CSOR W4231.002 Lecture 9: Apr 7, 2014 Spring 2014

Eleni Drinea

Today we will discuss algorithms for finding shortest paths on weighted graphs. First we will discuss Dijkstra's algorithm for finding single-source shortest paths in directed graphs with non-negative edge weights. Next we will discuss the Bellman-Ford algorithm for computing single-source shortest paths in directed graphs where edge weights may be negative. We will show how to modify the algorithm to detect negative cycles in such graphs. Finally we will discuss an algorithm by Floyd and Warshall that computes all-pairs shortest paths in graphs with real edge weights.

1 The single-source shortest paths problem

Consider a weighted, directed graph $G = (V, E, weight)$ where function $weight : E \to R$ maps edges to real-valued weights (edge weights may represent distances, cost, time, or in general some quantity that accumulates over a path and we would like to minimize).

The weight of a path is given by the sum of the weights of its edges. A shortest path from u to v in a weighted graph is a path of minimum weight among all paths from u to v. If there is no path from u to v, we use the convention that the shortest path from u to v has weight ∞.

Given a graph $G = (V, E)$ and a **source** vertex $s \in V$, the **single-source shortest paths problem** asks for a shortest path from s to every vertex $v \in V$. We denote the weight of a shortest path from s to u by $distance(u)$, where it is implied that distances are calculated from s.

Suppose we had an algorithm A to solve the single-source shortest paths problem. Then we could also solve the following problems.
1. **Single-pair shortest-path problem**: it turns out that all algorithms known for computing shortest path from s to u have to compute shortest paths from s to all vertices reachable from s.
2. **Single-destination shortest-paths problem:** we can find shortest paths from every vertex to a destination t by running A on G^r (G with edge directions reversed).
3. **All-pairs shortest-paths**: we can find a shortest path between every pair (u, v) of vertices by running A once from every vertex. In Section 2, we will show a faster algorithm to tackle this problem.

1.1 Dijkstra's algorithm for non-negative edge weights

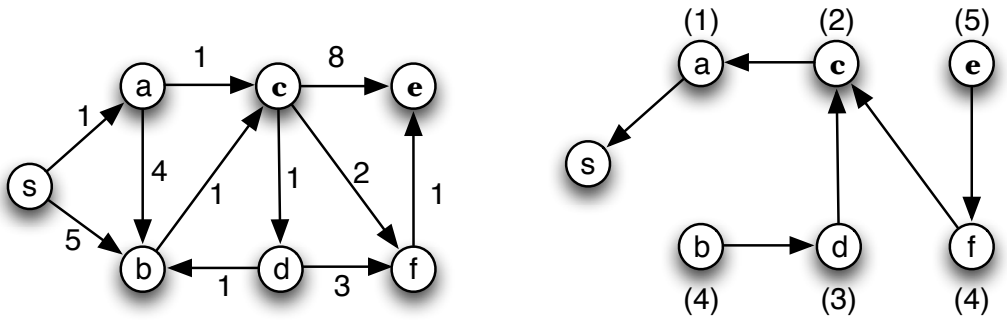
In directed graphs where edge weights are non-negative, Dijsktra's algorithm computes single-source shortest paths. At all times, the algorithm maintains a set S of vertices for which it has determined a shortest-path distance from s. Initially, this set only contains s, with $distance(s) = 0$. In every iteration, the algorithm goes over the nodes in $V - S$ and selects the node v that

1. has an incoming edge from some vertex in S and
2. minimizes the following quantity among all nodes $v \in V - S$

$$d(v) = \min_{e=(u,v):u \in S} distance(u) + weight(u, v).$$

It then adds v to S and stores $previous(v) = u$ to use for reconstructing the shortest path from s to v.

Figure 1(b) shows shortest paths and distances from s for all vertices in the graph in Figure 1(a) as computed by Dijkstra.



(a) A directed, weighted graph

(b) Shortest-paths and distances from s

Figure 1: Distances and shortest paths from s for the graph in Figure 1(a).

1.2 Correctness of Dijkstra's algorithm

Dijkstra's algorithm is a greedy algorithm: it always forms the shortest new s-v path by a path in S followed by a single edge. To prove optimality of Dijkstra we will show that the greedy solution "always stays ahead of any other solution". Specifically, we will show that each time the algorithm selects a path to some node v, that path is shorter than every other possible path from s to v.

Claim 1 *Consider the set S at any point in the algorithm's execution. For each u in S, the path P_u is a shortest s-u path.*

Note that if the claim is true, then correctness of Dijsktra's algorithm follows: applying the claim when the algorithm terminates (hence S=V) shows that Dijkstra correctly computes s-u paths for all $u \in V$.

Proof. By induction on the size of S.

- Base case: Initially $|S| = 1$ (when $S = \{s\}$). Then $distance(s) = 0$.
- Induction hypothesis: suppose the claim is true for $|S| = k$, that is, for every $u \in S$, P_u is a shortest s-u path.
- Induction step: the next node we add to S is v, by an edge (u, v) from some node $u \in S$. We want to show that P_v, which is P_u followed by (u, v) is a shortest s-v path.

  To this end, consider any other s-v path, call it P. We will show that it is at least as long as P_v. P must leave S somewhere since $v \notin S$: let y be the first node of P in $V - S$ and x the last node of P in S. Then the path to y follows the shortest path from s to x and then the edge (x, y). Since $y \notin S$ and the algorithm added v at this step and not y, it must be that

  $$distance(u) + weight(u, v) = d(v) \leq d(y) = distance(x) + weight(x, y).$$

  This means that there is no path from s to y through x that is shorter than P_v. So just the subpath from s to y through x in P is longer than P_v! Thus the full path P is longer as well (since edge weights are ≥ 0).

□

1.3 Implementation of Dijkstra

A straightforward implementation of the above algorithm follows.

Dijkstra-Version1($G = (V, E, weight), s \in V$)
  Initialize(G, s)
  $S = \{s\}$
  while $S \neq V$ **do**
    Select a node $v \in V - S$ with at least one edge from S that minimizes
      $d(v) = \min_{e=(u,v):u \in S} distance[u] + weight(u, v)$.
    $S = S \cup \{v\}$
    $distance[v] = d(v)$
    $previous[v] = u$
  end while

Initialize(G, s)
  for $v \in V$ **do**
    $distance[v] = \infty$
    $previous[v] = NIL$
  end for
  $distance[s] = 0$

The running time for this implementation is $O(mn)$ (*why?*).

An improvement would be to avoid recomputing $d(v)$ at every iteration of the while loop for every node in $V - S$. To this end, we could use $distance[u]$ to store a **conservative overestimate** of the true shortest s-u path at all times (starting from the extremely conservative overestimate $distance[u] = \infty$). We would also need update $distance[v]$ for all nodes v that have an incoming edge from u. The pseudocode for this improved procedure follows.

Dijkstra-Version2($G = (V, E, weight), s \in V$)
  Initialize(G, s)
  $S = \emptyset$
  while $S \neq V$ **do**
    Pick the node u such that $distance[u]$ is minimal among all nodes in $V - S$
    $S = S \cup \{u\}$
    for $(u, v) \in E$ **do**
      Update(u, v)
    end for
  end while

Update(u, v)
  if $distance[v] > distance[u] + weight(u, v)$ **then**
    $distance[v] = distance[u] + weight(u, v)$
    $previous[v] = u$
  end if

The running time for this implementation is $O(n^2)$ (*why?*).

Finally, we could further improve the above implementation when $m = o(n^2/\log n)$ by using a priority queue implemented as a binary min-heap, where we store vertex u with key the (over)estimate of its distance from s, given by $distance[u]$. The operations we will need are Insert, ExtractMin and

DecreaseKey (in Update()); each requires $O(\log n)$ time. The pseudocode follows.

Dijkstra-Version3($G = (V, E, weight), s \in V$)
  $Q = \{V; distance\}$
  $S = \emptyset$
  while $Q \neq \emptyset$ **do**
    $u = \text{ExtractMin}(Q)$
    $S = S \cup \{u\}$
    for $(u, v) \in E$ **do**
      Update(u, v)
    end for
  end while

The running time of this implementation is $O(n \log n + mlogn) = O(m \log n)$: it performs n Extract-Min operations, followed by at most m DecreaseKey.

The best implementation for Dijsktra's algorithm uses a Fibonacci heap, which is basically a heap organized so that Insert and DecreaseKey each requires $O(1)$ amortized time, thus it requires $O(n \log n + m)$ time.

1.4 Negative edge weights: why Dijkstra's algorithm fails

Intuitively, the reason why Dijkstra's algorithm fails for negative weights is that a path may start on expensive edges but then compensate along the way with cheap edges (e.g., see Figure 2). Specifically, in the proof of correctness of Claim 1, the last statement about the overall path P does not hold anymore: even if P_y is longer than P_v, the rest of P may contain negative edges that could still render P cheaper than P_v.

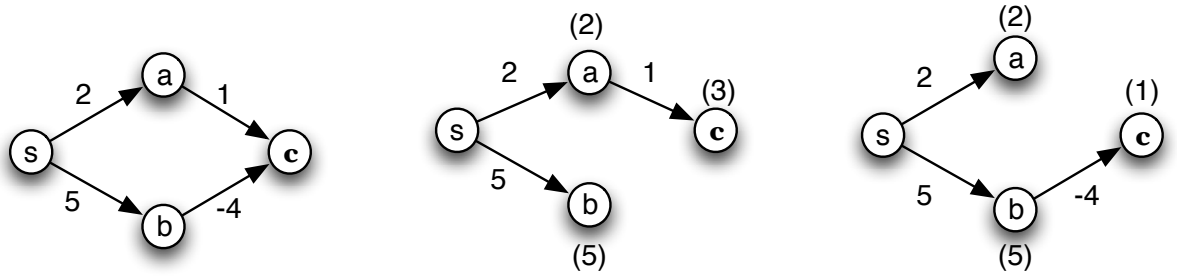


Figure 2: Dijkstra will first include a to S and then c, thus missing the shorter path from s to b to c.

Negative weight edges may cause bigger problems than the fact that Dijkstra does not provide the correct solution in their presence, namely, negative cycles (see Figure 3).

The problem of finding shortest paths in a graph with negative cycles has no answer. So we will want to detect such cycles.

1.5 A dynamic programming solution to finding shortest paths in graphs with real edge weights and no negative cycles (Bellman-Ford)

We argued that the problem of finding shortest paths in a graph with negative cycles has no answer. On the other hand, the problem is well defined for graphs with positive cycles or cycles with zero weight. Although an actual shortest path cannot contain a positive cycle (*why?*), it may contain a cycle of zero

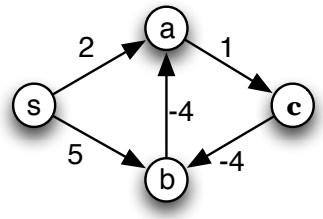


Figure 3: The weights of the shortest paths of a, c, b from s tend to $-\infty$ in the limit, as the cycle is traversed over and over again.

weight. However, if we ignore the latter, then we obtain a shortest path with the same weight as the path with the cycle and **fewer** edges.

Fact 1 *If G has no negative cycles, then there is a shortest path from s to v that is simple (i.e., has no cycles), and hence, has at most $n-1$ edges.*

Now suppose that G has no negative cycles and P is a shortest path from u to v. Then the shortest path problem exhibits optimal substructure, that is, any subpath of P must be shortest itself (e.g., see your textbook for a proof).

Fact 1 and optimal substructure of the shortest path problem motivate a dynamic programming solution to find single-source shortest-paths with negative weights **if** there are no negative cycles reachable from the source vertex s: to reach a vertex v from s, the optimal path can use at most $n-1$ edges.

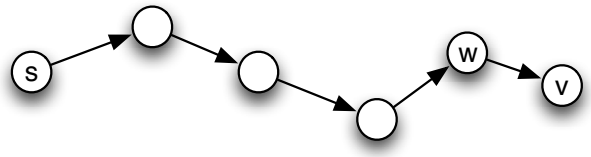


Figure 4: Shortest path from s to v using at most i edges.

So let $OPT(i, v)$ be the minimum cost of an s-v shortest path that uses at most i edges and consider an optimal path P from s to v that achieves $OPT(i, v)$ (see Figure 4).

- If the path uses at most $i-1$ edges, then

$$OPT(i, v) = OPT(i-1, v).$$

- If the path uses i edges, then

$$OPT(i, v) = \min_{w:(w,v)\in E}(OPT(i-1, w) + weight(w, v)).$$

We conclude the following recurrence

$$OPT(i, v) = \begin{cases} 0 & , \text{if } i = 0 \text{ and } v = s \\ \infty & , \text{if } i = 0 \text{ and } v \neq s \\ \min\left(OPT(i-1, v), \min_{w:(w,v)\in E}(OPT(i-1, w) + weight(w, v))\right) & , \text{if } i > 0 \end{cases}$$

Note that the above formulation assumes that the graph has no negative cycles (*how?*). The following pseudocode computes $OPT(i, v)$ in an $n \times n$ dynamic programming table $M[0, \ldots, n-1][1, \ldots, n]$, computed column by column, top down.

Bellman-Ford($G = (V, E, weight), s \in V$)
  $M[0, s] = 0$
  for $v \in V - \{s\}$ **do** $M[0, v] = \infty$
  end for
  for $i = 1, \ldots, n-1$ **do**
    for $v \in V$ *(in any order)* **do**
      $M[i, v] = \min\Big(M[i-1, v], \min_{w:(w,v)\in E}(M[i-1, w] + weight(w, v))\Big)$
    end for
  end for

The running time of this algorithm is $O(nm)$. The space is $\Theta(n^2)$ but can be improved significantly: even though it is obvious that we only need to store two columns of M at all times, one array M of size n suffices. Equivalently, we may drop the index i from $M[i, v]$ and only use it as a counter for the number of repetitions. Thus we have

$$M[v] = \min(M[v], \min_{w:(w,v)\in E}(M[w] + weight(w, v)))$$

It is not hard to see that throughout the algorithm, $M[v]$ is the length of some path from s to v and, after i rounds of updates, the value $M[v]$ is no larger than the length of the shortest path from s to v using at most i edges. Finally, if we want to reconstruct the actual shortest paths, we may use one extra array *previous* of size n.

1.6 Alternative Bellman-Ford formulation

An alternative formulation of the natural DP algorithm discussed in the previous section follows.

Bellman-Ford($G = (V, E, weight), s$)
  Initialize(G, s)
  for $i = 1, \ldots, n-1$ **do**
    for $(u, v) \in E$ **do**
      Update(u, v)
    end for
  end for

This simple algorithm uses the same time and space as the DP algorithm in the previous section. The intuition behind it is as follows (see your textbook for a formal proof of correctness).

Let $P = (s = v_0, v_1, \ldots, v_k = v)$ be a shortest path from s to v. If we update the edges on this shortest path in this order, that is, first (s, v_1), then (v_1, v_2), then (v_2, v_3) and so on and so forth, we are guaranteed to find $distance(v_i)$ for all intermediate v_i (by induction). Hence in the end we will find $distance(v_k) = distance(v)$.

So how do we guarantee a sequence of updates in this order occurs? Note that we don't care if an edge gets updated several times in between, or if other edges get updated as well: all we care is that the particular sequence of updates happens! Therefore, since a path consists of at most $n-1$ edges, update all edges $n-1$ times in a row.

1.7 Detecting Negative Cycles

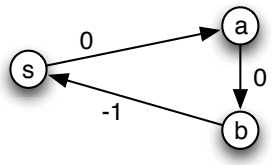
If the graph contains no negative cycles reachable from s, then a shortest s-v path may contain at most $n-1$ edges. Therefore,

$$OPT(i,v) = OPT(n-1,v) \text{ for all } v \text{ and } i \geq n. \tag{1}$$

If there is a negative cycle reachable from s, the weights of the shortest paths of the vertices on the cycle will keep decreasing, that is

$$\lim_{i\to\infty} OPT(i,v) = -\infty \text{ for all } v \text{ in the cycle.} \tag{2}$$

So how big does i have to be before we may conclude that the graph has no negative cycles?
To answer this question, we will first examine what happens if we update the values $OPT(i,v)$ once again, that is, $i = n$. A specific node in a negative cycle reachable from s may still satisfy $OPT(n,v) = OPT(n-1,v)$; e.g., see Figure 1.7, where $n = 3$. The shortest paths for a and b are only updated at $i = 4$ and $i = 5$ respectively, since the cycle has to be fully traversed once first, thus $OPT(3,a) = OPT(2,a)$ and $OPT(3,b) = OPT(2,b)$.



However, $OPT(3,s) \neq OPT(2,s)$ because the shortest path from s to itself gets updated at repetition $i = n = 3$. So this motivates the following question: *Can all nodes in a negative cycle reachable from s satisfy*

$$OPT(n,v) = OPT(n-1,v)?$$

Suppose they all did. Obviously every node in the graph that is not on the negative cycle also satisfies this equation. Then $OPT(n+1,v) = OPT(n-1,v)$ for all $v \in V$ since $OPT(n+1,v)$ values only depend on $OPT(n,v)$ values. Hence under the assumption that these values remained unchanged at iteration n, there is no reason why they should change at iteration $n+1$. The same holds for all future iterations, so

$$OPT(i,v) = OPT(n-1,v) \text{ for all } i \geq n.$$

Equation 2 states that the OPT values of nodes on a negative cycle reachable by s should become arbitrarily negative as i increases, we conclude that there cannot be a negative cycle that has a path from s, a contradiction. So we reached a contradiction that proves the following claim (the forward direction follows from our reasoning for equation 1).

Claim 2 *There is no negative cycle reachable from s if and only if $OPT(n,v) = OPT(n-1,v)$ for all nodes.*

So now we have a direct way to detect negative cycles:

1. Modify Bellman-Ford to compute (update) all $M[v]$ n times (instead of $n-1$).
2. If, after the n-th iteration, $M[v]$ changed for *any* v, then there is a negative cycle.

1.8 Shortest paths in a DAG

In DAGs, the problem of finding single-source shortest paths always has an answer (*why?*) and we can solve it faster than Bellman-Ford using the following algorithm.

DAG-Shortest-Paths($DAG\ G = (V, E, weight), s$)
  TopologicalSort(G)
  Initialize(G, s)
  for $u \in V$ in topological order **do**,
    for $(u, v) \in E$ **do**
      Update(u, v)
    end for
  end for

The correctness of this algorithm follows by an argument similar to the one in Section 1.6: specifically, suppose there is a path from s to v, say $P = (s = v_0, v_1, \ldots, v_k = v)$. Since the DAG is topologically sorted, we are updating the edges in the order $(s, v_0), (v_0, v_1), \ldots, (v_{k-1}, v)$. Therefore, we can show by induction that we are indeed computing $distance(v_i)$ for every i and, in particular, for $i = k$ we compute $distance(v)$.

The running time of this algorithm is $O(n + m)$.

2 All pairs shortest-paths intuition

Suppose we wanted to compute shortest paths between every pair of vertices in a directed, weighted graph G. As mentioned in Section 1, a straightforward solution to this problem is to run Bellman–Ford once for every vertex. This requires $O(n^2m)$ time. However we can do it faster, in $O(n^3)$ time, by using an algorithm by Floyd and Warshall.

This algorithm is again a DP algorithm with a slightly different formulation from the Bellman-Ford algorithm. The main idea is that any shortest path from s to v uses some **intermediate** vertices. For example, if $P = s, v_1, v_2, \ldots, v_k, t$ is such a path, then $v_1, \ldots, v_k$ are the intermediate vertices of P. To formulate subproblems, we will consider how a shortest path may be formed by considering progressively larger sets of possible intermediate vertices.

To make things clearer, let $V = \{1, 2, 3, \ldots, n\}$ and look at the shortest path from vertex i to vertex j, when intermediate vertices may only be from $\{1, 2, \ldots, k\}$. Let $L_k(i, j)$ be the length of this shortest path; then we want $L_n(i, j)$ for all i, j. As usual, we consider how the last vertex k of this set is involved in the shortest path under consideration.

1. Either k is not an intermediate vertex of path P, that is, P completely avoids k: in this case all intermediate vertices of P are from $\{1, \ldots, k-1\}$ and therefore a shortest path from i to j with intermediate nodes from $\{1, \ldots, k\}$ is a shortest path from i to j with intermediate nodes from $\{1, \ldots, k-1\}$.

2. Or, k is an intermediate vertex of path P. Then we may think of P as consisting of two subpaths $P_1 = i \to k$ and $P_2 = k \to j$ glued together (see Figure 5). Because vertex k is **no longer** an intermediate vertex of either subpath, now each subpath is itself a path with all intermediate vertices from $\{1, \ldots, k-1\}$.

Since shortest paths exhibit optimal substructure, based on our observations above, we obtain the following recurrence.

$$L_k(i, j) = \begin{cases} weight(i, j) & \text{, if } k = 0 \\ \min(L_{k-1}(i, j), L_{k-1}(i, k) + L_{k-1}(k, j)) & \text{, if } k \geq 1 \end{cases}$$

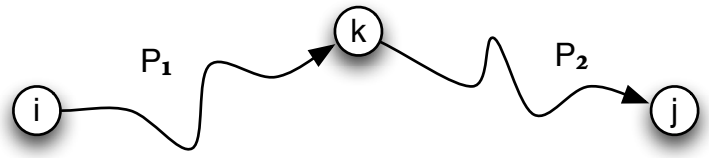


Figure 5: All intermediate vertices in $P = P_1P_2$ are from $\{1, \ldots, k\}$.

There are $\Theta(n^3)$ subproblems, each requiring $O(1)$ time to be computed from smaller subproblems. Similarly to Section 1.5, although it might look like we need to keep at least two matrices to compute $L_k(i, j)$, we can accomplish the computation by maintaining only one 2-dimensional dynamic programming matrix D (think about it), as in the following pseudocode. D is initialized to $weight(i, j)$, which corresponds to $k = 0$. That is, the algorithm checks if there is a direct edge between i and j. To perform this check in $O(1)$, the algorithm uses an adjacency matrix representation of G.

Floyd-Warshall($weight$)
for $i = 1$ to n **do**
  for $j = 1$ to n **do** $D[i, j] = weight(i, j)$
  end for
end for
for $k = 1$ to n **do**
  for $i = 1$ to n **do**
    for $j = 1$ to n **do**
      $D[i, j] = \min(D[i, j], D[i, k] + D[k, j])$
    end for
  end for
end for

The running time of this algorithm is $O(n^3)$ and the space is $\Theta(n^2)$.