CSOR W4231: Analysis of Algorithms (sec. 001) - Problem Set #5

Hongmin Zhu (hz2637) - `hz2637@columbia.edu`

November 18, 2019

Problem 1

Algorithm:

Apply *Breadth First Search* algorithm to find a path that connects all the nodes with a connected component of the graph G, and since nodes from the same connected component are reachable from each other, the output *new list* is the same for every node within that connected component. And the graph G may contain many connected components, so outside the *Breadth First Search*, there should be a *for-loop* to go over every possible start point for each connected component.

Pseudo-code:

```
GETLISTS ($G$)
for each $n \in G$
    $color[n] \leftarrow$ white
$lists \leftarrow null$
for $i \leftarrow 1$ to $n$
    $newlist \leftarrow null$
    if $color[i] ==$ white or $newlist == null$ then
        $Q \leftarrow i$
        $color[i] \leftarrow$ black
        while $Q \neq \emptyset$ do
            $u \leftarrow$ Dequeue(Q)
            $newlist \leftarrow$ add node $u$
            for each $v \in Adj[u]$ do
                if $color[v] ==$ white then
                    Enqueue($Q, v$)
        for each $u$ in $newlist$ do
            $lists[u] \leftarrow newlist$
```

```
17  return lists
```

Analysis:

Correctness. For every starting point we find, we use *Breadth First Search* to find all the nodes within this connected component, and we add these nodes to a list, since every node within the connected component is reachable from each other, output new list for every node is the same. So after the *Breadth First Search* is over, we will go over the nodes in the list, and generate output new list for every one.

To prevent duplication, we use *colorarray* to label each node, if the node is undiscovered, the color will be white, otherwise, the node will be black. Also, in the outer *for-loop*, we use this *colorarray* too, to skip the discovered node, and run *Breadth First Search* only for the starting point for a new connected component.

Time Complexity. The first *for-loop* takes $O(n)$ time to initial the *colorarray*, which is the same for every possible situation.

The worst case is the nodes in the graph G are all connected to each other, the running time for *Breadth First Search* is $O(n + e)$, the *for-loop* for creating output lists takes $O(n)$, and since the nodes are all connected to each other, there is only one starting point for the outer *for-loop*, so the running time is actually the time to do one *Breadth First Search.* The worst case running time is $O(n + e)$.

The best case is the nodes in the graph G are all disconnected to each other, so the running time to do *Breadth First Search* is constant, which is also the running time for creating output lists, the entire *for-loop* takes $\Theta(n)$ time, the number of edge here is 0, so the running time is equivalent to $\Theta(n + e)$, which means the lower bound for this algorithm is $\Omega(n + e)$

So, in summary, the algorithm has the optimal running time, which is $\Theta(n + e)$.

Problem 2

(a)

Proof is done by contradiction. Suppose graph G is bipartite and it has a cycle of odd length. First, let's say the odd cycle connects the vertices named $u_1, u_2, ..., u_k$, and k is an odd number. And we know that graph G is bipartite, the vertices from G belong to 2 sides, and there is no edge inside these 2 sides. Therefore, vertices from the odd cycle can also be split into 2 sides, let's say vertices $u_1, u_3, u_5..., u_k$ belongs to one side, and $u_2, u_4, ..., u_{k-1}$ belong to the other side. In order to form a cycle, vertices u_1 and u_k must be connected, which means there is an edge in one side of vertices connecting u_1 and u_k, which contradicts the property of bipartite of having no edges in each side of the vertices. So our assumption of graph G being bipartite and having a cycle of odd length is not correct. We can then say if a graph contains a cycle of odd length then it is not bipartite.

(b)

Algorithm:

Apply a modified BFS algorithm to identify if the graph is bipartite. The algorithm will color each node the different color from its parent, so in the *for-loop* of coloring adjacent nodes inside BFS, if there is a node that has the same color as its parent, we know that the graph is not bipartite, then we run a DFS algorithm and maintain a stack to get the cycle, otherwise, we will add nodes to its side (N_1 or N_2).

Pseudo-code:

```
isBipartite ($G$, $s$)
1   for each $n \in G$
2       $color[n] \leftarrow$ gray
3   $N_1, N_2 \leftarrow null$
4   $flag \leftarrow$ true
5   $Q \leftarrow s$
6   $color[s] \leftarrow$ white
7   $N_1 \leftarrow$ add $s$
8   while $Q \neq \emptyset$ do
9       $u \leftarrow$ Dequeue(Q)
10      for each $v \in Adj[u]$ do
11          if $color[v] ==$ gray
12              if $color[u] ==$ white then
13                  $color[v] ==$ black
14                  $N_2 \leftarrow$ add $v$
```

```
         else
            $color[v]$ == white
            $N_1 \leftarrow$ add $v$
      else
         if $color[v]$ == $color[u]$ then
            $flag \leftarrow$ false
            break
if $flag$ == false then
   for each $n \in G$
      $color[n] \leftarrow$ white
      $parent[n] \leftarrow$ NIL
      $stack \leftarrow null$
      $continue \leftarrow$ true
      getCycleDFS $(G, stack, continue, s)$
      for each $n \in stack$
         $cycle \leftarrow n$
         return $cycle$
else
   return $N_1, N_2$
```

```
getCycleDFS $(G, stack, continue, u)$
if $continue$ == false
   return
$color[u] \leftarrow$ gray
$stack \leftarrow u$
for each $v \in Adj[u]$ do
   if $color[v]$ == white then
      $parent[v] \leftarrow u$
      $stack \leftarrow v$
      $color[v] \leftarrow$ gray
      getCycleDFS $(G, stack, continue, v)$
   else if $color[v]$ == gray then
      if $parent[v] \neq u$ then
         $continue \leftarrow$ false          // we find a cycle
$color[u] \leftarrow$ black
```

Analysis:

Correctness. We apply a modified BFS to check if the graph is bipartite, and along the way, we add nodes to two different sides (N_1 and N_2). The nodes in N_1 will be colored white, and nodes in N_2 will be colored black. As all of the neighbors of node u should have the opposite color as u, so if we find a neighbor v with the same color as u, we know find a cycle.

We apply a modified DFS to output the cycle, we maintain a stack to keep all nodes that are visited but not done with processing (color gray), and if at some point of the recursion, we find a node v of u that has the color of gray and is not u itself, we will stop the recursion. The nodes we have now in the stack are nodes in the cycle.

Time Complexity. The dominant parts of the algorithm are BFS and DFS, and they both have the running time of $O(n + e)$, so the total running time of this algorithm is $O(n + e)$.

Problem 3

(a)

Algorithm:

We can construct a directed graph G from set V and set C, the variables in the set V are nodes in the directed graph and non-strict inequalities in the set C are the edges in the directed graph. The inequality $x_1 \leq x_3$ can be interpreted as the directed edge from x_3 to x_1. We will compute the strongly connected components of this graph, and group all of the variables in a strongly connected components together to one node, which can be done in a map. The reason behind this is all nodes in a strongly connected components with relation $\leq$ should have the same value.

Once we have grouped all strongly connected components, we construct a new graph of the components. This time, we see each single node that are not grouped in the last step as a component too. So an edge should connect component A and component B if there are variables $x_i \in A, x_j \in B$ that $x_i \leq x_j$.

We run a DFS search on this new graph again. The inequalities are inconsistent if there is still a cycle in the graph, otherwise, the inequalities are consistent.

Pseudo-code:

```
isConsistent ($G$)
  1   call DFS(G) to compute finishing times $f[u]$ for each vertex u
  2   $G^T \leftarrow$ traverse all adjacency lists and reverse $G$
  3   for each $u \in G^T.V$
  4       $u.color$ = White
  5       $u.\pi$ = NIL
  6   $time = 0$
  7   for each $u \in G^T.V$ in order of decreasing $f[u]$
  8       if $color[u]$ == white then
  9           $map \leftarrow (u, u)$       // key is each original node, value is grouped node
  10          Modified-DFS-Visit ($G^T$,$u$,$map$)
  11  construct a new graph $G' = (V', E')$. The nodes are each grouped component
stored in $map$, go over each inequality and add an edge to component A and compo-
nent B if there are variables $x_i \in A, x_j \in B$ that $x_i \leq x_j$.
  12  call DFS($G'$) to check if there are cycles
  13  return $false$ if a cycle is detected
  14  return $true$ if no cycle is detected
```

Modified-DFS-Visit (G,u,map)

```
1   $color[u] \leftarrow$ Gray
2   $time \leftarrow time + 1$
3   $d[u] \leftarrow time$
4   for each $v \in Adj[u]$ do
5      if $color[v] ==$ White then
6         $\pi[v] \leftarrow u$
7         $map \leftarrow (v, u)$
8         MODIFIED-DFS-VISIT ($G$,$v$,$map$)
9   $color[u] \leftarrow$ Black
10  $f[u] \leftarrow time \leftarrow time + 1$
```

Analysis:

Correctness. The first part of the algorithm is to compute the strongly connected components, then we group the strongly connected components to a single node and construct a new graph. If there is a cycle in the new graph, then it must contain a strict inequality otherwise the nodes would have been merged to a single node. Since if there is a cycle, a variable will be strictly smaller than itself along the results of inequalities.

Time Complexity. For finding the strongly connected component part, the DFS takes $O(n + m)$ time, traversing the lists to reverse graph takes $O(n + m)$ time, so the finding strongly connected components takes $O(n + m)$ time. For the second part of checking consistency, we also call a DFS search whose worst-case running time is $O(n + m)$. Therefore the algorithm has running time of $O(n + m)$.

(b)

Algorithm:

After the algorithm in (a), we have a graph G'. In the final step of the algorithm, we ran a DFS search, so we assume that G' is in the form of adjacency list and the list is ordered by topological order. Then we go over this graph, and assign values to each component. The value is decided based on its neighbor's value.

Pseudo-code:

```
GETSOLUTION ($G$)
1   $value \leftarrow$ an array initialized to all 1, the size equals to the size of the compo-
nents.
2   for $i \leftarrow 1$ to size of $G.V$
3      for $j \leftarrow Adj[i]$ do
4         if $i < j$ then
```

```
5            $value[j] = max(value[j], value[i] + 1)$
6        else then
7            $value[j] = max(value[j], value[i])$
8    for component $i \in G$
9        assign $value[i]$ to all variables of component $i$
```

Analysis:

Correctness. If the component has no incoming edge, then it should have value of 1. For inequality $i < j$ means component j must be at least one more than component i. While $i \leq j$ means the component j must be at least as large as component i.

Time Complexity. Creating *value* array takes $O(n)$ time, lines 2-7 traverse the graph, so it takes $O(n + m)$ time. The final step of assigning values to each variable takes $O(n)$ time. So in total, the algorithm has the running time of $O(n + m)$.

Problem 4

(a)

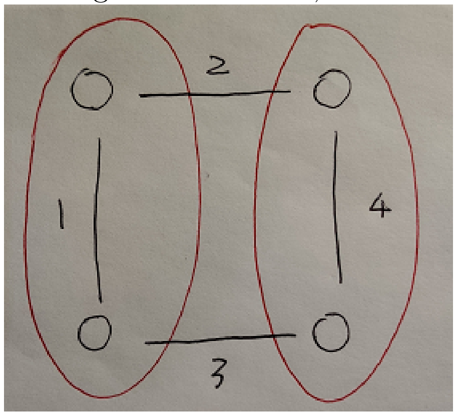
Proof is done by contradiction. Assume there are two minimum spanning tree of the graph G, MST_1, MST_2. These two MSTs have the same nodes but the edges are not completely the same. Let's say MST_1 has a set of edges E_1 and MST_2 has a set of edges E_2. Since E_1, E_2 are not completely the same, there must be an edge e_1 that belongs to E_1 but not E_2. If we add this e_1 to MST_2, there will be a cycle inside MST_2. The largest edge e' of this cycle can not be inside MST_2 since MST_2 is the minimum spanning tree. There are two cases: 1) $e' = e_1$, which means the largest edge of this cycle is in MST_1. 2) $e' \neq e_1$, which means the largest edge of this cycle is in MST_2. Both cases contradict the fact that MST_1 and MST_2 are minimum spanning trees and the largest edge of the cycle should not be inside the minimum spanning tree. Therefore, we can say the graph has a unique minimum spanning tree.

(b)

Assume T is a minimum spanning tree of graph G that does not contain edge (u, v). Since T is a spanning tree, there must be a an edge (x, y) connecting the two partitions. We take (x, y) out of the tree and add the edge (u, v). Since (u, v) is one of the minimum-weight edges across the partition, there will be two cases: 1) if edge $w(u, v) == w(x, y)$, then the new tree T' containing edge (u, v) is also a minimum spanning tree. 2) if edge $w(u, v) < w(x, y)$, then our assumption is not correct, the new tree containing edge (u, v) should be the minimum spanning tree.

(c)

The algorithm will fail, consider the following example:



if we apply the divide-and-conquer algorithm, the minimum weight will be 1+4+2 = 7, however, the correct minimum spanning tree should have the minimum weight should have $1 + 2 + 3 = 6$. The problem with the divide-and-conquer algorithm is that we cannot guarantee the algorithm will find the optimal way to divide the nodes each time.

Problem 5

(a)

Algorithm:

Since there is only one negative cost edge (u, v), we can first make its cost to be 0. Then we can use Dijkstra's algorithm to compute the shortest path from source s to vertex u and v. Let's say the shortest path from s to u is d_1, and shortest path from s to v is d_2, then what we do is to compare $d_1 + (u, v)$ (the original negative one) and d_2, the smaller one is the shortest path from source s to v.

Pseudo-code:

```
getPath ($G$,$w$,$s$,$u$,$v$)
    1    $cost \leftarrow w(u, v)$
    2    $w(u, v) \leftarrow 0$
    3    $Dijkstra(G, w, s)$
    4    return $min(d[u] + cost, d[v])$
```

Analysis:

Time Complexity. The algorithm has the same time complexity as Dijkstra's algorithm, which is $O(ElogV)$ if we use *heap* as the data structure.

(b)

Algorithm:

We see each currency is a node and each way to exchange between currencies is an edge, the edges are weighted by the exchange rate, which are values stored in the table T. So we can build a weighted directed graph $G = G(V, E)$, V is the set of currencies and the edges $(v_i, v_j), (v_i, u_j)$ are ways to do the exchange.

The goal is to find a sequence such that the product of the weights are larger than 1, we do the following transformations to make the problem solvable with the algorithm we learned in class:
$T[i_1, i_2] * T[i_2, i_3] ... T[i_{k-1}, i_k] * T[i_k, i_1] > 1$

$$\frac{1}{T[i_1, i_2]} * \frac{1}{T[i_2, i_3]} ... \frac{1}{T[i_{k-1}, i_k]} * \frac{1}{T[i_k, i_1]} < 1$$

$$log\frac{1}{T[i_1, i_2]} + log\frac{1}{T[i_2, i_3]} ... log\frac{1}{T[i_{k-1}, i_k]} + log\frac{1}{T[i_k, i_1]} < 0$$

so we can update the values in table T with $w(v_i, v_j) = -log(T[i_i, i_j])$, the goal is then changed to find a negative cycle in graph G.

We know that Bellman-ford algorithm can detect whether there is a negative cycle in the graph, so we will use it on graph G, if a negative cycle is detected, then we will use a recursion to get the sequence.

Pseudo-code:

```
getSequence ($T$)
  1  $G, w, s \leftarrow$ construct a graph from table $T$
  2  if Bellman-Ford ($G$,$w$,$s$) is $true$ then
  3     return $false$
  4  for each edge $(v_i, v_j) \in G.E$ do
  5     if $v_j.d > v_i.d + w(v_i, v_j)$ then
  6        $sequence \leftarrow empty$
  7        getList ($G, v_i, v_i.\pi$)
  8  return $sequence$

getList ($G$,$s$,$v$,$sequence$)
  1  if $v == s$ then
  2     $sequence \leftarrow$ add $s$
  3  else if $v.\pi ==$ NIL then
  4     return
  5  else getList ($G$,$s$,$v.\pi$,$sequence$)
  6     $sequence \leftarrow$ add $v$
```

Analysis:

Time Complexity. The first thing we need to do is to construct the graph from table T, this step needs to go over all the pairs of currencies which takes $O(n^2)$ time. The Bellman-ford algorithm takes $O(VE)$ time where $|V| = n, |E| = n^2$, so it takes $O(n^3)$ time. If the negative cycle is detected, we will go over the edges and get the sequence, the lines 4-7 take $O(n^2)$ time in the worst case to check all the edges, so in total, the algorithm proposed has running time of $O(n^3)$.