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.

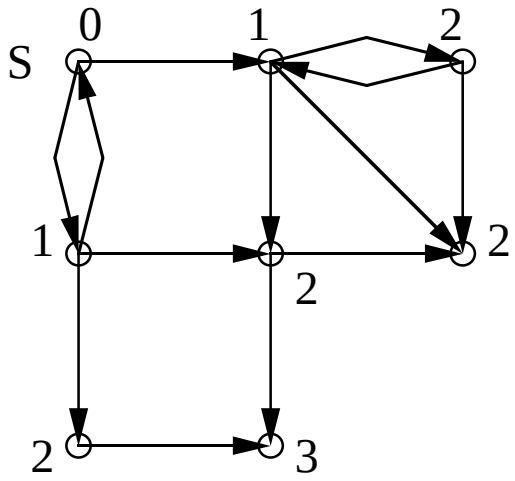

Figure 8.1: BFS of a directed graph

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 —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 ($G = (V, E, \text{length})$; $s \in V$)
    v, w: vertices
    dist: array$[V]$ of integer
    prev: array$[V]$ of vertices
    H: priority heap of V
    $H := \{s : 0\}$
    for $v \in V$ do
        dist$[v] := \infty$, prev$[v]$:=nil

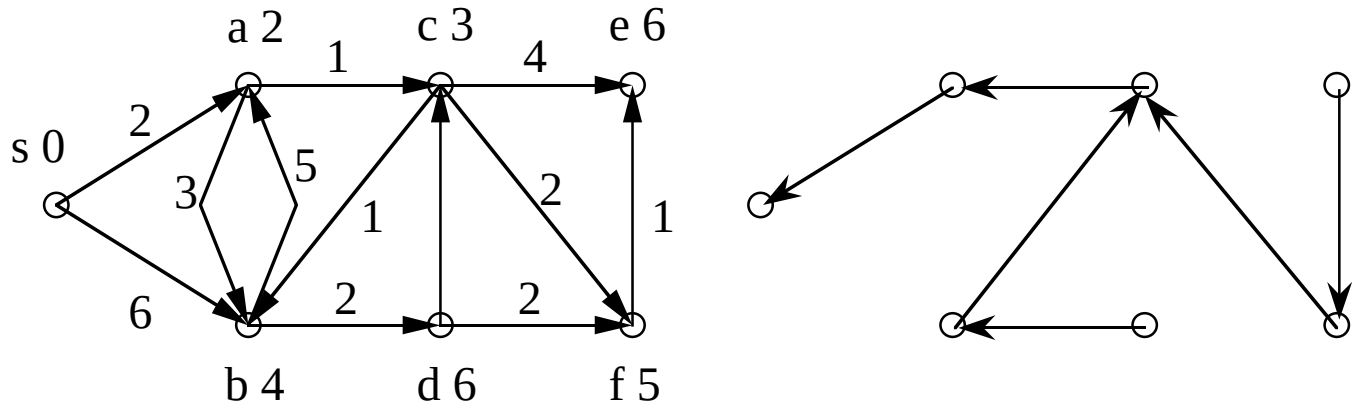

Figure 8.2: Shortest paths

```
rof
dist[s] := 0
while H ≠ ∅
    v := deletemin(h)
    for (v,w) ∈ E
        if dist[w] > dist[v]+ length(v,w)
           dist[w] := dist[v] + length(v,w), prev[w] := v, insert(w,dist[w],H)
        fi
    rof
end while end shortest paths 1
```

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 $|E|$ insert operations and $|V|$ deletemin operations on H, and so the running time depends on the implementation of the heap H. There are many ways to implement a heap. Even an unsophisticated implementation as a linked list of node/priority pairs yields an interesting time bound, $O(|V|^2)$ (see first line of the table below). A binary heap would give $O(|E|\log|V|)$.

Which of the two should we prefer? The answer depends on how *dense* or *sparse* our graphs are. In all graphs, $|E|$ is between $|V|$ and $|V|^2$. If it is $\Omega(|V|^2)$, then we should use the linked list version. If it is anywhere below $\frac{|V|^2}{\log|V|}$, we should use binary heaps.

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 more involved algorithms, which repeatedly update the values of dist.

We can describe a general paradigm for constructing shortest path algorithms with arbitrary edge weights. The algorithms use arrays dist and prev, and again we maintain the invariant that dist is always a conservative overestimate of the true distance from s. (Again, dist is initialized to ∞ for all nodes, except for s for which it is 0).

The algorithms maintain dist so that it is always a conservative overestimate; it will only update the a value when a suitable path is discovered to show that the overestimate can be lowered. That is, suppose we find a neighbor w of v, with dist$[v] >$ dist$[w]$ + length(w,v). Then we have found an actual path that shows the distance estimate is too conservative. We therefore repeatedly apply the following update rule.

procedure update ((w,v))
  edge (w,v)
  if dist$[v] >$ dist$[w]+$ length(w,v) then
    dist$[v]$:= dist$[w]$ + length(w,v), prev$[v]$:= w

A crucial observation is that this procedure is *safe*, in that it never invalidates our "invariant" that dist is a conservative overestimate.

The key idea is to consider how these updates along edges should occur. In Djikstra's algorithm, the edges are updated according to the time order of the imaginary explorers. But this only works with positive edge lengths.

A second crucial observation concerns how many updates we have to do. Let $a \neq s$ be a node, and consider the shortest path from s to a, say $s, v_1, v_2, \ldots, v_k = a$ for some k between 1 and $n-1$. If we perform update first on (s, v_1), later on (v_1, v_2), and so on, and finally on (v_{k-1}, a), then we are sure that dist(a) contains the true distance from s to a, and that the true shortest path is encoded in prev. (**Exercise:** Prove this, by induction.) We must thus find a sequence of updates that guarantee that these edges are updated in this order. We don't care if these or other edges are updated several times in between, since all we need is to have a sequence of updates that contains this particular subsequence. There is a very easy way to guarantee this: update all edges $|V|-1$ times in a row!

Algorithm Shortest Paths 2 ($G = (V, E, \text{length})$; $s \in V$)

```
    v, w: vertices
    dist: array[V] of integer
    prev: array[V] of vertices
    i: integer
    for v ∈ V do
        dist[v] := ∞, prev[v] :=nil
    rof
    dist[s] := 0
    for i = 1...n − 1
        for (w, v) ∈ E update(w, v)
end shortest paths 2
```

This algorithm solves the general single-source shortest path problem in $O(|V| \cdot |E|)$ time.

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. After the $|V|-1$ rounds of updates of all edges, do a last update. If any changes occur during this last round of updates, there is a negative cycle. This must be true, because if there were no negative cycles, $|V|-1$ rounds of updates would have been sufficient to find the shortest paths.

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.

Our algorithm is based on the same principle as our algorithm for negative weights. We are trying to find a sequence of updates, such that all shortest paths are its subsequences. But in a DAG we know that all shortest paths from s must go in the topological order of the DAG. All we have to do then is first topologically sort the DAG using a DFS, and then visit all edges coming out of nodes in the topological order. This algorithm solves the general single-source shortest path problem for DAG's in $O(m)$ time.

8.2 All pairs shortest paths via dynamic programming

Let G be a graph with positive edge weights. 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. Here we develop a different dynamic programming solution.

Our subproblems will be shortest paths 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.

We let the matrix $D_k[i.j]$ represent the length of the shortest path between i and j using intermediate nodes $1 \ldots k$. Initially, we set a matrix D_0 with the direct distances between nodes, given by d_{ij}. Then D_k is easily computed from the subproblems D_{k-1} as follows:

$$D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]).$$

The idea is the shortest path using intermediate nodes $1 \ldots k$ either completely avoids node k, in which case it has the same length as $D_{k-1}[i,j]$; or it goes through k, in which case we can glue together the shortest paths found from i to k and k to j using only intermediate nodes $1 \ldots k-1$ to find it.

It might seem that we need at least two matrices to code this, but in fact it can all be done in one loop. (**Exercise:** think about it!)

$D = (d_{ij})$, 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.

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.)