8.1 Introduction

Today we'll talk about *shortest path* algorithms. There are 3 main variations of the shortest path problem on graphs: the *point-to-point* shortest path problem (find the shortest path from s to t for given s to t); the *single source* shortest path problem (find the shortest path from a given source s to all other vertices); and the *all pairs* shortest path problem (find the shortest path between all pairs). Shortest path problems – particularly point-to-point problems – have become very important of late. Just think about whatever online mapping service you like to use. We'll actually focus on single source and all-pairs algorithms; these are often used in subroutines in various ways, even for point-to-point problems.

Background: Breadth First Search

Breadth First Search (BFS) is a way of exploring the part of the graph that is reachable from a particular vertex (s in the algorithm below).

```
Procedure BFS (G(V,E), s ∈ V)
    graph G(V,E)
    array[|V|] of integers dist
    queue q;
    dist[s] := 0
    inject(q,s)
    placed(s) := 1
    while size(q) > 0
        v := pop(q)
        previsit(v)
        for (v,w) ∈ E
            if placed(w) = 0 then
              inject(q,w)
              placed(w) := 1
              dist(w) = dist(v)+1
            fi
        rof
    end while
end BFS
```

BFS visits vertices in order of increasing distance from s. In fact, our BFS algorithm above labels each vertex with the *distance* from s, or the number of edges in the shortest path from s to the vertex. For example, applied to the graph in Figure 8.1, this algorithm labels the vertices (by the array dist) as shown.

Why are we sure that the array dist is the shortest-path distance from s? A simple induction proof suffices. It is certainly true if the distance is zero (this happens only at s). And, if it is true for $\text{dist}(v) = d$, then it can be easily shown to be true for values of dist equal to $d+1$ —any vertex that receives this value has an edge from a vertex with dist d, and from no vertex with lower value of dist. Notice that vertices not reachable from s will not be visited or labeled.

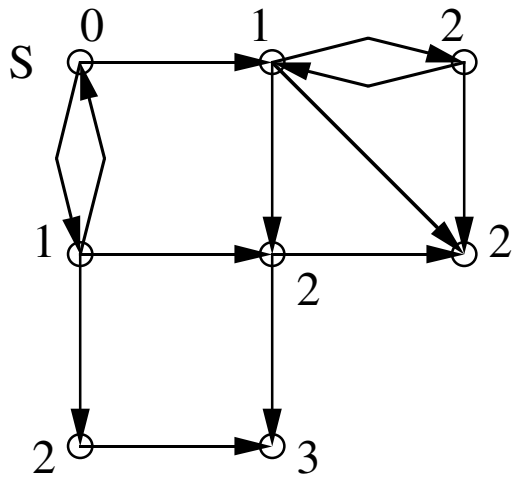

Figure 8.1: BFS of a directed graph

BFS runs, of course, in linear time $O(|E|)$, under the assumption that $|E| \geq |V|$. The reason is that BFS visits each edge exactly once, and does a constant amount of work per edge.

Single-Source Shortest Paths (SSSP) —Nonnegative Lengths

What if each edge (v,w) of our graph has a *length*, a positive integer denoted $\text{length}(v,w)$, and we wish to find the shortest paths from s to all vertices reachable from it? BFS offers a possible solution. We can subdivide each edge (u,v) into $\text{length}(u,v)$ edges, by inserting $\text{length}(u,v)-1$ "dummy" nodes, and then apply BFS to the new graph. This algorithm solves the shortest-path problem in time $O(\sum_{(u,v)\in E} \text{length}(u,v))$. Unfortunately, this can be very large —lengths could be in the thousands or millions. So we need to find a better way.

The problem is that this BFS-based algorithm will spend most of its time visiting "dummy" vertices; only

occasionally will it do something truly interesting, like visit a vertex of the original graph. What we would like to do is run this algorithm, but only do work for the "interesting" steps.

To do this, We need to generalize BFS. Instead of using a queue, we will instead use a *heap* or *priority queue* of vertices. As we have seen, a heap is a data structure that keeps a set of objects, where each object has an associated value. The operations a heap H implements include the following:

deletemin(H)	return the object with the smallest value
insert(x, y, H)	insert a new object x/value y pair in the structure
change(x, y, H)	if y is smaller than x's current value, change the value of object x to y

Each entry in the heap will stand for a projected future "interesting event" of our extended BFS. Each entry will correspond to a vertex, and its value will be the current projected time at which we will reach the vertex. Another way to think of this is to imagine that, each time we reach a new vertex, we can send an explorer down each adjacent edge, and this explorer moves at a rate of 1 unit distance per second. With our heap, we will keep track of when each vertex is due to be reached for the first time by some explorer. Note that the projected time until we reach a vertex can decrease, because the new explorers that arise when we reach a newly explored vertex could reach a vertex first (see node b in Figure 8.2). But one thing is certain: *the most imminent future scheduled arrival of an explorer must happen,* because there is no other explorer who can reach any vertex faster. The heap conveniently delivers this most imminent event to us.

As in all shortest path algorithms we shall see, we maintain two arrays indexed by V. The first array, dist$[v]$, will eventually contain the true distance of v from s. The other array, prev$[v]$, will contain the last node before v in the shortest path from s to v. Our algorithm maintains a useful invariant property: *at all times dist$[v]$ will contain a conservative over-estimate of the true shortest distance of v from s.* Of course dist$[s]$ is initialized to its true value 0, and all other dist's are initialized to ∞, which is a remarkably conservative overestimate. The algorithm is known as Djikstra's algorithm, named after the inventor.

```
Algorithm Djikstra (G = (V, E, length); s ∈ V)
    dist: array[V] of integers, intialized each to ∞
    prev: array[V] of vertices, initialized each to nil
    H := empty minheap
    for v ∈ V do
        H.insert(v, ∞) // insert v with key ∞
    H.decreasekey(s, 0)
    rof
```

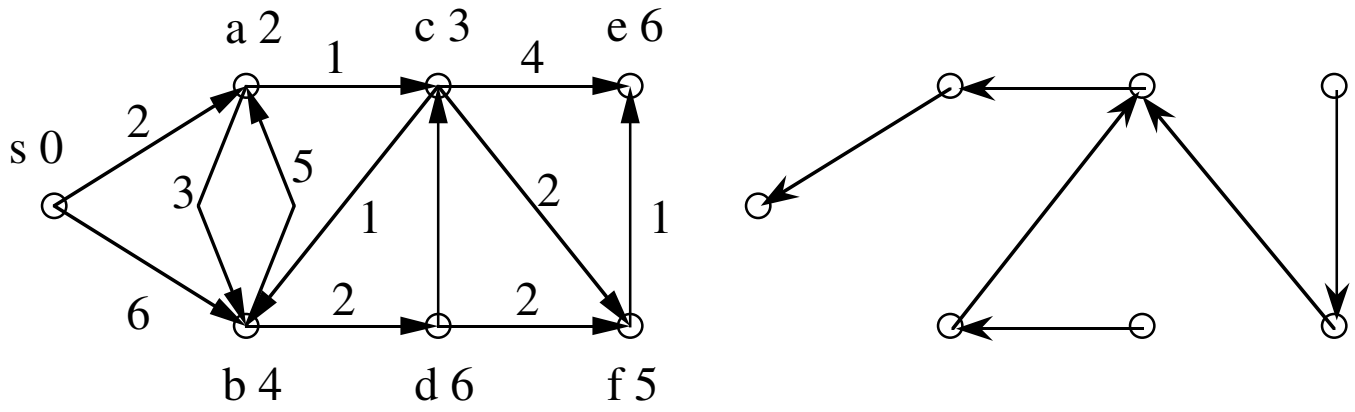

Figure 8.2: Shortest paths

```
while H ≠ ∅
    (v, v.key) := H.deletemin()
    dist[v] := v.key
    for (v, w) ∈ E
        if dist[w] > dist[v]+ length(v, w)
          H.decreasekey(w, dist[v] + length(v, w))
        fi
    rof
end while
return (dist, prev)
```

The algorithm, run on the graph in Figure 8.2, will yield the following heap contents (node: dist/priority pairs) at the beginning of the while loop: $\{s:0\}$, $\{a:2,b:6\}$, $\{b:5,c:3\}$, $\{b:4,e:7,f:5\}$, $\{e:7,f:5,d:6\}$, $\{e:6,d:6\}$, $\{e:6\}$, $\{\}$. The distances from s are shown in Figure 2, together with the *shortest path tree from s*, the rooted tree defined by the pointers prev.

What is the running time of this algorithm? The algorithm involves m decreasekey operations, and n deletemin and insertion operations, so the running time depends on the implementation of the heap H. There are many ways to implement a heap. Using a binary heap would give runtime $O(m \log n)$. A Fibonacci heap would give runtime $O(m + n \log n)$.

Current state of the art (SSSP with nonnegative edge lengths): The following results are all in the word RAM model, with nonnegative integer edge lengths. Thorup gave an algorithm for directed graphs running in time $O(m + n \log\log n)$ [Thorup04], which is currently the best known. He earlier showed that in *undirected* graphs there is an algorithm with running time time $O(m+n)$ [Thorup99], which is of course optimal. In [Thorup07], Thorup showed

that an integer sorting algorithm in word RAM running in time $O(nT)$ implies a minheap datastructure with running time $O(T)$ per operation. Combining this with the $O(n\sqrt{\log\log n})$ expected time randomized sorting algorithm from [HanT02] gives a Dijkstra implementation for directed graphs running in expected time $O(m\sqrt{\log\log n})$. Thorup stated it as an explicit open problem in the conclusion of [Thorup07] to give a reduction from minheaps to sorting such that insertion and deletemin each take $O(T)$ time, but decreasekey only takes $O(1)$ time. If such a reduction existed, then the expected running time using [HanT02] would improve to $O(m+n\sqrt{\log\log n})$.

Single-Source Shortest Paths: General Lengths

Our argument of correctness of our shortest path algorithm was based on the "time metaphor:" the most imminent prospective event (arrival of an explorer) must take place, exactly because it is the most imminent. This however would not work if we had *negative edges*. (Imagine explorers being able to arrive before they left!) If the length of edge (a,b) in Figure 2 were -1, the shortest path from s to b would have value 1, not 4, and our simple algorithm fails. Obviously, with negative lengths we need a different algorithm.

We will now develop the *Bellman-Ford* algorithm which solves this problem. It is a dynamic programming algorithm — we first explain its recursive variant (i.e. the memoization approach).

Suppose again that s is our source vertex that we are finding shortest paths from. Define a function $f(u,k)$ which is the length of the shortest path from s to u using at most k edges. First, let us suppose that we were promised that G has no negative weight cycles. In this case, the length of the shortest path from s to any u would be $f(u,n-1)$, since taking $k > n-1$ would imply a cycle, which cannot provide any benefit if there are no negative weight cycles. So, how can we calculate $f(u,n-1)$? Recursively!

$$f(u,k) = \begin{cases} 0, & \text{if } u = s, k = 0 \\ \infty, & \text{if } u \neq s, k = 0 \\ \min\left\{f(u,k-1), \min_{v\in V:(v,u)\in E} f(v,k-1) + \text{length}(v,u)\right\}, & \text{otherwise} \end{cases}$$

In words, the base case is $k = 0$ (a path with zero edges), in which the length of the shortest path is either 0 or ∞ depending on whether $u = s$ (we also use ∞ as the shortest path distance to represent the lack of a path). If $k > 0$, then the shortest path from s to u of length at most k is, by the law of the excluded middle, either of length less than k or exactly k. Thus we simply take the minimum of the two possibilities. The $f(u,k-1)$ term represents the first possibility. The other term in the minimum, which takes a min over $v \in V$, represents the other possibility: a path of length k is simply a path of length $k-1$, ending at some node v, followed by the edge v,u. Shortest paths have the property that all subpaths are themselves shortest paths (justify this to yourself as an exercise!), and thus

the path of length $k-1$ to v should itself be a shortest path. Note that above we actually take $f(v,k-1)+w(v,u)$, and $f(v,k-1)$ isn't actually a path of length exactly $k-1$, but rather simply *at most* $k-1$. This is OK though, since this only makes the minimum smaller and $f(v,k-1)+w(v,u)$ is still a valid upper bound on $f(u,k)$.

As an exercise, you may want to show that the straightforward recursive implementation of computation of $f(u,n-1)$ would take exponential time. But then we of course *memoize*. Now let us analyze running time and space. First, in pre-processing, we form the reverse graph G_{rev} by reversing all edges in E then store it in adjacency list representation; this will help us loop over $\min_{v\in V:(v,u)\in E}$ easily (it is just the neighborhood of u in G_{rev}. Now, fix some $0 \leq k \leq n-1$ and let us compute the total work to compute all $f(\cdot,k)$ values. For a fixed u, the total work through the loop to compute $f(u,k)$ is $O(1+\deg_{G_{rev}}(u))$, where $\deg_H$ is the out degree in graph H. Thus the total work done across all u is at most $\sum_{u\in V} C(1+\deg_{G_{rev}}(u)) = C(n+m) = O(m+n)$. Thus the total work done across all k is $O((m+n)n)$. If the graph is connected then $m\ gen-1$, thus simplifying the running time to $O(mn)$.

What about the space complexity? Storing a dynamic programming (DP) table `dp[u][k]` would take $O(n^2)$ space across all u,k. Observe though that $f(\cdot,k)$ values only depend on $f(\cdot,k-1)$ values, so a bottom-up dynamic programming approach would only need a DP table of size n. When including the $O(m+n)$ space to store the graph, the bottom-up DP approach thus uses space $O(m+n)$. It is also possible to implement the bottom-up DP approach simply to use time $O(mn)$ instead of $O((m+n)n)$, even in cases when $m \ll n$.

Negative Cycles

In fact, there is a further problem that negative edges can cause. Suppose the length of edge (b,a) in Figure 2 were changed to -5. The the graph would have a *negative cycle* (from a to b and back). On such graphs, it does not make sense to even *ask* the shortest path question. What is the shortest path from s to c in the modified graph? The one that goes directly from s to a to c (cost: 3), or the one that goes from s to a to b to a to c (cost: 1), or the one that takes the cycle twice (cost: -1)? And so on.

The shortest path problem is ill-posed in graphs with negative cycles. It makes no sense and deserves no answer. Our algorithm in the previous section works only in the absence of negative cycles. (Where did we assume no negative cycles in our correctness argument? Answer: When we asserted that a shortest path from s to a exists!) But it would be useful if our algorithm were able to *detect* whether there is a negative cycle in the graph, and thus to report reliably on the meaningfulness of the shortest path answers it provides.

This is easily done. To check whether there are negative weight cycles, we check to make sure that for all $u \in V$, $f(u,n) \geq f(u,n-1)$. If there are no negative weight cycles this test will clearly hold, since a length n path

must contain a cycle, and having no negative weight cycles means that this cannot be advantageous. For the other direction, suppose $f(u,n) \geq f(u,n-1)$ for all u. This is the same as saying $f(u,n-1) \leq f(v,n-1) + w(v,u)$ for all $u,v \in V$. Consider a cycle $v_1, v_2, \ldots, v_\ell$ in the graph. We just need to show this cycle has nonnegative total weight. For each i we have

$$f(v_{i+1}, n-1) \leq f(v_i, n-1) + w(v_i, v_{i+1})$$

where we treat "$\ell+1$" as 1 (wrapping around the cycle). Summing over all i from 1 to ℓ, we obtain the inequality $\sum_{i=1}^{\ell} w(v_i, v_{i+1}) \geq 0$, as desired.

Shortest Paths on DAG's

There are two subclasses of weighted graphs that automatically exclude the possibility of negative cycles: graphs with non-negative weights and DAG's. We have already seen that there is a fast algorithm when the weights are non-negative. Here we will give a *linear* algorithm for single-source shortest paths in DAG's.

Again this problem can be solved with memoization/dynamic programming. Fixing a source vertex s, define $f(u)$ to be the length of the shortest path from s to u. Then we have the recurrence

$$f(u,k) = \begin{cases} 0, & \text{if } s = u \\ \min\{\infty, \min_{v \in V:(v,u) \in E} f(v)\}, & \text{otherwise} \end{cases}$$

The running time and space with memoization are both $O(m+n)$.

All pairs shortest paths via dynamic programming

Suppose now we want to calculate the shortest paths between *every* pair of nodes. One way to do this is to run Dijkstra's algorithm several times, once for each node (if edge weights are nonnegative). This would take $O(mn + n^2 \log n)$ time using Fibonacci heaps. For graphs with negative edge weights, we could run Bellman-Ford from each source, taking time $O(mn^2)$. An algorithm called Johnson's algorithm can actually solve this problem in $O(mn + n^2 \log n)$ by running Bellman-Ford once followed by n Dijkstra calls, though we will not cover it in class. Here we develop a different solution, called the Floyd-Warshall algorithm. The running time will be $O(n^3)$.

Let us first assume that there are no negative weight cycles. Like with Bellman-Ford above, we can come up with a recursive solution. Let $f(u,v,k)$ be the length of the shortest path from u to v using *only* nodes $1 \ldots k$ as intermediate nodes. Of course when k equals the number of nodes in the graph, n, we will have solved the original problem. Again we can compute f recursively:

$$f(u,v,k) = \begin{cases} w(u,v), & \text{if } k = 0 \\ \min\{f(u,v,k-1), f(u,k,k-1) + f(k,v,k-1)\}, & \text{otherwise} \end{cases}$$

Again outside the base case we use the law of the excluded middle: either the shortest path when allowed to use intermediate nodes $1,\ldots,k$ uses node k or it doesn't. We simply take the minimum of the two possibilities. There are n^3 states and we do $O(1)$ work for each one (a minimum of two numbers) ignoring recursive calls. Thus with memoization the total running time would be $O(n^3)$. The space naively would unfortunately also be $\Theta(n^3)$, but a bottom-up DP implementation would only use $O(n^2)$ space since $f(\cdot,\cdot,k)$ values only depend on $f(\cdot,\cdot,k-1)$ values.

A short implementation of bottom-up Floyd-Warshall follows. We assume that d_{ij} is set to ∞ for all i,j such that $(i,j) \notin E$.

$D = (d_{ij})$, distance array, with weights from all i to all j
for $k = 1$ to n do
  for $i = 1$ to n do
    for $j = 1$ to n do
      $D[i,j] = \min(D[i,j], D[i.k] + D[k,j])$

Note that again we can keep an auxiliary array to recall the actual paths. We simply keep track of the last intermediate node found on the path from i to j. We reconstruct the path by successively reconstructing intermediate nodes, until we reach the ends.

What about if there are negative weight cycles?

Exercise. Show that there is a negative weight cycle in G if and only if for some $1 \le i \le n$, $D[i,i] < 0$ at the end of running Floyd Warshall.

Some Pointers on Point-to-Point Algorithms

Naturally, one can use Dijkstra's algorithm to solve the point-to-point shortest paths problem. Indeed, for non-negative distances, one can stop as soon as the other point is taken off the priority queue. But for real speed, one can optimize in various ways, or use alternative methods. Here are just a few ideas that have been used in algorithms for this problem.

- Bidirectional Dijkstra's algorithm: start searches from s and t and alternate between them, and terminate when

the searches meet in the middle. The stopping criterion is not trivial. The idea is that if the graph branches a lot, two "small" searches might require less exploration than one large one.

- Changing ditances with potential functions: one can associated a potential $\pi(x)$ with each vertex x and change each distance $d(u,v)$ to $d(u,v)+\pi(v)-\pi(u)$. If $\pi(s)=0$ for the source vertex and the new distances are nonnegative, then the shortest paths remain the same (check why) and Dijkstra's algorithm can still be used. Clever choices of potential functions can speed up the algorithm.

- Using landmarks and precomputation: by precomputing the distance between a small collection of landmark locations and nearby points, and precomputing all-pairs shortest paths between landmarks, one can quickly determine an upper bound on the shortest path between two points: add up the distance from the start point to its nearest landmark, the distance from the end point to its nearest landmark, and the distance between landmarks. Using this information, one can prune Djikstra's algorithm wehn finding point-to-point shortest paths.

- Hierarchical methods: In real world networks, there are often natural partitions of the underlying space. For example, for a long trip, it may be clear that first you have to get on a highway, then take the highway, then get off the highway. To cross a body of water, you may have to go through one of a small number of bridges. Taking advantage of such hierarchical features can speed up calculations.

Sections 1-3 of the survey at `http://research.microsoft.com/pubs/207102/MSR-TR-2014-4.pdf` has a lot of good background on this problem. (The rest of the survey examines the problem of traveling via public transport, where timetables are involved.)

References

[1] Yijie Han, Mikkel Thorup. Integer Sorting in $O(n\sqrt{\log\log n})$ Expected Time and Linear Space. *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 135–144, 2002.

[2] Mikkel Thorup. Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. *J. ACM* 46(3), pages 362–394, 1999.

[3] Mikkel Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *J. Comput. Syst. Sci.* 69(3), pages 330–353, 2004.

[4] Mikkel Thorup. Equivalence between priority queues and sorting. *J. ACM* 54(6), 2007.