CS 6363.003 Spring 2021 — Official Solutions

Homework 4 Problem 1

Describe and analyze an efficient algorithm to determine whether or not a given instance of Stick Clash can be won, *assuming the input graph $G = (V, E)$ is a DAG*. You may assume comparisons and basic arithmetic operations, including division, can be done in constant time each.

Solution: Suppose the knight is waiting at a vertex u, and their party has strength x. We should move them to a successor v of u from which they can eventually reach t given that the party enters v with only x strength. To know which v make that possible, we'll try to compute lower bounds on how much strength is needed to enter a vertex and eventually win the game.

For any vertex u, let $ForTheWin(u)$ denote the maximum value such that the knight can enter u and still win if and only if their strength is strictly greater than $ForTheWin(u)$ when they enter u. If a win is impossible, because there is no path from u to t, then $ForTheWin(u) := \infty$. Likewise, $ForTheWin(u) = -\infty$ if the game can be won no matter how little strength the knight has when entering u.

$ForTheWin(t) = -\infty$, because the game is won as soon as the knight's party enters t. For any other vertex u with outgoing edges, the knight's party must have enough strength to survive the feature at u (if there is one) and then continue on to a win from the least demanding of u's successors. So, if u has no feature,

$$ForTheWin(u) = \min_{u \to v} ForTheWin(v).$$

If u has friendly soldiers of strength $g(u)$, then

$$ForTheWin(u) = \min_{u \to v} ForTheWin(v) - g(u).$$

If u has enemy soldiers of strength $b(u)$, then

$$ForTheWin(u) = \max\left\{b(u), \min_{u \to v} ForTheWin(v) - b(u)\right\}.$$

If u has a spike trap of harm $h(u)$, then

$$ForTheWin(u) = \max\left\{h(u), \min_{u \to v} ForTheWin(v) + h(u)\right\}.$$

If u has a blessed shield, then

$$ForTheWin(u) = \min_{u \to v} ForTheWin(v)/2.$$

If u has some cursed bombs, then

$$ForTheWin(u) = 2 \cdot \min_{u \to v} ForTheWin(v).$$

Finally, if vertex u has no outgoing edges, then there is no path from u to t and $ForTheWin(u) = \infty$. Defining $\min_{\emptyset} := \infty$ implicitly covers that final.

The game can be won if and only if $ForTheWin(s) < 20$. There is only one subproblem per vertex of input graph $G = (V, E)$, and each subproblem $ForTheWin(u)$ relies only on successors of u. Therefore, we can solve all the subproblems in a postordering of G's vertices. The total time spent solving subproblems is proportional to the number of vertices and edges in G, so ***the whole algorithm will take $O(V + E)$ time***. Here's a pseudocode implementation.

```
CanWinStickClash($V, E$):
  for each vertex $u$ in postorder
      if $u = t$
          $u.ForTheWin \leftarrow -\infty$
      else
          $needed \leftarrow \infty$
          for each edge $u \to v$
              if $needed > v.ForTheWin$
                  $needed \leftarrow v.ForTheWin$
              if $u$ has friendly soldiers
                  $u.ForTheWin \leftarrow needed - g(u)$
              else if $u$ has enemy soldiers
                  $u.ForTheWin \leftarrow \max\{b(u), needed - b(u)\}$
              else if $u$ has a spike trap
                  $u.ForTheWin \leftarrow \max\{h(u), needed + h(u)\}$
              else if $u$ has a blessed shield
                  $u.ForTheWin \leftarrow needed/2$
              else if $u$ has cursed bombs
                  $u.ForTheWin \leftarrow 2 \cdot needed$
              else
              $u.ForTheWin \leftarrow needed$

      if $s.ForTheWin < 20$
          return “You can win!”
      else
          return “You’ll lose. Give up.”
```

■

Rubric: 10 points total: 5 points total for the recurrence; −1 for no justification, −2 for missing the base cases; 3 points for filling the table (0 if the recurrence is very wrong); 2 points for the running time analysis.

CS 6363.003 Spring 2021 **Homework 4 Problem 2** Official Solutions

(a) Describe an edge-weighted undirected graph that has a unique minimum spanning tree, even though two edges have equal weights.

Solution: Let $G = P_3$, the undirected path graph on three vertices and two edges. Give both edges a weight of 1. Graph G itself is its only (minimum) spanning tree even though both edges have the same weight. ■

Rubric: 2 points total. All or nothing.

(b) Let G be an arbitrary edge-weighted undirected graph and F be a subgraph of some minimum spanning tree of G. Let e be an arbitrary safe edge with respect to G and F. Prove the following extension of Erickson's Lemma 7.2:

Claim. *There exists a minimum spanning tree T of G such that $(F \cup \{e\}) \subseteq T$. Further, every minimum spanning tree $T \supset F$ contains e if there exists a component of F such that e is the only minimum-weight edge leaving that component.*

Solution: Let T be an arbitrary minimum spanning tree of G such that $F \subseteq T$. By definition, e is a lightest edge leaving some component S of F. Suppose T does not contain e. Spanning tree T contains a path between the endpoints of e. This path contains at least one edge e' with exactly one endpoint in S.

As in Erickson's Lemma 7.2, the subgraph $T' = T - e' + e$ is also a spanning tree. We have $e' \notin F$ by the definition of connected component S, implying $F \subseteq T'$ as well. Finally, $w(e') \geq w(e)$, implying $w(T') \leq w(T)$. We see T' is a minimum spanning tree such that $(F \cup \{e\}) \subseteq T'$.

Now, suppose e were the only edge of its weight leaving S. In this case, $w(e') > w(e)$, implying $w(T') < w(T)$. We have a contraction, implying T must have contained e all along. ■

Rubric: 3 points total.

(c) Describe and analyze an algorithm to determine whether or not a given edge-weight connected undirected graph has a unique minimum spanning tree.

Solution: As suggested by the hint, we'll modify Kruskal's algorithm. Let $G = (V, E)$ be the input graph. We begin by sorting the edges E in increasing order of weight. For any $w \in \mathbb{R}$, let $E_w := \{e \mid w(e) = w\}$ denote the edges of weight w. For each distinct edge weight w such that E_w contains one or more edges in increasing order, we will either add a subset of E_w to an intermediate spanning forest F or declare that G contains more than one minimum spanning tree. We will guarantee that immediately after processing each group E_w that either the minimum spanning tree is not unique, or F consists of exactly the subset of its edges with weight at most w.

Suppose we are about to process edge subset E_w. Assume our claim about F holds inductively for weights strictly less than w. As in class, each edge $e \in E_w$ is either safe or useless with respect to F. From part (b), we know each safe edge $e \in E_w$ belongs to *some* minimum spanning tree T such that $F \subseteq T$. Therefore, if the minimum spanning tree of G is unique, *all* safe edges of E_w must belong to it. We'll try to add them one by one to F as in standard Kruskal's algorithm, and report the minimum spanning tree is not unique if they cannot all be added.

Surprisingly, we *must* add all safe edges of E_w if we're able to do so in order to build any minimum spanning tree. Suppose we add only a strict subset of E_w to F. At least one component of the now larger forest F is incident to only one of the safe edges of E_w that we did not add; otherwise, we could take a "walk" from component to component along the safe edges we did not add, eventually reaching a component a second time and proving that adding all the safe edges of E_w would have created a cycle. Let $e \in E_w$ be the one safe edge incident to a component F that we just proved exists. By part (b), edge e must belong to every minimum spanning tree, so we would be wrong to exclude it after moving on from E_w.

From the above discussion, we see that our algorithm will eventually build the only minimum spanning tree if it never reports the tree is not unique. Therefore, we can safely report the minimum spanning tree is unique after successfully processing every subset E_w.

As in standard Kruskal's algorithm, we spend $O(E \log V)$ time sorting the edges by weight. Finding the groups of edges and checking which edges are safe within each group before attempting to add them to F only increases the running time of the rest of the algorithm by a constant factor. ***Therefore, the algorithm takes $O(E \log V)$ time total.*** ■

Rubric: 5 points total: 3 points for the algorithm; 1 point for justification, 1 point for running time analysis.

CS 6363.003 Spring 2021 Official Solutions
Homework 4 Problem 3

Suppose we are given a directed graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$ and two vertices s and t. You may assume G has no negative weight cycles.

(a) Describe and analyze an algorithm to find the shortest path from s to t when exactly one edge in G has negative weight.

Solution: Per the hint, we'll just run Dijkstra's algorithm without modification and then report the shortest path from s to t after it terminates. Correctness if immediate given we know Dijkstra's algorithm is correct, but the running time requires more explanation.

Let $u{\to}v$ be the only negative weight edge. Up until $u{\to}v$ is relaxed (if it ever is), it's like we're running Dijkstra on a graph with only non-negative weights, and everything takes $O(E \log V)$ time. Now, consider what happens when we relax $u{\to}v$. We'll do an INSERT or DECREASEKEY that causes the minimum key in the priority queue to be lower $dist(u)$, leading to a decrease between consecutive $dist$ values for EXTRACTIONs. However, *starting with that next EXTRACTION*, the $dist$ values of EXTRACTED vertices will go back to being non-decreasing, implying each vertex will be EXTRACTED at most one more time: Up until we check $u{\to}v$ again to see if its tense, the arguments of Erickson Lemmas 8.3 and 8.4 apply verbatim. However, to check if $u{\to}v$ is tense, we must first decrease $dist(u)$ and then INSERT u into the priority queue. But that would imply we found a shorter walk from s to u than we knew before. We already had the shortest walk that didn't include $u{\to}v$, so the new walk must include $u{\to}v$ and in particular u. We would only consider walks repeating a vertex if G had a negative cycle. ■

Rubric: 4 points total: 2 points for the algorithm; 1 point for justification, 1 point for running time analysis.
Simply references a correct solution to part (b) is sufficient for full credit.

(b) Describe and analyze an algorithm to find the shortest path from s to t when exactly k edges in G have negative weight.

Solution: We could just run Dijkstra's algorithm as in part (a), but the analysis gets very messy. Instead, we'll use the hint and modify Bellman-Ford.

The main intuition behind Bellman-Ford is that each iteration i of the main loop ends with us essentially learning about all shortest paths that happen to contain at most i edges. We'll apply a similar idea here, instead using each iteration i to learn about shortest paths containing at most i *negative weight* edges. Let DIJKSTRAPASS be a variant of Erickson's DIJKSTRA(s) that a) skips the first two lines which call INITSSSP(s) and INSERT(s, 0) and b) only relaxes edges with non-negative weights. We're maintain a single priority queue of vertices across multiple instantiations of DIJKSTRAPASS.

We begin by running INITSSSP(s) and INSERT(s, 0). We do a single DIJKSTRAPASS. Then, we repeat the following process k times: For each negative weight edges $u{\to}v$, we check if $u{\to}v$ is tense. If so, we RELAX($u{\to}v$) and either INSERT(v, $dist(v)$) or DESCREASEKEY(v, $dist(v)$)

depending on whether v it currently outside the priority queue. After processing all negative weight edges, we run DijkstraPass. In total, that's $k+1$ runs of DijkstraPass on the subgraph of non-negative weight edges. All the normal arguments for Dijkstra apply, so these runs take a total of $(k+1)\cdot O(E\log V) = O(kE\log V)$ time which subsumes the $k\cdot O(k\log V) = O(kE\log V)$ time spent relaxing negative weight edges and doing associated updates of the priority queue. ***The total running time is $O(kE\log V)$.***

For correctness, we claim that after the ith iteration of the main loop, for any vertex v, value $dist(v)$ is at most the length of the shortest walk using at most i *negative weight* edges. Indeed, before the first iteration, DijkstraPass computes all distances for walks using only non-negative weight edges. Suppose $i > 0$. By the end of the $(i-1)$st iteration, we inductively found distances at most the length of shortest walks using at most $i-1$ negative weight edges. We then relax all negative weight edges, so we now know all distances for paths using at most i negative weight edges that also end on a negative weight edge. The heads of any relaxed edges are added to the priority queue, so the call to DijkstraPass will successfully extend these paths to those using at most i edges and ending with non-negative weight edges. ■

Rubric: 6 points total: 3 points for the algorithm; 1.5 points for justification, 1.5 points for running time analysis.

Any correct $O(f(k)E\log V)$ time algorithm is worth full credit.

CS 6363.003 Spring 2021 — Official Solutions

Homework 4 Problem 4

Let $G = (V, E)$ be a directed graph with edge weights $w : E \to R$; edge weights could be positive, negative, or zero, but you may assume there are no negative weight cycles.

(a) Let $v \in V$ be an arbitrary vertex. Describe and analyze an algorithm that constructs a directed graph $G' = (V \setminus \{v\}, E')$ with weighted edges such that the shortest path distance between any two vertices in G' is equal to the shortest path distance between the same two vertices in G. Your algorithm should run in $O(V^2)$ time.

Solution: We describe the procedure RemoveVertex(V, E, w, v).

```
RemoveVertex(V, E, w, v):
    $V' \gets V \setminus \{v\}$
    $E' \gets \emptyset$
    for each $x \in V'$
        for each $y \in V'$
            add $x \to y$ to $E'$
            $w'(x \to y) \gets \min\{w(x \to y), w(x \to v) + w(v \to y)\}$
    return $(V', E', w')$
```

We will argue that the new weights preserve the shortest path distances by describing how to transform walks in G to cheaper walks in G' and vice versa. Then, one graph cannot have lower shortest path distances than the other. Consider a walk p in G between two vertices s and t. If p does not use v, then all its edges exist in G', and their weights are only lower. If it does use v, then let it contain the subwalk $x \to v \to y$. The subwalk from s to x, $x \to y$, and the subwalk from y to t together cost less than the total weight of p in G'.

Now, let p be a walk in G'. The same walk exists in G. For each edge $x \to y$ in p with $w'(x \to y) < w(x \to y)$, we can replace $x \to y$ with a walk $x \to v \to y$ of cost $w'(x \to y)$. After the replacements, we get a new walk in G of the same cost as p.

The running time is determined by the nested for loops over $V - 1$ vertices each, so it is $O(V^2)$. ■

Rubric: 4 points total. 2.5 points for the algorithm. 1 point for justification. 0.5 points for running time analysis.

(b) Now suppose we have already computed all shortest path distances in G'. Describe and analyze an algorithm to compute the shortest path distances in the original graph G from v to every other vertex, and from every other vertex to v, all in $O(V^2)$ time.

Solution: We describe the procedure RestoreVertex(V, E, w, v). We assume there is a global matrix $dist : V \times V \to \mathbb{R}$ that already stores the shortest-path distance $dist(x, y)$ for any pair of vertices x and y in G'.

```
RESTOREVERTEX($V, E, w, v$):
    $dist[v,v] \leftarrow 0$
    for each $z \in V \setminus \{v\}$
        $dist[v,z] \leftarrow \infty$
        for each $y \in V \setminus \{v\}$
            if $w(v\to y) + dist[y,z] < dist[v,z]$
                $dist[v,z] \leftarrow w(v\to y) + dist[y,z]$
    for each $x \in V \setminus \{v\}$
        $dist[x,v] \leftarrow \infty$
        for each $y \in V \setminus \{v\}$
            if $dist[x,y] + w(y\to v) < dist[x,v]$
                $dist[x,v] \leftarrow dist[x,y] + w(y\to v)$
```

The shortest path from v to any vertex z starts with some edge $v\to y$ and continues with a shortest path from y to z. The algorithm is just taking the minimum over the total length of all such paths. Similarly, the algorithm takes the minimum over the total lengths of all paths from some vertex x to y plus the edge weight $w(y\to v)$.

We now have two doubly nested for loops over $V - 1$ items, so the running time is $O(V^2)$. ■

Rubric: 3 points total. 2 points for the algorithm. 0.5 points for justification. 0.5 points for running time analysis. No penalty for not setting $dist[v,v]$.

(c) Combine parts (a) and (b) to describe and analyze another all-pairs shortest paths algorithm that runs in $O(V^3)$ time. (The resulting algorithm is *almost* the same as Floyd-Warshall.)

Solution: We describe the procedure RECURSIVEAPSP($V, E, w, dist$).

```
RECURSIVEAPSP($V, E, w$):
    if $V \neq \emptyset$
        Let $v$ be any vertex in $V$
        $(V', E', w') \leftarrow$ REMOVEVERTEX($V, E, w, v$)
        RECURSIVEAPSP($V', E', w'$)
        RESTOREVERTEX($V, E, w$)
```

If there are any vertices during a recursive call, then we remove one of them v using REMOVEVERTEX, preserving the other all pairs shortest path distances. We recursively compute the smaller set of distances by induction and finally use those distances to get distances to and from v.

Each recursive call involves at most V vertices, so all they each take $O(V^2)$ time to run the two subroutines described in earlier parts. The recursion tree is a path, so there are V recursive calls. The total running time is $O(V^3)$. ■

Rubric: 3 points total. 2 points for the algorithm. 0.5 points for justification. 0.5 points for running time analysis.