Homework 8

Graded

Student

Nick Zhu

Total Points

100 / 100 pts

Question 1

Problem 1

35 / 35 pts

✔ **– 0 pts** Correct

Question 2

Problem 2

35 / 35 pts

✔ **– 0 pts** Correct

Question 3

Problem 3

30 / 30 pts

3.1 **(a)** **9** / 9 pts

✔ **– 0 pts** Correct

3.2 **(b)** **7** / 7 pts

✔ **– 0 pts** Correct

3.3 **(c)** **7** / 7 pts

✔ **– 0 pts** Correct

3.4 **(d)** **7** / 7 pts

✔ **– 0 pts** Correct

Question assigned to the following page: 1

Problem 1 (Fast SSSP, 35 pts)

In class we observed that BFS solves single source shortest path on *unweighted* graphs in time $O(|V| + |E|)$. In this problem we will show how to solve SSSP in the same time complexity when G is a weighted graph with integer weights in the range $[1, c]$, where c is a constant. In SSSP for weighted graphs, we want to find the path that has the shortest path weight/distance, which is the sum of the weights of all the edges in the path. We can view SSSP for unweighted graphs as a special case where each edge has the same weight of 1.

Specifically, let c be a constant. Consider weighted graphs $G = (V, E, w)$ such that the weight $w(u, v)$ is an integer between 1 and c for each edge $(u, v) \in E$.

Design an algorithm to solve SSSP in time $O(|V| + |E|)$ on such graphs. Your algorithm should output `parent` encoding the shortest paths, and `dist` encoding the shortest path weight/distance of each node, as we saw in the BFS example in class.

Your algorithm should use BFS, as we learned in class, as a subroutine. Specifically, you should not modify the original BFS algorithm to devise your solution (think about accessing BFS in a black-box manner). Be sure to justify the correctness and runtime of your proposed algorithm.

Solution

Construct the Unweighted Graph G'

We want to eliminate weights:

1. Start with $G = (V, E)$. For each vertex $v \in V$, *copy* v into our new vertex set V'. Initially, $V' = \{\, v : v \in V \}$ (i.e., each "original" vertex is already in G').
2. For each edge $(u, v) \in E$ of weight $w(u, v) \in [1, c]$:
   (a) Create $w(u, v) - 1$ new "dummy" vertices $x_1, x_2, \ldots, x_{w(u,v)-1}$ (all distinct from each other and from all existing vertices in V').
   (b) In G', add an **unweighted** path of length $w(u, v)$ by linking:
   $$u \;-\; x_1 \;-\; x_2 \;-\; \cdots \;-\; x_{w(u,v)-1} \;-\; v.$$
   Thus, we replace a single weighted edge of cost $w(u, v)$ with a chain of $w(u, v)$ edges, each of which has weight 1 in G'.
   (c) Add all these new dummy vertices into V'. Add all the edges in the above chain into E'.

After doing this for *every* edge in E, we obtain an unweighted graph $G' = (V', E')$. By construction, every edge in G' has weight 1. Notice that because $w(u, v) \leq c$ for each original edge and c is a constant, we create at most $c - 1$ new vertices per edge, and each edge in E spawns at most c edges in E'. Therefore, the sizes of V' and E' satisfy

$$|V'| \;\leq\; |V| \;+\; (c-1)|E| \;=\; \mathcal{O}(|V| + |E|), \quad |E'| \;\leq\; c\,|E| \;=\; \mathcal{O}(|E|).$$

Consequently, running BFS on G' will cost

$$\mathcal{O}(|V'| + |E'|) \;=\; \mathcal{O}(|V| + |E|).$$

Question assigned to the following page: 1

Running `BFS` and Recovering Distances & Parents

Let us denote the standard BFS subroutine from class as $\texttt{BFS}(G', s)$. We proceed as follows:

1. Run $\texttt{BFS}(G', s)$ once, using the algorithm we learnt from the clss (i.e., the one with a queue, $\texttt{dist}[v]$, $\texttt{parent}[v]$, etc.).

2. For each vertex (original or dummy) in G', we get a BFS distance $\texttt{dist}'[\,\cdot\,]$. For every $v \in V$ (i.e., the "real" vertices in the original G), define

$$\texttt{dist}[v] \;=\; \texttt{dist}'[v],$$

   which is precisely the fewest number of edges on a path from s to v in G'. By construction, this number is also the sum of the (original) weights in G. In other words,

$$\texttt{dist}[v] \;=\; \min_{\pi: s \to v} \sum_{\text{(x,y) in } \pi} w(x, y) \quad \text{in the original graph } G.$$

3. To build the $\texttt{parent}[v]$ pointers for the original graph G, we observe the BFS tree in G'. Specifically, for each real vertex $v \in V$ (other than s), we look at the path in G' from s to v. If the predecessor of v in this path is a dummy vertex x_k, we follow its parent pointer(s) in G' until we reach another real vertex $u \in V$. We then define

$$\texttt{parent}[v] \;=\; u \quad \text{(the real vertex preceding } v \text{ on the shortest path)}.$$

   This way, we effectively "skip" dummy vertices (since they were only introduced to split a weighted edge into unit edges).

Hence, at the end of running one BFS on G', we have:

$$\texttt{dist}[v] \quad \text{and} \quad \texttt{parent}[v] \quad \text{for each real vertex } v \in V.$$

Correctness

- **Exactness of distances.** Because we subdivided each weighted edge of cost w into a chain of w unit-cost edges, the total BFS distance in G' from s to any vertex v is exactly the minimal sum of weights from s to v in the original graph G. Thus, our $\texttt{dist}[v]$ is the correct SSSP distance.

- **Parent pointers yield valid shortest paths in G.** By ignoring the dummy vertices and linking each vertex v to its nearest real predecessor u in G', we obtain a valid path in G. The sum of edge weights on that path is precisely $\texttt{dist}[v]$.

Running Time Analysis

Constructing G' from G requires iterating over all edges in E. Each edge of weight at most c spawns at most $(c-1)$ new vertices and c new edges. Since c is a constant,

$$|V'| = \mathcal{O}(|V| + |E|), \quad |E'| = \mathcal{O}(|E|).$$

Therefore, a single run of $\texttt{BFS}(G', s)$ costs

$$\mathcal{O}(|V'| + |E'|) \;=\; \mathcal{O}(|V| + |E|).$$

Question assigned to the following page: 1

After BFS, we do a quick pass to assign `parent`$[v]$ for real vertices v, which is also $\mathcal{O}(|V| + |E|)$ in total.

Hence, the overall time complexity of our algorithm is

$$\mathcal{O}(|V| + |E|).$$

Question assigned to the following page: 2

Problem 2 (Shortest Separable Path, 35 pts)

Let P be a program which, given as input a directed graph $G = (V, E)$ as well as two vertices $s, t \in V$, outputs the shortest path between them. Let $T(P)$ be the run-time of this program. Imagine that P is already highly optimized (say for running on a GPU) such that $T(P)$ runs significantly faster than your own shortest path algorithm. As such, we will use P to solve the following problem.

We are given a directed graph G where each of its edges is colored either red or blue. We want to find the shortest path from some vertex s to some other vertex t, with the requirement that our path must be *separable*. In order to be separable, the path we output must consist of first some number (possibly 0) of red edges followed by some number (possibly 0) of blue edges. In other words, once you use a blue edge, all following edges must be blue.

Design an algorithm to find the shortest separable path from s to t. Your algorithm should invoke P *only once* on a carefully constructed graph, and it should run in time $O(|V| + |E|) + T(P)$. You should *not* explore the graph yourself (i.e., do not implement BFS); use the optimized program P. Justify the correctness and runtime of your proposed algorithm.

(Hint: Consider the subgraph formed by restricting the edges to only the red ones. Then, consider the subgraph with only the blue edges. Notice that you are to spend some number of steps in the first subgraph and then move to the second subgraph and stay there. Can you combine the two graphs somehow?)

Solution

Constructing the Graph G'

We want to "simulate" the process of taking red edges and then, at some point, switching to blue edges by creating two copies of every vertex. We create two "states" for each vertex $v \in V$:

- v_R: representing that we are at vertex v while we are still allowed (or choosing) to use red edges.
- v_B: representing that we have switched and are now only using blue edges.

We now define G' as follows:

1. **Vertices.**
$$V' = \{v_R,\, v_B : v \in V\} \cup \{s', t'\}$$
where s' and t' are new vertices that we will use as the source and target, respectively.

2. **Edges.** We add edges to G' according to the following rules:

   (a) For each red edge $(u, v) \in E$ in G, add an edge
   $$(u_R, v_R) \quad \text{with weight 1.}$$
   This ensures that while remaining in the red segment the only allowable moves follow red edges.

Question assigned to the following page: 2

(b) For each blue edge $(u, v) \in E$ in G, add an edge

$$(u_B, v_B) \quad \text{with weight 1.}$$

This models the moves possible after the switch to blue.

(c) For each vertex $v \in V$, add a *switch edge*

$$(v_R, v_B) \quad \text{with weight 0.}$$

This edge allows a transition from taking red edges to taking blue edges at any vertex.

(d) To allow the possibility of starting with blue edges immediately (i.e., a separable path that uses 0 red edges), add an edge from the new source s' to s_B with weight 0. Similarly, add an edge from s' to s_R with weight 0 so that we can also start by taking red edges.

(e) Finally, to "collect" the end of the path in a single target, add edges from both copies of t to t' with weight 0:

$$(t_R, t') \quad \text{and} \quad (t_B, t').$$

Thus, G' has $2|V|+2$ vertices and at most $O(|V|+|E|)$ edges. Notice that every edge in G' has weight either 0 or 1. By construction, any valid path in G' from s' to t' corresponds to a separable path in G, as we now explain.

Correctness

Any path in G' from s' to t' must start at s' and immediately enter either s_R or s_B. There are three cases:

1. **All-blue path:** The path goes from s' to s_B (using the zero-cost edge) and then only follows blue edges (i.e., it remains in the B-layer) until reaching t_B and then the edge to t'. This directly corresponds to a separable path in G with 0 red edges.

2. **All-red path:** The path goes from s' to s_R and then takes red edges until reaching t_R, followed by the edge to t'. Since blue edges are not used, this path is separable (with 0 blue edges).

3. **Red then blue path:** The path starts at s' then goes to s_R, follows a sequence of red edges (remaining in the R-layer), and at some vertex v uses the zero-cost switch edge (v_R, v_B). After that, the path continues from v_B along blue edges until reaching t_B and finally goes to t'. This exactly simulates taking a sequence of red edges and then switching to blue edges.

In each case, the number of edges with weight 1 in the path equals the number of colored edges in the corresponding path in G. Consequently, the shortest path in G' (as computed by P) corresponds precisely to the shortest separable path in G.

Algorithm and Runtime

The final algorithm is as follows:

1. **Construct G':**

   - For every $v \in V$ create vertices v_R and v_B.

Question assigned to the following page: 2

- For each red edge $(u, v) \in E$, add the edge (u_R, v_R) with weight 1.
- For each blue edge $(u, v) \in E$, add the edge (u_B, v_B) with weight 1.
- For each $v \in V$, add the edge (v_R, v_B) with weight 0.
- Add new source s' and target t', add edges (s', s_R) and (s', s_B) with weight 0.
- Add edges (t_R, t') and (t_B, t') with weight 0.

This construction takes $O(|V| + |E|)$ time.

2. **Invoke the Shortest-Path Program:** Run P once on the graph G' with source s' and target t'. Let the returned path π' be the shortest path from s' to t' in G'. By our construction, π' encodes a separable path from s to t in G.

3. **Interpretation:** Remove the auxiliary vertices s' and t' and the state labels from the intermediate nodes. The resulting sequence of vertices in V is the shortest separable path from s to t in the original graph.

The overall running time is the sum of the time to construct G' (which is $O(|V| + |E|)$) and the time to run P on G' (i.e., $T(P)$). Thus, the total runtime is

$$O(|V| + |E|) + T(P).$$

Questions assigned to the following page: 3.1, 3.2, and 3.3

Problem 3 (Connected Components, 30 pts)

A *connected component* of an undirected graph $G = (V, E)$ is defined as a subset of vertices $S \subseteq V$ that is connected (under E) and maximal. By "maximal", it means that you cannot add any additional vertices to S while it remains connected. Put rigorously, there does not exist any strict superset $S' \supsetneq S$ that is connected. In this problem, we will study a few properties with regard to connected components, which eventually allow us to prove a useful graph theory statement.

(a) For an undirected graph with $|V| = n \geq 1$ vertices, what is the maximum number of connected components it can have? What about the minimum? Also, categorize which graphs would give you the corresponding maximum and minimum.

(b) Show that if you add an edge $\{u, v\}$, where u, v belongs to different connected components, the number of connected components in the graph will decrease by 1.

(c) Show that if you add an edge $\{u, v\}$, where u, v belongs to the same connected component, the resulting graph will contain a cycle.

(d) Combine the previous parts to show that an undirected graph with $|V|$ edges must contain a cycle. (*Hint: Consider adding the $|V|$ edges into the graph one after one.*)

Solution

(a)

Maximum Number: The maximum number of connected components occurs when the graph has no edges. In this case, every vertex is isolated, and each vertex forms its own connected component. Thus, the maximum number is

$$\max = n.$$

Minimum Number: The minimum number of connected components is achieved when the entire graph is connected. In that case, every vertex is reachable from any other vertex, so there is only one connected component.

$$\min = 1.$$

(b)

Suppose we have an edge $\{u, v\}$ that we add to G, where u belongs to a connected component S_1 and v belongs to a different connected component S_2. By the definition of connected components, there is no path connecting any vertex in S_1 to any vertex in S_2 in the original graph.

When we add the edge $\{u, v\}$, it provides a direct connection between S_1 and S_2. Therefore, any vertex reachable from u is now reachable from v and vice versa. In effect, S_1 and S_2 merge into a single connected component, and the overall number of components decreases by exactly 1.

(c)

Now, consider adding an edge $\{u, v\}$ where both u and v already belong to the same connected component. Since the graph is connected, there is already at least one path between u and v in the original graph. Denote this existing path by P_{uv}.

Questions assigned to the following page: 3.4 and 3.3

When we add the edge $\{u, v\}$, we now have two distinct paths between u and v: the original path P_{uv} and the new direct edge $\{u, v\}$. The existence of these two paths implies that the edge $\{u, v\}$ completes a cycle. Formally, the cycle is formed by concatenating the path P_{uv} with the edge $\{u, v\}$ to return to the starting vertex.

(d)

We now use the properties from parts (a)–(c) to establish that an undirected graph with $|V|$ edges must contain a cycle.

Argument:

1. Start with an empty graph on n vertices, which has n connected components (each vertex isolated).
2. Add edges one at a time. Every time an edge connects vertices from *different* connected components, it reduces the number of connected components by 1 (by part (b)). To obtain a connected (and acyclic) graph, which is a tree, one must add exactly $n - 1$ edges. In a tree, there are no cycles by definition.
3. Now, if we add one more edge (making a total of n edges), the graph is already connected. Therefore, by necessity, this extra edge must connect two vertices that are already in the same connected component. But by part (c), adding an edge within the same connected component creates a cycle.

Thus, an undirected graph with $|V|$ edges cannot remain acyclic; it must contain at least one cycle.