Shortest Paths

Gleb Evstropov
evstropovgleb@gmail.com

March 13, 2023

1 Notations

- A graph is typically denoted as G.
- $V(G)$ denotes the set of vertices (or nodes) of graph G.
- $E(G)$ denotes the set of edges (or arcs, typically assumes direction) of graph G.
- $u, v, w, s, t \in V(G)$ are common names to denote a vertex.
- Edges are typically denoted as e or uv. $uv \in E(G)$ means the edge between vertices u and v that belongs to the set of all edges of graph G.
- $c(uv)$, $w(uv)$, $l(uv)$ (or $c(e)$, $w(e)$, $l(e)$) are the common ways to refer to some additive measure, associated with the edges. They mean cost, weight and length, but basically mean the same thing.
- P is the common way to denote a path. $P_G(s, t)$ usually means a path from vertex s to vertex t in graph G. Sometimes it can mean the set of all such paths.
- C is the common way to denote a cycle. $v \in C$ means vertex v belongs to cycle C, $uv \in C$ means edge $e = uv$ belongs to cycle C. Same with paths. Sometimes you can see $v \in V(C)$ or $uv \in E(C)$.
- $w(P)$ denotes the weight of path P. By default it is assumed to be equal to the sum of individual weights of all edges forming path P, unless some other definition is explicitly mentioned. $w(P) = sum_{uv \in P} w(uv)$.
- $\rho(u, v)$ means the distance between vertices u and v, i.e. the minimum possible $w(P(u, v))$. If it is not clear which exactly graph is used to refer to the distance between u and v, notation $\rho_G(u, v)$ can be used.
- In competitive programming n is usually used for $|V(G)|$ and m is used for $|E(G)|$.
- $N(v)$ denotes the set of all neighbours of v. Formally, $N(v) = \{u : vu \in E(G)\}$.

2 BFS (Breadth-first search)

BFS is used to find the shortest path from one vertex s of **unweighted** graph G to all other vertices. Has linear time and space complexity.

Below are the key steps of the algorithm.

1. Initialize an array $d(v)$ to store the distances from s to v.
2. Initialize an array $vis(v)$ to mark vertices that were already visited. To save on variables, $d(v) = -1$ can be used to mark vertices yet unseen instead of $vis(v)$.
3. Initialize an array $from(v)$ if we need to be able to restore the shortest path.
4. Initialize a queue q.
5. Set $d(s) = 0$ and put s into the queue.
6. While q is not empty, extract one vertex v from the queue. For all $u \in N(v)$ check if $vis(u)$ is false. If so, set $vis(u)$ to true, set $d(u)$ to $d(v) + 1$, set $from(u) = v$ and push u into the queue.
7. At the end, all vertices u reachable from s will have $vis(u) = true$ and $d(u) = \rho(s, u)$.

BFS is often used to find the shortest sequence of actions required to achieve some state. That applies to problems where the total number of states is not large. Examples.

- Get from cell (x, y) to cell (x', y') in a table with some cells blocked. State is the current cell.
- Get from cell (x, y) to cell (x', y') in a table with some cells blocked. You are allowed to pass through a blocked cell no more than once. State is (a, b, c) where (a, b) encodes the current cell and c denotes the remaining number of times we can pass through a blocked cell.
- Get from cell (x, y) to cell (x', y') in a table with some cells blocked. You are allowed to pass through a blocked cell no more than once for each x consecutive moves. State can be (a, b, t) where (a, b) denotes the current cell and t denotes the number of moves since the latest passage through a blocked cell.

3 Dijkstra

Dijkstra is used to find the shortest path from one vertex s to all other vertices of some weighted graph G. Condition $w(uv) \geq 0$ must hold for all $uv \in E(G)$.

The two most common implementations of this algorithm have time complexity $O(n^2 + m)$ and $O(m \log n)$. The first is known for having a small constant factor. It can run for n up to $2 \cdot 10^4$ in just one second if implemented properly.

The key algorithm steps are as follows.

1. Initialize an array d to store the shortest path from s to each vertex v.
2. Set $d(v) = \inf$ for all $v \neq s$. Set $d(s) = 0$.
3. Initialize an array $mark$ to store the marks on whether v was already processed or not.
4. Initialize an array $from$ if you need to restore the shortest paths afterward.
5. Find vertex v that has $mark(v)$ not set and the value $d(v)$ is minimum possible.
6. Set $mark(v)$ to true. Now you know that $\rho(s, v) = d(v)$, so $d(v)$ will no longer change.
7. Consider all edges $vu \in E(G)$. Update $d(u) = \min(d(u), d(v) + w(vu))$. If you want to compute $from(v)$ you should use if clause instead of min function. If $d(v) + w(vu) < d(u)$ update $d(u) = d(v) + w(vu)$ and $from(u) = v$.
8. Go to step 5 until there are no unmarked vertices left. If the graph can contain vertices that can't be reached from s, you should also stop in case $d(v) = \inf$ at step 5.

Tricks to speed up the algorithm in some real-world cases.

1. Bidirectional Dijkstra. You can use this trick if you only need to find a shortest distance between a particular pair of vertices s and t (that is a very common case). Simultaneously run the algorithm from s and t until you find first vertex w such that both $\rho(s, w)$ and $\rho(w, t)$ are already computed.
2. A-star algorithm. Variation of Dijkstra. Suppose there is a function $\rho^*(u, v)$ such that $\rho^*(u, v) \geq 0$, $\rho^*(u, v) \leq \rho(u, v)$ and $\rho^*(u, v) \leq \rho^*(u, w) + \rho^*(w, v)$ for any three u, v and w. You can run Dijkstra's algorithm that always processes a vertex v with the smallest value of $d(v) + \rho * (v, t)$ instead of the smallest value of $d(v)$.

4 Ford-Bellman

Ford-Bellman is used to find the shortest paths from one vertex s to all other vertices of some weighted graph G. This algorithm works if there are no cycles of negative cost.

This algorithm time complexity is $O(nm)$ and space complexity $O(n)$.

Below are the key steps of the algorithm.

1. Initialize an array d to store the current distances. Set $d(v) = \inf$ for all $v \neq s$ and $d(s) = 0$.
2. Initialize an array $from$ if you want to restore the shortest paths afterward.

3. For each edge uv try to update $d(v)$ with $d(u)+w(uv)$. Note that for a bi-directional graph that means trying to update $d(u)$ with $d(v) + w(vu)$ as well.

4. Repeat the previous step n times.

5 Cycle of negative weight

Ford-Fulkerson algorithm can be modified to check whether the graph has a cycle of negative weight (and even restore it).

1. If there are no cycles of negative weight the algorithm computes the correct $d(v) = \rho(s, v)$ after n steps.

2. Run one more step. If at least one value of $d(v)$ changes, it means there is a negative-weight cycle in the graph.

3. Traverse the path of $from$ values starting from vertex v and you will restore the cycle.