Lecture 12 — Shortest Weighted Paths

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2013)

Lectured by Umut Acar — 21 February 2013

What was covered in this lecture:

- Weighted Graphs
- Priority First Search
- Weighted Shortest Paths
- Dijkstra's Algorithm – finished in next classs

1 Weighted Graph Representation

It is often necessary to associate weights or other values with the edges of a graph. Such a "weighted" or "edge-labeled" graph can be defined as a triple $G = (E, V, w)$ where $w : E \to \texttt{eVal}$ is a function mapping edges or directed edges to their values, and `eVal` is the set (type) of possible values. For a weighted graph `eVal` would typically be the real numbers, but for edge-labeled graphs they could be any type.

There are a few ways to represent a weighted or edge-labeled graph. The first representation translates directly from representing the function w. In particular we can use a table that maps each edge (a pair of vertex identifiers) to its value. This would have type

$$\texttt{eVal edgeTable}$$

That is, the keys of the table are edges and the values are of type `eVal`. For example, for a weighted graph we might have:

$$W = \{(0,1) \mapsto 0.7,\ (1,2) \mapsto -2.0,\ (0,2) \mapsto 1.5\}$$

This representation would allow us to look up the weight of an edge e using: $w(e) = \texttt{find}\ W\ e$.

Another way to associate values with edges is to use a structure similar to the adjacency set representation for unweighted graphs and associate a value with each out edge of a vertex. In particular, instead of associating with each vertex a set of out-neighbors, we can associate each vertex with a table that maps each out-neighbor to its value. It would have type:

$$(\texttt{eVal vertexTable})\ \texttt{vertexTable}\ .$$

The graph above would then be represented as:

$$G = \{0 \mapsto \{1 \mapsto 0.7,\ 2 \mapsto 1.5\},\ 1 \mapsto \{2 \mapsto -2.0\},\ 2 \mapsto \{\}\}\ .$$

We will mostly be using this second representation.

†Lecture notes by Umut A. Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

2 Priority First Search

Generalizing BFS and DFS, *priority first search* (also called best-first search) visits the vertices in some priority order. This priority order can either be static and decided ahead of time, or can be generated on the fly by the algorithm. Recall that a generic graph search maintains two sets: the set of visited vertices X, and the set of frontier vertices $F = N(X) \setminus X$ (in DFS the frontier is maintained implicitly on the recursion stack). The frontier vertices are the unvisited vertices we know about by having visited their in-neighbors. On each step we visit one or more vertices on the frontier, move them from F to X and add their unvisited out-neighbors to F. A priority first search can thus be seen as follows:

```
1  function pfs(X, F) =
2  if (F = ∅) then X
3  else let
4     val M = highest priority vertices in F
5     val X' = X ∪ M
6     val F' = (F ∪ N(M)) \ X'
7  in pfs(X', F') end
```

This would typically start with $X = \emptyset$ and the frontier F containing a single source.

One can imagine using such a scheme, for example, to explore the web in a way that prioritizes the more interesting or relevant pages. The idea is to keep a ranked list of unvisited outgoing links based on how interesting the tag on the link appears to be. Then, when choosing what page to visit next, follow the link with the highest rank. When visiting the page, scratch it off the list and add all its unvisited outgoing links to the list with ranks.

Several famous graph algorithms are instances of priority first search. For example, Dijkstra's algorithm for finding single-source shortest paths (SSSP), discussed next and Prim's algorithm for finding Minimum Spanning Trees (MST). Priority First Search is a greedy technique since it greedily selects among the choices in front of it (the frontier) based on some cost function (the priorities) and never backs up. Algorithms based on PFS are hence often referred to as greedy algorithms.

2.1 Shortest Weighted Paths

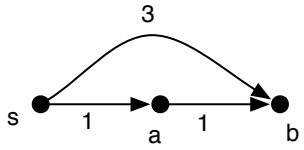
The single-source shortest path (SSSP) problem is to find the shortest (weighted) path from a source vertex s to every other vertex in the graph. Consider a weighted graph $G = (V, E, w)$, $w : E \to \mathbb{R}$. The graph can be either directed or undirected. For convenience we define $w(u, v) = \infty$ if $(u, v) \notin E$. The *weight of a path* is the sum of the weights of the edges along that path.

Definition 2.1 (The Single-Source Shortest Path (SSSP) **Problem**)**.** Given a weighted graph $G = (V, E, w)$ and a source vertex s, the *single-source shortest path (SSSP) problem* is to find the shortest weighted path from s to every other vertex in V.

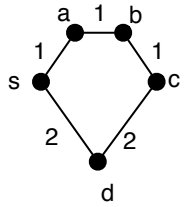
For a weighted graph G we will use $\delta_G(u, v)$ to indicate the weight of the shortest path from u to v. Dijkstra's algorithm solves the SSSP problem when all the weights on the edges are non-negative (i.e. $w : E \to \mathbb{R}_+ \cup \{0\}$). It is a very important algorithm both because shortest paths have many

applications, and also because it is a very elegant example of an efficient greedy algorithm that generates optimal solutions on a nontrivial task.

Before describing Dijkstra's algorithm, we would like to understand why we need a new algorithm for the weighted case. Why, in particular, doesn't BFS work on weighted graphs? Consider the following directed graphs on 3 nodes:



In this example, BFS would visit b then a. This means when we visit b, we assign it an incorrect weight of 3. Since BFS never visit it again, we'll never update it to the correct value. Note that we cannot afford to visit it again as it will be much more expensive. This problem still persists even if the graph is geometric (and satisfies the triangle inequality), as the following example shows:



But why does BFS work in unweighted case? The key idea is that it works outwards from the source. For each frontier F_i, it has the correct unweighted distance from source to each vertex in the frontier. It then can determine the correct distance for unencountered neighbors that are distance one further away (on the next frontier).

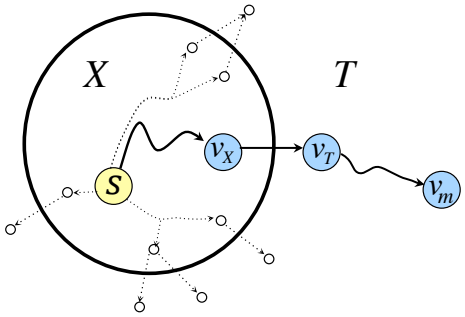
Let's consider using a similar approach when the graphs has non-negative edge weights. Starting from the source vertex s, for which vertex can we safely say we know its shortest path from s? The vertex v that is the closest neighbor of s. There could not be a shorter path to v, since such a path would have to go through one of the neighbors that is further away from s and that path cannot get shorter because none of the edge weights are negative. More generally, if we know the shortest path distances for a set of vertices, how can we determine the shortest path to another vertex? This is the question that Dijkstra's algorithm answers.

3 Dijkstra's Algorithm for SSSP

The key property used by Dijkstra's algorithm is that for a set of vertices $X \subset V$ that include s and the rest of the vertices ($T = V \setminus X$), the closest vertex in T from s based on paths that only go through X is also an overall closest vertex in T. This property will allow us to select the next closest vertex by only considering the vertices we have already visited. Defining $\delta_{G,X}(s, v)$ as the shortest path length from s to v in G that only goes through vertices in X (except for v), the property can be stated more formally and proved as follows:

Lemma 3.1 (Dijkstra's Property)**.** *Consider a (directed) weighted graph $G = (V, E)$, $w : E \to \mathbb{R}_+ \cup \{0\}$, and a source vertex $s \in V$. For any partitioning of the vertices V into X and $T = V \setminus X$ with $s \in X$,*

$$\min_{t \in T} \delta_{G,X}(s,t) = \min_{t \in T} \delta_G(s,t) \,.$$



Proof. Consider a vertex $v_m \in T$ such that $\delta_G(s, v_m) = \min_{t \in T} \delta_G(s,t)$, and a shortest path from s to v_m in G. The path must cross from X to T at some point using some edge (v_X, v_T). Since subpaths of shortest paths are shortest paths, the subpath from s to v_T is the shortest path to v_T and, since edges are non-negative, distances don't decrease along the path implying $\delta_G(s, v_T) \leq \delta_G(s, v_m)$ (it could be that $v_T = v_m$). Furthermore since the shortest path to v_T only goes through X, $\delta_{G,X}(s, v_T) = \delta_G(s, v_T)$. Together we have:

$$\min_{t \in T} \delta_{G,X}(s,t) \leq \delta_{G,X}(s, v_T) = \delta_G(s, v_T) \leq \delta_G(s, v_m) = \min_{t \in T} \delta_G(s,t) \,.$$

Also $\min_{t \in T} \delta_{G,X}(s,t) \geq \min_{t \in T} \delta_G(s,t)$ since we are only restricting paths by requiring them to go through X. Together we have the equality. □

This lemma implies that if we have already visited a set of vertices X we can find a new shortest path to a vertex in $T = V \setminus X$ by just considering the path lengths through X to a neighbor of X. In particular we want to pick a vertex t that minimizes $\delta_{G,X}(s,t)$. This suggests an algorithm for shortest paths based on priority first search using the priority $P(v) = \delta_{G,X}(s,u)$. Also note that $\delta_{G,X}(u) = \min_{v \in V}(\delta_G(v) + w(v,t))$. Indeed this gives us Dijkstra's algorithm, at least in the abstract:

Definition 3.2 (Dijkstra's Algorithm)**.** Given a weighted graph $G = (V, E, w)$ and a source s, Dijkstra's algorithm is priority first search on G starting at s using priority $P(v) = \min_{v \in V}(d(v) + w(v,t))$ (to be minimized) and setting $d(v) = P(v)$ when v is visited and $d(s) = 0$.

Lemma 3.3. *When Dijkstra's algorithm returns, $d(v) = \delta_G(s,v)$ for all reachable v.*

Proof. We prove that $d(x) = \delta_G(s,x)$ for $x \in X$ by induction on the size of X. The base case is true since $d(s) = 0$. Assuming it is true for size i, then the algorithm adds the vertex v that minimizes $P(v) = \delta_{G,X}(s,v)$. By Lemma 3.1 $\delta_{G,X}(s, f\,v) = \delta_G(s,v)$ giving $d(v) = \delta_G(s,v)$ for $i+1$. □

A curious aspect of Dijkstra's Algorithm. Although Dijkstra's algorithms visits the vertices in increasing (technically non-decreasing) order of distance, this feature does not play any role in the correctness of the algorithm. For example if someone during the algorithm removed an arbitrary set of vertices from X along with their distances and then continued running the algorithm, it would still run correctly. This is because Lemma 3.1 makes no requirement about how X and T are partitioned other than $s \in X$.

We now discuss how to implement this abstract algorithm efficiently using a priority queue to maintain $P(v)$. We use a priority queue that supports `deleteMin` and `insert`. The algorithm is as follows:

```
1  function dijkstra(G,s) =
2  let
3    function dijkstra'(X, Q) =
4      case PQ.deleteMin (Q) of
5          (NONE, _) ⇒ X
6        | (SOME(d,v),Q') ⇒
7          if ((v, _) ∈ X) then dijkstra'(X, Q')
8          else let
9              X' = X ∪ {(v,d)}
10           function relax (Q,(u,w)) = PQ.insert (d+w,u) Q
11             Q'' = iter relax Q' (N_G(v))
12         in dijkstra'(X', Q'') end
13 in
14   dijkstra'({}, PQ.insert (0,s) {})
15 end
```

The algorithm maintains the visited set X as a table mapping each visited vertex u to $d(u) = \delta_G(s,u)$. It also maintains a priority queue Q that keeps the frontier prioritized based on the shortest distance from s through vertices in X. On each round, the algorithm selects the vertex x with least distance d in the priority queue (Line 4 in the code) and, if it hasn't already been visited, visits it by adding $(x \mapsto d)$ to the table of visited vertices (Line 9), and then adds all its neighbors v to Q along with the priority $d(x) + w(x,v)$ (i.e. the distance to v through x) (Lines 10 and 11). Note that a neighbor might already be in Q since it could have been added by another of its in-neighbors. Q can therefore contain duplicate entries for a vertex, but what is important is that the minimum distance will always be pulled out first. Line 7 checks to see whether a vertex pulled from the priority queue has already been visited and discard it if it has. This algorithm is just a concrete implementation of the previously described Dijkstra's algorithm.

Dijkstra variants. A variant of Dijkstra's algorithm is to decrease the priority of the neighbors instead of adding duplicates to the priority queue. This requires a more powerful priority queue that supports a `decreaseKey` function. Another variant is to instead of visiting just the closest vertex, to visit all the equally closest vertices on each round, as suggested by the more abstract algorithm. This

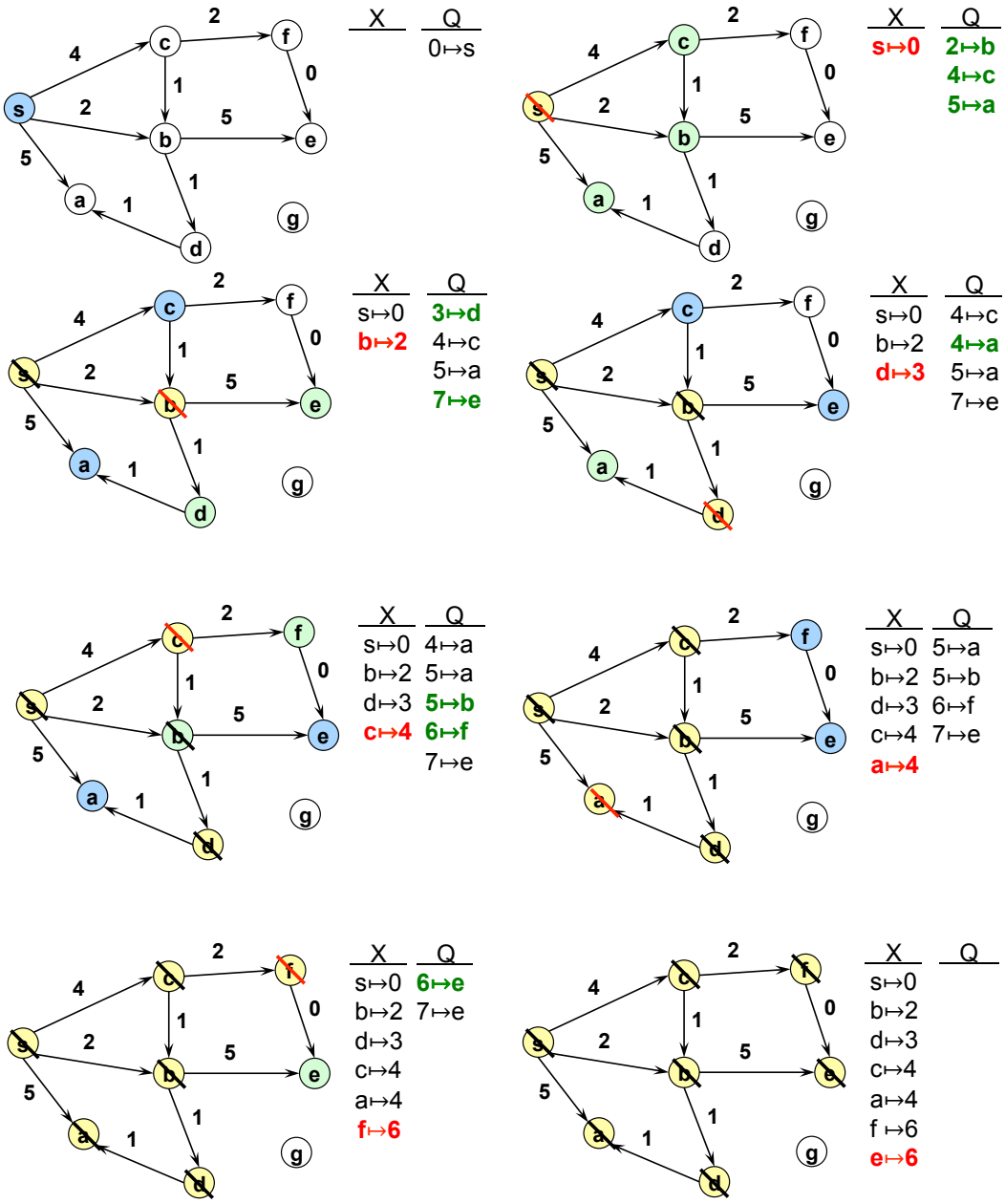


Figure 1: An example of Dijkstra's algorithm initialization and the end of each round. Notice that after visiting s, b, and d there are two distances in Q for a corresponding to the two paths from s to a; only the shortest distance will be added to X. In the next step, both f and b are added to Q, even though b is already visited. After visiting a, the redundant entries for a and b are discarded before visiting f. Finally, e is visited and the remaining element in Q is discarded. The vertex g is never visited as it is unreachable from s. If the priority queue supports a function that can return all minimum value keys, then the rounds for visiting c and a can be done in parallel, since they have the same shortest distance. Finally, notice that the distances in X never decrease.

variant has the advantage that the vertices can be visited in parallel. If all edges have unit weight, for example, this variant does the same as parallel BFS, visiting one level at a time. The amount of parallelism, however, depends on how many vertices have equal distance. This variant requires that the priority queue supports a function that returns all minimum valued keys.

Costs of Dijkstra's Algorithm. We now consider the cost of Dijkstra's algorithm by counting up the number of operations. In the code we have put a box around each operation. The `PQ.insert` in Line 14 is called only once, so we can ignore it. Of the remaining operations, The `iter` and $N_G(v)$ on line 11 are on the graph, Lines 7 and 9 are on the table of visited vertices, and Lines 4 and 10 are on the priority queue. For the priority queue operations, we have only discussed one cost model which, for a queue of size n, requires $O(\log n)$ work and span for each of `PQ.insert` and `PQ.deleteMin`. We have no need for a `meld` operation here. For the graph, we can either use a tree-based table or an array to access the neighbors[1] There is no need for single threaded array since we are not updating the graph. For the table of distances to visited vertices we can use a tree table, an array sequence, or a single threaded array sequences. The following table summarizes the costs of the operations, along with the number of calls made to each operation. There is no parallelism in the algorithm so we only need to consider the sequential execution of the calls.

Operation	Line	# of calls	PQ	Tree Table	Array	ST Array
`deleteMin`	4	$O(m)$	$O(\log m)$	-	-	-
`insert`	10	$O(m)$	$O(\log m)$	-	-	-
Priority Q total			$O(m \log m)$	-	-	-
`find`	7	$O(m)$	-	$O(\log n)$	$O(1)$	$O(1)$
`insert`	9	$O(n)$	-	$O(\log n)$	$O(n)$	$O(1)$
Distances total			-	$O(m \log n)$	$O(n^2)$	$O(m)$
$N_G(v)$	11	$O(n)$	-	$O(\log n)$	$O(1)$	-
`iter`	11	$O(m)$	-	$O(1)$	$O(1)$	-
Graph access total			-	$O(m + n \log n)$	$O(m)$	-

We can calculate the total number of calls to each operation by noting that the body of the let starting on Line 8 is only run once for each vertex. This means that Line 9 and $N_G(v)$ on Line 11 are only called $O(n)$ times. Everything else is done once for every edge.

Based on the table one should note that when using either tree tables or single threaded arrays the cost is no more than the cost of the priority queue operations. Therefore there is no asymptotic advantage of using one over the other, although there might be a constant factor speedup that is not insignificant. One should also note that using regular purely functional arrays is not a good idea since then the cost is dominated by the insertions and the algorithm runs in $\Theta(n^2)$ work.

The total work for Dijkstra's algorithm using a tree table $O(m \log m + m \log n + m + n \log n) = O(m \log n)$ since $m \leq n^2$.

[1] We could also use a hash table, but we have not yet discussed them.

4 SML code

Here we present the SML code for Dijkstra.

```
functor TableDijkstra(Table : TABLE) =
struct
  structure PQ = Default.RealPQ
  type vertex = Table.key
  type 'a table = 'a Table.table
  type weight = real
  type 'a pq = 'a PQ.pq
  type graph = (weight table) table

  (* Out neighbors of vertex v in graph G *)
  fun N(G : graph, v : vertex) =
    case (Table.find G v) of
      NONE => Table.empty()
    | SOME(ngh) => ngh

  fun Dijkstra(u : vertex, G : graph) =
    let
      val insert = Table.insert (fn (x,_) => x)

      fun Dijkstra'(Distances : weight table,
                    Q         : vertex pq) =
        case (PQ.deleteMin(Q)) of
            (NONE, _)       => Distances
          | (SOME(d, v), Q) =>
            case (Table.find Distances v) of

              (* if distance already set, then skip vertex *)
              SOME(_) => Dijkstra'(Distances, Q)

            | NONE =>
              let
                val Distances' = insert (v, d) Distances
                fun relax (Q,(u,l)) = PQ.insert (d+l, u) Q

                (* relax all the out edges *)
                val Q' = Table.iter relax Q (N(G,v))
              in
                Dijkstra'(Distances', Q')
              end
    in
      Dijkstra'(Table.empty(), PQ.singleton (0.0, u))
    end
end
```