HW 8 Solution

CS/ECE 374, Spring 2019 Version: **1.1**

22 (100 pts.) Graph Morphing

In the following, let G be a graph with n vertices and m edges.

22.A. (20 pts.) Given an *undirected* graph G, and two vertices $s, t \in \mathsf{V}(\mathsf{G})$, describe a linear time algorithm that directs each edge of G so that the resulting directed graph is a DAG, s is a source of this DAG, and t is a sink.

Solution:

Pick any ordering π of the vertices of G, where the first vertex is s, and last vertex is t (i.e., $\pi(s) = 1$ and $\pi(t) = n$). Orient an edge $uv \in \mathsf{E}(\mathsf{G})$ as (u, v) if $\pi(u) < \pi(v)$, and as (v, u) otherwise. Clearly, there are only incoming edges in t, so it is a sink, and similarly there are only outgoing edges from s, so it is a source. There are no cycles in this graph, since traversing any edge strictly increases the value of π associated with the current vertex - and a cycle would imply the existence of an edge that decreases this value.

22.B. (10 pts.) For a connected undirected graph G, a ***bridge*** is an edge that its removal disconnects the graph. Using **DFS** describe a linear time algorithm that decides if a bridge exists, and if so it outputs it.

Solution:

Let us do **DFS** traversal of G, and compute for every vertex its visit time interval $I(v) = [\mathrm{pre}(v), \mathrm{post}(v)]$. In addition, let $\mathrm{parent}(u)$ denote the parent of u in the **DFS**, and let

$$\Gamma(u) = \{v \mid uv \in \mathsf{E}(\mathsf{G}) \text{ and } v \neq \mathrm{parent}(u)\}.$$

Note, that $\Gamma(u)$ includes all the children of u in the **DFS**, but it might also include some ancestors of u. Compute during the **DFS** the quantity

$$\alpha(u) = \min\left(\min_{v \in \Gamma(u)} \mathrm{pre}(v), \quad \min_{v \in \mathrm{children}(u)} \alpha(v), \quad \mathrm{pre}(u) \right),$$

where $\mathrm{children}(u)$ is the set of children of u in the **DFS** tree. The quantity $\alpha(u)$ is the minimum pre-order time of any vertex reachable from u, where you travel down a **DFS** edge, and up on a single back edge. (However, you are not allowed to travel directly on the edge back to the parent.)

Clearly, $\alpha(u)$ can be computed in linear time during the computation of the **DFS**.

Claim 8.1. *Let r be the root of the* **DFS** *of G, where G is an undirected connected graph. The graph G has a bridge $\iff$ there exists a vertex $u \neq r$, such that $\alpha(u) \in I(u)$.*

Proof: $\implies$ Let uv be the bridge in the graph, and assume that the **DFS** first visits u. But then, $I(v) \subsetneq I(u)$. Let T be the computed **DFS** tree of G. Let T_v be the subtree of v. Since uv is a bridge, no vertex of T_v has an edge to a vertex of $\mathsf{V}(\mathsf{T}) \setminus (\mathsf{V}(\mathsf{T}_v) \cup \{u\})$. But this implies that $\alpha(x)$, for all $x \in \mathsf{T}_v$, is contained in $I(v)$. In particular, this holds for v itself. Namely, $\alpha(v) \in I(v)$.

$\Longleftarrow$ Assume that there exists $\alpha(v) \in I(v)$, for some $v \neq r$, and furthermore, let v be vertex maximizing this value. Let $z = \text{parent}(v)$ be the parent of v in the **DFS**. Clearly, the edge zv is a bridge. Indeed, if not, there is some descendant u of v in T_v, such that u is connected by an edge to some other vertex, say x that is in $\mathsf{T} \setminus \mathsf{T}_v$. We have that $\text{pre}(x) < \text{pre}(v)$, as otherwise x would be in the subtree of v, but this implies that $\alpha(u) < \text{pre}(v)$, which is a contradiction. ■

As such, the above implies that we can easily detect if a bridge in a graph exists, by just scanning all the vertices in the graph, and checking if $\alpha(v) \in I(v)$ for any $v \in \mathsf{V}(\mathsf{G}) - r$. Clearly, this can be done in linear time after the **DFS** is done. If we find such a vertex v, then $v'v$ is a bridge, where $v' = \text{parent}(v)$.

22.C. (20 PTS.) Prove that for an *undirected* connected graph G with n vertices and m edges, the edges can be directed so that the resulting graph is strongly connected $\Longrightarrow$ there are no bridges in G. (the claim holds with $\Longleftrightarrow$. This is the easier direction.)

Solution:

Proof: Assume for the sake of contradiction that there is a bridge uv in G, then it connects two parts of the graph P_1 and P_2, and assume that in the orientation of the graph the edge is oriented as (u, v), where $u \in P_1$ and $v \in P_2$. But then, there is no path between vertices of P_2 and vertices of P_1. Namely, the oriented graph is not strongly connected. ■

22.D. (50 PTS.) Given an undirected connected graph G that has no bridges, describe how to modify the **DFS** algorithm so that it outputs an orientation of the edges of G so that the resulting graph is strongly connected. Prove the correctness of your algorithm.

Solution:

The algorithm works as hinted above – we perform a **DFS** in the graph from an arbitrary vertex r, and verify that there are no bridges. Next, we orient the edges, where **DFS** edges point downward, and back edges points upward (here, the root r of the **DFS** tree is on top). Let H be the resulting directed graph.

Claim 8.2. *If G is an undirected connected graph, then H is strongly connected.*

Proof: Since G is connected, the **DFS** tree includes all the vertices in the graph, and in particular all of them are reachable from r, since the edges of the **DFS** tree are oriented away from r.

The other direction is more fun. We claim that from any vertex $u \in \mathsf{V}(\mathsf{G})$ there is a walk in H that goes to the root r. To this end, recall that for any vertex $u \neq r$, we have that $\alpha(u) < \text{pre}(u)$. Namely, one can reach from u a vertex with an earlier pre-ordering time by going down to some descendant in the **DFS** tree, and then using one single back edge. In particular, let $u_1 = \text{back}(u)$ be this earlier (in the pre-order) vertex reachable from $u_0 = u$. If $\text{pre}(u_1) > \text{pre}(r)$, then we can continue in this fashion, setting $u_2 = \text{back}(u_1)$. We continue in this fashion. Each time we do this, we jump backward in time (no need to use a car to do this). Since the graph is finite, sooner or later, the process would stop in the root r of the tree, as $\text{pre}(r)$ is the minimal pre-ordering time in the graph. We conclude that there is a path from u to r, which implies that every vertex in H can reason r, and vice versa, which implies that H is strongly connected. ■

Proof: (Alternative.) Let G^{SCC} be the meta graph of the strong connected components. Let S be the strong connected component that contains r, the root of the **DFS**. Since the graph is connected, S can reach all the vertices in the graph. Namely it is a source in the meta graph. Let $X = \mathsf{V}(\mathsf{G}) \setminus S$.

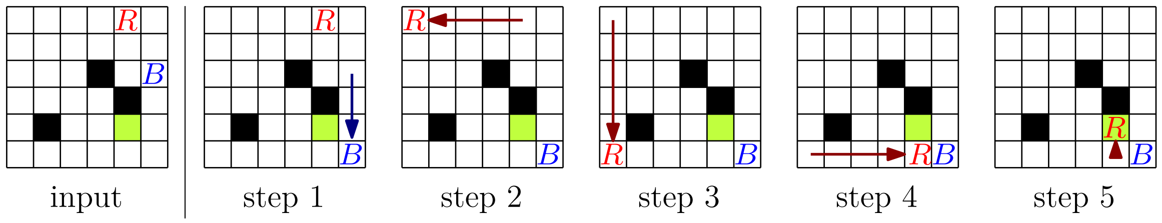
Let (s, x) the first edge constructed by the **DFS** such that $s \in S$ and $X \in X$. Since sx is not a bridge, there an edge $x's'$ that lies on a cycle in G that contains also the edge sx, where $x' \in X$ and $s' \in S$. The **DFS** visiting x, would reach x', which would then try to orient it as (x', s') and put it in the **DFS** tree. Otherwise, it is a back edge, but then s' is an ancestor of x' in the **DFS** tree, and the edge $x's'$ would be oriented as (x', s') (namely, we orient this edge in the same direction in both scenarios).

But this implies that there is a path from r to x' – since r can reach all vertices in H, and there is a path from x' to s', and from s' to r (since they are both in S). It follows that x' and r are in the same strong connected component, which is a contradiction. ■

23 (100 PTS.) Robot motion planning.

Consider the following puzzle. You are given a red and blue robots and an $n \times n$ grid – the robots are initially located in two prespecified grid cells. Some grid cells are walls (and marked as such). One cell is marked as the *target*.

In each turn, one of the robots has to move from its current position as far as possible upward, downward, right, or left, stopping just before either (i) hitting the outer wall of the grid, (ii) the other robot, or (iii) an obstacle. The task is get one of the robots to the target cell. Here is an example of an input, and a valid solution (which is not unique in this case):



23.A. (30 PTS.) You are given as input n, a list of obstacle locations, the target, and the initial locations of the robots. Describe and analyze an efficient algorithm to determine the minimum number of steps required to reach the target if possible.

Solution:

Define an directed graph $\mathsf{G} = (V, E)$ where every vertex v represents the position of the red and blue robots on the $n \times n$ grid, i.e., $u = (i_R, j_R, i_B, j_B)$. There is a directed edge (u, v) if there is a valid move of the red robot or blue robot. For example, there is an edge $(i, j_R, i, j_B) \rightarrow (i, j_B - 1, i, j_B)$ if $j_R < j_B$ and there are no ob stables in the positions (i, j) for $j_R \leq j \leq j_B$ (R can not move any further, because it just hit B). Let H be the resulting graph.

Computing the edges of the graph requires a bit of cleverness. We can do this in $O(n^2)$ time, by starting with all the free grid cells that their top vertical edge is blocking (i.e., because its other side is an obstacle, or it is part of the wall of the grid [Mexico paid for this wall, BTW]), and perform **BFS** from these cells, where we are allowed to go only downward. This would compute for every free grid cell the longest vertical move it can do. Doing it for all possible directions results in knowing for each grid cell, which moves are allowed (ignoring the location of the other robot). Now, given a configuration of the two robots, it is straightforward to compute the at most 8 edges adjacent to it, by using this precomputed information (and taking into account the robots location). Namely, after $O(n^2)$ preprocessing, we can compute each edge of H in constant time.

Let the position of the target be (i_T, j_T). Then, the game ends if we reach any of the vertices (i, j, i_T, j_T) or (i_T, j_T, i, j) of all i and j. To this end, let

$$X = \left\{ (i, j, i_T, j_T), (i_T, j_T, i, j) \;\middle|\; i, j \in [\![n]\!] \right\}.$$

Note that there are no outgoing edges from any target vertex. Add a vertex w and an edge from all the vertices of X to w.

Run **BFS** on G starting from the vertex u representing the initial prespecified locations of the two robots. If no vertex of X is reachable from u, return impossible. Otherwise return the length of the shortest path from u to any vertex of X (i.e., the shortest path from u to w). The minimum number of steps is 1 minus the length of the shortest path.

There are $|\mathsf{V}(\mathsf{H})| = O(n^4)$ vertices. Each vertex has at most 8 outgoing edges, then $|\mathsf{E}(\mathsf{H})| = O(n^4)$. Constructing the graph and running **BFS** takes $O(|\mathsf{V}| + |\mathsf{E}|) = O(n^4)$ time.

In the monotone variant of the problem, one can choose the initial positions of the robots. In each step, one of the robots is being moved. One of the robots can move only right or down, and the other robot can move only up and left. As before a robot must stop just before hitting something.

23.B. (30 PTS.) Monotone variant longest path.

You are given as input n, a list of obstacle locations, and a target cell. Describe and analyze an algorithm to determine the initial locations of the robots that would maximize the number of turns required to reach the target (for the monotone variant). You can assume there is a solution for one of the possible initial configurations. The running time of your algorithm should be polynomial in n.

Solution:

Define the same graph $\mathsf{H} = (\mathsf{V}, \mathsf{E})$ as in part **(23.A.)**, but for one robot, edges exist only for moving right or down. For the other robot, only the edges for moving left or up are in the graph. Note that the new graph is a DAG since the robots can never move in the opposite directions.

The problem now is simply to find the longest path in the DAG that ends at a vertex of X (or more precisely, ends in w). This can be readily be done using topological sort. Indeed, assume that $v_1, \ldots, v_n$ be the topological ordering of the vertices, such that for all edges in the graph (v_i, v_j), we have that $i > j$.

Compute for each v_i the length of the longest path starting at v_i, and let $\ell(v_i)$ denote this quantity. Observe that

$$\ell(v_i) = \max_{(v_i,v_j)\in\mathsf{E}(\mathsf{G})} (1 + \ell(v_j)),$$

where $\ell(v_i) = 0$ if it is a sink – here, the max goes over indices $j < i$, so computing $\ell(v_i)$ only depends on the values $\ell(v_1), \ldots, \ell(v_{i-1})$ which were already computed, if we compute these values with increasing value of i. As such this quantity can be computed in $O(n+m)$ time, by scanning the vertices of G according to their topological order, where $m = |\mathsf{V}(\mathsf{G})|$. The running time follows since we scan an edge only once during the algorithm execution.

23.C. (40 PTS.) Monotone variant longest stuck.

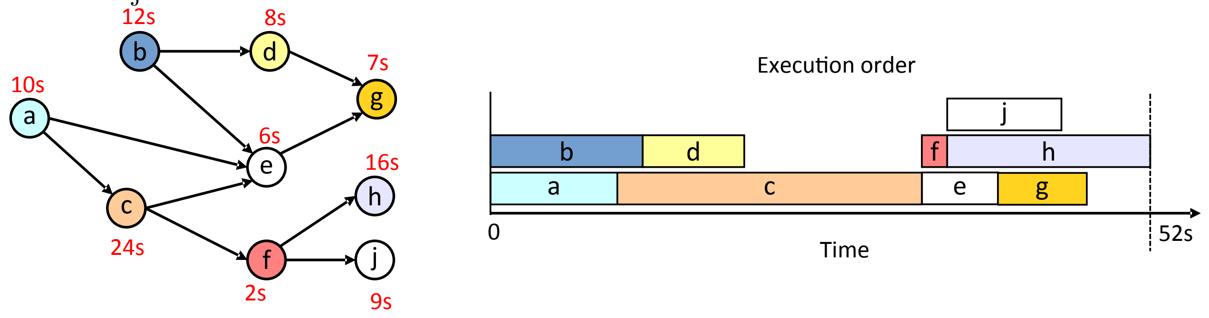
You are given as input n, a list of obstacle locations, and a target cell. Describe and analyze an algorithm to determine the initial locations of the robots that would maximize the number of turns before either a robot reaches the target, or both robots get stuck (the running time of your algorithm should be polynomial in n).

Solution:

The solution follows (**23.B.**), except that we add configuration to the set X. Specifically, let X' be the set of all sinks in H. The set of sinks can be computed in linear time by scanning the graph, and marking all the states that do not have any outgoing edges out of them. Let $Y = X \cup X'$, where X is defined in (**23.A.**). We connect all the vertices of Y to w (except w, naturally). We are now back to the problem of part (**23.B.**) of computing the longest path in the resulting DAG (which would end in w). This overall takes $O(n^4)$ time, as before.

24 (100 PTS.) Run DAG run!

Suppose we are given a DAG G with n vertices and m edges. The vertices represent jobs and whose edges represent precedence constraints. That is, each edge (u, v) indicates that job u must be completed before job v begins. Each node v also has a weight $T(v)$ indicating the time required to execute job v.



For all the following questions, you can assume that you have as many processors as you want (that work independently in parallel).

24.A. (30 PTS.) Describe an algorithm to determine the shortest interval of time in which all jobs in G can be executed.

Solution:

The shortest interval will take at least as long as the maximum weighted path in the DAG. That is the path with the maximum sum of weights $T(v)$ of the vertices in the path. Any shorter interval would violate the precedence constraints. In the example, the solution is $10 + 24 + 2 + 16 = 52$, with path $a \to c \to f \to h$.

The shortest interval will take at most as long as the maximum weighted path in the DAG i.e., all jobs can be finished in this interval because the jobs on any path do no violate precedence constraints. It is easy to show by contradiction that if any job cannot be finished within this interval, then there must be a path with a larger sum of weights.

Again, we can use dynamic programming to find the maximum weight path in linear time by topologically sorting the graph. Define $MWP(v)$ as the maximum weighted path ending in v.

```
// Maximum weight path.
MaximumWeightPath($G$):
  for each vertex $v \in V$ in topological order do
    $MWP(v) \leftarrow T(v)$
    for each edge $(u, v) \in E$ do
      $MWP(v) \leftarrow \max\{MWP(v), MWP(u) + T(v)\}$
  return $\max\{MWP(v) \mid v \in V\}$
```

Topological sort runs in $O(m+n)$ using DFS and the above algorithm runs in $O(m+n)$ since it touches every vertex and every edge exactly once.

24.B. (40 PTS.) Suppose the first job starts at time 0. Describe an algorithm to determine, for each vertex v, the earliest time when job v can begin.

Solution:

The earliest start time of job $v \in V$ is the shortest interval that is needed to finish all precedents of v. Hence, the problem is just to compute the maximum weighted path that ends in v and then subtract $T(v)$. The algorithm is thus similar to part A with a minor difference shown below. It runs in $O(m+n)$ time. Define $EST(v)$ as the earliest start time of v.

```
// Earliest start time of all jobs.
EarliestStartTime($G$):
  for each vertex $v \in V$ in topological order do
    $EST(v) \leftarrow 0$
    for each edge $(u,v) \in E$ do
      $EST(v) \leftarrow \max\{EST(v), EST(u) + T(u)\}$
  return $EST(v)$ for all $v \in V$
```

24.C. (30 PTS.) Now describe an algorithm to determine, for each vertex v, the latest time when job v can begin without violating the precedence constraints or increasing the overall completion time (computed in part (A)), assuming that every job that has precedence on v starts at its earliest start time (computed in part (B)).

Solution:

The latest start time of job $v \in V$ should guarantee that the complete time of this job should not be later than the latest start time of any of its followers, i.e. any $u \in V$ where $(v,u) \in E$. Define S as the shortest interval computed in part A. The idea is similar to parts A and B. However, we work our way back from the sinks. The algorithm provided below takes $O(m+n)$ time.

```
// Latest start time of all jobs.
LatestStartTime($G$):
  for each vertex $v \in V$ in reverse topological order do
    if $v$ is sink then
      $LST(v) \leftarrow S - T(v)$
    else
      $LST(v) \leftarrow \infty$
    for each edge $(v,u) \in E$ do
      $LST(v) \leftarrow \min\{LST(v), LST(u) - T(v)\}$
  return $LST(v)$ for all $v \in V$
```