Algorithm Design, Analysis & Complexity

Lecture 5 - Graph Algorithms

Koushik Pal

University of Toronto

June 1, 2021

Shortest Path on Weighted Graph

Definition

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$.
Let $P = \langle v_0, v_1, \ldots, v_k \rangle$ be a path. The weight of P is defined as

$$w(P) := \sum_{i=0}^{k-1} w(v_i, v_{i+1}).$$

The shortest path weight $\delta(u, v)$ from u to v is defined as

$$\delta(u,v) := \begin{cases} \min\{w(P) \mid P \text{ is a path from } u \text{ to } v\} & \text{if such a path exists} \\ \infty & \text{otherwise.} \end{cases}$$

Shortest Path Problem

Problem

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, compute a shortest path from a source node s to a destination node t.

Optimal substructure of a shortest path

Let $P = \langle v_0, \ldots, v_k \rangle$ be a shortest path from v_0 to v_k. Then $\langle v_i, \ldots, v_j \rangle$ is a shortest path from v_i to v_j for any $0 \le i < j \le k$.

Variations

The shortest path problem has a few variants.

1. Single source shortest path
2. Single destination shortest path
3. All pairs shortest path
4. Negative weights

Single source shortest path on nonnegative weights

```
procedure DIJKSTRA(G = (V, E, w), s)
    Define array d of size |V|
    Set d[v] = ∞ for all v ∈ V
    Set d[s] = 0
    Define array Π of size |V|
    Set Π[v] = NIL for all v ∈ V
    Let Q be a priority queue initialized with (v, d[v]) for v ∈ V
    S := ∅
    while Q ≠ ∅ do
        u = Extract-Min(Q)
        S = S ∪ {u}
        for each vertex v ∈ Adj[u] do
            if d[v] > d[u] + w[u, v] then
                d[v] = d[u] + w[u, v]
                Π[v] = u
                Decrease-Key(Q, v)
```

Proof of correctness

Theorem

After the DIJKSTRA *algorithm terminates, we get*

$$d[v] = \delta(s, v) \textit{ for all } v \in V.$$

Proof.

By mathematical induction on the size of S.

Base Case. When $|S| = 1$, then $s \in S$ and $\delta(s, s) = 0 = d[s]$.

Ind. Hyp. Assume $\delta(s, x) = d[x]$ for all $x \in S$ when $|S| = i$.

Ind. Step. Show $\delta(s, u) = d[u]$, where u is the $(i + 1)^{th}$ element added to S.
Note that for u to be added to S, it has to be removed from Q, which means at this stage it has the minimum d value, i.e.,

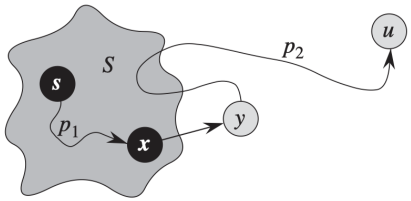
$$d[u] \leq d[z] \text{ for all } z \notin S.$$

Proof of correctness

Assume, for a contradiction, that there is a shortest path P from s to u of length $\delta(s,u) < d[u]$. Let $e = (x,y)$ be the edge where P crosses the boundary of S for the first time.

Observe the following:

1. $\delta(s,x) = d[x]$, since $x \in S$ (by Ind. Hyp.)
2. $d[u] \le d[y]$ (by the algorithm)
3. $d[y] \le d[x] + w[x,y]$ (by the algorithm)



By combining all these inequalities, we get

$$\delta(s,u) < d[u] \le d[y] \le d[x] + w[x,y] = \delta(s,x) + w[x,y] \le \delta(s,u)$$

(the last inequality is because of nonnegative weights), which yields the necessary contradiction.

Hence, $d[u] = \delta(s,u)$. $\square$

Complexity

- The Extract-Min operation runs n times, and takes $\Theta(n \lg n)$ computations.
- The Decrease-Key operation runs m times, and takes $\Theta(m \lg n)$ computations, if implemented with a binary heap.

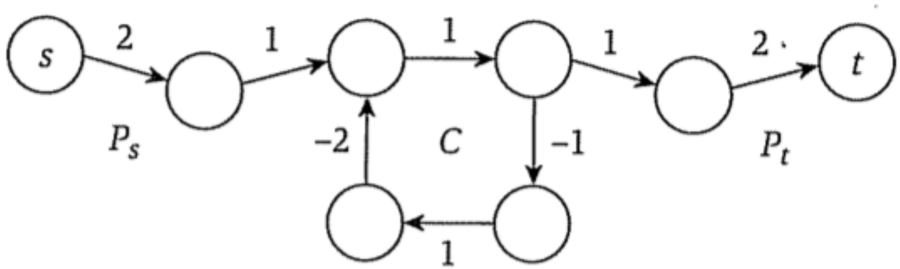
Thus, total time complexity if Q is implemented via a binary heap is $\Theta(m \lg n)$.

But if Q is implemented using Fibonacci heap, then the amortized cost of running all the Decrease-Key operations drops to $\Theta(n \lg n)$. The inside loop still runs m times across all runs of the outer loop, giving a total complexity of $\Theta(m + n \lg n)$.

Single-destination shortest path on negative weights

Unfortunately, DIJKSTRA doesn't work when the weights are allowed to be negative. Having the weights to be nonnegative is a crucial part in the proof of DIJKSTRA.

When weights are allowed to be negative, there can be more complications. For example, if there is a negative weight cycle (all edges on the cycle have negative weights), then the shortest path weight between two vertices on the cycle is $-\infty$, i.e., there is no "shortest" path between those two vertices.



Fortunately, that's the only hindrance to having a shortest path!

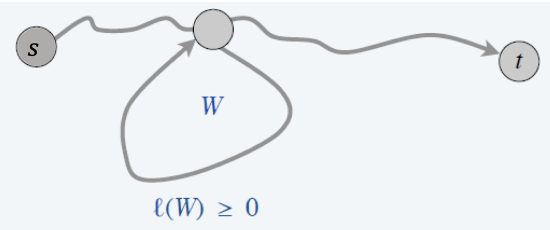
Why?

Lemma

If $G = (V, E)$ has no negative cycle and t is reachable from s, then there is a shortest path from s to t that is simple (i.e., does not repeat nodes), and hence has at most $|V|-1$ edges.

Proof.

Since every cycle has nonnegative weight, the shortest path P from s to t with the fewest number of edges does not repeat any vertex v.



For if P did repeat a vertex v, the portion of P between consecutive visits to v can be removed, resulting in a path of no greater cost. $\square$

DP Solution

We will use DP to solve the shortest path problem on weighted graphs with negative weights.

Define $OPT(i, v)$ to be the shortest path weight from v to t using at most i edges. By Lemma 2, our goal is to compute the value of $OPT(n-1, s)$.

Let P be an optimal path representing $OPT(i, v)$. Now, two things can happen:

1. P uses at most $i-1$ edges. In this case, $OPT(i, v) = OPT(i-1, v)$.
2. P uses i edges with the first edge being (v, u). In this case, $OPT(i, v) = w(v, u) + OPT(i-1, u)$.

This gives the following recursive formula: if $i > 0$, then

$$OPT(i, v) = \min\{OPT(i-1, v), \min_{u \in V}\{OPT(i-1, u) + w(v, u)\}\}.$$

DP Solution

Define an array M such that $M[i, v]$ denotes $OPT(i, v)$.
Now, we write a bottom-up iterative algorithm to compute M.

---

procedure $\textsc{ShortestPath}(G = (V, E, w), t)$
    Let $n := |V|, m := |E|$
    Define 2-D array $M[0, \ldots, n-1, v_0, \ldots, v_{n-1}]$
    $M[0, t] = 0$ and $M[0, v] = \infty$ for all $v \in V \setminus \{t\}$
    for $i = 1, \ldots, n-1$ **do**
        for $v \in V$ **do**
            $M[i, v] = M[i-1, v]$
            for $u \in Adj[v]$ **do**
                if $M[i, v] > M[i-1, u] + w[v, u]$ **then**
                    $M[i, v] = M[i-1, u] + w[v, u]$
    return M

Complexity: $\Theta(n \sum_{v \in V} n_v) = \Theta(nm)$.
Here n_v is the degree of node v (number of edges going out of v).

Space Complexity

Space complexity: $\Theta(n^2)$.

But a close look at the algorithm shows that we don't need to store $M[i, v]$ for all values of i. We just need the values at stage $i-1$ to compute the values for stage i.

Hence, we can use a 1-D array, and rewrite the above algorithm as follows.

Bellman-Ford Algorithm

procedure $\textsc{Bellman-Ford}(G = (V, E, w), t)$

Let $n := |V|, m = |E|$

Define 1-D array $M[v_0, \ldots, v_{n-1}]$

$M[t] = 0$ and $M[v] = \infty$ for all $v \in V \setminus \{t\}$

for $i = 1, \ldots, n-1$ **do**

  for $v \in V$ **do**

    for $u \in Adj[v]$ **do**

      if $M[v] > M[u] + w[v, u]$ **then**

        $M[v] = M[u] + w[v, u]$

for each edge $(v, u) \in E$ **do**

  if $M[v] > M[u] + w[v, u]$ **then**

    return $\textsc{Nil}$

return M

Time Complexity: $\Theta(nm)$.
Space Complexity: $\Theta(n)$.

Bellman-Ford Analysis

Bellman-Ford not only computes M for a graph with negative weights, but it also returns whether the graph has a negative cycle or not. If there is a negative cycle, it returns Nil; otherwise, it returns the computed array M.

Claim

If there is a negative weight cycle in G that can reach t, Bellman-Ford returns Nil.

Proof.

Assume G contains a negative weight cycle that can reach t. Let this cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$ where $v_0 = v_k$. Then

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0.$$

Assume for a contradiction that Bellman-Ford does not return Nil. Thus, $M[v_{i-1}] \leq M[v_i] + w(v_{i-1}, v_i)$ for $i = 1, \ldots, k$.

Bellman-Ford Analysis

Proof (cont.)

Summing the inequalities around the cycle c gives

$$\begin{aligned}
\sum_{i=1}^{k} M[v_{i-1}] &\leq \sum_{i=1}^{k} \Big(M[v_i] + w(v_{i-1}, v_i) \Big) \\
&= \sum_{i=1}^{k} M[v_i] + \sum_{i=1}^{k} w(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} M[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) \ \text{(since } v_0 = v_k\text{)}.
\end{aligned}$$

This implies $\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$, a contradiction. Hence, Bellman-Ford returns Nil. □

Exercise: Modify Bellman-Ford to find an actual shortest path from any vertex v to t.

All-pairs shortest path

Let $G = (V, E)$ be a weighted directed graph with $|V| = n$ and $|E| = m$. The goal now is to find a shortest path from *any* vertex to *any* other vertex in G.

An obvious thing to do is to run a single source shortest path algorithm from every vertex in the graph.

On non-negative weighted graph, we can apply $\textsc{Dijkstra}$ from each vertex. This has a complexity of $\Theta(n^2 \lg n + nm)$.

On a graph with negative weights but no negative cycle, we can apply $\textsc{Bellman-Ford}$ from each vertex. This has a complexity of $\Theta(n^2 m)$.

Question: Can we do any better?

Generalization of Bellman-Ford

Define $\ell_{ij}^{(m)} :=$ shortest path weight from vertex v_i to vertex v_j that contains at most m edges. Then

$$\ell_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j, \end{cases}$$

and for $m \geq 1$,

$$\begin{aligned} \ell_{ij}^{(m)} &= \min\left\{\ell_{ij}^{(m-1)}, \min_{1\leq k\leq n, k\neq j}\{\ell_{ik}^{(m-1)} + w(k,j)\}\right\} \\ &= \min_{1\leq k\leq n}\{\ell_{ik}^{(m-1)} + w(k,j)\} \quad (\text{since } w(j,j) = 0) \end{aligned}$$

We now use this recurrence to compute a series of matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, where $L^{(k)} = (\ell_{ij}^{(k)})$, starting with $L^{(1)} = W = (w(i,j))$ and ending with $L^{(n-1)}$ which contains the actual shortest path weights.

Generalization of Bellman-Ford

procedure SLOW-ALL-PAIRS-SHORTEST-PATHS(W)

  $n :=$ number of rows of W

  $L^{(1)} := W$

  for $k = 2, \ldots, n-1$ **do**

    $L^{(k)} =$ EXTEND-SHORTEST-PATHS $(L^{(k-1)}, W)$

  return $L^{(n-1)}$

Generalization of Bellman-Ford

```
procedure EXTEND-SHORTEST-PATHS(L, W)
    n := number of rows of L
    Let L' be a new n × n matrix
    for i = 1, . . . , n do
        for j = 1, . . . , n do
            L'[i, j] = ∞
            for k = 1, . . . , n do
                if L'[i, j] > L[i, k] + W[k, j] then
                    L'[i, j] = L[i, k] + W[k, j]
    return L'
```

Complexity

The complexity of the EXTEND-SHORTEST-PATHS is $\Theta(n^3)$.

Thus, the complexity of SLOW-ALL-PAIRS-SHORTEST-PATHS is $\Theta(n^4)$.

Unfortunately, this is no better than running BELLMAN-FORD from every vertex.

However, the complexity can be improved to $\Theta(n^3 \lg n)$ by computing $L^{(n-1)}$ as follows:

$$\begin{aligned}
L^{(1)} &= W \\
L^{(2)} &= W \cdot W = W^2 \\
L^{(4)} &= W^2 \cdot W^2 = W^4 \\
L^{(8)} &= W^4 \cdot W^4 = W^8 \\
&\vdots \\
L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}}.
\end{aligned}$$

A different DP solution

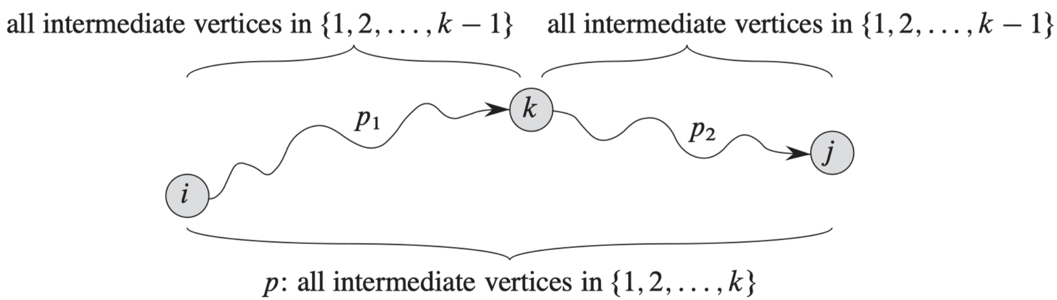
Redefining the DP subproblems in a different way gives us a better algorithm.

Let $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$.
Define $d_{ij}^{(k)}$:= shortest path weight from vertex i to vertex j for which all intermediate vertices are in the set $\{1, \ldots, k\}$.
(An intermediate vertex on a path from i to j is any vertex on the path other than i or j.)
For $k = 0$, we have $d_{ij}^{(0)} = w(i, j)$ (direct edges).

Now, consider a shortest path P from i to j where all the intermediate vertices are in the set $\{1, \ldots, k\}$. Then two things can happen:

1. k is not an intermediate vertex on P: in this case, $d_{ij}^{(k)} = d_{ij}^{(k-1)}$.
2. k is an intermediate vertex on P: in this case, $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

A different DP solution



Consequently, we obtain the following recurrence:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}.$$

We now use this recurrence to compute a series of matrices $D^{(0)}, D^{(1)}, \ldots, D^{(n)}$, where $D^{(k)} = (d_{ij}^{(k)})$, starting with $D^{(0)} = W = (w(i,j))$ and ending with $D^{(n)}$ which contains the actual shortest path weights.

Floyd-Warshall Algorithm

procedure $\textsc{Floyd-Warshall}(W)$
    $n :=$ number of rows of W
    $D^{(0)} := W$
    for $k = 1, \ldots, n$ **do**
        Let $D^{(k)}$ be a new $n \times n$ matrix
        for $i = 1, \ldots, n$ **do**
            for $j = 1, \ldots, n$ **do**
                if $D^{(k-1)}[i,j] > D^{(k-1)}[i,k] + D^{(k-1)}[k,j]$ **then**
                    $D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j]$
                else
                    $D^{(k)}[i,j] = D^{(k-1)}[i,j]$
    return $D^{(n)}$

Complexity: $\Theta(n^3)$.

Exercise: Modify $\textsc{Floyd-Warshall}$ to find an actual shortest path from any vertex i to any vertex j.

Johnson's algorithm for sparse graphs

If we have a graph with negative weights, then one might hope to somehow re-weight the edges to nonnegative weights such that the shortest paths do not change. The following lemma gives one such way.

Lemma

Given a weighted directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $h : V \to \mathbb{R}$ be any function mapping vertices to real numbers. For each edge $(u, v) \in E$, define

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v).$$

Let $P = \langle v_0, v_1, \ldots, v_k \rangle$ be any path from v_0 to v_k. Then P is a shortest path from v_0 to v_k with weight function w if and only if it is a shortest path with weight function $\hat{w}$.
Furthermore, G has a negative weight cycle using w if and only if G has a negative weight cycle using $\hat{w}$.

Proof of lemma

Proof.

$$\begin{aligned}
\hat{w}(P) &= \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} \Big(w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i) \Big) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\
&= w(P) + h(v_0) - h(v_k).
\end{aligned}$$

Thus, the new weight of any path P depends on the old weight of P and the values of the function h at the starting and the ending vertices. Since all path weights between two given vertices change by a constant amount, the shortest paths do not change under the new weight function $\hat{w}$.

If P is a cycle, then $v_0 = v_k$, and hence $w(P) = \hat{w}(P)$. This proves the second statement of the claim. □

How to define h?

The big question now is: how to come up with a function h so that all negative weights under w become nonnegative under $\hat{w}$?

The following idea works!
Given a graph $G = (V, E, w)$, introduce a new vertex s and make a new graph $G' = (V', E', w')$, where

$$\begin{aligned} V' &= V \cup \{s\} \\ E' &= E \cup \{(s, v) \mid v \in V\} \\ w'(s, v) &= 0 \text{ for all } v \in V. \end{aligned}$$

Assume G and G' have no negative-weight cycles.
Define $h(v) = \delta(s, v)$ for all $v \in V'$ (where $\delta(s, v)$ stands for the shortest path weight from s to v).
By triangle inequality of shortest path weights, it follows that $h(v) \le h(u) + w(u, v)$ for all $(u, v) \in E'$.
Thus, $w'(u, v) := w(u, v) + h(u) - h(v) \ge 0$ for all $(u, v) \in E'$.

Johnson's Algorithm

```
procedure JOHNSON(G, w)
    Define G' where V' = V ∪ {s}, E' = E ∪ {(s, v) | v ∈ V}
    Set w(s, v) = 0 for all v ∈ V
    if BELLMAN-FORD(G', w, s) == NIL then
        return NIL
    else
        for v ∈ V do
            Set h(v) := δ(s, v) as computed by BELLMAN-FORD
        for (u, v) ∈ E do
            Set ŵ(u, v) := w(u, v) + h(u) − h(v)
        Let D = (d_uv) be a new n × n matrix
        for u ∈ V do
            Run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v)  ∀v ∈ V
            for v ∈ V do
                D[u, v] = δ̂(u, v) − (h(u) − h(v))
        return D
```

Complexity

Note that we run $\textsc{Bellman-Ford}$ only once from the vertex s. We use that to compute h, and consequently, $\hat{w}$.

After that we run $\textsc{Dijkstra}$ n times, since all the weights $\hat{w}(u, v)$ are now nonnegative.

Hence, the total complexity of $\textsc{Johnson}$ is $\Theta(n^2 \lg n + nm)$.

For sparse graphs (where $m \ll n^2$), this is better than $\Theta(n^3)$.