Lecture Outline 3
Topics in Graph Algorithms (CSCI5320-22S)

CAI Leizhen
Department of Computer Science and Engineering
The Chinese University of Hong Kong
`lcai@cse.cuhk.edu.hk`

January 21, 2022

All-pairs Shortest Paths

Keywords: distance, negative cycle, single-source shortest paths: Dijkstra's algorithm, and Bellman-Ford algorithm, all-pairs shortest paths: Johnson's algorithm, Floyd-Warshall algorithm, matrix "multiplication", dynamic programming, and edge-disjoint paths.

1. Let $G = (V, E; w)$ be a weighted graph with $w : E \to R$. We assume that G is a digraph (i.e., a directed graph) when we study shortest paths.

   Question 3.1: How do we convert an undirected graph into a digraph?

   Question 3.2: Why do we allow negative weights?

2. The *distance* from u to v, denoted $d(u, v)$, is the length of a shortest path from u to v. The *length* of a path P is the number of edges in P for unweighted graphs, and the sum of weights of edges in P for weighted graphs.

   Question 3.3: Does a path allow repetition of vertices/edges?

3. **Review of single-source shortest paths**

   Dijkstra's algorithm: for weighted graph G without negative edges ($O(m + n \log n)$ time using Fibonacci heap).

   Dijkstra's algorithm computes shortest paths from the source vertex s to all vertices reachable from s in a way similar to BFS: the algorithm maintains the set S of visited vertices, and visits the next vertex by selecting an edge $v'v$, where $v' \in S$ and $v \in V - S$, such that $v'v$ is on a shortest (s, v)-path. For this purpose, the algorithm assigns each vertex v a label $d[v]$ which is an upper bound of $d(s, v)$ and decreased to $d(s, v)$ once v is visited. We also maintain a rooted tree T on S such that for each vertex $v \in S$, the unique (s, v)-path in T is a shortest (s, v)-path in G.

   The key step in Dijkstra's algorithm is to choose a vertex u outside S with minimum $d[u]$, and for every edge uv change $d[v]$ to $\min\{d[v], d[u] + w(uv)\}$.

Question 3.4: How can Dijkstra's algorithm fail to work for graphs with negative edges?

Bellman-Ford algorithm: for general weighted graphs ($O(mn)$ time).

Repeats the following step $n-1$ times: for every edge uv do **Refine**(uv).

Refine(uv): if $d[v] > d[u] + w(uv)$ then $d[v] \leftarrow d[u] + w(uv)$.

Question 3.5: How do we know that indeed $d[v] = d(s, v)$ after $n-1$ iterations?

Question 3.6: Can we use the ideas in Bellman-Ford algorithm to modify Dijkstra's algorithm to make it work for general weighted graphs?

4. **Negative cycle**: a cycle whose total weight is negative. Distances may not be well defined when a graph contains negative cycles, and Bellman-Ford algorithm can be used to detect a negative cycle in a graph.

5. **All-pairs shortest paths**: Run single-source algorithm for every vertex.

   Use Bellman-Ford ($O(mn^2)$ time) and use Dijkstra's algorithm when no negative edges ($O(mn + n^2 \log n)$ time).

6. **Johnson's algorithm**: reweighting method in $O(mn + n^2 \log n)$ time.

   Change the weight w of $G = (V, E; w)$ to a new weight w' such that
   (a) shortest paths in G remain unchanged, and
   (b) for every edge $e \in E$, $w'(e) \geq 0$.

   Let $h : V \to R$ be an arbitrary function, and for each edge $uv \in E$, set $w'(uv) = w(uv) + h(u) - h(v)$. Let $G' = (V, E; w')$.

   Lemma. P is a shortest path in G' iff it is a shortest path in G, and G' has a negative cycle iff G does.

   Therefore any function h satisfies (a), and we use the following method to find an h that also satisfies (b).

   Construct G^* from G by adding a vertex s and edges sv with weight 0 for every vertex v of G. Set $h(v) = d_{G^*}(s, v)$ for all $v \in V$ (note that G^* has a negative cycle iff G does).

   By the triangle inequality of distance, we have

   $$d_{G^*}(s, v) \leq d_{G^*}(s, u) + d_{G^*}(u, v) \leq d_{G^*}(s, u) + w(uv),$$

   implying $w'(uv) \geq 0$.

   Johnson's algorithm

   Step 1. Run Bellman-Ford algorithm on G^*.

   Step 2. If G^* has no negative cycle, then set $h(v) = d_{G^*}(s, v)$ for all $v \in V$ and set $w'(uv) = w(uv) + h(u) - h(v)$ for every edge $uv \in E$.

   Step 3. For every vertex of G, run Dijkstra's algorithm on $G' = (V, E; w')$.

   Step 4. For every pair u, v of vertices, compute $d_G(u, v)$ from $d_{G^*}(u, v)$.

Question 3.7: Can you find a different method to obtain a required h?

7. **Floyd-Warshall algorithm**: dynamic programming in $O(n^3)$ time.

   Let $V = \{1, \dots, n\}$, and $w(ij)$ the weight of edge ij. Define $d_{i,j}^{(k)}$ to be the weight of a shortest (i,j)-path with internal vertices in $\{1, \dots, k\}$. Then $d(i,j) = d_{i,j}^{(n)}$.

   Recurrence: $d_{i,j}^{(0)} = w(ij)$ and $d_{i,j}^{(k)} = \min\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\}$.

   We can easily compute $d_{i,j}^{(n)}$ by dynamic programming in $O(n^3)$ time.

   Question 3.8: How can we compute pairwise shortest paths instead of distances?

8. **Matrix "multiplication"**: $O(n^3 \log n)$ time.

   Let $V = \{1, \dots, n\}$, and let $W = [w_{ij}]_{n \times n}$ be the weighted adjacency matrix of G, i.e.,

   $$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ w(ij) & \text{if } i \neq j \text{ and } ij \in E \\ \infty & \text{otherwise.} \end{cases}$$

   Let $l_{ij}^{(t)}$ be the weight of a shortest (i,j)-path that uses at most t edges. Then we have the following recurrence: $l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$ and $l_{ij}^{(t)} = \min_{1 \le k \le n}\{l_{ik}^{(t-1)} + w(kj)\}$.

   We can easily compute distance $d(i,j) = l_{ij}^{(n-1)}$ by dynamic programming in $O(n^4)$ time, and we can speed it up by matrix "multiplication".

   For two $n \times n$ matrices A and B, define $A \star B$ to be matrix C with $c_{ij} = \min_k\{a_{ik} + b_{kj}\}$. Compare this with the normal matrix multiplication $c_{ij} = \sum_k a_{ik} \times b_{kj}$, we see the correspondences "$\min$" $\leftrightarrow$ "$\sum$" and "$+$" $\leftrightarrow$ "$\times$".

   Let $L^{(t)} = [l_{ij}^{(t)}]_{n \times n}$. Then

   $$L^{(0)} = [l_{ij}^{(0)}], L^{(1)} = L^{(0)} \star W = W, L^{(2)} = L^{(1)} \star W = W^2, \dots, L^{(n-1)} = L^{(n-2)} \star W = W^{n-1},$$

   where $W^i = W \star \dots \star W$ with i W's.

   We can compute W^{n-1} by repeated squaring $W, W^2, W^4, W^8, \dots$ $O(\log n)$ times to obtain W^{n-1}. Note that, when G has no negative cycle, W^{n-1} converges and doesn't change after that.

9. **Edge-disjoint paths** (Suurballe 1974) $O(m + n \log n)$ time.

   Task: For a pair (s,t) of vertices in a weighted digraph $G = (V, E; w)$, find a pair of edge-disjoint (s,t)-paths of minimum total length.

   For simplicity, we assume $w(e) \ge 0$ and G is antisymmetric, i.e., $uv \in E$ implies that $vu \notin E$. We use the reweighting idea to solve our problem.

   Step 1 . Use Dijkstra's algorithm to find a shortest-path tree T with root s, and compute $d(s,v)$ for every vertex v.

Step 2 . For every edge uv, define $w'(uv) = (d(s,u) + w(uv)) - d(s,v)$.

Step 3 . Construct from G a new graph G' by reversing directions of all edges in the (s,t)-path P in T.

Step 4 . Run Dijkstra's algorithm on G' with new weight w' to find a shortest (s,t)-path P' in G'.

Step 5 . Take the union of edges in P and P', discarding every edge in P whose reversal appears in P', and group them into two (s,t)-paths.

Remarks. The new weight $w'(e) \geq 0$ by the triangle inequality of distances. Step 5 is guaranteed by augmenting paths and min-cut max-flow theorem for network flows.

10. **Tree 1-spanner** (Cai and Corneil 1995)

Task: For a weighted graph $G = (V, E; w)$ with $w : E \to R^+$, determine whether G contains a *tree 1-spanner*, i.e., a spanning tree T such that for every pair $\{u, v\}$ of vertices, $d_G(u,v) = d_T(u,v)$.

Fact 1: If G contains a tree 1-spanner T, then T is the unique minimum spanning tree of G.

Fact 2: T is a tree 1-spanner iff for every non-tree edge xy, $w(xy) \geq d_T(x,y)$.

Algorithm: Find a minimum spanning tree T of G and then verify that T preserves pairwise distances of G.

For verification, it takes $O(n^3)$ time if we compute all-pairs distances in T and G. We can reduce the verification time to $O(m+n)$ as follows:

Step 1. Arbitrarily choose a vertex r of T as the root.

Step 2. For every vertex v, compute $d_T(r,v)$.

Step 3. For every non-tree edge xy, check if $w(xy) \geq d_T(x,y)$.

Step 2 takes $O(n)$ time by BFS. For Step 3, we note that for any two vertices x and y,

$$d_T(x,y) = d_T(r,x) + d_T(r,y) - 2d_T(r, LCA(x,y)),$$

where $LCA(x,y)$ denotes the least common ancestor of x and y. We can use an algorithm of Harel and Tarjan to compute $LCA(x,y)$ in $O(1)$ time. It follows that Step 3 takes $O(m+n)$ time.

11. **References**

[1] Cormen et. al., Introduction to Algorithms (2nd Ed) Chapter 25, MIT Press, 2001.

[2] Suurballe and Tarjan, A quick method for finding shortest pairs of disjoint paths, Networks, Vol. 14 325-336, 1984.