Notes 7 for CS 170

1 Dijkstra's Algorithm

Suppose each edge (v, w) of our graph has a *weight*, a positive integer denoted weight(v, w), and we wish to find the shortest from s to all vertices reachable from it.[1]

We will still use BFS, but instead of choosing which vertices to visit by a queue, which pays no attention to how far they are from s, we will use a *heap*, or *priority queue*, of vertices. The priority depends on our current best estimate of how far away a vertex is from s: we will visit the closest vertices first.

These distance estimates will always *overestimate* the actual shortest path length from s to each vertex, but we are guaranteed that the *shortest* distance estimate in the queue is actually the true shortest distance to that vertex, so we can correctly mark it as finished.

As in all shortest path algorithms we shall see, we maintain two arrays indexed by V. The first array, `dist[v]`, contains our overestimated distances for each vertex v, and will contain the true distance of v from s when the algorithm terminates. Initially, `dist[s]`=0 and the other `dist[v]`$= \infty$, which are sure-fire overestimates. The algorithm will repeatedly decrease each `dist[v]` until it equals the true distance. The other array, `prev[v]`, will contain the last vertex before v in the shortest path from s to v. The pseudo-code for the algorithm is in Figure 1

The procedure `insert(w,H)` must be implemented carefully: if w is already in H, we do not have to actually insert w, but since w's priority `dist[w]` is updated, the position of w in H must be updated.

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

1.1 What is the complexity of Dijkstra's algorithm?

The algorithm involves $|E|$ insert operations (in the following, we will call m the number $|E|$) on H and $|V|$ deletemin operations on H (in the following, we will call n the number $|V|$), and so the running time depends on the implementation of the heap H, so let us discuss this implementation. There are many ways to implement a heap.[2] Even the most

[1] What if we are interested only in the shortest path from s to a specific vertex t? As it turns out, all algorithms known for this problem also give us, as a free byproduct, the shortest path from s to all vertices reachable from it.

[2] In all heap implementations we assume that we have an array of pointers that give, for each vertex, its position in the heap, if any. This allows us to always have at most one copy of each vertex in the heap. Each time `dist[w]` is decreased, the `insert(w,H)` operation finds w in the heap, changes its priority, and possibly moves it up in the heap.

```
algorithm Dijkstra(G=(V, E, weight), s)
  variables:
    v,w:  vertices (initially all unmarked)
    dist:  array[V] of integer
    prev:  array[V] of vertices
    heap of vertices prioritized by dist
  for all v ∈ V do { dist[v] := ∞, prev[v] :=nil}
  H:={s} , dist[s] :=0 , mark(s)
  while H is not empty do
  {
    v := deletemin(H) , mark(v)
    for each edge (v,w) in E out of v do
      {
        if w unmarked and dist[w] > dist[v] + weight[v,w] then
          {
            dist[w] := dist[v] + weight[v,w]
            prev[w] := v
            insert(w,H)
          }
      }
  }
```

Figure 1: Dijkstra's algorithm.

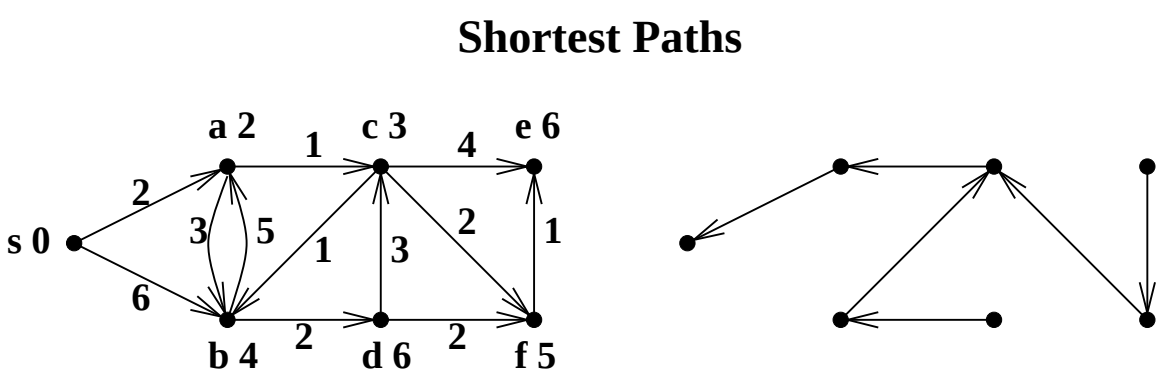

Figure 2: An example of a shortest paths tree, as represented with the `prev[]` vector.

unsophisticated one (an amorphous set, say a linked list of vertex/priority pairs) yields an interesting time bound, $O(n^2)$ (see first line of the table below). A binary heap gives $O(m \log n)$.

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

heap implementation	deletemin	insert	$n\times$deletemin$+m\times$insert
linked list	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(m \log n)$

1.2 Why does Dijkstra's algorithm work?

Here is a sketch. Recall that the inner loop of the algorithm (the one that examines edge (v, w) and updates `dist`) examines each edge in the graph exactly once, so at any point of the algorithm we can speak about the subgraph $G' = (V, E')$ examined *so far*: it consists of all the nodes, and the edges that have been processed. Each pass through the inner loop adds one edge to E'. We will show that the following property of `dist[]` is an invariant of the inner loop of the algorithm:

> `dist[w]` is the minimum distance from s to w in G', and
> if $\mathtt{v_k}$ is the k-th vertex marked, then $\mathtt{v_1}$ through $\mathtt{v_k}$ are the k closest vertices to `s` in G, and the algorithm has found the shortest paths to them.

We prove this by induction on the number of edges $|E'|$ in E'. At the very beginning, before examining *any* edges, $E' = \emptyset$ and $|E'| = 0$, so the correct minimum distances are `dist[s] = 0` and `dist[w] =` ∞, as initialized. And `s` is marked first, with the distance to it equal to zero as desired.

Now consider the next edge (v, w) to get $E'' = E' \cup (v, w)$. Adding this edge means there is a new way to get to `w` from `s` in E'': from `s` to `v` to `w`. The shortest way to do this is `dist[v] + weight(v,w)` by induction. Also by induction the shortest path to get to `w` not using edge (`v`, `w`) is `dist[w]`. The algorithm then replaces `dist[w]` by the minimum of its old value and possible new value. Thus `dist[w]` is still the shortest distance to `w` in E''. We still have to show that all the other `dist[u]` values are correct; for this we need to use the heap property, which guarantees that (`v`, `w`) is an edge coming out of the node `v` in the heap *closest* to the source `s`. To complete the induction we have to show that adding the edge (v, w) to the graph G' does not change the values of the distance of *any* other vertex `u` from `s`. `dist[u]` could only change if the shortest path from `s` to `u` had previously gone through `w`. But this is impossible, since we just decided that `v` was closer to `s` than `w` (since the shortest path to `w` is via `v`), so the heap would not yet have marked `w` and examined edges out of it.

2 Negative Weights—Bellman-Ford Algorithm

Our argument of correctness of our shortest path algorithm was based on the fact that adding edges can only make a path longer. This however would not work if we had *negative*

Shortest Path with Negative Edges

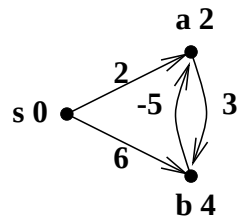

Figure 3: A problematic instance for Dijkstra's algorithm.

edges: If the weight of edge (b, a) in figure 3 were -5, instead of 5 as in the last lecture, the first event (the arrival of BFS at vertex a, with `dist[a]=2`) would not be suggesting the correct value of the shortest path from s to a. Obviously, with negative weights we need more involved algorithms, which repeatedly update the values of dist.

The basic information updated by the negative edge algorithms is the same, however. They rely on arrays `dist` and `prev`, of which `dist` is always a conservative overestimate of the true distance from s (and is initialized to ∞ for all vertices, except for s for which it is 0). The algorithms maintain `dist` so that it is always such a conservative overestimate. This is done by the same scheme as in our previous algorithm: Whenever "tension" is discovered between vertices v and w in that `dist(w)>dist(v)+weight(v,w)`—that is, when it is discovered that `dist(w)` is a more conservative overestimate than it has to be—then this "tension" is "relieved" by this code:

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

One crucial observation is that this procedure is *safe*, it never invalidates our "invariant" that `dist` is a conservative overestimate. Our algorithm for negative edges will consist of many updates of the edges.

A second crucial observation is the following: Let $a \neq s$ be a vertex, 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. 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, all we need is to have a sequence of updates that contains this particular subsequence.

And there is a very easy way to guarantee this: *Update all edges $n-1$ times in a row!* Here is the algorithm:

```
algorithm BellmanFord(G=(V, E, weight):  graph with weights; s:  vertex)
dist:  array[V] of integer; prev:  array[V] of vertices
for all v ∈ V do { dist[v] := ∞, prev[v] := nil}
```

```
for i := 1,...,n-1 do
  { for each edge (v,w) ∈ E do update[v,w] }
```

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

3 Negative Cycles

In fact, if the weight of edge (b, a) in the figure above were indeed changed to -5, then there would be a bigger problem with the graph of this figure: It 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 a in the modified graph? The one that goes directly from s to b to a (cost: 1), or the one that goes from s to b to a to b to a (cost: -1), or the one that takes the cycle twice (cost: -3)? And so on.

The shortest path problem is ill-posed in graphs with negative cycles. It is still an interesting question to ask however. If the vertices represent currencies (dollars, marks, francs, etc.) and the edge weights represent (logarithms of) exchange rates, then a negative cycle means that it is possible to start with \$100, say, buy and sell a sequence of other currencies, coming back to dollars, and ending up with more than \$100. This process may be repeated, and is called *arbitrage*. In the real world, the exchange rates keep changing, so the problem is more challenging. See Question 25-3 in CLR (24-3 in CLRS).

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 as follows: After the $n - 1$ rounds of updates of all edges, do a last round. If anything is changed during this last round of updates—if, that is, there is still "tension" in some edges—this means that there is no well-defined shortest path (because, if there were, $n - 1$ rounds would be enough to relieve all tension along it), and thus there is a negative cycle reachable from s.