Unit 12: Shortest Path Problems

Agenda:

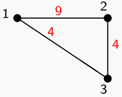
- Single-Source Shortest Path problem (SSSP)
  - Dijkstra's algorithm for the non-negative weights case
  - Bellman-Ford algorithm for graphs with no negative cycles
- All-Pairs Shortest Path problem (APSP)
  - Floyd-Washall algorithm

Reading:

- CLRS : 643-650, 658-664, 651-655, 684-686, 693-699

Shortest path problems:

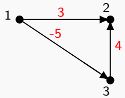
- BFS recall: outputs every s-to-v shortest path
  - s — start vertex
  - v — reachable vertex from s (residing in a same connected component)
  - shortest — # edges
  - running time $\Theta(n + m)$
- But what if the edges of the graph have weights?
  In this case, shortest-path in terms of #edges isn't shortest weighted path.



  E.g. shortest weighted-path distance between 1 and 2 is 8.
- The weight of a path = sum of weights on edges on path.
  Note: if there is no path between two nodes, the distance is set to ∞ ...
- The SHORTEST PATH problem: find the shortest path in an edge-weighted graph connecting s and t.
- Turns out, shortest-path has a few properties that allow us to infer in the process **all** shortest-paths from a node s to any other node in the graph.
  So we study the SINGLE-SOURCE SHORTEST PATH (SSSP) problem:
  Given an edge-weighted graph G and a source s, find out for each vertex $v \in V(G)$ a shortest paths from s to v.
- Variants:
  - edge weights: non-negative *vs.* arbitrary weights

Shortest Paths

- A shortest path from u to v: out of all $u \to v$ paths, it is a path of minimal weight. (Can be more than one.)
- But a shortest path must satisfy subpath optimality: if $(u_0, u_1, ..., \underbrace{u_i, ..., u_j}_{\text{optimal}}, ..., u_k)$ is a shortest $u_0 \to u_k$ path, then $(u_i, u_{i+1}, ..., u_j)$ is a shortest $u_i \to u_j$ path.
- we denote $d(u, v)$ as the shortest weighted path length from u to v
- Shortest paths remain well-defined if there are negative weight directed edges... (why not undirected?)



- ...as long as there isn't a negative weight path from a node to itself.
- So we assume **no negative cycles**.
- Shortest path distances — d is a metric:
  - For any u, we have $d(u, u) = 0$
  - For any u, v, we have $d(u, v) = d(v, u)$ (if G is undirected)
  - For any u, v, w, we have $d(u, w) \leq d(u, v) + d(v, w)$.

Common Outline of Single Source Shortest Path Algorithms

- Our shortest paths algorithm starts at a source s.
- It maintains a $dist$ attribute for each vertex that will serve as the estimation of the shortest-path distance.
- During the execution of the algorithm, we **always** have $u.dist \geq d(s,u)$.
- We start with `init()`: set $s.dist \leftarrow 0$ and $u.dist \leftarrow \infty$ for any $u \neq s$.
- We only update the $dist$ attribute by `Relaxing`

  <u>`procedure relax`(u,v)</u> `**`u,v `two adjacent nodes`
  `if` $(v.dist > u.dist + w(u,v))$ `then`
  $\quad v.dist \leftarrow u.dist + w(u,v)$
- Claim: any algorithm that starts with `init()` and only updates $dist$ using `relax()` must always satisfy $v.dist \geq d(s,v)$ for any v.
- Proof: by induction on the number of times `relax()` is invoked.
  Base case: invoked 0 times — claim holds through `init()`.
  Induction step: If `relax`(u,v) doesn't change $v.dist$ we are done.
  Otherwise, we now have

$$v.dist = u.dist + w(u,v) \geq d(s,u) + w(u,v) \geq d(s,v)$$

- Corollary: Since `relax()` cannot increases $v.dist$, so if and when we set $v.dist = d(s,v)$ we keep $v.dist$ unchanged.

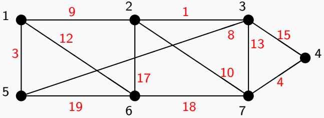
Dijkstra's SSSP algorithm:

- For graphs with non-negative weights (both directed and undirected)
- Idea in Dijkstra's algorithm:
  - Maintains a set S of vertices for which we know the shortest path. Initially, $S = \{s\}$, at the end: $S = V$.
  - Which vertex from $\bar{S} = V \setminus S$ should we pick?
  - Dijsktra: the greedy solution — the node in $\bar{S}$ with minimum $dist$.
- procedure dijkstra(G, w, s) **$G = (V, E)$, w =weights

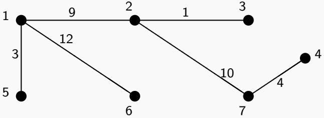
```
foreach v do                                    **initialization
    v.dist ← ∞
    v.predec ← NIL
s.dist ← 0
Build Min-Priority-Queue Q on all nodes, key = dist
            ** nodes in Q are nodes we are not yet sure about
            ** namely Q holds nodes in S̄
while (Q ≠ ∅) do
    u ← ExtractMin(Q)                           **s dequeued first
    foreach v neighbor of u do
        if (v.dist > u.dist + w(u,v)) then
            v.dist ← u.dist + w(u,v)            **a Relax() call
            v.predec ← u
            decrease-key(Q, v, v.dist)
```

Dijkstra's SSSP algorithm — an example:

- Input graph G:



- $\texttt{dijkstra}(G, 1)$:



- $\texttt{dijkstra}(G, 1)$ trace:

v	1	2	3	4	5	6	7
$v.dist/v.predec$	$0/\texttt{NIL}$	$\infty/\texttt{NIL}$	$\infty/\texttt{NIL}$	$\infty/\texttt{NIL}$	$\infty/\texttt{NIL}$	$\infty/\texttt{NIL}$	$\infty/\texttt{NIL}$
1 dequeued	$0/\texttt{NIL}$	$9/1$	$\infty/\texttt{NIL}$	$\infty/\texttt{NIL}$	$3/1$	$12/1$	$\infty/\texttt{NIL}$
5 dequeued	$0/\texttt{NIL}$	$9/1$	$11/5$	$\infty/\texttt{NIL}$	$3/1$	$12/1$	$\infty/\texttt{NIL}$
2 dequeued	$0/\texttt{NIL}$	$9/1$	$10/2$	$\infty/\texttt{NIL}$	$3/1$	$12/1$	$19/2$
3 dequeued	$0/\texttt{NIL}$	$9/1$	$10/2$	$25/3$	$3/1$	$12/1$	$19/2$
6 dequeued	$0/\texttt{NIL}$	$9/1$	$10/2$	$25/3$	$3/1$	$12/1$	$19/2$
7 dequeued	$0/\texttt{NIL}$	$9/1$	$10/2$	$23/7$	$3/1$	$12/1$	$19/2$

Dijkstra's SSSP algorithm — Correctness:

- LI: "at the start of each while-loop iteration $u.dist = d(s,u)$ for all $u \notin Q$."
- Initialization: The statement is vacuously true when Q holds all nodes. The statement is clearly true for the first vertex taken out of Q: the source s.
- Termination: At the end of the while-loop, Q is empty, so we have found all shortest-paths distances from s.
- Maintenance:
  - Suppose LI holds at the beginning of the iteration. Denote u as the node `Extract-Min(`Q`)` takes out.
  - ASOC $d(s,u) < u.dist$.
  - Look at a shortest-path $s \to u$. $s \notin Q$ but $u \in Q$. Let (x,y) be the first edge such that $x \notin Q$ but $y \in Q$.
    (y might be u, x might be s.)
  - First, $d(s,y) \leq d(s,u)$ (subpath optimality + non negative weights)
  - Second, when we took x outof Q we made sure $y.dist \leq x.dist + w(x,y)$
  - Since $x \notin Q$ in the beginning of the iteration, then $x.dist = d(s,x)$.
  - Thus $y.dist \leq d(s,x) + w(x,y) = d(s,y)$ (Again, subpath optimality)
  - Altogether: $y.dist = d(s,y) \leq d(s,u) < u.dist$
  - If $y = u$ — immediate contradiction.
    If $y \neq u$ — then `Extract-Min(`Q`)` should return y not u.
    Contradiction in any case.

Dijkstra's SSSP algorithm — analysis:

- $|V| = n$ and $|E| = m$
- Running time:
  - `init()`— takes $O(n)$ time.
  - Initializing the priority-queue — $O(n)$
  - For each node, `ExtractMin` takes $O(\log(n))$ time.
  - ... and we search for all neighbors ($O(deg(v))$ in the adjacency list model, $O(n)$ in the adjacency matrix model)
  - For every edge, we examine the edge atmost twice (each time we take its endpoints from Q) and invoke `relax()` at most once, and decrease the key at most once.
    So $O(\log(n))$ work per edge.
- In the adjacency-list model

$$O(n)+O(n)+O(n\log(n))+O(\sum_v deg(v))+O(m\log(n)) = O((n+m)\log(n))$$

- In the adjacency-matrix model

$$O(n) + O(n) + O(n(\log(n) + n)) + O(m\log(n)) = O(n^2 + m\log(n))$$

- There exists a more refined implementation of the PQ (with Fibonacci heaps) that gives runtime $O(n\log(n) + m)$

Dealing with Negative Weights

- Consider a $s \to v_k$ shortest path: $(s, v_1, v_2, ..., v_k)$ (has k edges)
- Subpath optimality: this is also a shortest path for any $s \to v_i$.
- Suppose that among the `relax()` calls of our algorithm it also made a subsequence of calls (in this order):
  $\texttt{relax}(s, v_1)$, $\texttt{relax}(v_1, v_2)$, $\texttt{relax}(v_2, v_3)$, ... , $\texttt{relax}(v_{k-1}, v_k)$,
  - The first call sets
    $v_1.dist \leq 0 + w(s, v_1) = d(s, v_1) \qquad \Rightarrow v_1.dist = d(s, v_1)$
  - The second call sets
    $v_2.dist \leq v_1.dist + w(v_1, v_2) = d(s, v_2) \Rightarrow v_2.dist = d(s, v_2)$
  - The third call sets
    $v_3.dist \leq v_2.dist + w(v_2, v_3) = d(s, v_3) \Rightarrow v_3.dist = d(s, v_3)$
  - ...the k-th call sets $v_k.dist = d(s, v_k)$
- How can we make sure we have such k calls?
- Brute force solution: makes all the `relax()` calls on all edges k times.
- What's k? At most $n - 1$...

Bellman-Ford Algorithm

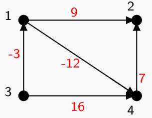
- procedure Bellman-Ford(G, s)

```
foreach v ∈ V(G) do
    v.dist ← ∞                              ** Initialization
    v.predec ← NIL
s.dist ← 0
for (i from 1 to n − 1) do
    foreach edge (u, v) do
        if (v.dist > u.dist + w(u, v)) then
            v.dist ← u.dist + w(u, v)       ** Relax()
            v.predec ← u
foreach edge (u, v) do
    if (v.dist > u.dist + w(u, v)) then
        return ‘‘negative-cycle’’
```

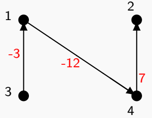
- Why is the negative cycle-check true? (last 3 lines)
  - Along a negative cycle, we keep relaxing and relaxing infinitely many times...
- Runtime on a graph with n nodes and m edges:
  $O(n \cdot m)$ in the adjacency list model
  $O(n^2 + n \cdot m)$ in the adjacency matrix model (first construct the array of edges in $O(n^2)$ time)

Bellman-Ford algorithm — an example:

- Input graph G:



- `Bellman-Ford`$(G, 3)$:

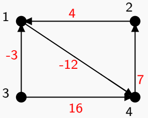


- `Bellman-Ford`$(G, 3)$ trace:
  Edge order: $(4, 2), (1, 2), (1, 4), (3, 1), (3, 4)$

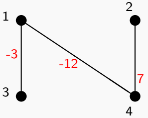
v	1	2	3	4
$v.dist/v.predec$	∞/NIL	∞/NIL	0/NIL	∞/NIL
round 1	$-3/3$	∞/NIL	0/NIL	16/3
round 2	$-3/3$	6/1	0/NIL	$-15/1$
round 3	$-3/3$	$-8/4$	0/NIL	$-15/1$

Bellman-Ford algorithm — an example with negative cycle:

- Input graph G:



- `Bellman-Ford`$(G, 3)$:



- `Bellman-Ford`$(G, 3)$ trace:
  Edge order: $(4, 2), (2, 1), (1, 4), (3, 1), (3, 4)$

v $v.dist/v.predec$	1 ∞/NIL	2 ∞/NIL	3 $0/\text{NIL}$	4 ∞/NIL
`round 1`	$-3/3$	∞/NIL	$0/\text{NIL}$	$16/\text{NIL}$
`round 2`	$-3/3$	$23/4$	$0/\text{NIL}$	$-15/1$
`round 3`	$-3/3$	$-8/4$	$0/\text{NIL}$	$-15/1$
`checkup round`	$-4/2$	$-8/4$	$0/\text{NIL}$	$-15/1$

All-Pairs Shortest-Path Problem:

- General case (weights can be negative but no negative cycle);
- Output: shortest path between every pair u, v of vertices.
- If the graph has no negative weights: run Dijsktra n times with each vertex as a source.
- Runtime:
  - $O(n(n+m)\log(n))$.
  - ($O(n^2\log(n)+nm)$ with the improved implementation that uses a Fibonacci-heap.)
- If the graph has negative weights: run Bellman-Ford n times with each vertex as a source.
  - $O(n^2m)$.
- Can we do better?

Floyd-Warshall's algorithm for All-Pairs-Shortest path:

- General case (weights can be negative but no negative cycle);
- Idea: Use dynamic programming.
  Define $d[i, j, k]$ to be the length of shortest path from i to j for which all intermediate vertices are in $\{1, \ldots, k\}$, for every $1 \leq i, j \leq n$ and $k \leq n$.
- When $k = 0 \Longrightarrow$ no intermediate vertex, so: $d[i, j, 0] = w(i, j)$ if the edge (i, j) exists, ∞ otherwise.
- For general $k \geq 1$:
  - If the path does not contain k, inter. vertices only from $\{1, \ldots, k-1\}$: $d[i, j, k-1]$.
  - If the path contains k, it has two parts: one goes from i to k with intermediate only from $\{1, \ldots, k-1\}$, followed by a path from k to j using only from $\{1, \ldots, k-1\}$: $d[i, k, k-1] + d[k, j, k-1]$.
- Recurrence for $k \geq 1$:

$$d[i, j, k] = \min \begin{cases} d[i, j, k-1] & \text{path doesn't use vertex } k \\ d[i, k, k-1] + d[k, j, k-1] & \text{path does use vertex } k \end{cases}$$

- We compute the table bottom-up, starting from smaller values of k to larger values.

Floyd-Warshall's algorithm:

- procedure Floyd-Warshall(G)

```
for (i from 1 to n) do
   for (j from 1 to n) do
      if ((i,j) is an edge) do
         d[i,j,0] ← w(i,j)
         b[i,j] ← 0          ** b[i,j] keeps the break point vertex
      else
         d[i,j,0] ← ∞
         b[i,j] ← ⊥          ** ⊥ stands for "don't know"
for (k from 1 to n) do
   for (i from 1 to n) do
      for (j from 1 to n) do
         if (d[i,j,k-1] > d[i,k,k-1] + d[k,j,k-1]) then
            d[i,j,k] ← d[i,k,k-1] + d[k,j,k-1]
            b[i,j] ← k
         else
            d[i,j,k] ← d[i,j,k-1]
```

- Running time: $\Theta(n^3)$

Floyd-Warshall's algorithm:

- To print the actual path between i and j

```
procedure Print-FW(i, j)
if (b[i, j] = 0) then
   Print ‘‘(i, j)’’
else if (b[i, j] = ⊥) then
   Print ‘‘no path’’
else
   Print-FW(i, b[i, j])    ** First print the path i → b[i, j]
   Print-FW(b[i, j], j)    ** then print the path b[i, j] → j
```

- Correctness follows from the correctness of the Floyd-Warshall algorithm (the fact that the matrix b doesn't contain loops).
- Running time:
  - The recurrence relation:
    If the shortest-path between i and j is of length 1 then $T(1) = O(1)$.
    If the shortest-path between i and j is of length $k > 1$ then we invoke 2 recursions: to print the path from i to $b[i, j]$ and then to print the path from $b[i, j]$ to k. Denoting the length of the path from i to $b[i, j]$ as ℓ we have that
    $$T(k) = O(1) + O(\ell) + O(k - \ell)$$
  - You can guess-and-test and see this solves to $\Theta(k)$.
    As $k \leq n - 1$ the runtime is $O(n)$.
  - Runtime of printing all path of all $\binom{n}{2}$ possible i and js: $O(n^3)$.