Problem Set 4 Solution

All parts are due Tuesday, November 15, 2016 at 11:59PM.

Name: Jisoo Min
Collaborators:

Part A

Problem 4-1. [30 points] Runaround

(a) [3 points] Solution: First assume that the destination base has a cost of zero. Then for all edges \((u, v) \in E\), set the edge cost \(\delta(u, v)\) as the cost of node \(v\).

(b) [10 points] Solution:
Summary: In \(O(VE)\), we need to determine if a mission from Base A to Base B can be completed with a given initial armor integrity value \(C\).
Algorithm: Use a modified Bellman-Ford algorithm to detect positive weight cycles.

- modification 1: Initialize Base A with an integrity value of \(C\) instead of zero and all other bases with \(-\infty\) instead of \(\infty\).
- modification 2: Reverse relax (i.e. update to maximum) \(v.d = u.d + w(u, v)\) when \(v.d < u.d + w(u, v)\) and \(u.d + w(u, v) \geq 0\) (notation adopted from lecture).

If positive weight cycle is detected after the modified Bellman-Ford algorithm:
We conclude that Speedy can infinitely repair itself and happily get to Base B.
If positive weight cycle is not detected after the modified Bellman-Ford algorithm:
(1) Speedy can complete the mission if the integrity value for Base B is not \(-\infty\).
(2) Speedy cannot complete the mission if the integrity value for Base B is \(-\infty\).
Correctness: Initialization with \(-\infty\) and reverse relaxation enable positive weight cycle detection. Speedy’s armor integrity never drops below zero on a valid path because only non-negative values are valid updates.
Run Time: \(O(VE)\) is achieved with the modified Bellman-Ford algorithm; \(O(V)\) initialization and \(O(V)\) iterations of \(O(E)\) relaxations.

(c) [5 points] Solution: Speedy has not returned, so we know there is no path for Speedy to return to home base. If he is not calling for help, Speedy must be running around in a zero-weight cycle that does not require his armor integrity to dip below zero.
(d) [12 points] Solution:
Summary: Knowing that Speedy started at Base $B$, we need to find out where he could be stuck in. In other words, we need to find all bases involved in zero weight cycles.
Algorithm: First we complete SSSP (single source shortest path) modified Bellman-Ford algorithm (discussed above) with Base $B$ as the source. All edges not involved in positive weight cycle should be fully relaxed.
We then iterate through all edges once more to narrow down options;
- if $v.d = -\infty$, then delete node $v$ (unreachable node)
- elif $u.d + w(u, v) < 0$, then delete edge $(u, v)$ (See Figure 1)
- elif $v.d \neq u.d + w(u, v)$, then delete node $u$ (See Figure 1)

Figure 1: Edge cases (cycle within another cycle)

This process cuts out negative weight cycles and unreachable nodes. Finally, we run DFS to find out where the cycles are. The cycles can only be zero weight cycles. Correctness: The key is that there is no reachable positive weight cycle. If there was one, Speedy would have returned home. When we narrow down options, we handle edge cases of having a subloop within another loop (See Figure 1). So by the time we run DFS, we have already ruled out both positive and negative weight cycles.
Run Time: Bellman-Ford takes $O(VE)$, one extra round of edge relaxations takes $O(E)$, and DFS takes $O(V + E)$. We get $O(VE)$-time algorithm overall.
Problem 4-2. [25 points] It's Hanna Barbera Time!

(a) [10 points] **Solution:**

Summary: Given a DAG (Directed Acyclic Graph), we need to find the least-effort path for Jerry to get from location $s$ to the fridge $f$ without getting caught by Tom. The algorithm should run in $O(V + E)$.

Algorithm: First run a modified BFS to see if there is a valid, safe path from $s$ to $f$. We do not extend the path if the node $(x, y)$ is an unsafe spot and $x + y \equiv 0 \pmod{3}$ (See Figure 3).
Once we know there is a valid path from \( s \) to \( f \), create a new DAG \( G' \) as directed below (See Figure 4).

Set vertices in \( G' \) as following:

- \( \forall v = (x, y) \in V \) draw \( v \) in \( G' \) unless both \( v \not\in H \) and \( x + y \equiv 0 \pmod{3} \).
- \( \forall v = (x, y) \in V \) mark \( v \) in \( G' \) if \( v \not\in H \) and \( x + y \equiv 0 \pmod{3} \).

Set edges in \( G' \) as following:

- \( \forall (u, v) \in E \), draw \( (u, v) \) of \( w(u, v) \) in \( G' \) if both \( u \) and \( v \) are not marked.

Then carefully select the order of relaxation by topological sorting \( G' \) so that one iteration of Bellman-Ford edge relaxation is sufficient to calculate the least effort path from \( s \) to \( f \). Choose \( u \) in topologically sorted order, and relax \( \{(u, v) \mid v \in \text{Adj}(u)\} \).

Correctness: First we can assume \( s, f \in H \). Tom only glances over every three minutes and Jerry only moves one step north or one step east per minute. Jerry just needs to make sure that he does not end up in an unsafe spot on time \( 3k \) \((k \in \mathbb{N})\). So any path from \( s \) to \( f \) on \( G' \) is always valid and safe. On a topologically sorted DAG, one iteration through edges in sorted order is enough to find the shortest paths, because we never modify \( u.d \) once we have relaxed all edges \( \in \{(u, v) \mid v \in \text{Adj}(u)\} \).

Run Time: It takes \( O(V + E) \) to check if there exists a safe path from \( s \) to \( f \) using modified BFS. Then it takes \( O(V + E) \) to create \( G' \) because we have constant operations for all vertices and edges iterations. In \( O(V + E) \), we obtain a topological sort using DFS. One iteration of relaxation costs \( O(E) \). So we achieve \( O(V + E) \).
(b) [15 points] Solution:
Summary: Jerry wants to safely get from $s$ to $f$ with the least effort. Now he waits at vertices for as many minutes as he wants. The algorithm should run in $O(V \log V + E)$.

Algorithm: First iterate through all $(x, y) \in H$ to see if there is another hiding spot (or $f$) within three edges. If all $v \in \{(x + 1, y), (x + 2, y), (x + 3, y), (x, y + 1), (x + 1, y + 1), (x + 2, y + 1), (x, y + 2), (x + 1, y + 2), (x, y + 3)\}$ are unsafe, then Jerry cannot safely get from $s$ to $f$.

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{figure5}
\caption{Check for hiding spots within three edges (i.e. circled nodes)}
\end{figure}

Once we know there is a valid path from $s$ to $f$, we initialize subgraphs $G_0, G_1, G_2$ for $G''$ as directed below ($G_i$: state graph for $i$ minutes remaining until Tom looks over). Set vertices in $G''$ as following:

- Include one copy of each $s$ and $f$ in $G''$.
- $\forall v \in H \setminus \{s, f\}$, include a copy called $v_i$ in $G_i$ for $i = 0, 1, 2$.
- $\forall v \in V \setminus H$, include a copy called $v_i$ in $G_i$ for $i = 1, 2$.

Set edges in $G''$ as following (*draw edge $(u, v)$ only if both $u$ and $v$ exist in $G''$):

- $\forall (s, v) \in E$, *draw $(s, v_i)$ of $w(s, v)$ in $G''$ for $i = 0, 1, 2$.
- $\forall (u, f) \in E$, *draw $(u_i, f)$ of $w(u, f)$ in $G''$ for $i = 0, 1, 2$.
- $\forall (u, v) \in E, u \neq s, v \neq f$, *draw $(u_0, v_1), (u_1, v_2), (u_2, v_0)$ of $w(u, v)$ in $G''$.
- $\forall v \in H \setminus \{s, f\}$, *draw $(v_0, v_1), (v_1, v_2), (v_2, v_0)$ of zero-effort in $G''$.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure6}
\caption{Example}
\end{figure}
Run Dijkstra’s algorithm with a Fibonacci heap on $G''$ with $s$ as the source node.

Correctness: As in part (a), first we can assume $s, f \in H$. We rule out the case where Jerry cannot get to $f$ safely. In $G''$ construction, it is okay to assume that Jerry waits in hiding spots only, because the only constraint is that Jerry has to be in a hiding spot on time $3k$ ($k \in \mathbb{N}$). Whether Jerry can wait in non-safe spots does not affect the required effort. Waiting in nodes creates cycles but Dijkstra’s algorithm has no problem working with cycles. With $G''$ that guarantees at least one valid path from $s$ to $f$, Dijkstra’s algorithm with a Fibonacci heap will find the least effort path.

Run Time: We have $O(V)$ iterations of examining hiding spots and $O(1)$ check for nine vertices within three edges. Then it takes $O(V + E)$ to create $G'$ because we have constant operations for all vertices and edges iterations. $G''$ has $O(3V)$ vertices and $O(3E + 3V)$ edges. Running Dijkstra’s algorithm with a Fibonacci heap produces $O((3V) \log(3V) + 3E + 3V)$ which can be simplified to $O(V \log V + E)$.

Problem 4-3. [15 points] Rubinfeld’s Cube

(a) [5 points] Solution:
Summary: Given an undirected, unweighted graph $G = (V, E)$ and a starting configuration $c \in V$, we need to find a minimum-length sequence of configurations that solves the cube. The algorithm should run in $O(V + E)$.
Algorithm: We use BFS to find the shortest path from $c$ to the solved configuration.
Correctness: All moves are reversible and require equal effort. On undirected, unweighted graphs, we can find the shortest path from one node to another using BFS.
Run Time: $O(V + E)$ is achieved with standard BFS.

(b) [10 points] Solution:
Summary: Given an undirected, unweighted graph $G = (V, E)$, we need to preprocess $G$ in $O(VE)$ to output in $O(k)$ time a minimum-length sequence of configurations that transforms $c_1$ into $c_2$, where $k$ is the length of this shortest sequence.
Algorithm: BFS for all $v \in V$. Then output $k$ steps in transforming $c_1$ into $c_2$.
Correctness: Completion of BFS for all vertex in $V$ allows us to provide the shortest path for any $c_1$ and $c_2$.
Run Time: We get $O(V^2 + VE)$ with $O(V)$ iterations of $O(V + E)$-time algorithm BFS. With $V^2 = O(E)$ (: $|V| \leq |E| + 1$), we achieve $O(VE)$. Outputting $k$ steps require $O(k)$ with $k$ iterations of $O(1)$.

Part B

Problem 4-4. [30 points] $k$th shortest paths

(a) Submitted on alg.csail.mit.edu

(b) Submitted on alg.csail.mit.edu
(c) [10 points] Solution:

Summary: All-pairs $k$th-shortest-path for $k \in \mathbb{N}$.

Algorithm: Maintain $k$ 2D matrix arrays. Initialize the shortest path matrix with edge lengths and all other matrices with infinity values. Using the Floyd-Warshall idea as in part (a) and (b), update values as we consider taking a path with intermediate node $x$. If new shortest is the shortest, update all $k$ shortest. If it is greater than the shortest but less than the second-shortest, update second-shortest and up to $k$th shortest. And following the same logic, there are $k + 1$ cases. We just make sure that when $u = x$ or $x = v$ (where $x$ is the newly-considering-intermediate node), we check all $k^2$ possibilities as our new candidate.

Correctness: By keeping track of all $k$ shortest values and updating values as we consider more intermediate nodes, we make sure to get the $k$th shortest by the end. We check for loops by considering all $k^2$ possibilities.

Run Time: $O(kV^3)$ with more checks in the loop.

Eppstein has a great algorithm... (not my idea)

Algorithm: Use Eppstein’s Algorithm for all $v \in V$.

Correctness: [Eppstein's Algorithm Link]

Run Time: $O(VE + V^2 \log V + kV^2)$