Dijkstra's Algorithm

Last time we saw two methods to solve the all-pairs shortest path problem: Min-plus matrix powering in $O(n^3 \log n)$ time and the Floyd-Warshall algorithm in $O(n^3)$ time. Neither of these algorithms were able to take advantage of a *sparse* input graph – a situation where the number of edges e might be $o(n^2)$.

Today we'll see a dynamic programming algorithm due to Dijkstra that solves the *single-source* shortest path problem in $O(e + n \log n)$ time. (Actually, we'll only discuss an implementation that takes slightly longer, $O(e \log n)$ time.) If $e = o(n^2)$, we can solve the all-pairs problem in $o(n^3)$ time by solving the single-source problem for each of the n possible sources.

The basic strategy is a bit of a hybrid of ideas we've seen already. As in the Prim algorithm discussed in HW#2, we expand a set of vertices from just the source to the entire set, making a locally optimal choice at each step and ensuring that we have a globally optimal solution inside the set. As in the Floyd-Warshall algorithm, at intermediate stages we will consider the shortest paths among those that use only a given set of intermediate vertices. But in contrast to Floyd-Warshall, these sets of intermediate vertices will emerge from the algorithm rather than being predetermined.

Let G be our graph and s our designated source. We let X be our growing set of vertices, originally just $\{s\}$ and eventually all of V.

Our key data structure will be an array d that will give a distance $d[v]$ for each vertex v. At any time during the run of the algorithm, $d[v]$ will be the distance from s to v using *only intermediate vertices in* X. Thus initially $d[v]$ is:

- 0, if $v = s$
- the length of the edge from s to v, if any
- ∞, if there is no such edge

The basic step of the algorithm will be to choose another vertex to add to X, and update the array d accordingly. We make a *greedy* choice of vertex, letting u be the vertex in $V \setminus X$ such that $d[u]$ is smallest, and setting X to be $X \cup \{u\}$.

The update step is to take each vertex v in $V \setminus X$ and consider whether adding u to X requires us to change $d[v]$. It suffices to look at $d[u]$ plus the length of the edge from u to v, if any, and replace $d[v]$ by this if it is an improvement.

Why couldn't some other path through X from s to v be better than this? To improve on the path that gave rise to the old value of $d[v]$, it would have to visit u, and could visit u only once (since we are assuming non-negative weights). The best way to u through X takes distance $d[u]$. If we went from s to u and then to some other vertex w in X before going to v, we would have been better off going directly from s to w, since:

Lemma: As X grows, $d[v]$ never increases. It reaches its final value when v enters X, and at that time every vertex in X has distance at most $d[v]$ and every vertex outside X has distance at least $d[v]$.

Correctness of Dijkstra's Algorithm

If we believe that $d[v]$ always represents the shortest distance from s by paths that only use intermediate vertices in X, then by the time that $X = V$ we must have $d[v]$ as the shortest distance overall. But just to be sure, let's look at an arbitrary v and make sure that $d[v]$ gets set correctly.

First note that $d[v]$ can never be set too *small*, since the array d is only set on the evidence of an actual path from s. We must make sure that any *other* path from s to v must have length at least the final value of $d[v]$.

If there are any vertices where $d[v]$ is set incorrectly, let v be one whose $d[v]$ is smallest. Then we can assume that all u with $d[u] < d[v]$ have their distances correct. At the time that v is added to X, all such u must already be in X, since v is the d-minimum vertex not in X. Now consider any path from s to v. Let (x, y) be the edge on which is first leaves X (we may have $s = x$ or $y = v$). If $y \neq v$, the length of the path is at least $d[y]$, and $d[y] \geq d[v]$. Otherwise the length of the path is at least $d[x] + e(x, v)$. When x entered X, $d[v]$ was compared against this very value. So again we know that this other path has length at least $d[v]$.

Implementation of Dijkstra's Algorithm

While we maintain the array d, we also have to keep the set $V \setminus X$ and carry out two operations on it:

- Find and remove the item u of $V \setminus X$ such that $d[u]$ is minimal, and
- Find an item v in $V \setminus X$ and decrease $d[v]$ if necessary.

These are the two characteristic operations of a **priority queue**. If we have a set of tasks, for example, each with a priority, we may have to find and perform the task of highest priority, or increase the priority of some task. (We could also consider decreasing a priority, but we don't need that for Dijkstra and it adds complications.)

In the Dijkstra algorithm we will set up a priority queue of at most $n - 1$ items. We will need to carry out $n - 1$ REMOVE-MIN operations, and e DECREASE-KEY operations. (Why e? Because we only need to update $d[v]$ if there is an edge to v from the vertex u currently being added to X.)

A simple implementation of a priority queue uses a **heap**. A heap is a binary tree of items, where each item has a value that is smaller than or equal to that of either of its children. (That's the case for a **min-heap** – in a **max-heap** the parent's value would be greater than or equal to the childrens'.) We also insist that the tree be as close as possible to balanced, with the bottom level of leaves left-justified.

Clearly the item with the smallest value must be at the root. We can easily remove this item, but we then have to restore the heap property for the smaller remaining tree. What we do is to move the rightmost leaf to the root, then **promote** items over their parents as necessary to restore the heap property. To begin, if either of the root's children is smaller than the root then we swap the root with the smaller of the two. Then we look at the old root's new children and swap the old root with the smaller of them if needed. In this way the root item (the original rightmost leaf) moves down until it is in an acceptable position. This might take $O(\log n)$ operations.

When an item has its value decreased, the shape of the heap has not changed but that item might need to move up. We can have it do so by swapping it with its parent until the heap property is restored. We can check that doing this gets rid of the single violation of the heap property without creating new ones.

With a heap implementation, connected by pointers to a single array of d values, the Dijkstra algorithm has $O(n)$ phases and $O(e)$ total updates of d values, taking $O(e \log n)$ time total.

In principle, but not generally in practice, the performance of the Dijkstra algorithm can be improved if we implement the priority queue using **Fibonacci heaps**. Such a priority queue keeps a number of heaps, linked by pointers, instead of just one. Individual operations on the queue may take more than constant time, but it turns out that the *amortized time* for a sequence of operations is $O(1)$ per operation. That is, any sequence of s REMOVE-MIN and DECREASE-KEY operations on the priority queue takes $O(s)$ total time. The proof of this involves a potential function, like the one we discussed for the tree-compression algorithm for the union-find problem.

With this improvement, the running time of Dijkstra's algorithm drops to $O(e)$ on a weighted graph with n vertices and e edges. In practice, the overhead of Fibonacci heaps makes them impractical, and ordinary heaps are usually used.

The Bellman-Ford Algorithm

Here is another dynamic programming algorithm for the all-pairs shortest-path problem, due to Bellman and Ford. It is slightly slower than Dijkstra's algorithm but is more general because it can handle **negative weights** as long as there are no **negative cycles**.

Why would we care about shortest paths where distances could be negative? Consider a graph where the nodes represent various different assets and the edges represent possible transactions trading one asset for another. The edge weight could represent the potential profit or loss from a transaction, and these weights could change rapidly with the market. A "shortest" path from i to j would represent the most economical way to convert asset i to asset j. A negative cycle would represent an arbitrage opportunity, which we would expect not to persist for very long if the traders are paying attention.

CLRS and Kleinberg–Tardos give different presentations of the Bellman-Ford algorithm, and I will follow KT here. (This algorithm isn't in the Adler notes.) We'll solve the all-pairs problem by solving the n single-**destination** problems separately.

Fix a destination vertex t. For any vertex v and number i, define $Opt(v, i)$ to be the shortest distance from v to t using a path of *at most* i *edges*. Thus $Opt(v, 1)$ is the length of the edge from v to t if there is one, and $Opt(v, n-1)$ is the actual shortest-path distance. (If there are no negative cycles, the shortest path must be a **simple path**, never revisiting a vertex.) Note that we also compute $Opt(t, i)$ for each i – if one of these is ever negative we have found a negative cycle and the calculation fails.

All we need to do is to compute $Opt(v, i+1)$ for each v, given that we have already found $Opt(v, i)$ for each v. This is easy. One candidate for $Opt(v, i+1)$ is just $Opt(v, i)$ – it may be that no path with $i+1$ edges is shorter than the best path with i edges. For every edge (v, w) out of v, we have another candidate path, whose length is the edge weight $e(v, w)$ plus $Opt(w, i)$. We find the smallest path weight among these candidates and set $Opt(v+1)$ to that.

Correctness is obvious – by induction on i we prove that each value $Opt(v, i)$ is indeed the least weight of any path of i or fewer edges from v to t.

How long does this take? We have a phase of computation for each i, and in each phase we must check one candidate for each vertex and one for each edge, requiring $O(n+e)$ time ($O(e)$ for a connected graph). Thus our total time for the single-destination problem is $O(ne)$, and for the all-pairs problem $O(n^2e)$. By contrast, Dijkstra took $O(e \log n)$ for a single source and $O(en \log n)$ for all pairs, but could not handle negative weights.

Another advantage of Bellman-Ford is its **local** nature. Each node has its own estimate of the distance to t, which it can maintain by getting information from its neighbors. Algorithms similar to this are used for routing in computer networks, particular when the membership of the network and the relative distances change frequently.

Seidel's Algorithm

When we used min-plus matrix powering to solve the all-pairs shortest path problem in $O(n^3 \log n)$ time, we noted that we were forced to use ordinary matrix multiplication rather than subcubic methods such as Strassen's. Could we, at least in principle, use subcubic matrix multiplication to improve the running time for all-pairs shortest path?

What about just the problem of determining the transitive closure of a graph? Warshall's algorithm takes $O(n^3)$ time, and boolean matrix powering takes $O(n^3 \log n)$ using ordinary matrix multiplication. And the boolean semiring, like the min-plus semiring, does not support subtraction, so we cannot use the Strassen method directly.

But there *is* a way to use **integer** matrix multiplication to simulate the boolean version. If A and B are boolean matrices, and we compute $C = AB$ using integer matrix multiplication, then c_{ij} tells us the *number* of paths from i to j. All we need to do is change any nonzero entries of C to ones, and we have the correct boolean result.

We have to be a bit careful in our implementation with regard to how we represent integers as finite bit strings. For example, suppose we used integer matrix powering to compute the transitive closure of a graph, representing the integers as Java `long` variables, and the graph happened to have exactly 2^{64} paths from i to j? But we know that the integer product of *two* boolean matrices can have no entry larger than n, so we could carry out the computation modulo $n + 1$ or any larger number. We could implement powering by repeated squaring, converting each integer matrix result to a boolean matrix before proceeding.

Theorem: Let $\mu(n)$ be the time complexity of integer matrix multiplication. Then two n by n boolean matrices may be multiplied in $O(\mu(n))$ time, and we can compute the reflexive transitive closure of an n-vertex graph in $O(\mu(n) \log n)$ time.

Today we will present an algorithm due to Seidel that uses subcubic matrix multiplication to attack the all-pairs shortest path problem. But this algorithm only works on a special case of the problem, when the graph is *connected*, *undirected*, and *unweighted*. The matrices we multiply, two at a time, will be either boolean or integer. The integer matrices will have relatively small entries, at most n^2.

Remember that Strassen's method, which we saw, shows $\mu(n)$ to be $O(n^{2.81\ldots})$, and the best known method (due to Coppersmith and Winograd) shows that $\mu(n) = O(n^{2.376})$. The best lower bound is only $\Omega(n^2)$, and the true asymptotic value of $\mu(n)$ is unknown.

What will a single matrix multiplication do for us? If A is the matrix of G, then the boolean A^2 tells us about paths of length *exactly two*. This suggests a recursive attack on the problem. Let G_2 be the graph that consists of the edges of G together with an edge short-cutting every two-step path (except for those from vertices to themselves). The adjacency matrix $M[G_2]$ of G_2 is easy to calculate in $O(\mu(n))$ time – we take A^2, component-wise OR it with A, and zero out the main diagonal. This takes only $O(n^2)$ on top of the one boolean matrix multiplication.

Now we'll recursively apply our algorithm to find the shortest-path distance matrix of G_2. The distance from i to j in G_2 is about half the distance from i to j in G – more precisely, it is the *ceiling* of half the distance. This is because a path of length $2d$ in G corresponds to a path of d shortcut edges in G_2, and a path of length $2d + 1$ in G corresponds to a path of d shortcut edges and one ordinary edge in G_2.

In order to compute the exact distances in G, then, we will need only one bit of additional information for each i and j – whether the shortest-path distance from i to j is odd or even. We define a matrix $P[G]$ to be $D[G]$ reduced modulo 2. So $P[G]$ will have a 1 in the (i, j) entry exactly when the shortest-path distance from i to j is *odd*. For example, the diagonal entries of $P[G]$ will all be 0, since the distance from a vertex to itself is 0, and the entries where A has ones will also have ones.

The resulting algorithm to compute $D[G]$ is a bit bizarre:

- Compute $M[G_2]$ in $O(\mu(n))$ time.
- If $M[G_2](i, j) = 1$ whenever $i \neq j$, return the answer.
- Otherwise compute $D'[G] = D[G_2]$ recursively.
- Compute $P[G]$ from $D'[G]$ in time $O(\mu(n))$ by a method to be given later.
- Return $D[G] = 2D'[G] - P[G]$.

This leaves us with the questions:

- Why is this correct?
- How long does it take?
- How do we get $P[G]$ from $D'[G]$ in $O(\mu(n))$ time?