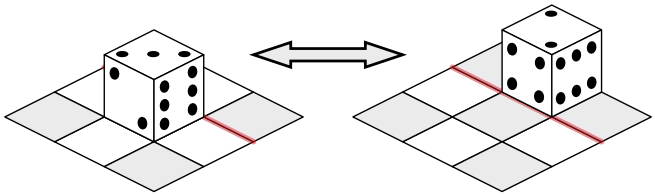
    In a single turn, you may move either token up, right, down, or left *by a distance determined by the* ***other*** *token*. For example, if the red token is on a square labeled 3, then you may move the blue token 3 steps up, 3 steps left, 3 steps right, or 3 steps down. However, you may not move a token off the grid or to the same square as the other token.



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

    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 above, your algorithm would return the number 5.

11. 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.

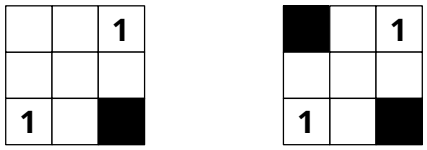


Rolling a die.

    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.]

12. ***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.

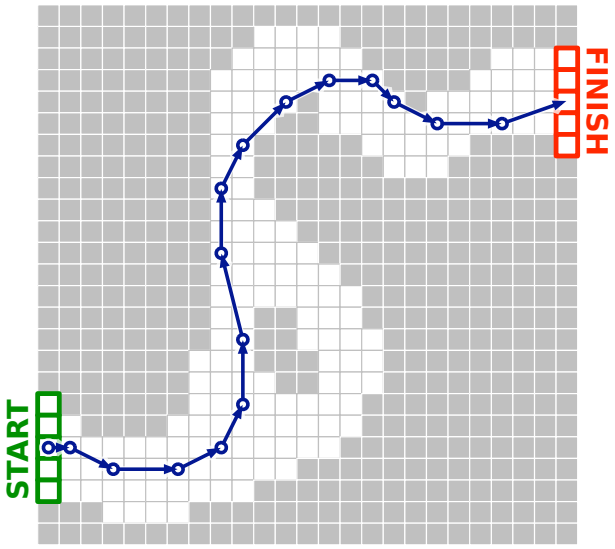
Each car has a *position* and a *velocity*, both with integer x- and y-coordinates. A subset of grid squares is marked as the *starting area*, and another subset is marked as the *finishing area*. The initial position of each car is chosen by the player somewhere in the starting area; the initial velocity 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.

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

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



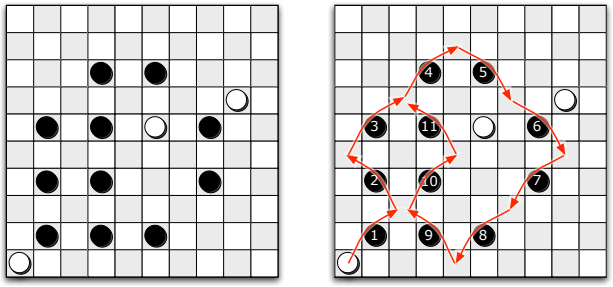
A 16-step Racetrack run, on a 25×25 track. This is *not* the shortest run on this track.

*13. Draughts (also known as 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.

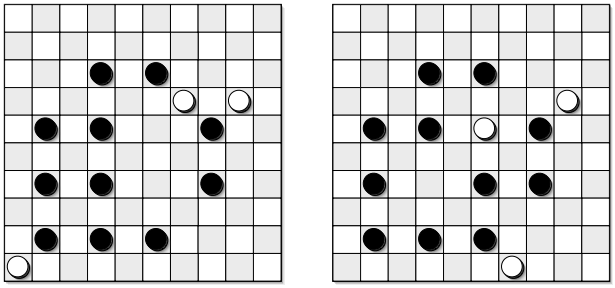
Consider the following simple version of the game, essentially American checkers or British draughts, but where every piece is a king.[5] Pieces can be moved in any of the four diagonal directions, either one or two steps at a time. On each turn, a player either *moves* one of her pieces one step diagonally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

[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!

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* **??***. Parity, parity, parity.]*



White wins in one turn.



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

Ts'ui Pe must have said once: I am withdrawing to write a book.
And another time: I am withdrawing to construct a labyrinth.
Every one imagined two works;
to no one did it occur that the book and the maze were one and the same thing.

— Jorge Luis Borges, "El jardín de senderos que se bifurcan" (1942)
English translation ("The Garden of Forking Paths") by Donald A. Yates (1958)

"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)

At some point, the learning stops and the pain begins.

— Rao Kosaraju

19 Depth-First Search

Recall from the previous lecture the recursive formulation of depth-first search in undirected graphs.

```
DFS(v):
  if v is unmarked
      mark v
      for each edge vw
          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 define parent pointers and other useful information about the vertices. This additional information is computed by 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)
```

We can search any *connected* graph by unmarking all vertices and then calling DFS(s) for an arbitrary start vertex s. As we argued in the previous lecture, the subgraph of all parent edges $v \to parent(v)$ defines a spanning tree of the graph, which we consider to be rooted at the start vertex s.

Lemma 1. *Let T be a depth-first spanning tree of a connected undirected graph G, computed by calling DFS(s). For any node v, the vertices that are marked during the execution of DFS(v) are the proper descendants of v in T.*

Proof: T is also the recursion tree for DFS(s). □

Lemma 2. *Let T be a depth-first spanning tree of a connected undirected graph G. For every edge vw in G, either v is an ancestor of w in T, or v is a descendant of w in T.*

Proof: Assume without loss of generality that v is marked before w. Then w is unmarked when DFS(v) is invoked, but marked when DFS(v) returns, so the previous lemma implies that w is a proper descendant of v in T. □

Lemma 2 implies that any depth-first spanning tree T divides the edges of G into two classes: *tree* edges, which appear in T, and *back* edges, which connect some node in T to one of its ancestors.

19.1 Counting and Labeling Components

For graphs that might be disconnected, we can compute a depth-first spanning *forest* by calling the following wrapper function; again, we introduce a generic black-box subroutine PREPROCESS to perform any necessary preprocessing for the POSTVISIT and POSTVISIT functions.

```
DFSALL(G):
  PREPROCESS(G)
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          DFS(v)
```

With very little additional effort, we can count the components of a graph; we simply increment a counter inside the wrapper function. Moreover, we can also record which component contains each vertex in the graph by passing this counter to DFS. The single line $comp(v) \leftarrow count$ is a trivial example of PREVISIT. (And the absence of code after the for loop is a vacuous example of POSTVISIT.)

```
COUNTANDLABEL(G):
  count ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          count ← count + 1
          LABELCOMPONENT(v, count)
  return count
```

```
LABELCOMPONENT(v, count):
  mark v
  comp(v) ← count
  for each edge vw
      if w is unmarked
          LABELCOMPONENT(w, count)
```

It should be emphasized that depth-first search is not specifically required here; any other instantiation of our earlier generic traversal algorithm ("whatever-first search") can be used to count components in the same asymptotic running time. However, most of the other algorithms we consider in this note *do* specifically require *depth*-first search.

19.2 Preorder and Postorder Labeling

You should already be familiar with preorder and postorder traversals of rooted trees, both of which can be computed using from depth-first search. Similar traversal orders can be defined for arbitrary graphs by passing around a counter as follows:

```
PREPOSTLABEL(G):
  for all vertices v
      unmark v
  clock ← 0
  for all vertices v
      if v is unmarked
          clock ← LABELCOMPONENT(v, clock)
```

```
LABELCOMPONENT(v, clock):
  mark v
  pre(v) ← clock
  clock ← clock + 1
  for each edge vw
      if w is unmarked
          clock ← LABELCOMPONENT(w, clock)
  post(v) ← clock
  clock ← clock + 1
  return clock
```

Equivalently, if we're willing to use (shudder) global variables, we can use our generic depth-first-search algorithm with the following subroutines PREPROCESS, PREVISIT, and POSTVISIT.

```
PREPROCESS(G):
  clock ← 0
```

```
PREVISIT(v):
  pre(v) ← clock
  clock ← clock + 1
```

```
POSTVISIT(v):
  post(v) ← clock
  clock ← clock + 1
```

Consider two vertices u and v, where u is marked after v. Then we must have $pre(u) < pre(v)$. Moreover, Lemma 1 implies that if v is a descendant of u, then $post(u) > post(v)$, and otherwise, $pre(v) > post(u)$. Thus, for any two vertices u and v, the intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or nested; in particular, if uv is an edge, Lemma 2 implies that the intervals must be nested.

19.3 Directed Graphs and Reachability

The recursive algorithm requires only one minor change to handle directed graphs:

```
DFSALL(G):
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          DFS(v)
```

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

However, we can no longer use this modified algorithm to count components. Suppose G is a single directed path. Depending on the order that we choose to visit the nodes in DFSALL, we may discover any number of "components" between 1 and n. All that we can guarantee is that the "component" numbers computed by DFSALL do not increase as we traverse the path. In fact, the real problem is that the *definition* of "component" is only suitable for *undirected* graphs.

Instead, for directed graphs we rely on a more subtle notion of ***reachability***. We say that a node v is *reachable* from another node u in a directed graph G—or more simply, that u can reach v—if and only if there is a directed path in G from u to v. Let ***Reach(u)*** denote the set of vertices that are reachable from u (including u itself). A simple inductive argument proves that $Reach(u)$ is precisely the subset of nodes that are marked by calling DFS(u).

19.4 Directed Acyclic Graphs

A ***directed acyclic graph*** or ***dag*** is a directed graph with no directed cycles. Any vertex in a dag that has no incoming vertices is called a ***source***; any vertex with no outgoing edges is called a ***sink***. Every dag has at least one source and one sink (Do you see why?), but may have more than one of each. For example, in the graph with n vertices but no edges, every vertex is a source and every vertex is a sink.

We can check whether a given directed graph G is a dag in $O(V + E)$ time as follows. First, to simplify the algorithm, we add a single artificial source s, with edges from s to every other vertex. Let $G + s$ denote the resulting augmented graph. Because s has no outgoing edges, no directed cycle in $G + s$ goes through s, which implies that $G + s$ is a dag if and only if G is a dag. Then we preform a depth-first search of $G + s$ starting at the new source vertex s; by construction every other vertex is reachable from s, so this search visits every node in the graph.

Instead of vertices being merely marked or unmarked, each vertex has one of three statuses—NEW, ACTIVE, or DONE—which depend on whether we have started or finished the recursive depth-first search at that vertex. (Since this algorithm never uses parent pointers, I've removed the line "$parent(w) \leftarrow v$".)

```
IsAcyclic(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      status(v) ← NEW
  return IsAcyclicDFS(s)
```

```
IsAcyclicDFS(v):
  status(v) ← ACTIVE
  for each edge v→w
      if status(w) = ACTIVE
          return FALSE
      else if status(w) = NEW
          if IsAcyclicDFS(w) = FALSE
              return FALSE
  status(v) ← DONE
  return TRUE
```

Suppose the algorithm returns FALSE. Then the algorithm must discover an edge $v \to w$ such that $status(w) = \text{ACTIVE}$. The active vertices are precisely the vertices currently on the recursion stack, which are all ancestors of the current vertex v. Thus, there is a directed path from w to v, and so the graph has a directed cycle.

On the other hand, suppose G has a directed cycle. Let w be the first vertex in this cycle that we visit, and let $v \to w$ be the edge leading into v in the same cycle. Because there is a directed path from w to v, we must call IsAcyclicDFS(v) during the execution of IsAcyclicDFS(w), unless we discover some other cycle first. During the execution of IsAcyclicDFS(v), we consider the edge $v \to w$, discover that $status(w) = \text{ACTIVE}$. The return value FALSE bubbles up through all the recursive calls to the top level.

We conclude that IsAcyclic(G) returns TRUE if and only if G is a dag.

19.5 Topological Sort

A ***topological ordering*** of a directed graph G is a total order $\prec$ on the vertices such that $u \prec v$ for every edge $u \to v$. Less formally, a topological ordering arranges the vertices along a horizontal line so that all edges point from left to right. A topological ordering is clearly impossible if the graph G has a directed cycle—the rightmost vertex of the cycle would have an edge pointing to the left! On the other hand, every dag has a topological order, which can be computed by either of the following algorithms.

```
TopologicalSort(G):
  n ← |V|
  for i ← 1 to n
      v ← any source in G
      S[i] ← v
      delete v and all edges leaving v
  return S[1..n]
```

```
TopologicalSort(G):
  n ← |V|
  for i ← n down to 1
      v ← any sink in G
      S[1] ← v
      delete v and all edges entering v
  return S[1..n]
```

The correctness of these algorithms follow inductively from the observation that *deleting* a vertex cannot *create* a cycle.

This simple algorithm has two major disadvantages. First, the algorithm actually destroys the input graph. This destruction can be avoided by simply marking the "deleted" vertices, instead of actually deleting them, and defining a vertex to be a source (sink) if none of its incoming (outgoing) edges come from (lead to) an unmarked vertex. The more serious problem is that finding a source vertex seems to require $\Theta(V)$ time in the worst case, which makes the running time of this algorithm $\Theta(V^2)$. In fact, a careful implementation of this algorithm computes a topological ordering in $O(V + E)$ time without removing any edges.

But there is a simpler linear-time algorithm based on our earlier algorithm for deciding whether a directed graph is acyclic. The new algorithm is based on the following observation:

Lemma 3. *For any directed acyclic graph G, the first vertex marked* Done *by* IsAcyclic(G) *must be a sink.*

Proof: Let v be the first vertex marked Done during an execution of IsAcyclic. For the sake of argument, suppose v has an outgoing edge $v \to w$. When IsAcyclicDFS first considers the edge $v \to w$, there are three cases to consider.

- If $status(w) =$ Done, then w is marked Done before v, which contradicts the definition of v.
- If $status(w) =$ New, the algorithm calls TopoSortDFS(w), which (among other computation) marks w Done. Thus, w is marked Done before v, which contradicts the definition of v.
- If $status(w) =$ Active, then G has a directed cycle, contradicting our assumption that G is acyclic.

In all three cases, we have a contradiction, so v must be a sink. □

Thus, to topologically sort a dag G, it suffice to list the vertices in the *reverse* order of being marked Done. For example, we could push each vertex onto a stack when we mark it Done, and then pop every vertex off the stack.

```
TopologicalSort(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      status(v) ← New
  TopoSortDFS(s)
  for i ← 1 to V
      S[i] ← Pop
  return S[1..V]
```

```
TopoSortDFS(v):
  status(v) ← Active
  for each edge v→w
      if status(w) = New
          ProcessBackwardDFS(w)
      else if status(w) = Active
          fail gracefully
  status(v) ← Done
  Push(v)
  return True
```

But maintaining a separate data structure is actually overkill. In most applications of topological sort, an explicit sorted list of the vertices is not our actual goal; instead, we want to performing some fixed computation at each vertex of the graph, either in topological order or in reverse topological order. In this case, it is not necessary to *record* the topological order. To process the graph in *reverse* topological order, we can just process each vertex at the end of its recursive depth-first search.

```
ProcessBackward(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      status(v) ← New
  ProcessBackwardDFS(s)
```

```
ProcessBackwardDFS(v):
  status(v) ← Active
  for each edge v→w
      if status(w) = New
          ProcessBackwardDFS(w)
      else if status(w) = Active
          fail gracefully
  status(v) ← Done
  Process(v)
```

If we already *know* that the input graph is acyclic, we can simplify the algorithm by simply marking vertices instead of labeling them Active or Done.

```
ProcessDagBackward(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      unmark v
  ProcessDagBackwardDFS(s)
```

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

Except for the addition of the artificial source vertex s, which we need to ensure that every vertex is visited, this is just the standard depth-first search algorithm, with PostVisit renamed to Process!

The simplest way to process a dag in *forward* topological order is to construct the ***reversal*** of the input graph, which is obtained by replacing each each $v \to w$ with its reversal $w \to v$. Reversing a directed cycle gives us another directed cycle with the opposite orientation, so the reversal of a dag is another dag. Every source in G becomes a sink in the reversal of G and vice versa; it follows inductively that every topological ordering for the reversal of G is the reversal of a topological ordering of G. The reversal of any directed graph can be computed in $O(V + E)$ time; the details of this construction are left as an easy exercise.

19.6 Every Dynamic Programming Algorithm?

Our topological sort algorithm is arguably the model fora wide class of dynamic programming algorithms. Recall that the ***dependency graph*** of a recurrence has a vertex for every recursive subproblem and an edge from one subproblem to another if evaluating the first subproblem requires a recursive evaluation of the second. The dependency graph must be acyclic, or the naïve recursive algorithm would never halt. Evaluating any recurrence with memoization is exactly the same as performing a depth-first search of the dependency graph. In particular, a vertex of the dependency graph is 'marked' if the value of the corresponding subproblem has already been computed, and the black-box subroutine Process is a placeholder for the actual value computation.

However, there are some minor differences between most dynamic programming algorithms and topological sort.

- First, in most dynamic programming algorithms, the dependency graph is *implicit*—the nodes and edges are not given as part of the input. But this difference really is minor; as long as we can enumerate recursive subproblems in constant time each, we can traverse the dependency graph exactly as if it were explicitly stored in an adjacency list.
- More significantly, most dynamic programming recurrences have highly structured dependency graphs. For example, the dependency graph for edit distance is a regular grid with diagonals, and the dependency graph for optimal binary search trees is an upper triangular grid with all possible rightward and upward edges. This regular structure lets us hard-wire a topological order directly into the algorithm, so we don't have to compute it at run time.

Conversely, we can use depth-first search to build dynamic programming algorithms for problems with less structured dependency graphs. For example, consider the ***longest path*** problem, which asks for the path of *maximum* total weight from one node s to another node t in a directed graph G with weighted edges. The longest path problem is NP-hard in general directed graphs, by an easy reduction from the traveling salesman problem, but it is easy to solve in linear time if the input graph G is acyclic, as follows. For any node s, let $LLP(s,t)$ denote the Length of the Longest Path in G from s to t. If G is a dag, this function satisfies the recurrence

$$LLP(s,t) = \begin{cases} 0 & \text{if } s = t, \\ \max_{s \to v} (\ell(s \to v) + LLP(v,t)) & \text{otherwise,} \end{cases}$$

where $\ell(v \to w)$ is the given weight ("length") of edge $v \to w$. In particular, if s is a *sink* but not equal to t, then $LLP(s,t) = \infty$. The dependency graph for this recurrence is the input graph G itself: subproblem $LLP(u,t)$ depends on subproblem $LLP(v,t)$ if and only if $u \to v$ is an edge in G. Thus, we can evaluate this recursive function in $O(V+E)$ time by performing a depth-first search of G, starting at s.

```
LongestPath(s, t):
  if s = t
      return 0

  if LLP(s) is undefined
      LLP(s) ← ∞
      for each edge s→v
          LLP(s) ← max{LLP(v), ℓ(s→v) + LongestPath(v, t)}
  return LLP(s)
```

A surprisingly large number of dynamic programming problems (but *not* all) can be recast as optimal path problems in the associated dependency graph.

19.7 Strong Connectivity

Let's go back to the proper definition of connectivity in directed graphs. Recall that one vertex u can *reach* another vertex v in a graph G if there is a directed path in G from u to v, and that $Reach(u)$ denotes the set of all vertices that u can reach. Two vertices u and v are ***strongly connected*** if u can reach v and v can reach u. Tedious definition-chasing implies that strong connectivity is an equivalence relation over the set of vertices of any directed graph, just as connectivity is for undirected graphs. The equivalence classes of this relation are called the ***strongly connected components*** (or more simply, the ***strong components***) of G. If G has a single strong component, we call it ***strongly connected***. G is a directed acyclic graph if and only if every strong component of G is a single vertex.

It is straightforward to compute the strong component containing a single vertex v in $O(V+E)$ time. First we compute $\mathit{Reach}(v)$ by calling $\mathit{DFS}(v)$. Then we compute $\mathit{Reach}^{-1}(v) = \{u \mid v \in \mathit{Reach}(u)\}$ by searching the reversal of G. Finally, the strong component of v is the intersection $\mathit{Reach}(v) \cap \mathit{Reach}^{-1}(v)$. In particular, we can determine whether the entire graph is strongly connected in $O(V+E)$ time.

We can compute *all* the strong components in a directed graph by wrapping the single-strong-component algorithm in a wrapper function, just as we did for depth-first search in undirected graphs. However, the resulting algorithm runs in $O(VE)$ time; there are at most V strong components, and each requires $O(E)$ time to discover. Surely we can do better! After all, we only need $O(V+E)$ time to decide whether every strong component is a single vertex.

19.8 Strong Components in Linear Time

For any directed graph G, the ***strong component graph scc(G)*** is another directed graph obtained by contracting each strong component of G to a single (meta-)vertex and collapsing parallel edges. The strong component graph is sometimes also called the *meta-graph* or *condensation* of G. It's not hard to prove (hint, hint) that $\mathit{scc}(G)$ is always a dag. Thus, in principle, it is possible to topologically order the strong components of G; that is, the vertices can be ordered so that every *backward* edge joins two edges in the same strong component.

Let C be any strong component of G that is a sink in $\mathit{scc}(G)$; we call C a *sink component*. Every vertex in C can reach every other vertex in C, so a depth-first search from any vertex in C visits every vertex in C. On the other hand, because C is a sink component, there is no edge from C to any other strong component, so a depth-first search starting in C visits *only* vertices in C. So if we can compute all the strong components as follows:

```
StrongComponents(G):
  count ← 0
  while G is non-empty
      count ← count + 1
      v ← any vertex in a sink component of G
      C ← OneComponent(v, count)
      remove C and incoming edges from G
```

At first glance, finding a vertex in a sink component *quickly* seems quite hard. However, we can quickly find a vertex in a *source* component using the standard depth-first search. A source component is a strong component of G that corresponds to a source in $\mathit{scc}(G)$. Specifically, we compute *finishing times* (otherwise known as post-order labeling) for the vertices of G as follows.

```
DFSAll(G):
  for all vertices v
      unmark v
  clock ← 0
  for all vertices v
      if v is unmarked
          clock ← DFS(v, clock)
```

```
DFS(v, clock):
  mark v
  for each edge v→w
      if w is unmarked
          clock ← DFS(w, clock)
  clock ← clock + 1
  finish(v) ← clock
  return clock
```

Lemma 4. *The vertex with largest finishing time lies in a source component of G.*

Proof: Let v be the vertex with largest finishing time. Then $\mathrm{DFS}(v, \mathit{clock})$ must be the last direct call to DFS made by the wrapper algorithm DFSAll.

Let C be the strong component of G that contains v. For the sake of argument, suppose there is an edge $x \to y$ such that $x \notin C$ and $y \in C$. Because v and y are strongly connected, y can reach v, and therefore x can reach v. There are two cases to consider.

- If x is already marked when DFS(v) begins, then v must have been marked during the execution of DFS(x), because x can reach v. But then v was already marked when DFS(v) was called, which is impossible.
- If x is not marked when DFS(v) begins, then x must be marked during the execution of DFS(v), which implies that v can reach x. Since x can also reach v, we must have $x \in C$, contradicting the definition of x.

We conclude that C is a source component of G. □

Essentially the same argument implies the following more general result.

Lemma 5. *For any edge $v \to w$ in G, if* finish(v) < finish(w)*, then v and w are strongly connected in G.*

Proof: Let $v \to w$ be an arbitrary edge of G. There are three cases to consider.If w is unmarked when DFS(v) begins, then the recursive call to DFS(w) finishes w, which implies that $finish(w) < finish(v)$. If w is still active when DFS(v) begins, there must be a path from w to v, which implies that v and w are strongly connected. Finally, if w is finished when DFS(v) begins, then clearly $finish(w) < finish(v)$. □

This observation is consistent with our earlier topological sorting algorithm; for *every* edge $v \to w$ in a directed acyclic graph, we have $finish(v) > finish(w)$.

It is easy to check (hint, hint) that any directed G has exactly the same strong components as its reversal $rev(G)$; in fact, we have $rev(scc(G)) = scc(rev(G))$. Thus, if we order the vertices of G by their finishing times in DFSALL($rev(G)$), the *last* vertex in this order lies in a sink component of G. Thus, if we run DFSALL(G), visiting vertices in reverse order of their finishing times in DFSALL($rev(G)$), then each call to DFS visits exactly one strong component of G.

Putting everything together, we obtain the following algorithm to count and label the strong components of a directed graph in $O(V + E)$ time, first discovered (but never published) by Rao Kosaraju in 1978, and then independently rediscovered by Micha Sharir in 1981. The Kosaraju-Sharir algorithm has two phases. The first phase performs a depth-first search of the reversal of G, pushing each vertex onto a stack when it is finished. In the second phase, we perform another depth-first search of the original graph G, considering vertices in the order they appear on the stack.

```
KosarajuSharir(G):
  ⟨⟨Phase 1: Push in finishing order⟩⟩
  unmark all vertices
  for all vertices v
      if v is unmarked
          clock ← RevPushDFS(v)
  ⟨⟨Phase 2: DFS in stack order⟩⟩
  unmark all vertices
  count ← 0
  while the stack is non-empty
      v ← Pop
      if v is unmarked
          count ← count + 1
          LabelOneDFS(v, count)
```

```
RevPushDFS(v):
  mark v
  for each edge v→u in rev(G)
      if u is unmarked
          RevPushDFS(u)
  Push(v)
```

```
LabelOneDFS(v, count):
  mark v
  label(v) ← count
  for each edge v→w in G
      if w is unmarked
          LabelOneDFS(w, count)
```

With further minor modifications, we can also compute the strongly connected component graph $scc(G)$ in $O(V+E)$ time.

Exercises

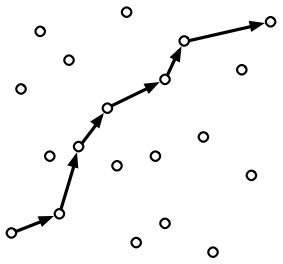
0. (a) Describe an algorithm to compute the reversal $rev(G)$ of a directed graph in $O(V+E)$ time.

   (b) Prove that for any directed graph G, the strong component graph $scc(G)$ is a dag.

   (c) Prove that for any directed graph G, we have $scc(rev(G)) = rev(scc(G))$.

   (d) Suppose S and T are two strongly connected components in a directed graph G. Prove that $finish(u) < finish(v)$ for all vertices $u \in S$ and $v \in T$.

1. A ***polygonal path*** is a sequence of line segments joined end-to-end; the endpoints of these line segments are called the ***vertices*** of the path. The ***length*** of a polygonal path is the sum of the lengths of its segments. A polygonal path with vertices $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ is ***monotonically increasing*** if $x_i < x_{i+1}$ and $y_i < y_{i+1}$ for every index i—informally, each vertex of the path is above and to the right of its predecessor.



A monotonically increasing polygonal path with seven vertices through a set of points

Suppose you are given a set S of n points in the plane, represented as two arrays $X[1..n]$ and $Y[1..n]$. Describe and analyze an algorithm to compute the length of the maximum-length monotonically increasing path with vertices in S. Assume you have a subroutine $\textsc{Length}(x, y, x', y')$ that returns the length of the segment from (x, y) to (x', y').

2. 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) A *transitive reduction* G^{TR} is a graph with the smallest possible number of edges whose transitive closure is G^T. (The same graph may have several transitive reductions.) Describe an efficient algorithm to compute the transitive reduction of G.

3. One of the oldest[1] algorithms for exploring graphs was proposed by Gaston Tarry in 1895. The input to Tarry's algorithm is a directed graph G that contains both directions of every edge; that is, for every edge $u \to v$ in G, its reversal $v \to u$ is also an edge in G.

```
TARRY(G):
  unmark all vertices of G
  color all edges of G white
  s ← any vertex in G
  RECTARRY(s)
```

```
RECTARRY(v):
  mark v                                  ⟨⟨"visit v"⟩⟩
  if there is a white arc v→w
      if w is unmarked
          color w→v green
      color v→w red                       } ⟨⟨"traverse v→w"⟩⟩
      RECTARRY(w)
  else if there is a green arc v→w
      color v→w red                       } ⟨⟨"traverse v→w"⟩⟩
      RECTARRY(w)
```

We informally say that Tarry's algorithm "visits" vertex v every time it marks v, and it "traverses" edge $v \to w$ when it colors that edge red and recursively calls RECTARRY(w).

   (a) Describe how to implement Tarry's algorithm so that it runs in $O(V + E)$ time.

   (b) Prove that no directed edge in G is traversed more than once.

   (c) When the algorithm visits a vertex v for the kth time, exactly how many edges into v are red, and exactly how many edges out of v are red? *[Hint: Consider the starting vertex s separately from the other vertices.]*

   (d) Prove each vertex v is visited at most $\deg(v)$ times, except the starting vertex s, which is visited at most $\deg(s) + 1$ times. This claim immediately implies that TARRY(G) terminates.

   (e) Prove that when TARRY(G) ends, the last visited vertex is the starting vertex s.

   (f) For every vertex v that TARRY(G) visits, prove that all edges into v and out of v are red when TARRY(G) halts. *[Hint: Consider the vertices in the order that they are marked for the first time, starting with s, and prove the claim by induction.]*

   (g) Prove that TARRY(G) visits every vertex of G. This claim and the previous claim imply that TARRY(G) traverses every edge of G exactly once.

4. Consider the following variant of Tarry's graph-traversal algorithm; this variant traverses green edges without recoloring them red and assigns two numerical labels to every vertex:

[1] Even older graph-traversal algorithms were described by Charles Trémaux in 1882, by Christian Wiener in 1873, and (implicitly) by Leonhard Euler in 1736. Wiener's algorithm is equivalent to depth-first search in a connected undirected graph.

```
TARRY2(G):
  unmark all vertices of G
  color all edges of G white
  s ← any vertex in G
  RECTARRY(s, 1)
```

```
RECTARRY2(v, clock):
  if v is unmarked
      pre(v) ← clock;  clock ← clock + 1
      mark v
  if there is a white arc v→w
      if w is unmarked
          color w→v green
      color v→w red
      RECTARRY2(w, clock)
  else if there is a green arc v→w
      post(v) ← clock;  clock ← clock + 1
      RECTARRY2(w, clock)
```

Prove or disprove the following claim: When TARRY2(G) halts, the green edges define a spanning tree and the labels $pre(v)$ and $post(v)$ define a preorder and postorder labeling that are all consistent with a single depth-first search of G. In other words, prove or disprove that TARRY2 produces the same output as depth-first search.

5. For any two nodes u and v in a directed acyclic graph G, the ***interval*** $\boldsymbol{G[u,v]}$ is the union of all directed paths in G from u to v. Equivalently, $G[u,v]$ consists of all vertices x such that $x \in Reach(u)$ and $v \in Reach(x)$, together with all the edges in G connecting those vertices.

   Suppose we are given a directed acyclic graph G, in which every edge has a numerical weight, which may be positive, negative, or zero. Describe an efficient algorithm to find the maximum-weight interval in G, where the weight of any interval is the sum of the weights of its vertices. *[Hint: Don't try to be clever.]*

6. Let G be a directed acyclic graph with a unique source s and a unique sink t.

   (a) A *Hamiltonian path* in G is a directed path in G that contains every vertex in G. Describe an algorithm to determine whether G has a Hamiltonian path.

   (b) Suppose the *vertices* of G have weights. Describe an efficient algorithm to find the path from s to t with maximum total weight.

   (c) Suppose we are also given an integer ℓ. Describe an efficient algorithm to find the maximum-weight path from s to t, such that the path contains at most ℓ edges. (Assume there is at least one such path.)

   (d) Suppose the vertices of G have integer labels, where $label(s) = -\infty$ and $label(t) = \infty$. Describe an algorithm to find the path from s to t with the maximum number of edges, such that the vertex labels define an increasing sequence.

   (e) Describe an algorithm to compute the number of distinct paths from s to t in G. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

7. Let G and H be directed acyclic graphs, whose vertices have labels from some fixed alphabet, and let $A[1..\ell]$ be a string over the same alphabet. Any directed path in G has a label, which is a string obtained by concatenating the labels of its vertices.

   (a) Describe an algorithm that either finds a path in G whose label is A or correctly reports that there is no such path.

(b) Describe an algorithm to find the *number* of paths in G whose label is A. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

(c) Describe an algorithm to find the longest path in G whose label is a subsequence of A.

(d) Describe an algorithm to find the *shortest* path in G whose label is a *supersequence* of A.

(e) Describe an algorithm to find a path in G whose label has minimum edit distance from A.

(f) Describe an algorithm to find the longest string that is both a label of a directed path in G and the label of a directed path in H.

(g) Describe an algorithm to find the longest string that is both a *subsequence* of the label of a directed path in G and a *subsequence* of the label of a directed path in H.

(h) Describe an algorithm to find the shortest string that is both a *supersequence* of the label of a directed path in G and a *supersequence* of the label of a directed path in H.

(i) Describe an algorithm to find the longest path in G whose label is a palindrome.

(j) Describe an algorithm to find the longest palindrome that is a subsequence of the label of a path in G.

(k) Describe an algorithm to find the shortest palindrome that is a supersequence of the label of a path in G.

8. Suppose two players are playing a turn-based game on a directed acyclic graph G with a unique source s. Each vertex v of G is labeled with a real number $\ell(v)$, which could be positive, negative, or zero. The game starts with three tokens at s. In each turn, the current player moves one of the tokens along a directed edge from its current node to another node, and the current player's score is increased by $\ell(u) \cdot \ell(v)$, where u and v are the locations of the two tokens that did *not* move. At most one token is allowed on any node except s at any time. The game ends when the current player is unable to move (for example, when all three tokens lie on sinks); at that point, the player with the higher score is the winner.

   Describe an efficient algorithm to determine who wins this game on a given labeled graph, assuming both players play optimally.

*9. Let $x = x_1 x_2 \ldots x_n$ be a given n-character string over some finite alphabet Σ, and let A be a deterministic finite-state machine with m states over the same alphabet.

(a) Describe and analyze an algorithm to compute the length of the longest subsequence of x that is accepted by A. For example, if A accepts the language $(\texttt{AR})^*$ and $x = \texttt{ABRACADABRA}$, your algorithm should output the number 4, which is the length of the subsequence ARAR.

(b) Describe and analyze an algorithm to compute the length of the shortest supersequence of x that is accepted by A. For example, if A accepts the language $(\texttt{ABCDR})^*$ and $x = \texttt{ABRACADABRA}$, your algorithm should output the number 25, which is the length of the supersequence ABCDRABCDRABCDRABCDRABCDR.

10. Not *every* dynamic programming algorithm can be modeled as finding an optimal path through a directed acyclic graph; the most obvious counterexample is the optimal binary search tree problem. But every dynamic programming problem does traverse a dependency graph in reverse topological order, performing some additional computation at every vertex.

    (a) Suppose we are given a directed acyclic graph G where every node stores a numerical search key. Describe and analyze an algorithm to find the largest binary search tree that is a subgraph of G.

    (b) Let G be a directed acyclic graph with the following features:

    - G has a single source s and several sinks $t_1, t_2, \ldots, t_k$.
    - Each edge $v \to w$ has an associated numerical value $p(v \to w)$ between 0 and 1.
    - For each non-sink vertex v, we have $\sum_w p(v \to w) = 1$.

    The values $p(v \to w)$ define a random walk in G from the source s to some sink t_i; after reaching any non-sink vertex v, the walk follows edge $v \to w$ with probability $p(v \to w)$. Describe and analyze an algorithm to compute the probability that this random walk reaches sink t_i, for every index i. (Assume that any arithmetic operation requires $O(1)$ time.)

We must all hang together, gentlemen, or else we shall most assuredly hang separately.

— Benjamin Franklin, at the signing of the Declaration of Independence (July 4, 1776)

It is a very sad thing that nowadays there is so little useless information.

— Oscar Wilde

A ship in port is safe, but that is not what ships are for.

— Rear Admiral Grace Murray Hopper

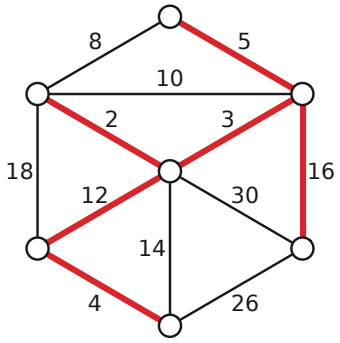
20 Minimum Spanning Trees

20.1 Introduction

Suppose we are given a connected, undirected, *weighted* graph. This is a graph $G = (V, E)$ together with a function $w : E \to \mathbb{R}$ that assigns a real *weight* $w(e)$ to each edge e, which may be positive, negative, or zero. Our task is to find the ***minimum spanning tree*** of G, that is, the spanning tree T that minimizes the function

$$w(T) = \sum_{e \in T} w(e).$$

To keep things simple, I'll assume that all the edge weights are distinct: $w(e) \neq w(e')$ for any pair of edges e and e'. Distinct weights guarantee that the minimum spanning tree of the graph is unique. Without this condition, there may be several different minimum spanning trees. For example, if all the edges have weight 1, then *every* spanning tree is a minimum spanning tree with weight $V - 1$.



A weighted graph and its minimum spanning tree.

If we have an algorithm that assumes the edge weights are unique, we can still use it on graphs where multiple edges have the same weight, as long as we have a consistent method for breaking ties. One way to break ties consistently is to use the following algorithm in place of a simple comparison. ShorterEdge takes as input four integers i, j, k, l, and decides which of the two edges (i, j) and (k, l) has "smaller" weight.

```
SHORTEREDGE(i, j, k, l)
  if w(i, j) < w(k, l)          then return (i, j)
  if w(i, j) > w(k, l)          then return (k, l)
  if min(i, j) < min(k, l)      then return (i, j)
  if min(i, j) > min(k, l)      then return (k, l)
  if max(i, j) < max(k, l)      then return (i, j)
  ⟨⟨if max(i,j) < max(k,l)⟩⟩     return (k, l)
```

20.2 The Only Minimum Spanning Tree Algorithm

There are several different methods for computing minimum spanning trees, but really they are all instances of the following generic algorithm. The situation is similar to the previous lecture, where we saw that depth-first search and breadth-first search were both instances of a single generic traversal algorithm.

The generic minimum spanning tree algorithm maintains an acyclic subgraph F of the input graph G, which we will call an *intermediate spanning forest*. F is a subgraph of the minimum spanning tree of G, and every component of F is a minimum spanning tree of its vertices. Initially, F consists of n one-node trees. The generic algorithm merges trees together by adding certain edges between them. When the algorithm halts, F consists of a single n-node tree, which must be the minimum spanning tree. Obviously, we have to be careful about *which* edges we add to the evolving forest, since not every edge is in the minimum spanning tree.

The intermediate spanning forest F induces two special types of edges. An edge is ***useless*** if it is not an edge of F, but both its endpoints are in the same component of F. For each component of F, we associate a ***safe*** edge—the minimum-weight edge with exactly one endpoint in that component. Different components might or might not have different safe edges. Some edges are neither safe nor useless—we call these edges *undecided*.

All minimum spanning tree algorithms are based on two simple observations.

Lemma 1. *The minimum spanning tree contains every safe edge.*

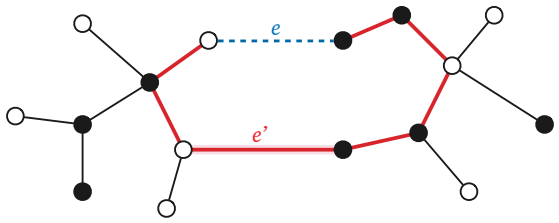
Proof: In fact we prove the following stronger statement: For *any* subset S of the vertices of G, the minimum spanning tree of G contains the minimum-weight edge with exactly one endpoint in S. We prove this claim using a greedy exchange argument.

Let S be an arbitrary subset of vertices of G; let e be the lightest edge with exactly one endpoint in S; and let T be an arbitrary spanning tree that does *not* contain e. Because T is connected, it contains a path from one endpoint of e to the other. Because this path starts at a vertex of S and ends at a vertex not in S, it must contain at least one edge with exactly one endpoint in S; let e' be *any* such edge. Because T is acyclic, removing e' from T yields a spanning *forest* with exactly two components, one containing each endpoint of e. Thus, adding e to this forest gives us a new spanning tree $T' = T - e' + e$. The definition of e implies $w(e') > w(e)$, which implies that T' has smaller total weight than T. We conclude that T is not the minimum spanning tree, which completes the proof. □

Lemma 2. *The minimum spanning tree contains no useless edge.*

Proof: Adding any useless edge to F would introduce a cycle. □

Our generic minimum spanning tree algorithm repeatedly adds one or more safe edges to the evolving forest F. Whenever we add new edges to F, some undecided edges become safe, and



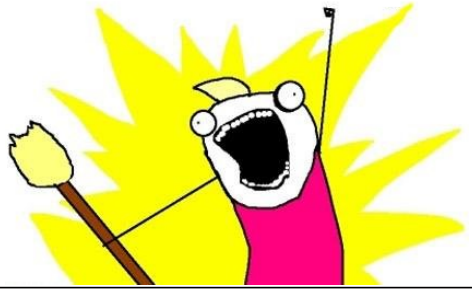
Proving that every safe edge is in the minimum spanning tree. Black vertices are in the subset S.

others become useless. To specify a particular algorithm, we must decide which safe edges to add, and we must describe how to identify new safe and new useless edges, at each iteration of our generic template.

20.3 Borvka's Algorithm

The oldest and arguably simplest minimum spanning tree algorithm was discovered by Borvka in 1926, long before computers even existed, and practically before the invention of graph theory![1] The algorithm was rediscovered by Choquet in 1938; again by Florek, Łukaziewicz, Perkal, Stienhaus, and Zubrzycki in 1951; and again by Sollin some time in the early 1960s. Because Sollin was the only Western computer scientist in this list—Choquet was a civil engineer; Florek and his co-authors were anthropologists—this is often called "Sollin's algorithm", especially in the parallel computing literature.

The Borvka/Choquet/Florek/Łukaziewicz/Perkal/Stienhaus/Zubrzycki/Sollin algorithm can be summarized in one line:



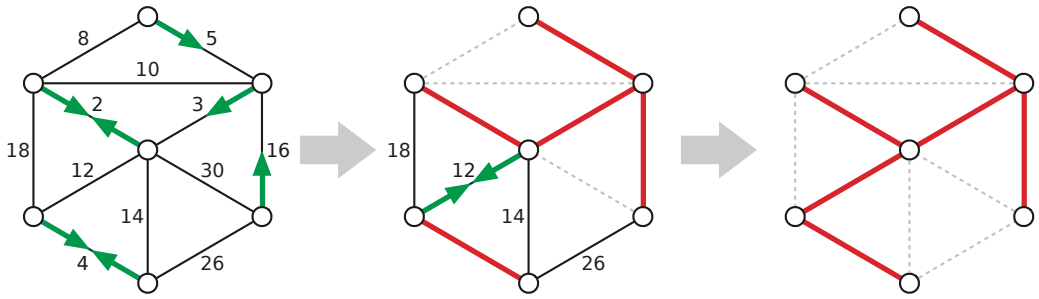
BORVKA: Add ***ALL*** the safe edges[2] and recurse.

We can find all the safe edge in the graph in $O(E)$ time as follows. First, we count the components of F using whatever-first search, using the standard wrapper function. As we count, we label every vertex with its component number; that is, every vertex in the first traversed component gets label 1, every vertex in the second component gets label 2, and so on.

If F has only one component, we're done. Otherwise, we compute an array $S[1..V]$ of edges, where $S[i]$ is the minimum-weight edge with one endpoint in the ith component (or a sentinel value NULL if there are less than i components). To compute this array, we consider each edge uv in the input graph G. If the endpoints u and v have the same label, then uv is useless. Otherwise, we compare the weight of uv to the weights of $S[label(u)]$ and $S[label(v)]$ and update the array entries if necessary.

[1] Leonard Euler published the first graph theory result, his famous theorem about the bridges of Königsburg, in 1736. However, the first textbook on graph theory, written by Dénes König, was not published until 1936.

[2] See also: Allie Brosh, "This is Why I'll Never be an Adult", *Hyperbole and a Half*, June 17, 2010. Actually, just go see everything in *Hyperbole and a Half*. And then go buy the book. And an extra copy for your cat.



Borůvka's algorithm run on the example graph. Thick edges are in F.
Arrows point along each component's safe edge. Dashed (gray) edges are useless.

```
BORVKA(V, E):
  F = (V, ∅)
  count ← COUNTANDLABEL(F)
  while count > 1
      ADDALLSAFEEDGES(E, F, count)
      count ← COUNTANDLABEL(F)
  return F
```

```
ADDALLSAFEEDGES(E, F, count):
  for i ← 1 to count
      S[i] ← NULL        ⟨⟨sentinel: w(NULL) := ∞⟩⟩
  for each edge uv ∈ E
      if label(u) ≠ label(v)
          if w(uv) < w(S[label(u)])
              S[label(u)] ← uv
          if w(uv) < w(S[label(v)])
              S[label(v)] ← uv
  for i ← 1 to count
      if S[i] ≠ NULL
          add S[i] to F
```

Each call to TRAVERSEALL requires $O(V)$ time, because the forest F has at most $V-1$ edges. Assuming the graph is represented by an adjacency list, the rest of each iteration of the main while loop requires $O(E)$ time, because we spend constant time on each edge. Because the graph is connected, we have $V \le E+1$, so each iteration of the while loop takes $O(E)$ time.

Each iteration reduces the number of components of F by at least a factor of two—the worst case occurs when the components coalesce in pairs. Since F initially has V components, the while loop iterates at most $O(\log V)$ times. Thus, the overall running time of Borvka's algorithm is $\boldsymbol{O(E \log V)}$.

Despite its relatively obscure origin, early algorithms researchers were aware of Borvka's algorithm, but dismissed it as being "too complicated"! As a result, despite its simplicity and efficiency, Borvka's algorithm is rarely mentioned in algorithms and data structures textbooks. On the other hand, Borvka's algorithm has several distinct advantages over other classical MST algorithms.

- Borvka's algorithm often runs faster than the $O(E \log V)$ worst-case running time. In arbitrary graphs, the number of components in F can drop by significantly more than a factor of 2 in a single iteration, reducing the number of iterations below the worst-case $\lceil \log_2 V \rceil$. A slight reformulation of Borvka's algorithm (actually closer to Borvka's original presentation) actually runs in $O(E)$ time for a broad class of interesting graphs, including graphs that can be drawn in the plane without edge crossings. In contrast, the time analysis for the other two algorithms applies to *all* graphs.

- Borvka's algorithm allows for significant parallelism; in each iteration, each component of F can be handled in a separate independent thread. This implicit parallelism allows for even faster performance on multicore or distributed systems. In contrast, the other two classical MST algorithms are intrinsically serial.

- There are several more recent minimum-spanning-tree algorithms that are faster even in the worst case than the classical algorithms described here. *All* of these faster algorithms are generalizations of Borvka's algorithm.

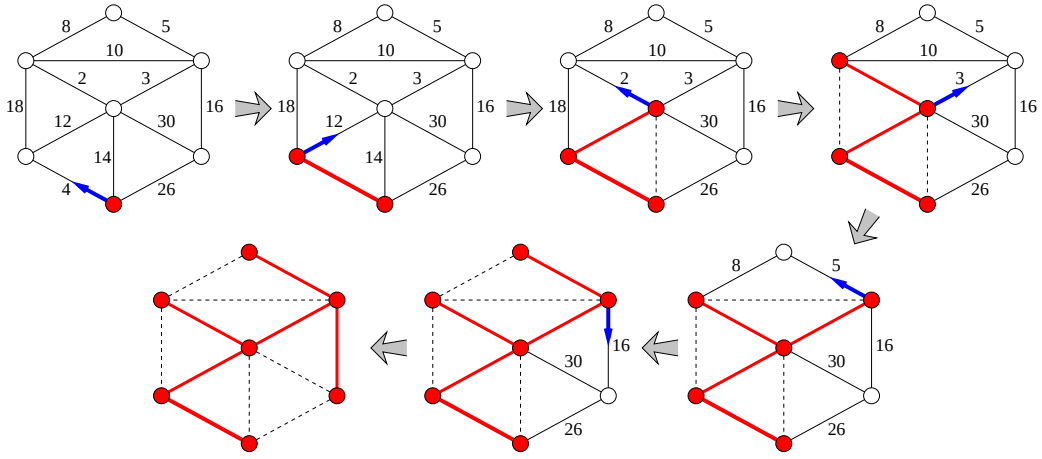
In short, if you ever need to implement a minimum-spanning-tree algorithm, use Borvka. On the other hand, if you want to *prove things about* minimum spanning trees effectively, you really need to know the next two algorithms as well.

20.4 Jarník's ("Prim's") Algorithm

The next oldest minimum spanning tree algorithm was first described by the Czech mathematician Vojtch Jarník in a 1929 letter to Borvka; Jarník published his discovery the following year. The algorithm was independently rediscovered by Kruskal in 1956, by Prim in 1957, by Loberman and Weinberger in 1957, and finally by Dijkstra in 1958. Prim, Loberman, Weinberger, and Dijkstra all (eventually) knew of and even cited Kruskal's paper, but since Kruskal also described two other minimum-spanning-tree algorithms in the same paper, *this* algorithm is usually called "Prim's algorithm", or sometimes "the Prim/Dijkstra algorithm", even though by 1958 Dijkstra already had another algorithm (inappropriately) named after him.

In Jarník's algorithm, the forest F contains only one nontrivial component T; all the other components are isolated vertices. Initially, T consists of an arbitrary vertex of the graph. The algorithm repeats the following step until T spans the whole graph:

> JARNÍK: Repeatedly add T's safe edge to T.



Jarník's algorithm run on the example graph, starting with the bottom vertex.
At each stage, thick edges are in T, an arrow points along T's safe edge, and dashed edges are useless.

To implement Jarník's algorithm, we keep all the edges adjacent to T in a priority queue. When we pull the minimum-weight edge out of the priority queue, we first check whether both of its endpoints are in T. If not, we add the edge to T and then add the new neighboring edges to the priority queue. In other words, Jarník's algorithm is another instance of the generic graph traversal algorithm we saw last time, using a priority queue as the "bag"! If we implement the algorithm this way, the algorithm runs in $O(E \log E) = \boldsymbol{O(E \log V)}$ time.

*20.5 Improving Jarník's Algorithm

We can improve Jarník's algorithm using a more advanced priority queue data structure called a *Fibonacci heap*, first described by Michael Fredman and Robert Tarjan in 1984. Fibonacci heaps support the standard priority queue operations INSERT, EXTRACTMIN, and DECREASEKEY. However, unlike standard binary heaps, which require $O(\log n)$ time for every operation, Fibonacci heaps support INSERT and DECREASEKEY in constant *amortized* time. The amortized cost of EXTRACTMIN is still $O(\log n)$.

To apply this faster data structure, we keep *vertices* in the priority queue instead of edge, where the key for each vertex v is either the minimum-weight edge between v and the evolving tree T, or ∞ if there is no such edge. We can INSERT all the vertices into the priority queue at the beginning of the algorithm; then, whenever we add a new edge to T, we may need to decrease the keys of some neighboring vertices.

To make the description easier, we break the algorithm into two parts. JARNÍK INIT initializes the priority queue; JARNÍK LOOP is the main algorithm. The input consists of the vertices and edges of the graph, plus the start vertex s. For each vertex v, we maintain both its key $key(v)$ and the incident edge $edge(v)$ such that $w(edge(v)) = key(v)$.

```
JARNÍK(V, E, s):
  JARNÍKINIT(V, E, s)
  JARNÍKLOOP(V, E, s)
```

```
JARNÍKINIT(V, E, s):
  for each vertex v ∈ V \ {s}
      if (v, s) ∈ E
          edge(v) ← (v, s)
          key(v) ← w(v, s)
      else
          edge(v) ← NULL
          key(v) ← ∞
      INSERT(v)
```

```
JARNÍKLOOP(V, E, s):
  T ← ({s}, ∅)
  for i ← 1 to |V| − 1
      v ← EXTRACTMIN
      add v and edge(v) to T
      for each neighbor u of v
          if u ∉ T and key(u) > w(uv)
              edge(u) ← uv
              DECREASEKEY(u, w(uv))
```

The operations INSERT and EXTRACTMIN are each called $O(V)$ times once for each vertex except s, and DECREASEKEY is called $O(E)$ times, at most twice for each edge. Thus, if we use a Fibonacci heap, the improved algorithm runs in $\boldsymbol{O(E + V \log V)}$ ***time***, which is faster than Borvka's algorithm unless $E = O(V)$.

In practice, however, this improvement is rarely faster than the naive implementation using a binary heap, unless the graph is extremely large and dense. The Fibonacci heap algorithms are quite complex, and the hidden constants in both the running time and space are significant—not outrageous, but certainly bigger than the hidden constant 1 in the $O(\log n)$ time bound for binary heap operations.

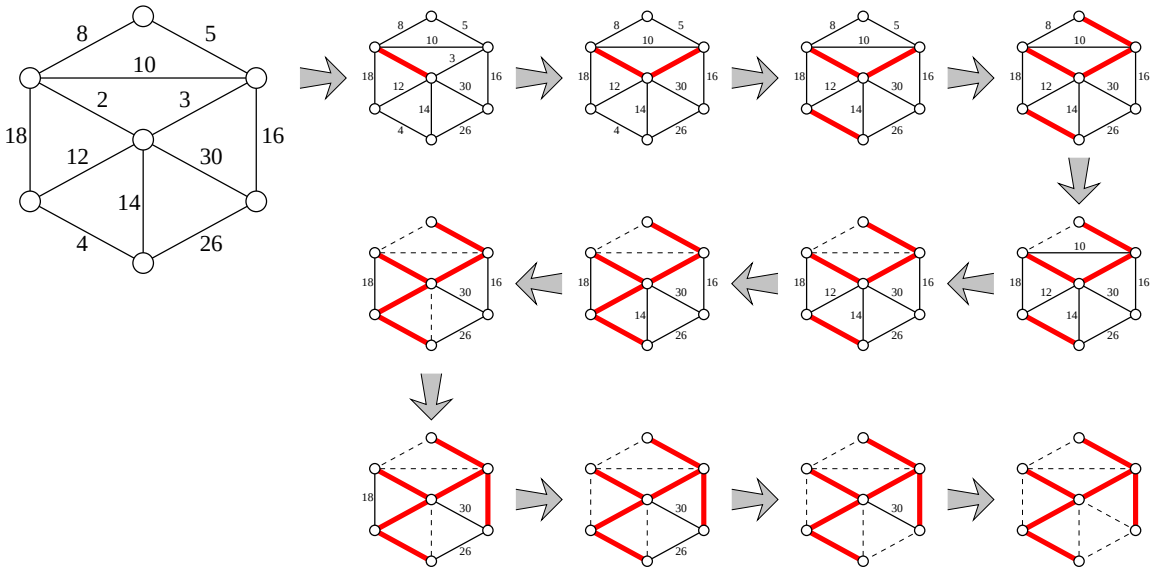
20.6 Kruskal's Algorithm

The last minimum spanning tree algorithm I'll discuss was first described by Kruskal in 1956, in the same paper where he rediscovered Jarnik's algorithm. Kruskal was motivated by "a typewritten translation (of obscure origin)" of Borvka's original paper, claiming that Borvka's algorithm was "unnecessarily elaborate".[3] This algorithm was also rediscovered in 1957 by Loberman and

[3]To be fair, Borvka's original paper *was* unnecessarily elaborate, but in his followup paper, also published in 1927, simplified his algorithm essentially to its current modern form. Kruskal was apparently unaware of Borvka's second paper. Stupid Iron Curtain.

Weinberger, but somehow avoided being renamed after them.

> KRUSKAL: Scan all edges in increasing weight order; if an edge is safe, add it to F.



Kruskal's algorithm run on the example graph. Thick edges are in F. Dashed edges are useless.

Since we examine the edges in order from lightest to heaviest, any edge we examine is safe if and only if its endpoints are in different components of the forest F. To prove this, suppose the edge e joins two components A and B but is not safe. Then there would be a lighter edge e' with exactly one endpoint in A. But this is impossible, because (inductively) any previously examined edge has both endpoints in the same component of F.

Just as in Borvka's algorithm, each component of F has a "leader" node. An edge joins two components of F if and only if the two endpoints have different leaders. But unlike Borvka's algorithm, we do not recompute leaders from scratch every time we add an edge. Instead, when two components are joined, the two leaders duke it out in a nationally-televised no-holds-barred steel-cage grudge match.[4] One of the two emerges victorious as the leader of the new larger component. More formally, we will use our earlier algorithms for the UNION-FIND problem, where the vertices are the elements and the components of F are the sets. Here's a more formal description of the algorithm:

```
KRUSKAL(V, E):
  sort E by increasing weight
  F ← (V, ∅)
  for each vertex v ∈ V
      MAKESET(v)
  for i ← 1 to |E|
      uv ← ith lightest edge in E
      if FIND(u) ≠ FIND(v)
          UNION(u, v)
          add uv to F
  return F
```

[4] Live at the Assembly Hall! Only $49.95 on Pay-Per-View![5]

[5] Is Pay-Per-View still a thing?

In our case, the sets are components of F, and $n = V$. Kruskal's algorithm performs $O(E)$ FIND operations, two for each edge in the graph, and $O(V)$ UNION operations, one for each edge in the minimum spanning tree. Using union-by-rank and path compression allows us to perform each UNION or FIND in $O(\alpha(E, V))$ time, where α is the not-quite-constant inverse-Ackerman function. So ignoring the cost of sorting the edges, the running time of this algorithm is $O(E\,\alpha(E, V))$.

We need $O(E \log E) = O(E \log V)$ additional time just to sort the edges. Since this is bigger than the time for the UNION-FIND data structure, the overall running time of Kruskal's algorithm is $\boldsymbol{O(E \log V)}$, exactly the same as Borvka's algorithm, or Jarník's algorithm with a normal (non-Fibonacci) heap.

Exercises

1. Most classical minimum-spanning-tree algorithms use the notions of "safe" and "useless" edges described in the lecture notes, but there is an alternate formulation. Let G be a weighted undirected graph, where the edge weights are distinct. We say that an edge e is ***dangerous*** if it is the longest edge in some cycle in G, and ***useful*** if it does not lie in any cycle in G.

   (a) Prove that the minimum spanning tree of G contains every useful edge.

   (b) Prove that the minimum spanning tree of G does not contain any dangerous edge.

   (c) Describe and analyze an efficient implementation of the "anti-Kruskal" MST algorithm: Examine the edges of G in *decreasing* order; if an edge is dangerous, remove it from G. *[Hint: It won't be as fast as Kruskal's algorithm.]*

2. Let $G = (V, E)$ be an arbitrary connected graph with weighted edges.

   (a) Prove that for any partition of the vertices V into two disjoint subsets, the minimum spanning tree of G includes the minimum-weight edge with one endpoint in each subset.

   (b) Prove that for any cycle in G, the minimum spanning tree of G *excludes* the maximum-weight edge in that cycle.

   (c) Prove or disprove: The minimum spanning tree of G includes the minimum-weight edge in *every* cycle in G.

3. Throughout this lecture note, we assumed that no two edges in the input graph have equal weights, which implies that the minimum spanning tree is unique. In fact, a weaker condition on the edge weights implies MST uniqueness.

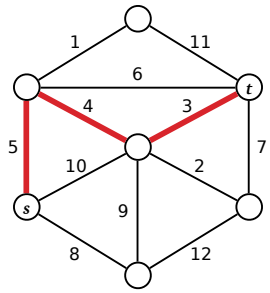
   (a) Describe an edge-weighted graph that has a unique minimum spanning tree, even though two edges have equal weights.

   (b) Prove that an edge-weighted graph G has a *unique* minimum spanning tree if and only if the following conditions hold:

   - For any partition of the vertices of G into two subsets, the minimum-weight edge with one endpoint in each subset is unique.
   - The maximum-weight edge in any cycle of G is unique.

(c) Describe and analyze an algorithm to determine whether or not a graph has a unique minimum spanning tree.

4. Consider a path between two vertices s and t in an undirected weighted graph G. The *bottleneck length* of this path is the maximum weight of any edge in the path. The *bottleneck distance* between s and t is the minimum bottleneck length of any path from s to t. (If there are no paths from s to t, the bottleneck distance between s and t is ∞.)



The bottleneck distance between s and t is 5.

Describe an algorithm to compute the bottleneck distance between *every* pair of vertices in an arbitrary undirected weighted graph. Assume that no two edges have the same weight.

5. (a) Describe and analyze an algorithm to compute the *maximum*-weight spanning tree of a given edge-weighted graph.

   (b) A *feedback edge set* of an undirected graph G is a subset F of the edges such that every cycle in G contains at least one edge in F. In other words, removing every edge in F makes the graph G acyclic. Describe and analyze a fast algorithm to compute the minimum weight feedback edge set of of a given edge-weighted graph.

6. Suppose we are given both an undirected graph G with weighted edges and a minimum spanning tree T of G.

   (a) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is decreased.

   (b) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is increased.

   In both cases, the input to your algorithm is the edge e and its new weight; your algorithms should modify T so that it is still a minimum spanning tree. *[Hint: Consider the cases $e \in T$ and $e \notin T$ separately.]*

7. (a) Describe and analyze and algorithm to find the *second smallest spanning tree* of a given graph G, that is, the spanning tree of G with smallest total weight except for the minimum spanning tree.

   *(b) Describe and analyze an efficient algorithm to compute, given a weighted undirected graph G and an integer k, the k spanning trees of G with smallest weight.

8. We say that a graph $G = (V, E)$ is *dense* if $E = \Theta(V^2)$. Describe a modification of Jarník's minimum-spanning tree algorithm that runs in $O(V^2)$ time (independent of E) when the input graph is dense, using only simple data structures (and in particular, *without* using a Fibonacci heap).

9. (a) Prove that the minimum spanning tree of a graph is also a spanning tree whose maximum-weight edge is minimal.

   *(b) Describe an algorithm to compute a spanning tree whose maximum-weight edge is minimal, in $O(V + E)$ time. *[Hint: Start by computing the median of the edge weights.]*

10. Consider the following variant of Borvka's algorithm. Instead of counting and labeling components of F to find safe edges, we use a standard disjoint set data structure. Each component of F is represented by an up-tree; each vertex v stores a pointer $parent(v)$ to its parent in the up-tree containing v. Each leader vertex $\bar{v}$ also maintains an edge $safe(\bar{v})$, which is (eventually) the lightest edge with one endpoint in $\bar{v}$'s component of F.

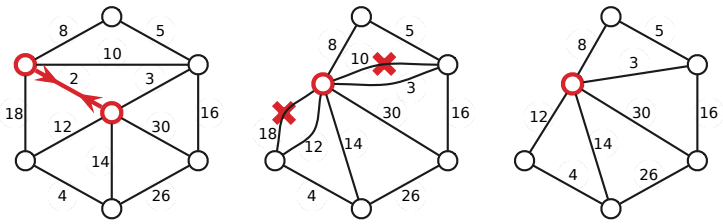
```
BORVKA(V, E):
  F = ∅
  for each vertex v ∈ V
      parent(v) ← v
  while FINDSAFEEDGES(V, E)
      ADDSAFEEDGES(V, E, F)
  return F
```

```
FINDSAFEEDGES(V, E):
  for each vertex v ∈ V
      safe(v) ← NULL
  found ← FALSE
  for each edge uv ∈ E
      ū ← FIND(u); v̄ ← FIND(v)
      if ū ≠ v̄
          if w(uv) < w(safe(ū))
              safe(ū) ← uv
          if w(uv) < w(safe(v̄))
              safe(v̄) ← uv
          found ← TRUE
  return done
```

```
ADDSAFEEDGES(V, E, F):
  for each vertex v ∈ V
      if safe(v) ≠ NULL
          xy ← safe(v)
          if FIND(x) ≠ FIND(y)
              UNION(x, y)
              add xy to F
```

Prove that if FIND uses path compression, then each call to FINDSAFEEDGES and ADDSAFEEDGES requires only $O(V + E)$ time. *[Hint: It doesn't matter how UNION is implemented! What is the depth of the up-trees when FINDSAFEEDGES ends?]*

11. Minimum-spanning tree algorithms are often formulated using an operation called *edge contraction*. To contract the edge uv, we insert a new node, redirect any edge incident to u or v (except uv) to this new node, and then delete u and v. After contraction, there may be multiple parallel edges between the new node and other nodes in the graph; we remove all but the lightest edge between any two nodes.



Contracting an edge and removing redundant parallel edges.

The three classical minimum-spanning tree algorithms can be expressed cleanly in terms of contraction as follows. All three algorithms start by making a clean copy G' of the input graph G and then repeatedly contract safe edges in G; the minimum spanning tree consists of the contracted edges.

- BORVKA: Mark the lightest edge leaving each vertex, contract all marked edges, and recurse.
- JARNÍK: Repeatedly contract the lightest edge incident to some fixed root vertex.
- KRUSKAL: Repeatedly contract the lightest edge in the graph.

(a) Describe an algorithm to execute a single pass of Borvka's contraction algorithm in $O(V + E)$ time. The input graph is represented in an adjacency list.

(b) Consider an algorithm that first performs k passes of Borvka's contraction algorithm, and then runs Jarník's algorithm (*with* a Fibonacci heap) on the resulting contracted graph.

  i. What is the running time of this hybrid algorithm, as a function of V, E, and k?

  ii. For which value of k is this running time minimized? What is the resulting running time?

(c) Call a family of graphs *nice* if it has the following properties:

- A nice graph with n vertices has only $O(n)$ edges.
- Contracting an edge of a nice graph yields another nice graph.

For example, graphs that can be drawn in the plane without crossing edges are nice; Euler's formula implies that any planar graph with n vertices has at most $3n-6$ edges.

Prove that Borüvka's contraction algorithm computes the minimum spanning tree of any nice n-vertex graph in $O(n)$ time.

Well, ya turn left by the fire station in the village and take the old post road by the reservoir and. . . no, that won't do.

Best to continue straight on by the tar road until you reach the schoolhouse and then turn left on the road to Bennett's Lake until. . . no, that won't work either.

East Millinocket, ya say? Come to think of it, you can't get there from here.

— Robert Bryan and Marshall Dodge,
Bert and I and Other Stories from Down East (1961)

Hey farmer! Where does this road go?
Been livin' here all my life, it ain't gone nowhere yet.
Hey farmer! How do you get to Little Rock?
Listen stranger, you can't get there from here.
Hey farmer! You don't know very much do you?
No, but I ain't lost.

— Michelle Shocked, "Arkansas Traveler" (1992)

21 Shortest Paths

21.1 Introduction

Suppose we are given a weighted *directed* graph $G = (V, E, w)$ with two special vertices, and we want to find the shortest path from a *source* vertex s to a *target* vertex t. That is, we want to find the directed path p starting at s and ending at t that minimizes the function

$$w(p) := \sum_{u \to v \in p} w(u \to v).$$

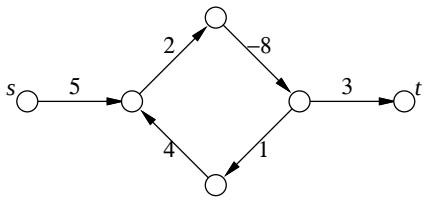
For example, if I want to answer the question "What's the fastest way to drive from my old apartment in Champaign, Illinois to my wife's old apartment in Columbus, Ohio?", I might use a graph whose vertices are cities, edges are roads, weights are driving times, s is Champaign, and t is Columbus.[1] The graph is directed, because driving times along the same road might be different in different directions. (At one time, there was a speed trap on I-70 just east of the Indiana/Ohio border, but only for eastbound traffic.)

Perhaps counter to intuition, we will allow the weights on the edges to be negative. Negative edges make our lives complicated, because the presence of a negative *cycle* might imply that there is no shortest path. In general, a shortest path from s to t exists if and only if there is *at least one* path from s to t, but there is no path from s to t that touches a negative cycle. If any negative cycle is reachable from s and can reach t, we can always find a shorter path by going around the cycle one more time.

Almost every algorithm known for solving this problem actually solves (large portions of) the following more general ***single source shortest path*** or ***SSSP*** problem: Find the shortest path from the source vertex s to *every* other vertex in the graph. This problem is usually solved by finding a ***shortest path tree*** rooted at s that contains all the desired shortest paths.

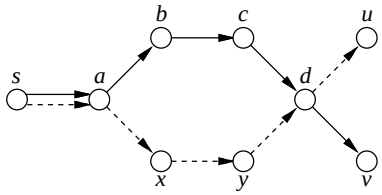
It's not hard to see that if shortest paths are unique, then they form a tree, because any subpath of a shortest path is itself a shortest path. If there are multiple shortest paths to some

[1] West on Church, north on Prospect, east on I-74, south on I-465, east on Airport Expressway, north on I-65, east on I-70, north on Grandview, east on 5th, north on Olentangy River, east on Dodridge, north on High, west on Kelso, south on Neil. Depending on traffic. We both live in Urbana now.



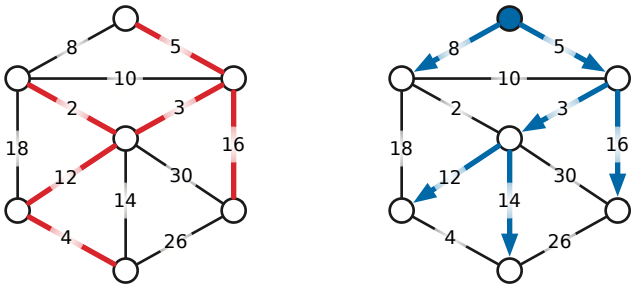
There is no shortest path from s to t.

vertices, we can always choose one shortest path to each vertex so that the union of the paths is a tree. If there are shortest paths to two vertices u and v that diverge, then meet, then diverge again, we can modify one of the paths without changing its length so that the two paths only diverge once.



If $s \to a \to b \to c \to d \to v$ and $s \to a \to x \to y \to d \to u$ are shortest paths, then $s \to a \to b \to c \to d \to u$ is also a shortest path.

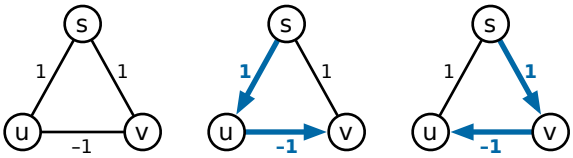
Although they are both optimal spanning trees, shortest-path trees and minimum spanning trees are very different creatures. Shortest-path trees are rooted and directed; minimum spanning trees are unrooted and undirected. Shortest-path trees are most naturally defined for directed graphs; only undirected graphs have minimum spanning trees. If edge weights are distinct, there is only one minimum spanning tree, but every source vertex induces a different shortest-path tree; moreover, it is possible for *every* shortest path tree to use a different set of edges from the minimum spanning tree.



A minimum spanning tree and a shortest path tree (rooted at the top vertex) of the same undirected graph.

21.2 Warning!

Throughout this lecture, we will explicitly consider *only* directed graphs. All of the algorithms described in this lecture also work for undirected graphs with some minor modifications, *but only if negative edges are prohibited*. Dealing with negative edges in undirected graphs is considerably more subtle. We cannot simply replace every undirected edge with a pair of directed edges, because this would transform any negative edge into a short negative cycle. Subpaths of an undirected shortest path that contains a negative edge are *not* necessarily shortest paths; consequently, the set of all undirected shortest paths from a single source vertex may not define a tree, even if shortest paths are unique.



An undirected graph where shortest paths from s are unique but do not define a tree.

A complete treatment of undirected graphs with negative edges is beyond the scope of this lecture (if not the entire course). I will only mention that a *single* shortest path in an undirected graph with negative edges can be computed in $O(VE + V^2 \log V)$ time, by a reduction to maximum weighted matching.

21.3 The Only SSSP Algorithm

Just like graph traversal and minimum spanning trees, there are several different SSSP algorithms, but they are all special cases of the a single generic algorithm, first proposed by Lester Ford in 1956, and independently by George Dantzig in 1957.[2] Each vertex v in the graph stores two values, which (inductively) describe a *tentative* shortest path from s to v.

- $dist(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path, or ∞ if there is no such path.
- $pred(v)$ is the predecessor of v in the tentative shortest $s \rightsquigarrow v$ path, or NULL if there is no such vertex.

In fact, the predecessor pointers automatically define a tentative shortest path *tree*; they play exactly the same role as the parent pointers in our generic graph traversal algorithm. At the beginning of the algorithm, we already know that $dist(s) = 0$ and $pred(s) = \text{NULL}$. For every vertex $v \neq s$, we initially set $dist(v) = \infty$ and $pred(v) = \text{NULL}$ to indicate that we do not know of *any* path from s to v.

During the execution of the algorithm, we call an edge $u \to v$ ***tense*** if $dist(u) + w(u \to v) < dist(v)$. If $u \to v$ is tense, the tentative shortest path $s \rightsquigarrow v$ is clearly incorrect, because the path $s \rightsquigarrow u \to v$ is shorter. Our generic algorithm repeatedly finds a tense edge in the graph and *relaxes* it:

```
RELAX(u→v):
  dist(v) ← dist(u) + w(u→v)
  pred(v) ← u
```

When there are no tense edges, the algorithm halts, and we have our desired shortest path tree.

The correctness of Ford's generic relaxation algorithm follows from the following series of claims:

1. For every vertex v, the distance $dist(v)$ is either ∞ or the length of some walk from s to v. This claim can be proved by induction on the number of relaxations.
2. If the graph has no negative cycles, then $dist(v)$ is either ∞ or the length of some *simple path* from s to v. Specifically, if $dist(v)$ is the length of a walk from s to v that contains a directed cycle, that cycle must have negative weight. This claim implies that if G has no negative cycles, the relaxation algorithm eventually halts, because there are only a finite number of simple paths in G.

[2] Specifically, Dantzig showed that the shortest path problem can be phrased as a linear programming problem, and then described an interpretation of his simplex method in terms of the original graph. His description is equivalent to Ford's relaxation strategy.

3. If no edge in G is tense, then for every vertex v, the distance $dist(v)$ is the length of the predecessor path $s \to \cdots pred(pred(v)) \to pred(v) \to v$. Specifically, if v violates this condition but its predecessor $pred(v)$ does not, the edge $pred(v) \to v$ is tense.

4. If no edge in G is tense, then for every vertex v, the path $s \to \cdots pred(pred(v)) \to pred(v) \to v$ is a shortest path from s to v. Specifically, if v violates this condition but its predecessor u *in some shortest path* does not, the edge $u \to v$ is tense. This claim also implies that if the G has a negative cycle, then some edge is *always* tense, so the generic algorithm never halts.

So far I haven't said anything about how we detect which edges can be relaxed, or in what order we relax them. To make this easier, we refine the relaxation algorithm slightly, into something closely resembling the generic graph traversal algorithm. We maintain a "bag" of vertices, initially containing just the source vertex s. Whenever we take a vertex u from the bag, we scan all of its outgoing edges, looking for something to relax. Finally, whenever we successfully relax an edge $u \to v$, we put v into the bag. Unlike our generic graph traversal algorithm, we do not mark vertices when we visit them; the same vertex could be visited many times, and the same edge could be relaxed many times.

```
InitSSSP(s):
  dist(s) ← 0
  pred(s) ← Null
  for all vertices v ≠ s
      dist(v) ← ∞
      pred(v) ← Null
```

```
GenericSSSP(s):
  InitSSSP(s)
  put s in the bag
  while the bag is not empty
      take u from the bag
      for all edges u→v
          if u→v is tense
              Relax(u→v)
              put v in the bag
```

Just as with graph traversal, different "bag" data structures for the give us different algorithms. There are three obvious choices to try: a stack, a queue, and a priority queue. Unfortunately, if we use a stack, the resulting algorithm performs $\Theta(2^V)$ relaxation steps in the worst case! (Proving this is a good homework problem.) The other two possibilities are much more efficient.

21.4 Dijkstra's Algorithm

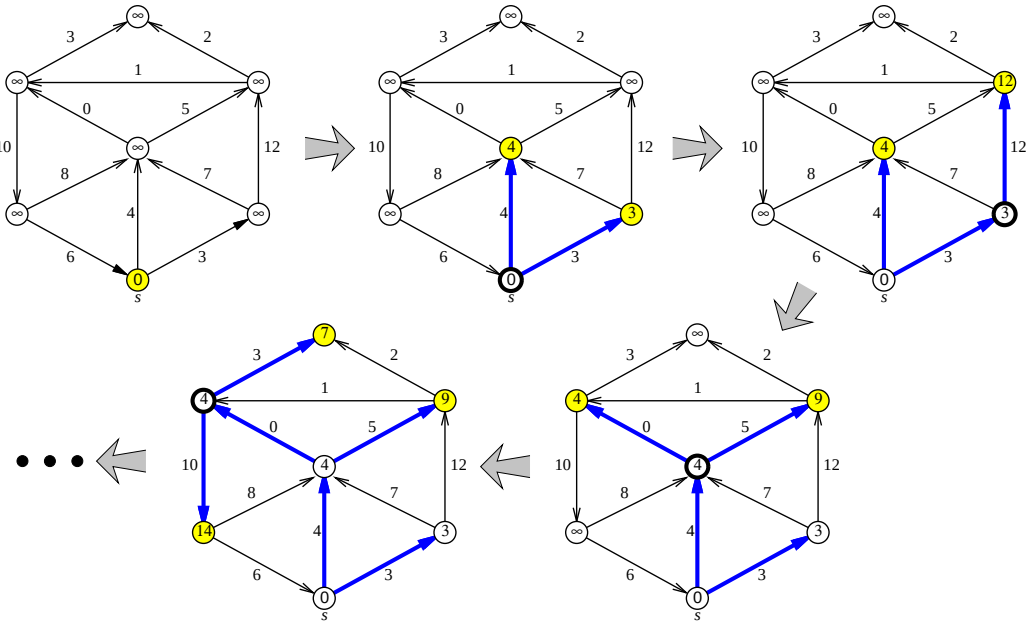
If we implement the bag using a priority queue, where the key of a vertex v is its tentative distance $dist(v)$, we obtain an algorithm first "published" in 1957 by a team of researchers at the Case Institute of Technology, in an annual project report for the Combat Development Department of the US Army Electronic Proving Ground. The same algorithm was later independently rediscovered and actually publicly published by Edsger Dijkstra in 1959. A nearly identical algorithm was also described by George Dantzig in 1958.

Dijkstra's algorithm, as it is universally known[3], is particularly well-behaved if the graph has no negative-weight edges. In this case, it's not hard to show (by induction, of course) that the vertices are scanned in increasing order of their shortest-path distance from s. It follows that each vertex is scanned at most once, and thus that each edge is relaxed at most once. Since the key of each vertex in the heap is its tentative distance from s, the algorithm performs a DecreaseKey operation every time an edge is relaxed. Thus, the algorithm performs at most E DecreaseKeys.

[3]I will follow this common convention, despite the historical inaccuracy, partly because I don't think anybody wants to read about the "Leyzorek-Gray-Johnson-Ladew-Meaker-Petry-Seitz algorithm", and partly because papers that aren't actually *publically* published don't count.

Similarly, there are at most V INSERT and EXTRACTMIN operations. Thus, if we store the vertices in a Fibonacci heap, the total running time of Dijkstra's algorithm is $\boldsymbol{O(E + V \log V)}$; if we use a regular binary heap, the running time is $\boldsymbol{O(E \log V)}$.

This analysis assumes that no edge has negative weight. Dijkstra's algorithm (in the form I'm presenting here[4]) is still *correct* if there are negative edges, but the worst-case running time could be exponential. (Proving this unfortunate fact is a good homework problem.) On the other hand, in practice, Dijkstra's algorithm is usually quite fast even for graphs with negative edges.



Four phases of Dijkstra's algorithm run on a graph with no negative edges.
At each phase, the shaded vertices are in the heap, and the bold vertex has just been scanned.
The bold edges describe the evolving shortest path tree.

21.5 The A^* Heuristic

A slight generalization of Dijkstra's algorithm, commonly known as the A^* algorithm, is frequently used to find a shortest path from a single source node s to a single target node t. This heuristic was first described in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael. A^* uses a black-box function GUESSDISTANCE(v, t) that returns an estimate of the distance from v to t. The only difference between Dijkstra and A^* is that the key of a vertex v is $dist(v) + \text{GUESSDISTANCE}(v, t)$.

The function GUESSDISTANCE is called *admissible* if GUESSDISTANCE(v, t) never overestimates the actual shortest path distance from v to t. If GUESSDISTANCE is admissible and the actual edge weights are all non-negative, the A^* algorithm computes the actual shortest path from s to t at least as quickly as Dijkstra's algorithm. In practice, the closer GUESSDISTANCE(v, t) is to the real distance from v to t, the faster the algorithm. However, in the worst case, the running time is still $O(E + V \log V)$.

The heuristic is especially useful in situations where the actual graph is not known. For example, A^* can be used to find optimal solutions to many puzzles (15-puzzle, Freecell, Shanghai,

[4]Most algorithms textbooks, Wikipedia, and even Dijkstra's original paper present a version of Dijkstra's algorithm that gives incorrect results for graphs with negative edges, because it *never* visits the same vertex more than once. I've taken the liberty of correcting Dijkstra's mistake. Even Dijkstra would agree that a correct algorithm that is sometimes slow (and in practice, *rarely* slow) is better than a fast algorithm that doesn't always work.

Sokoban, Atomix, Rush Hour, Rubik's Cube, Racetrack, . . .) and other path planning problems where the starting and goal configurations are given, but the graph of all possible configurations and their connections is not given explicitly.

21.6 Shimbel's Algorithm

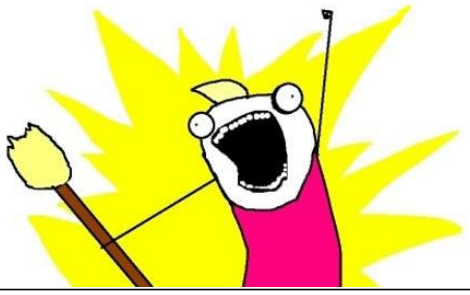
If we replace the heap in Dijkstra's algorithm with a FIFO queue, we obtain an algorithm first sketched by Shimbel in 1954, described in more detail by Moore in 1957, then independently rediscovered by Woodbury and Dantzig in 1957 and again by Bellman in 1958. Because Bellman explicitly used Ford's formulation of relaxing edges, this algorithm is almost universally called "Bellman-Ford", although some early sources refer to "Bellman-Shimbel". Shimbel's algorithm is efficient even if there are negative edges, and it can be used to quickly detect the presence of negative cycles. If there are no negative edges, however, Dijkstra's algorithm is faster. (In fact, in practice, Dijkstra's algorithm is often faster even for graphs with negative edges.)

The easiest way to analyze the algorithm is to break the execution into *phases*, by introducing an imaginary *token*. Before we even begin, we insert the token into the queue. The current phase ends when we take the token out of the queue; we begin the next phase by reinserting the token into the queue. The 0th phase consists entirely of scanning the source vertex s. The algorithm ends when the queue contains *only* the token. A simple inductive argument (hint, hint) implies the following invariant for every integer i and vertex v:

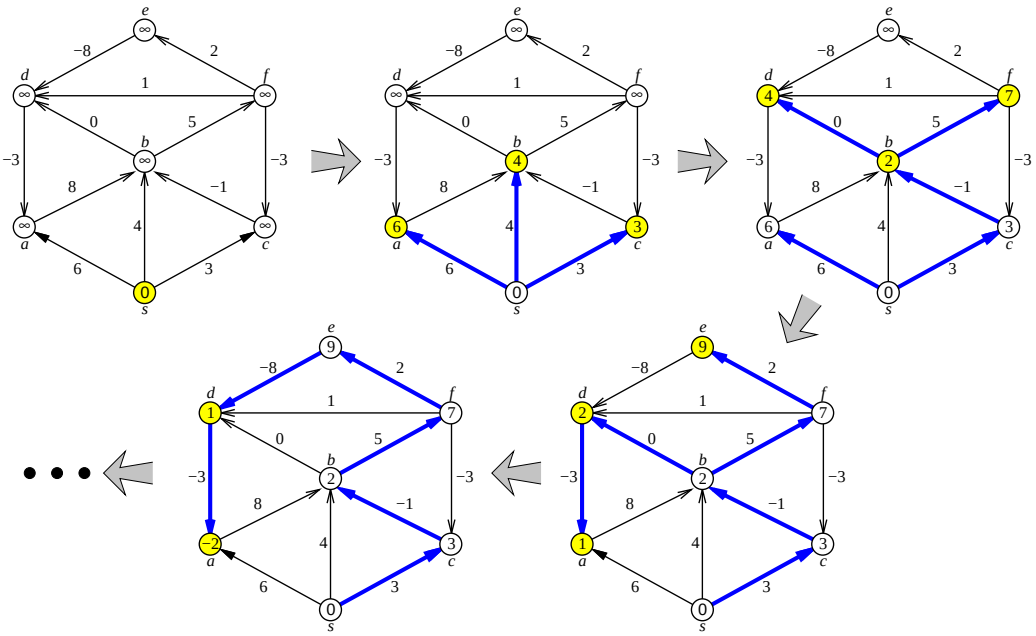
> **After i phases of the algorithm, $dist(v)$ is *at most* the length of the shortest walk from s to v consisting of *at most* i edges.**

Since a shortest path can only pass through each vertex once, either the algorithm halts before the Vth phase, or the graph contains a negative cycle. In each phase, we scan each vertex at most once, so we relax each edge at most once, so the running time of a single phase is $O(E)$. Thus, the overall running time of Shimbel's algorithm is $\boldsymbol{O(VE)}$.

Once we understand how the phases of Shimbel's algorithm behave, we can simplify the algorithm considerably by producing the same behavior on purpose. Instead of performing a partial breadth-first search of the graph in each phase, we can simply scan through the adjacency list directly, relaxing every tense edge we find in the graph.



SHIMBEL: Relax ***ALL*** the tense edges and recurse.



Four phases of Shimbel's algorithm run on a directed graph with negative edges.
Nodes are taken from the queue in the order $s \diamond a\ b\ c \diamond d\ f\ b \diamond a\ e\ d \diamond d\ a \diamond\diamond$, where $\diamond$ is the end-of-phase token.
Shaded vertices are in the queue at the end of each phase. The bold edges describe the evolving shortest path tree.

```
ShimbelSSSP(s)
  InitSSSP(s)
  repeat V times:
      for every edge u→v
          if u→v is tense
              Relax(u→v)
  for every edge u→v
      if u→v is tense
          return "Negative cycle!"
```

This is how most textbooks present "Bellman-Ford".[5] The $O(VE)$ running time of this formulation of the algorithm should be obvious, but it may be less clear that the algorithm is still correct. In fact, correctness follows from exactly the same invariant as before:

> **After i phases of the algorithm, $dist(v)$ is *at most* the length of the shortest walk from s to v consisting of *at most* i edges.**

As before, it is straightforward to prove by induction (hint, hint) that this invariant holds for every integer i and vertex v.

21.7 Shimbel's Algorithm as Dynamic Programming

Shimbel's algorithm can also be recast as a dynamic programming algorithm. Let $dist_i(v)$ denote the length of the shortest path $s \rightsquigarrow v$ consisting of at most i edges. It's not hard to see that this

[5] In fact, this is essentially the formulation proposed by both Shimbel and Bellman. Bob Tarjan recognized in the early 1980s that Shimbel's algorithm is equivalent to Dijkstra's algorithm with a queue instead of a heap.

function obeys the following recurrence:

$$dist_i(v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ \min \left\{ \begin{array}{c} dist_{i-1}(v), \\ \min\limits_{u \to v \in E} (dist_{i-1}(u) + w(u \to v)) \end{array} \right\} & \text{otherwise} \end{cases}$$

For the moment, let's assume the graph has no negative cycles; our goal is to compute $dist_{V-1}(t)$. We can clearly memoize this two-parameter function into a two-dimensional array. A straightforward dynamic programming evaluation of this recurrence looks like this:

```
ShimbelDP(s)
  dist[0,s] ← 0
  for every vertex v ≠ s
      dist[0,v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i,v] ← dist[i−1,v]
          for every edge u→v
              if dist[i,v] > dist[i−1,u] + w(u→v)
                  dist[i,v] ← dist[i−1,u] + w(u→v)
```

Now let us make two minor changes to this algorithm. First, we remove one level of indentation from the last three lines. This may change the order in which we examine edges, but the modified algorithm still computes $dist_i(v)$ for all i and v. Second, we change the indices in the last two lines from $i-1$ to i. This change may cause the distances $dist[i,v]$ to approach the true shortest-path distances more quickly than before, but the algorithm is still correct.

```
ShimbelDP2(s)
  dist[0,s] ← 0
  for every vertex v ≠ s
      dist[0,v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i,v] ← dist[i−1,v]
      for every edge u→v
          if dist[i,v] > dist[i,u] + w(u→v)
              dist[i,v] ← dist[i,u] + w(u→v)
```

Now notice that the iteration index i is completely redundant! We really only need to keep a one-dimensional array of distances, which means we don't need to scan the vertices in each iteration of the main loop.

```
ShimbelDP3(s)
  dist[s] ← 0
  for every vertex v ≠ s
      dist[v] ← ∞
  for i ← 1 to V − 1
      for every edge u→v
          if dist[v] > dist[u] + w(u→v)
              dist[v] ← dist[u] + w(u→v)
```

The resulting algorithm is almost identical to our earlier algorithm SHIMBELSSSP! The first three lines initialize the shortest path distances, and the last two lines check whether an edge is tense, and if so, relaxes it. The only feature missing from the new algorithm is explicit maintenance of predecessors, but that's easy to add.

Exercises

0. Prove that the following invariant holds for every integer i and every vertex v: After i phases of Shimbel's algorithm (in either formulation), $dist(v)$ is at most the length of the shortest path $s \rightsquigarrow v$ consisting of at most i edges.

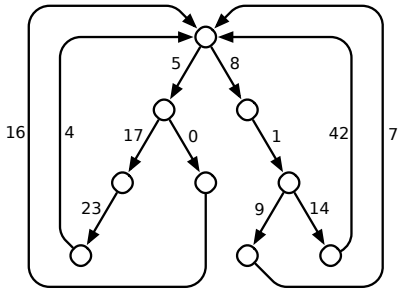
1. Let G be a directed graph with edge weights (which may be positive, negative, or zero), and let s be an arbitrary vertex of G.

   (a) Suppose every vertex v stores a number $dist(v)$. Describe and analyze an algorithm to determine whether $dist(v)$ is the shortest-path distance from s to v, for every vertex v.

   (b) Suppose instead that every vertex $v \neq s$ stores a pointer $pred(v)$ to another vertex in G. Describe and analyze an algorithm to determine whether these predecessor pointers define a single-source shortest path tree rooted at s.

   Do ***not*** assume that G contains no negative cycles.

2. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has a non-negative weight.



A looped tree.

   (a) How much time would Dijkstra's algorithm require to compute the shortest path between two vertices u and v in a looped tree with n nodes?

   (b) Describe and analyze a faster algorithm.

3. Suppose we are given an undirected graph G in which every *vertex* has a positive weight.

   (a) Describe and analyze an algorithm to find a *spanning tree* of G with minimum total weight. (The total weight of a spanning tree is the sum of the weights of its vertices.)

   (b) Describe and analyze an algorithm to find a *path* in G from one given vertex s to another given vertex t with minimum total weight. (The total weight of a path is the sum of the weights of its vertices.)

4. For any edge e in any graph G, let $G \setminus e$ denote the graph obtained by deleting e from G.

   (a) Suppose we are given a directed graph G in which the shortest path σ from vertex s to vertex t passes through *every* vertex of G. Describe an algorithm to compute the shortest-path distance from s to t in $G \setminus e$, for *every* edge e of G, in $O(E \log V)$ time. Your algorithm should output a set of E shortest-path distances, one for each edge of the input graph. You may assume that all edge weights are non-negative. *[Hint: If we delete an edge of the original shortest path, how do the old and new shortest paths overlap?]*

   *(b) Let s and t be *arbitrary* vertices in an arbitrary *undirected* graph G. Describe an algorithm to compute the shortest-path distance from s to t in $G \setminus e$, for *every* edge e of G, in $O(E \log V)$ time. Again, you may assume that all edge weights are non-negative.

5. Let $G = (V, E)$ be a connected directed graph with non-negative edge weights, let s and t be vertices of G, and let H be a subgraph of G obtained by deleting some edges. Suppose we want to reinsert exactly one edge from G back into H, so that the shortest path from s to t in the resulting graph is as short as possible. Describe and analyze an algorithm that chooses the best edge to reinsert, in $O(E \log V)$ time.

6. When there is more than one shortest path from one node s to another node t, it is often convenient to choose a shortest path with the fewest edges; call this the ***best path*** from s to t. Suppose we are given a directed graph G with positive edge weights and a source vertex s in G. Describe and analyze an algorithm to compute *best* paths in G from s to every other vertex.

*7. (a) Prove that Ford's generic shortest-path algorithm (while the graph contains a tense edge, relax it) can take exponential time in the worst case when implemented with a stack instead of a priority queue (like Dijkstra) or a queue (like Shimbel). Specifically, for every positive integer n, construct a weighted directed n-vertex graph G_n, such that the stack-based shortest-path algorithm call RELAX $\Omega(2^n)$ times when G_n is the input graph. *[Hint: Towers of Hanoi.]*

   (b) Prove that Dijkstra's shortest-path algorithm can require exponential time in the worst case when edges are allowed to have negative weight. Specifically, for every positive integer n, construct a weighted directed n-vertex graph G_n, such that Dijkstra's algorithm calls RELAX $\Omega(2^n)$ times when G_n is the input graph. *[Hint: This is relatively easy if you've already solved part (a).]*

8. (a) Describe and analyze a modification of Shimbel's shortest-path algorithm that actually returns a negative cycle if any such cycle is reachable from s, or a shortest-path tree if there is no such cycle. The modified algorithm should still run in $O(VE)$ time.

   (b) Describe and analyze a modification of Shimbel's shortest-path algorithm that computes the correct shortest path distances from s to every other vertex of the input graph, even if the graph contains negative cycles. Specifically, if any walk from s to v contains a negative cycle, your algorithm should end with $dist(v) = -\infty$; otherwise, $dist(v)$ should contain the length of the shortest path from s to v. The modified algorithm should still run in $O(VE)$ time.

*(c) Repeat parts (a) and (b), but for Ford's generic shortest-path algorithm. You may assume that the unmodified algorithm halts in $O(2^V)$ steps if there is no negative cycle; your modified algorithms should also run in $O(2^V)$ time.

*9. Describe and analyze an efficient algorithm to compute the *number* of shortest paths between two specified vertices s and t in a directed graph G whose edges have positive weights. *[Hint: Which edges of G can lie on a shortest path from s to t?]*

10. You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cites that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving toward each other. But where *exactly* should you meet?

    You are given a weighted graph $G = (V, E)$, where the vertices V represent cities and the edges E represent roads that directly connect cities. Each edge e has a weight $w(e)$ equal to the time required to travel between the two cities. You are also given a vertex p, representing your starting location, and a vertex q, representing your friend's starting location.

    Describe and analyze an algorithm to find the target vertex t that allows you and your friend to meet as quickly as possible.

11. After a grueling algorithms midterm, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Champaign-Urbana. Unfortunately, there isn't a single bus that visits both your exam building and your home; you must transfer between bus lines at least once.

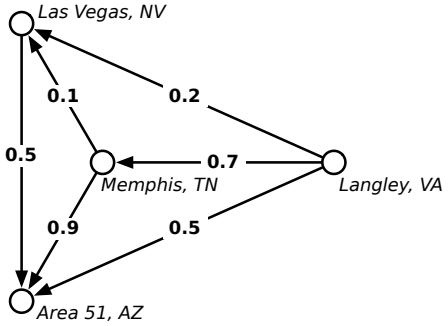
    Describe and analyze an algorithm to determine the sequence of bus rides that will get you home as early as possible, assuming there are b different bus lines, and each bus stops n times per day. Your goal is to minimize your *arrival time*, not the time you actually spend traveling. Assume that the buses run exactly on schedule, that you have an accurate watch, and that you are too tired to walk between bus stops.

12. After graduating you accept a job with Aerophobes-Я-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of all the flights on the planet.

    Suppose one of your customers wants to fly from city X to city Y. Describe an algorithm to find a sequence of flights that minimizes the *total time in transit*—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights. *[Hint: Modify the input data and apply Dijkstra's algorithm.]*

13. Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road *won't* be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Nevada. What route should he take so that he has the least chance of being abducted?

More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t.



For example, with the probabilities shown above, if Mulder tries to drive directly from Langley to Area 51, he has a 50% chance of getting there without being abducted. If he stops in Memphis, he has a $0.7 \times 0.9 = 63\%$ chance of arriving safely. If he stops first in Memphis and then in Las Vegas, he has a $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$ chance of being abducted! (That's how they got Elvis, you know.) Although this example is a dag, your algorithm must handle *arbitrary* directed graphs.

14. On an overnight camping trip in Sunnydale National Park, you are woken from a restless sleep by a scream. As you crawl out of your tent to investigate, a terrified park ranger runs out of the woods, covered in blood and clutching a crumpled piece of paper to his chest. As he reaches your tent, he gasps, "Get out. . . while. . . you. . . ", thrusts the paper into your hands, and falls to the ground. Checking his pulse, you discover that the ranger is stone dead.

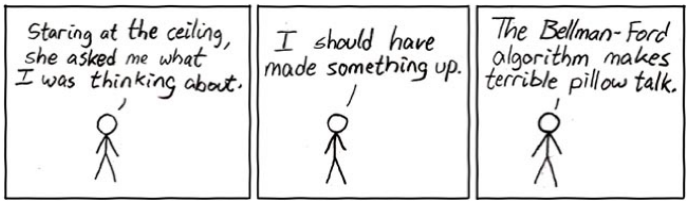
    You look down at the paper and recognize a map of the park, drawn as an undirected graph, where vertices represent landmarks in the park, and edges represent trails between those landmarks. (Trails start and end at landmarks and do not cross.) You recognize one of the vertices as your current location; several vertices on the boundary of the map are labeled EXIT.

    On closer examination, you notice that someone (perhaps the poor dead park ranger) has written a real number between 0 and 1 next to each vertex and each edge. A scrawled note on the back of the map indicates that a number next to an edge is the probability of encountering a vampire along the corresponding trail, and a number next to a vertex is the probability of encountering a vampire at the corresponding landmark. (Vampires can't stand each other's company, so you'll never see more than one vampire on the same trail or at the same landmark.) The note warns you that stepping off the marked trails will result in a slow and painful death.

    You glance down at the corpse at your feet. Yes, his death certainly looked painful. Wait, was that a twitch? Are his teeth getting longer? After driving a tent stake through the undead ranger's heart, you wisely decide to leave the park immediately.

    Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the total *expected number* of vampires encountered along

the path is as small as possible. *Be sure to account for **both** the vertex probabilities **and** the edge probabilities!*



— Randall Munroe, *xkcd* (http://xkcd.com/69/)

The tree which fills the arms grew from the tiniest sprout;
the tower of nine storeys rose from a (small) heap of earth;
the journey of a thousand li commenced with a single step.

— Lao-Tzu, *Tao Te Ching*, chapter 64 (6th century BC), translated by J. Legge (1891)

And I would walk five hundred miles,
And I would walk five hundred more,
Just to be the man who walks a thousand miles
To fall down at your door.

— The Proclaimers, "Five Hundred Miles (I'm Gonna Be)", *Sunshine on Leith* (2001)

Almost there. . . Almost there. . .

— Red Leader [Drewe Henley], *Star Wars* (1977)

2 All-Pairs Shortest Paths

In the previous lecture, we saw algorithms to find the shortest path from a source vertex s to a target vertex t in a directed graph. As it turns out, the best algorithms for this problem actually find the shortest path from s to every possible target (or from every possible source to t) by constructing a shortest path tree. The shortest path tree specifies two pieces of information for each node v in the graph:

- $dist(v)$ is the length of the shortest path (if any) from s to v;
- $pred(v)$ is the second-to-last vertex (if any) the shortest path (if any) from s to v.

In this lecture, we want to generalize the shortest path problem even further. In the *all pairs shortest path* problem, we want to find the shortest path from *every* possible source to *every* possible destination. Specifically, for every pair of vertices u and v, we need to compute the following information:

- $dist(u, v)$ is the length of the shortest path (if any) from u to v;
- $pred(u, v)$ is the second-to-last vertex (if any) on the shortest path (if any) from u to v.

For example, for any vertex v, we have $dist(v, v) = 0$ and $pred(v, v) = \text{NULL}$. If the shortest path from u to v is only one edge long, then $dist(u, v) = w(u \to v)$ and $pred(u, v) = u$. If there is *no* shortest path from u to v—either because there's no path at all, or because there's a negative cycle—then $dist(u, v) = \infty$ and $pred(v, v) = \text{NULL}$.

The output of our shortest path algorithms will be a pair of $V \times V$ arrays encoding all V^2 distances and predecessors. Many maps include a distance matrix—to find the distance from (say) Champaign to (say) Columbus, you would look in the row labeled 'Champaign' and the column labeled 'Columbus'. In these notes, I'll focus almost exclusively on computing the distance array. The predecessor array, from which you would compute the actual shortest paths, can be computed with only minor additions to the algorithms I'll describe (hint, hint).

2.1 Lots of Single Sources

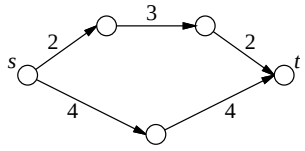
The obvious solution to the all-pairs shortest path problem is just to run a single-source shortest path algorithm V times, once for every possible source vertex! Specifically, to fill in the one-dimensional subarray $dist[s, \cdot]$, we invoke either Dijkstra's or Shimbel's algorithm starting at the source vertex s.

```
ObviousAPSP(V, E, w):
  for every vertex s
      dist[s, ·] ← SSSP(V, E, w, s)
```

The running time of this algorithm depends on which single-source shortest path algorithm we use. If we use Shimbel's algorithm, the overall running time is $\boldsymbol{\Theta(V^2E)} = O(V^4)$. If all the edge weights are non-negative, we can use Dijkstra's algorithm instead, which decreases the running time to $\boldsymbol{\Theta(VE + V^2 \log V)} = O(V^3)$. For graphs with negative edge weights, Dijkstra's algorithm can take exponential time, so we can't get this improvement directly.

2.2 Reweighting

One idea that occurs to most people is increasing the weights of all the edges by the same amount so that all the weights become positive, and then applying Dijkstra's algorithm. Unfortunately, this simple idea doesn't work. Different paths change by different amounts, which means the shortest paths in the reweighted graph may not be the same as in the original graph.



Increasing all the edge weights by 2 changes the shortest path s to t.

However, there is a more complicated method for reweighting the edges in a graph. Suppose each vertex v has some associated *cost* $c(v)$, which might be positive, negative, or zero. We can define a new weight function w' as follows:

$$w'(u \to v) = c(u) + w(u \to v) - c(v)$$

To give some intuition, imagine that when we leave vertex u, we have to pay an exit tax of $c(u)$, and when we enter v, we get $c(v)$ as an entrance gift.

Now it's not too hard to show that the shortest paths with the new weight function w' are exactly the same as the shortest paths with the original weight function w. In fact, for *any* path $u \rightsquigarrow v$ from one vertex u to another vertex v, we have

$$w'(u \rightsquigarrow v) = c(u) + w(u \rightsquigarrow v) - c(v).$$

We pay $c(u)$ in exit fees, plus the original weight of of the path, minus the $c(v)$ entrance gift. At every intermediate vertex x on the path, we get $c(x)$ as an entrance gift, but then immediately pay it back as an exit tax!

2.3 Johnson's Algorithm

Johnson's all-pairs shortest path algorithm finds a cost $c(v)$ for each vertex, so that when the graph is reweighted, every edge has non-negative weight.

Suppose the graph has a vertex s that has a path to every other vertex. Johnson's algorithm computes the shortest paths from s to every other vertex, using Shimbel's algorithm (which doesn't care if the edge weights are negative), and then sets $c(v) \leftarrow dist(s, v)$, so the new weight of every edge is

$$w'(u \to v) = dist(s, u) + w(u \to v) - dist(s, v).$$

Why are all these new weights non-negative? Because otherwise, Shimbel's algorithm wouldn't be finished! Recall that an edge $u \to v$ is *tense* if $dist(s, u) + w(u \to v) < dist(s, v)$, and that single-source shortest path algorithms eliminate all tense edges. The only exception is if the graph has a negative cycle, but then shortest paths aren't defined, and Johnson's algorithm simply aborts.

But what if the graph *doesn't* have a vertex s that can reach everything? No matter where we start Shimbel's algorithm, some of those vertex costs will be infinite. Johnson's algorithm avoids this problem by adding a new vertex s to the graph, with zero-weight edges going from s to every other vertex, but *no* edges going back into s. This addition doesn't change the shortest paths between any other pair of vertices, because there are no paths into s.

So here's Johnson's algorithm in all its glory.

```
JohnsonAPSP(V, E, w):
  create a new vertex s
  for every vertex v
      w(s→v) ← 0
      w(v→s) ← ∞
  dist[s, ·] ← Shimbel(V, E, w, s)
  if Shimbel found a negative cycle
      fail gracefully
  for every edge (u, v) ∈ E
      w'(u→v) ← dist[s, u] + w(u→v) − dist[s, v]
  for every vertex u
      dist[u, ·] ← Dijkstra(V, E, w', u)
      for every vertex v
          dist[u, v] ← dist[u, v] − dist[s, u] + dist[s, v]
```

The algorithm spends $\Theta(V)$ time adding the artificial start vertex s, $\Theta(VE)$ time running Shimbel, $O(E)$ time reweighting the graph, and then $\Theta(VE + V^2 \log V)$ running V passes of Dijkstra's algorithm. Thus, the overall running time is $\boldsymbol{\Theta(VE + V^2 \log V)}$.

2.4 Dynamic Programming

There's a completely different solution to the all-pairs shortest path problem that uses dynamic programming instead of a single-source algorithm. For *dense* graphs where $E = \Omega(V^2)$, the dynamic programming approach eventually leads to the same $O(V^3)$ running time as Johnson's algorithm, but with a much simpler algorithm. In particular, the new algorithm avoids Dijkstra's algorithm, which gets its efficiency from Fibonacci heaps, which are rather easy to screw up in the implementation. **In the rest of this lecture, I will assume that the input graph contains no negative cycles.**

As usual for dynamic programming algorithms, we first need to come up with a recursive formulation of the problem. Here is an "obvious" recursive definition for $dist(u, v)$:

$$dist(u, v) = \begin{cases} 0 & \text{if } u = v \\ \min\limits_{x \to v} \big(dist(u, x) + w(x \to v) \big) & \text{otherwise} \end{cases}$$

In other words, to find the shortest path from u to v, we consider all possible last edges $x \to v$ and recursively compute the shortest path from u to x. **Unfortunately, this recurrence doesn't work!** To compute $dist(u, v)$, we may need to compute $dist(u, x)$ for every other vertex x. But to compute $dist(u, x)$, we may need to compute $dist(u, v)$. We're stuck in an infinite loop!

To avoid this circular dependency, we need an additional parameter that decreases at each recursion, eventually reaching zero at the base case. One possibility is to include the number of edges in the shortest path as this third magic parameter, just as we did in the dynamic programming formulation of Shimbel's algorithm. Let $dist(u, v, k)$ denote the length of the shortest path from u to v that uses *at most* k *edges*. Since we know that the shortest path between any two vertices has at most $V - 1$ vertices, $dist(u, v, V - 1)$ is the actual shortest-path distance. As in the single-source setting, we have the following recurrence:

$$dist(u, v, k) = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{if } k = 0 \text{ and } u \neq v \\ \min\limits_{x \to v} \big(dist(u, x, k-1) + w(x \to v) \big) & \text{otherwise} \end{cases}$$

Turning this recurrence into a dynamic programming algorithm is straightforward. To make the algorithm a little shorter, let's assume that $w(v \to v) = 0$ for every vertex v. Assuming the graph is stored in an adjacency list, the resulting algorithm runs in $\boldsymbol{\Theta(V^2E)}$ ***time***.

```
DynamicProgrammingAPSP(V, E, w):
  for all vertices u
      for all vertices v
          if u = v
              dist[u, v, 0] ← 0
          else
              dist[u, v, 0] ← ∞

  for k ← 1 to V − 1
      for all vertices u
          dist[u, u, k] ← 0
          for all vertices v ≠ u
              dist[u, v, k] ← ∞
              for all edges x→v
                  if dist[u, v, k] > dist[u, x, k − 1] + w(x→v)
                      dist[u, v, k] ← dist[u, x, k − 1] + w(x→v)
```

This algorithm was first sketched by Shimbel in 1955; in fact, this algorithm is just running V different instances of Shimbel's single-source algorithm, one for each possible source vertex. Just as in the dynamic programming development of Shimbel's single-source algorithm, we don't actually need the inner loop over vertices v, and we only need a two-dimensional table. After the kth iteration of the main loop in the following algorithm, $dist[u, v]$ lies between the true shortest path distance from u to v and the value $dist[u, v, k]$ computed in the previous algorithm.

```
ShimbelAPSP(V, E, w):
  for all vertices u
      for all vertices v
          if u = v
              dist[u, v] ← 0
          else
              dist[u, v] ← ∞
  for k ← 1 to V − 1
      for all vertices u
          for all edges x→v
              if dist[u, v] > dist[u, x] + w(x→v)
                  dist[u, v] ← dist[u, x] + w(x→v)
```

2.5 Divide and Conquer

But we can make a more significant improvement. The recurrence we just used broke the shortest path into a slightly shorter path and a single edge, by considering all predecessors. Instead, let's break it into two shorter paths at the *middle* vertex of the path. This idea gives us a different recurrence for $dist(u, v, k)$. Once again, to simplify things, let's assume $w(v \to v) = 0$.

$$dist(u, v, k) = \begin{cases} w(u \to v) & \text{if } k = 1 \\ \min_x \big(dist(u, x, k/2) + dist(x, v, k/2) \big) & \text{otherwise} \end{cases}$$

This recurrence only works when k is a power of two, since otherwise we might try to find the shortest path with a fractional number of edges! But that's not really a problem, since $dist(u, v, 2^{\lceil \lg V \rceil})$ gives us the overall shortest distance from u to v. Notice that we use the base case $k = 1$ instead of $k = 0$, since we can't use half an edge.

Once again, a dynamic programming solution is straightforward. Even before we write down the algorithm, we can tell the running time is $\mathbf{\Theta(V^3 \log V)}$—we consider V possible values of u, v, and x, but only $\lceil \lg V \rceil$ possible values of k.

```
FastDynamicProgrammingAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)
  for i ← 1 to ⌈lg V⌉          ⟨⟨k = 2^i⟩⟩
      for all vertices u
          for all vertices v
              dist[u, v, i] ← ∞
              for all vertices x
                  if dist[u, v, i] > dist[u, x, i − 1] + dist[x, v, i − 1]
                      dist[u, v, i] ← dist[u, x, i − 1] + dist[x, v, i − 1]
```

This algorithm is ***not*** the same as V invocations of any single-source algorithm; in particular, the innermost loop does not simply relax tense edges. However, we can remove the last dimension of the table, using $dist[u, v]$ everywhere in place of $dist[u, v, i]$, just as in Shimbel's single-source algorithm, thereby reducing the space from $O(V^3)$ to $O(V^2)$.

```
FastShimbelAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)
  for i ← 1 to ⌈lg V⌉
      for all vertices u
          for all vertices v
              for all vertices x
                  if dist[u, v] > dist[u, x] + dist[x, v]
                      dist[u, v] ← dist[u, x] + dist[x, v]
```

This faster algorithm was discovered by Leyzorek *et al.* in 1957, in the same paper where they describe Dijkstra's algorithm.

2.6 Aside: 'Funny' Matrix Multiplication

There is a very close connection (first observed by Shimbel, and later independently by Bellman) between computing shortest paths in a directed graph and computing powers of a square matrix. Compare the following algorithm for multiplying two $n \times n$ matrices A and B with the inner loop of our first dynamic programming algorithm. (I've changed the variable names in the second algorithm slightly to make the similarity clearer.)

```
MatrixMultiply(A, B):
  for i ← 1 to n
      for j ← 1 to n
          C[i, j] ← 0
          for k ← 1 to n
              C[i, j] ← C[i, j] + A[i, k] · B[k, j]
```

```
APSPInnerLoop:
  for all vertices u
      for all vertices v
          D'[u, v] ← ∞
          for all vertices x
              D'[u, v] ← min{D'[u, v], D[u, x] + w[x, v]}
```

The *only* difference between these two algorithms is that we use addition instead of multiplication and minimization instead of addition. For this reason, the shortest path inner loop is often referred to as 'funny' matrix multiplication.

DynamicProgrammingAPSP is the standard iterative algorithm for computing the $(V-1)$th 'funny power' of the weight matrix w. The first set of for loops sets up the 'funny identity matrix', with zeros on the main diagonal and infinity everywhere else. Then each iteration of the second main for loop computes the next 'funny power'. FastDynamicProgrammingAPSP replaces this iterative method for computing powers with repeated squaring, exactly like we saw at the beginning of the semester. The fast algorithm is simplified slightly by the fact that unless there are negative cycles, every 'funny power' after the Vth is the same.

There are faster methods for multiplying matrices, similar to Karatsuba's divide-and-conquer algorithm for multiplying integers. (Google for 'Strassen's algorithm'.) Unfortunately, these algorithms us subtraction, and there's no 'funny' equivalent of subtraction. (What's the inverse operation for min?) So at least for general graphs, there seems to be no way to speed up the inner loop of our dynamic programming algorithms.

Fortunately, this isn't true. There a beautiful randomized algorithm, discovered by Alon, Galil, Margalit, and Noar[1], that computes all-pairs shortest paths in undirected graphs in $O(M(V)\log^2 V)$ expected time, where $M(V)$ is the time to multiply two $V \times V$ integer matrices. A simplified version of this algorithm for *unweighted* graphs was discovered by Seidel.[2]

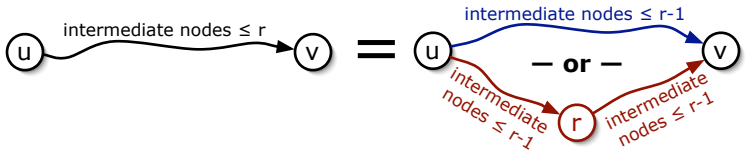
2.7 Floyd-(Roy-Kleene-)Warshall

Our fast dynamic programming algorithm is still a factor of $O(\log V)$ slower than Johnson's algorithm. A different formulation that removes this logarithmic factor was proposed in 1962 by Robert Floyd, slightly generalizing an algorithm of Stephen Warshall published earlier in the same year. (In fact, Warshall's algorithm was independently discovered by Bernard Roy in 1959, but the underlying technique was used even earlier by Stephen Kleene[3] in 1951.) Warshall's (and Roy's and Kleene's) insight was to use a different third parameter in the dynamic programming recurrence.

Number the vertices arbitrarily from 1 to V. For every pair of vertices u and v and every integer r, we define a path $\pi(u, v, r)$ as follows:

> $\pi(u, v, r) :=$ the shortest path from u to v where every intermediate vertex (that is, every vertex except u and v) is numbered at most r.

If $r = 0$, we aren't allowed to use any intermediate vertices, so $\pi(u, v, 0)$ is just the edge (if any) from u to v. If $r > 0$, then either $\pi(u, v, r)$ goes through the vertex numbered r, or it doesn't. If $\pi(u, v, r)$ does contain vertex r, it splits into a subpath from u to r and a subpath from r to v, where every intermediate vertex in these two subpaths is numbered at most $r-1$. Moreover, the subpaths are as short as possible with this restriction, so they must be $\pi(u, r, r-1)$ and $\pi(r, v, r-1)$. On the other hand, if $\pi(u, v, r)$ does not go through vertex r, then every intermediate vertex in $\pi(u, v, r)$ is numbered at most $r-1$; since $\pi(u, v, r)$ must be the *shortest* such path, we have $\pi(u, v, r) = \pi(u, v, r-1)$.



Recursive structure of the restricted shortest path $\pi(u, v, r)$.

This recursive structure implies the following recurrence for the length of $\pi(u, v, r)$, which we will denote by $dist(u, v, r)$:

$$dist(u, v, r) = \begin{cases} w(u \to v) & \text{if } r = 0 \\ \min\left\{dist(u, v, r-1),\ dist(u, r, r-1) + dist(r, v, r-1)\right\} & \text{otherwise} \end{cases}$$

[1] Noga Alon, Zvi Galil, Oded Margalit*, and Moni Naor. Witnesses for Boolean matrix multiplication and for shortest paths. *Proc. 33rd FOCS* 417-426, 1992. See also Noga Alon, Zvi Galil, Oded Margalit*. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences* 54(2):255–262, 1997.

[2] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400-403, 1995. This is one of the few algorithms papers where (in the conference version at least) the algorithm is completely described and analyzed *in the abstract* of the paper.

[3] Pronounced "clay knee", not "clean" or "clean-ee" or "clay-nuh" or "dimaggio".

We need to compute the shortest path distance from u to v with no restrictions, which is just $dist(u, v, V)$. Once again, we should immediately see that a dynamic programming algorithm will implement this recurrence in $\mathbf{\Theta(V^3)}$ time.

```
FloydWarshall(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)

  for r ← 1 to V
      for all vertices u
          for all vertices v
              if dist[u, v, r − 1] < dist[u, r, r − 1] + dist[r, v, r − 1]
                  dist[u, v, r] ← dist[u, v, r − 1]
              else
                  dist[u, v, r] ← dist[u, r, r − 1] + dist[r, v, r − 1]
```

Just like our earlier algorithms, we can simplify the algorithm by removing the third dimension of the memoization table. Also, because the vertex numbering was chosen arbitrarily, there's no reason to refer to it explicitly in the pseudocode.

```
FloydWarshall2(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)

  for all vertices r
      for all vertices u
          for all vertices v
              if dist[u, v] > dist[u, r] + dist[r, v]
                  dist[u, v] ← dist[u, r] + dist[r, v]
```

Now compare this algorithm with FastShimbelAPSP. Instead of $O(\log V)$ passes through all triples of vertices, FloydWarshall2 only requires a single pass, but only because it uses a different nesting order for the three for-loops!

2.8 Converting DFAs to regular expressions

Floyd's algorithm is a special case of a more general method for solving problems involving paths between vertices in graphs. The earliest example (that I know of) of this technique is an 1951 algorithm of Stephen Kleene to convert a deterministic finite automaton into an equivalent regular expression.

Recall that a deterministic finite automaton (DFA) formally consists of the following components:

- A finite set Σ, called the ***alphabet***, and whose elements we call ***symbols***.
- A finite set Q, whose elements are called ***states***.
- An ***initial state*** $s \in Q$.
- A subset $A \subseteq Q$ of ***accepting states***.
- A ***transition function*** $\delta : Q \times \Sigma \to Q$.

The ***extended transition function*** $\delta^* : Q \times \Sigma^* \to Q$ is recursively defined as follows:

$$\delta^*(q, w) := \begin{cases} q & \text{if } w = \varepsilon, \\ \delta^*(\delta(q, a), x) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^*. \end{cases}$$

Finally, a DFA ***accepts*** a string $w \in \Sigma^*$ if and only if $\delta^*(s, w) \in A$.

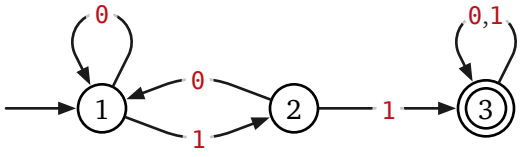
Equivalently, a DFA is a directed (multi-)graph with labeled edges whose vertices are the states, such that each vertex (state) has exactly one outgoing edge (transition) labeled with each symbol in Σ. There is a special "start" vertex s, and a subset A of the vertices are marked as "accepting". For any string $w \in \Sigma^*$, there is a unique walk starting at s whose sequence of edge labels is w. The DFA accepts w if and only if this walk ends at a state in A.

Kleene described the following algorithm to convert DFAs into equivalent regular expressions. Suppose we are given a DFA M with n states, where (without loss of generality) each state is identified by an integer between 1 and n. Let $\boldsymbol{L(i, j, r)}$ denote the set of all strings that describe walks in M that start at state i and end at state j, such that every intermediate state has index at most r. Thus, the language accepted by M is precisely

$$L(M) = \bigcup_{q \in A} L(s, q, n).$$

We prove inductively that every language $L(i, j, r)$ is regular, by recursively constructing a regular expression $R(i, j, r)$ that represents $L(i, j, r)$. There are two cases to consider.

- First, suppose $r = 0$. The language $L(i, j, 0)$ contains the labels walks from state i to state j that do not pass through *any* intermediate states. Thus, every string in $L(i, j, 0)$ has length at most 1. Specifically, for any symbol $a \in \Sigma$, we have $a \in L(i, j, 0)$ if and only if $\delta(i, a) = j$, and we have $\varepsilon \in L(i, j, 0)$ if and only if $i = j$. Thus, $L(i, j, 0)$ is always finite, and therefore regular.



An example DFA

For example, the DFA sown on the next page defines the following regular languages $L(i, j, 0)$.

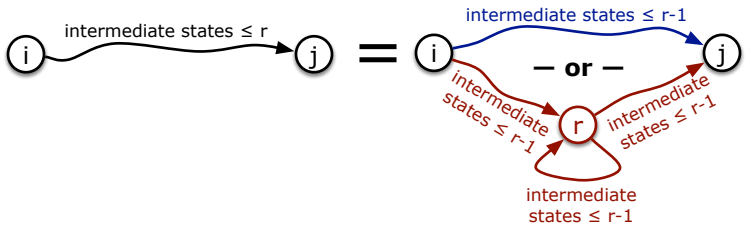
$$\begin{array}{lll} R[1,1,0] = \varepsilon + 0 & R[2,1,0] = 0 & R[3,1,0] = \varnothing \\ R[1,2,0] = 1 & R[2,2,0] = \varepsilon & R[3,2,0] = \varnothing \\ R[1,3,0] = \varnothing & R[2,3,0] = 1 & R[3,3,0] = \varepsilon + 0 + 1 \end{array}$$

- Now suppose $r > 0$. Each string $w \in L(i, j, r)$ describes a walk from state i to state j where every intermediate state has index at most r. If this walk does not pass through state r, then $w \in L(i, j, r-1)$ by definition. Otherwise, we can split w into a sequence of substrings $w = w_1 \cdot w_2 \cdots w_\ell$ at the points where the walk visits state r. These substrings have the following properties:
  - The prefix w_1 describes a walk from state i to state r and thus belongs to $L(i, r, r-1)$.

- The suffix w_ℓ describes a walk from state r to state j and thus belongs to $L(r, j, r-1)$.
- For every other index k, the substring w_k describes a walk from state r to state r and thus belongs to $L(r, r, r-1)$.

We conclude that

$$L(i,j,r) \;=\; L(i,j,r-1) \;\cup\; L(i,r,r-1) \bullet L(r,r,r-1)^* \bullet L(r,j,r-1).$$



Recursive structure of the regular language $L(i,j,r)$.

Putting these pieces together, we can recursively define a regular ***expression*** $R(i,j,r)$ that describes the language $L(i,j,r)$, as follows:

$$R(i,j,r) := \begin{cases} \varepsilon + \sum_{\delta(i,a)=j} a & \text{if } r = 0 \text{ and } i = j \\ \sum_{\delta(i,a)=j} a & \text{if } r = 0 \text{ and } i \neq j \\ R(i,j,r-1) + R(i,r,r-1) \bullet R(r,r,r-1)^* \bullet R(r,j,r-1) & \text{otherwise} \end{cases}$$

Kleene's algorithm evaluates this recurrence bottom-up using the natural dynamic programming algorithm. We memoize the previous recurrence into a three-dimensional array $R[1..n, 1..n, 0..n]$, which we traverse by increasing r in the outer loop, and in arbitrary order in the inner two loops.

```
KLEENE(Σ, n, δ, F):
  ⟨⟨Base cases⟩⟩
  for i ← 1 to n
      for j ← 1 to n
          if i = j then R[i,j,0] ← ε else R[i,j,0] ← ∅
          for all symbols a ∈ Σ
              if δ[i,a] = j
                  R[i,j,0] ← R[i,j,0] + a
  ⟨⟨Recursive cases⟩⟩
  for r ← 1 to n
      for i ← 1 to n
          for j ← 1 to n
              R[i,j,r] ← R[i,j,r-1] + R[i,r,r-1] • R[r,r,r-1]* • R[r,j,r-1]
  ⟨⟨Assemble the final result⟩⟩
  R ← ∅
  for q ← 0 to n-1
      if q ∈ F
          R ← R + R[1,q,n-1]
      return R
```

For purposes of analysis, let's assume the alphabet Σ has constant size. Assuming each alternation ($+$), concatenation ($\bullet$), and Kleene closure (*) operation requires constant time, the entire algorithm runs in $\boldsymbol{O(n^3)}$ ***time***.

However, regular expressions over an alphabet Σ are normally represented either as standard strings (arrays) over the larger alphabet $\Sigma \cup \{+, \bullet, *, (,), \varepsilon\}$, or as regular expression *trees*, whose internal nodes are $+$, $\bullet$, and $*$ operators and whose leaves are symbols and εs. In either representation, the regular expressions in Kleene's algorithm grow in size by roughly a factor of 4 in each iteration of the outer loop, at least in the worst case. Thus, in the worst case, each regular expression $R[i, j, r]$ has size $O(4^r)$, the size of the final output expression is $O(4^n n)$, and entire algorithm runs in $\boldsymbol{O(4^n n^2)}$ ***time***.

So we shouldn't do this. After all, the running time is exponential, and exponential time is bad. Right? Moreover, this exponential dependence is unavoidable; Hermann Gruber and Markus Holzer proved in 2008[4] that there are n-state DFAs over the binary alphabet $\{0, 1\}$ such that any equivalent regular expression has length $2^{\Omega(n)}$.

Well, maybe it's not so bad. The output regular expression has exponential size *because it contains multiple copies of the same subexpressions*; similarly, the regular expression tree has exponential size *because it contains multiples copies of several subtrees*. But it's precisely this exponential behavior that we use dynamic programming to avoid! In fact, it's not hard to modify Kleene's algorithm to compute a ***regular expression dag*** of size $O(n^3)$, ***in*** $\boldsymbol{O(n^3)}$ ***time***, that (intuitively) contains each subexpression $R[i, j, r]$ only once. This regular expression dag has exactly the same relationship to the regular expression *tree* as the dependency graph of Kleene's algorithm has to the recursion tree of its underlying recurrence.

Exercises

1. All of the algorithms discussed in this lecture fail if the graph contains a negative cycle. Johnson's algorithm detects the negative cycle in the initialization phase (via Shimbel's algorithm) and aborts; the dynamic programming algorithms just return incorrect results. However, all of these algorithms can be modified to return correct shortest-path distances, even in the presence of negative cycles. Specifically, if there is a path from vertex u to a negative cycle and a path from that negative cycle to vertex v, the algorithm should report that $dist[u, v] = -\infty$. If there is no directed path from u to v, the algorithm should return $dist[u, v] = \infty$. Otherwise, $dist[u, v]$ should equal the length of the shortest directed path from u to v.

   (a) Describe how to modify Johnson's algorithm to return the correct shortest-path distances, even if the graph has negative cycles.

   (b) Describe how to modify the Floyd-Warshall algorithm (FLOYDWARSHALL2) to return the correct shortest-path distances, even if the graph has negative cycles.

2. All of the shortest-path algorithms described in this note can also be modified to return an explicit description of some negative cycle, instead of simply reporting that a negative cycle exists.

   (a) Describe how to modify Johnson's algorithm to return either the matrix of shortest-path distances or a negative cycle.

   (b) Describe how to modify the Floyd-Warshall algorithm (FLOYDWARSHALL2) to return either the matrix of shortest-path distances or a negative cycle.

[4] Hermann Gruber and Markus Holzer. Finite automata, digraph connectivity, and regular expression size. *Proc. 35th ICALP*, 39–50, 2008.

If the graph contains more than one negative cycle, your algorithms may choose one arbitrarily.

3. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero. Suppose the vertices of G are partitioned into k disjoint subsets $V_1, V_2, \ldots, V_k$; that is, every vertex of G belongs to exactly one subset V_i. For each i and j, let $\delta(i, j)$ denote the minimum shortest-path distance between vertices in V_i and vertices in V_j:

$$\delta(i, j) = \min \left\{ dist(u, v) \;\middle|\; u \in V_i \text{ and } v \in V_j \right\}.$$

Describe an algorithm to compute $\delta(i, j)$ for all i and j in time $O(V^2 + kE \log E)$.

4. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero.

   (a) How could we delete an arbitrary vertex v from this graph, without changing the shortest-path distance between any other pair of vertices? Describe an algorithm that constructs a directed graph $G' = (V', E')$ with weighted edges, where $V' = V \setminus \{v\}$, and the shortest-path distance between any two nodes in H is equal to the shortest-path distance between the same two nodes in G, in $O(V^2)$ time.

   (b) Now suppose we have already computed all shortest-path distances in G'. Describe an algorithm to compute the shortest-path distances from v to every other vertex, and from every other vertex to v, in the original graph G, in $O(V^2)$ time.

   (c) Combine parts (a) and (b) into another all-pairs shortest path algorithm that runs in $O(V^3)$ time. (The resulting algorithm is *not* the same as Floyd-Warshall!)

5. In this problem we will discover how you, too, can be employed by Wall Street and cause a major economic collapse! The *arbitrage* business is a money-making scheme that takes advantage of differences in currency exchange. In particular, suppose that 1 US dollar buys 120 Japanese yen; 1 yen buys 0.01 euros; and 1 euro buys 1.2 US dollars. Then, a trader starting with \$1 can convert his money from dollars to yen, then from yen to euros, and finally from euros back to dollars, ending with \$1.44! The cycle of currencies $\$ \to ¥ \to € \to \$$ is called an ***arbitrage cycle***. Of course, finding and exploiting arbitrage cycles before the prices are corrected requires extremely fast algorithms.

   Suppose n different currencies are traded in your currency market. You are given the matrix $R[1..n, 1..n]$ of exchange rates between every pair of currencies; for each i and j, one unit of currency i can be traded for $R[i, j]$ units of currency j. (Do *not* assume that $R[i, j] \cdot R[j, i] = 1$.)

   (a) Describe an algorithm that returns an array $V[1..n]$, where $V[i]$ is the maximum amount of currency i that you can obtain by trading, starting with one unit of currency 1, assuming there are no arbitrage cycles.

   (b) Describe an algorithm to determine whether the given matrix of currency exchange rates creates an arbitrage cycle.

   (c) Modify your algorithm from part (b) to actually return an arbitrage cycle, if it exists.

*6. Let $G = (V, E)$ be an undirected, unweighted, connected, n-vertex graph, represented by the adjacency matrix $A[1..n, 1..n]$. In this problem, we will derive Seidel's sub-cubic algorithm to compute the $n \times n$ matrix $D[1..n, 1..n]$ of shortest-path distances using fast matrix multiplication. Assume that we have a subroutine MatrixMultiply that multiplies two $n \times n$ matrices in $\Theta(n^\omega)$ time, for some unknown constant $\omega \geq 2$.[5]

(a) Let G^2 denote the graph with the same vertices as G, where two vertices are connected by a edge if and only if they are connected by a path of length at most 2 in G. Describe an algorithm to compute the adjacency matrix of G^2 using a single call to MatrixMultiply and $O(n^2)$ additional time.

(b) Suppose we discover that G^2 is a complete graph. Describe an algorithm to compute the matrix D of shortest path distances in $O(n^2)$ additional time.

(c) Let D^2 denote the (recursively computed) matrix of shortest-path distances in G^2. Prove that the shortest-path distance from node i to node j is either $2 \cdot D^2[i, j]$ or $2 \cdot D^2[i, j] - 1$.

(d) Suppose G^2 is not a complete graph. Let $X = D^2 \cdot A$, and let $\deg(i)$ denote the degree of vertex i in the original graph G. Prove that the shortest-path distance from node i to node j is $2 \cdot D^2[i, j]$ if and only if $X[i, j] \geq D^2[i, j] \cdot \deg(i)$.

(e) Describe an algorithm to compute the matrix of shortest-path distances in G in $O(n^\omega \log n)$ time.

[5] The matrix multiplication algorithm you already know runs in $\Theta(n^3)$ time, but this is not the fastest algorithm known. The current record is $\omega \approx 2.3727$, due to Virginia Vassilevska Williams. Determining the smallest possible value of ω is a long-standing open problem; many people believe there is an undiscovered $O(n^2)$-time algorithm for matrix multiplication.