CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)

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*.

Solution: Let maxStrength[v] represent the maximum strength that knight party can achieve when it reaches vertex v.

maxStrength[v] = 20 if v = s,

or = max(f(v, maxStrength[u]) | maxStrength[u] > 0 and u->v belongs to E) o.w.

where f(v, maxStrength[u]) represent the corresponding resulting strength when it reach to vertex v.

```
StickClash(s, t):
  for each node v in topological order
      if v = s
          v.maxStrength = 20
      else
          v.maxStrength = -inf
          for each edge from u -> v
              if u.maxStrength > 0
                  v.maxStrength = max (v.maxStrength, f(v, u.maxStrength))
  return t.maxStrength > 0
```

```
f(v, strength):
  if feature(v) is friendly soldier
      return strength + g(v)
  else if feature(v) is enemy soldier
      if strength > b(v)
          return strength + b(v)
      else
          return -inf
  else if feature(v) is trap
      return strength - h(v)
  else if feature(v) is bless shield
      return 2 * strength
  else if feature(v) bomb
      return strength / 2
```

■

Time complexity: The total running time is $O(V + E)$ since the topological sort takes O(V+E) time and the dynamic programming takes O(V+E) time as well since each edge is considered exactly once.

CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)
Homework 4 Problem 2

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

Solution: Given a graph G(V,E), if there always exist a unique least weighted edge connecting two arbitrarily assigned component C1(V1,E1) and component C2(V2, E2), the graph has the unique minimum spanning tree. In other words, when there is only one minimum-weighted edge leaving any component in F belonging to T, the graph has the unique minimum spanning tree.

■

(b) Prove the following: 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: Part1 Proof: Let S be an arbitrary subset of vertices of G that having one endpoint for edge e. Let T be an arbitrary spanning tree that does not contain e. Because T is connected, it contains a path from one endpoint of e to the other. Because this path starts at a vertex of S and ends at a vertex not in S, it must contain at least one edge with exactly one endpoint in S. Let e' be any such edge. Because T is acyclic, removing e' from T yields a spanning forest with exactly two components, one containing each endpoint of e. Thus, adding e to this forest gives us a new spanning tree T' = T - e' + e. The definition of e implies w(e) <= w(e'), which implies that T' can have the same total weight as T. Thus, T' can also be a minimum spanning tree.

part2 proof: Let T be an arbitrary spanning tree that does not contain e. And let S be an subset of vertices of G that constitute a component in F and where e has one endpoint in that component. Because T is connected, it contains a path from one endpoint of e to the other. Because this path starts at a vertex of S and ends at a vertex not in S, it must contain at least one edge with exactly one endpoint in S. Let e' be any such edge. Because T is acyclic, removing e' from T yields a spanning forest with exactly two components, one containing each endpoint of e. Thus, adding e to this forest gives us a new spanning tree T' = T - e' + e. Since e is the only minimum-weight edge leaving that component, w(e) < w(e'), which implies that T' has a smaller total weight than T. Thus, T is not the minimum spanning tree of G, which lead to a contradiction. ■

(c) Describe and analyze an algorithm to determine whether or not a given edge-weight connected undirected graph has a unique minimum spanning tree. *[Hint: Modify Kruskal's algorithm.]*

Solution:

```
UniqueMST(V, E, w):
  notate each edge with a unique id
  sort E by increasing weight
  F <- (V,emptySet)
  for each vertex v
      MAKESET(v)
  previousEdge <- Null
  for i <- 1 to |E|
      uv <- ith lightest edge in E
      if FIND(u) = FIND(v)
          if previousEdge not Null and w(uv) = w(previousEdge)
              id(wx), w(wx) <- id and weight of the edge wx having the minimum weight in the cycle
              if w(wx) = w(uv) and id(wx) != id(uv)
                  return not unique
      else
          UNION(u,v)
          add uv to F
          previousEdge = uv
  return unique
```

Time complexity: $O(VE)$ because in the worst case inside the we need to traverse the potential cycle at most E - 2 times, and each traversal requires O(V) time.

■

CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)
Homework 4 Problem 3

(a) Describe and analyze an algorithm to find the shortest path from s to t when exactly one edge in G has negative weight. *[Hint: Modify Dijkstra's algorithm. Or don't.]*

Solution:

```
ONENEGSSSP(V, E, w, s, t):
  G' <- remove the negative edge u -> v in the graph
  dist'(t) <- run dijkstra CLRS version on the G'
  if dist'(u) + w(u -> v) + dist'(v -> t) < dist'(t)
      dist(t) = dist'(u) + w(u -> v) + dist'(v -> t)
  else
      dist(t) = dist'(t)
```

Time complexity: $O(ElogV)$ because there the dijkstra requires O(ElogV) time and the extra relaxation takes constant time. ■

(b) Describe and analyze an algorithm to find the shortest path from s to t when exactly k edges in G have negative weight. Any $O(f(k)E\log V)$ time algorithm where f is a function of k is worth full credit, but an $O(kE\log V)$ time algorithm may be faster and easier to analyze than those with worse dependency on k.

Solution:

```
KNEGSSSP(V, E, w, s, t):
  INITSSSP(s)
  for all vertices u
      INSERT(u, dist[u])
  for i <- 1 to k
      while the priority queue is not empty
          u <- EXTRACTMIN()
          for all edges u -> v
              if u -> v is tense
                  RELAX(u -> v)
              if v is in the priority queue
                  DECREASEKEY(v, dist(v))
      for all vertices u
          INSERT(u, dist[u])
```

Time complexity: $O(kElogV)$ because the dijkstra requires O(ElogV) time and there are k loops, the insertion inside the loop takes O(VlogV) time. So the total time complexity is O(kElogV) + O(kVlogV) = O(kElogV) ■

CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)
Homework 4 Problem 4

(a) 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:

```
BuildGraph(V, E, w, v):
    copy every edge from G to G' except for those begin or end at vertex v
    for all vertices u in G
        if edge u -> v exists
        for all vertices k in G
            if edge v -> k exists
            add edge u -> k of w(u -> v) + w(v -> k) to G'
```

Time complexity: $O(V^2)$ because nested loops are needed to general new edge weights and copy original ones. ∎

(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: Let dist[i][j] represent the already computed shortest path distances between nodes i and j.

```
ComputeDist(V, E, dist, v):
    for all vertices i in G
        for all vertices j in G
            if i != v
                if edge j -> v exist
                    dist[i][v] = min(dist[i][v], dist[i][j] + dist[j][v])
                if edge v -> j exist
                    dist[v][i] = min(dist[v][i], dist[j][i] + dist[v][j])
```

Time complexity: $O(V^2)$ because we need to determine whether to relax edges that going through vertex 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:

```
AllPairShortestPath(V, E):
    for all vertices u
        for all vertices v
            if u = v
                dist[u][v] <- 0
            else
                dist[u][v] <- +inf
    return helper(V,E,dist)
```

```
HELPER(V, E, dist):
  if V has only one node
      return dist
  pick a vertex v in V
  V',E' <- BuildGraph(V,E,w,v)
  dist <- HELPER (V',E',dist)
  dist <- ComputeDist(V,E,dist,v)
return dist
```

Time complexity: $O(V^3)$ because each at each level, it required $O(V^2)$ complexity, and there are V levels in total. The overall complexity is therefore $O(V^3)$. This can be proved by induction. Suppose we already computed the all pair shortest path in subgraph G', then ComputeDist will correctly calculate the all pair shortest path in G. Base case is when graph has one node, directly return. ■