CSCI-GA.1170-001/002 Fundamental Algorithms December 5, 2019

Solutions to Problem 1 of Homework 11 (9+8 points)

Name: jingshuai jiang (jj2903) *Due: Thursday, 5 pm on December 5*

Collaborators: NetID1, NetID2

You are given a set V of n variables, $\{x_1, \ldots, x_n\}$ and a set C of m strict inequalities, i.e, of the form $x_i < x_j$. The set C of inequalities is called *consistent* over the set of positive integers $\mathbb{Z}^+$ iff there exists an assignment of positive integer values in such a way that it satisfies all the inequalities. For example, $x_1 < x_2, x_2 < x_3, x_3 < x_1$ is not consistent but $x_1 < x_2, x_2 < x_3$ is consistent. The goal of this question is to formulate an algorithm to determine whether C is consistent or not and then assign minimum possible value that satisfies these constraints over the set of positive integers.

(a) (4 points) Give an $O(m + n)$ algorithm to determine whether the set C of inequalities is consistent. State precisely the asymptotic running time of your algorithm in terms of n and m. Argue the correctness of your algorithm. (**Hint**: Represent this as a graph and use one of the graph algorithms. You are not allowed to assign weights to edges.)

Solution: In this problem, we should think of these variables as vertices of graph and these inequalities as edges. If this set C of inequalities is consistent, then our graph should be acyclic. Then we can turn this problem into a graph problem, wihch determines whether there is a cycle in this graph.

```
1 DFSconsistent(G)
2     for each u in V
3           u.color = white
4           u.π = NIL
5     for each u in V
6           if u.color == white
7                 if IsConsistent(G,u)==false, return false
8     return True
```

```
1 IsConsistent(G, u)
2     u.color = GRAY
3     for each v in G.adj[u]
3           if v.color == GRAY, return false;
4           if v.color ==white
5                 if IsConsistent(G,v)==false, return false;
7     u.color = black
8     return true;
```

In this code i am using an dfs search. If we met a back edge then there is a cycle in this graph. That means it is not consistent. By using the three color, we can know whether we have a backedge by pending on whether we met a gray node. The running time will be O(m+n). □

(b) (5 points) Give an $O(m+n)$ algorithm Assign-Value to find the minimum possible solution over the set of *positive* integers, assuming C is consistent. Briefly argue the correctness and the runtime of your algorithm. (**Hint**: Use topological sort.)

```
1 Driver(G)
2     for v in V
3           v.value = 1
3     Assign-Value(G)
```

Solution:

```
1 reverse-topological-sort(G)
2     call DFS(G) to computer finishing time v.f for each vertex v
3     as each vertex is finished, insert it onto the front of a linked list
4     reverse this linked list
```

```
1 Assign-Value(G)
2     list = reverse-topological-sort(G)
3     for node from head to tail
4           if node.adj is not NIL,then v.value = max_{u∈adj[v]} u.value + 1
```

We turn this problem into a graph problem. The symbol $x1 < x2$ means there is an edge from $x2$ to $x1$. And we use topological sort to determine which node finish first.

1: if the node has no adjcent list, which means it has no constraints on its value. Then we just need to assign the samllest value to it.

2: If the node does have adjcent list, we need to check all its neighbours and since it should be bigger than its neighbours . Then we just need to take the maximum value of all its neighbours and add 1 to this maximum value. Since we have used the topological sort, we have already calculated the value of this nodes neighbours.

The runtime of this algorithm will be the same as topological sort, which is O(V+E) = O(m+n). □

While we spoke about strict inequalities, we can also look at weak inequalities. These typically involve the operators $\leq, \geq$. Indeed, a strict equality of the form $x_i < x_j$ can be rewritten as $x_i \leq x_j - 1$ provided x_i, x_j are over the set of integers. Therefore, one can look at the generalized case where C contains inequalities of the form $x_i \leq x_j - c$ where $c \geq 0$.

(c) **(Extra Credit)** (8 points) Consider the above generalization where the set C contains inequalities of the form $x_i \leq x_j - c$ where $c \geq 0$. Design an $O(m+n)$ algorithm to determine if the set C is consistent. Further, design an $O(m+n)$ algorithm that provides the least possible satisfying assignment, provided C is consistent. Argue correctness of your algorithms. You may describe the algorithm in words. (**Hint**: Consider adding edge weights. Use the SCC algorithm.)

Solution: Problem transformation: The variables will be viewed as vertices of graph. The weak inequalities will be viewed as edges among these vertices. And the constant c will be viewed as the weight of these edges.

We use these rules to build a graph G

Consistency: First we run DFS on this newly built graph.

During the process of DFS, when we traverse each node v in the adjlist of vertice u, if this v.color is gray, then we will know there is a cycle in this graph. If not all the weights on these edge are equal to zero. Then this graph is not consistent. Otherwise we continue our DFS. If there is no this unusual cycle then it is consistent

Assign-Value:

□

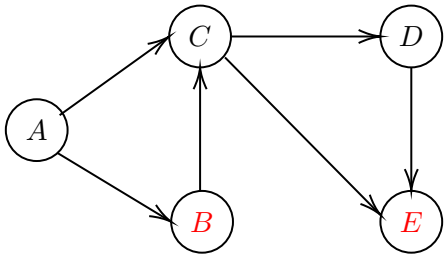
CSCI-GA.1170-001/002 Fundamental Algorithms December 5, 2019

Solutions to Problem 2 of Homework 11 (10 Points)

Name: jingshuai jiang (jj2903) *Due: Thursday, 5 pm on December 5*

Collaborators: NetID1, NetID2

Let us assume that you are given a directed graph $G = (V, E)$. Each $v \in V$ is assigned $v.w \in \{0, 1\}$. Let the cost of a path $v_1, \ldots, v_k$ be the number of times the path switches in weights. More concretely, $\sum_{i=1}^{k-1} |v_i.w - v_{i+1}.w|$. Note that some of these nodes might be the same, i.e, the path can revisit nodes already visited. Consider the following graph:



Let the nodes labeled in red have a weight 1. Therefore, $A.w = C.w = D.w = 0$ and $B.w = E.w = 1$. The path $A \to C \to D \to E$ has a cost of 1 while the path $A \to B \to C \to D \to E$ has a cost of 2.

(a) (5 points) Design an algorithm that takes as input an *acyclic* graph $G = (V, E)$ and returns the maximum cost of a path in G. This algorithm should run in time $O(|V| + |E|)$. Assume that G is represented as an adjacency list. Briefly argue correctness of the algorithm. (**Hint**: Begin by using an algorithm we covered in class. Then, define $M[v]$ to be the maximum cost of a path starting at v. Use a simple dynamic programming idea to construct this array M.)

Solution:

```
1 MAXIMUM-COST(G)
2     call DFS(G) to computer finishing time v.f for each vertex v
3     as each vertex is finished, insert it onto the front of a linked list
4     from vertices v from tail to head
5         if v has no adjcent vertices,then M[v] = 0
6         else M[v] = max_{u∈v.adj} {M[u] + |v.w − u.w|}
7     return the maximum value of all M
```

Firstly we use topological sort to sort this whole graph. Then from tail to head we construct the array M. If the vertice v has no adjcent veritces, then this v must be end to a path. Then the maximum cost of a path starting at this end point is 0. If this vertice v has some adjcent vertices, we will choose the maximum cost among its all adjcent veritces. Since we are using

topological sort. Then the maximum cost of a vertice's adjcent vertice must be calculated before this vertice. After that we just need to return the maximum value among all the M[V].

The runtime of this algorithm will be $O(|V| + |E|)$.

□

(b) (5 points) Now generalize your algorithm to solve this problem on any *arbitrary* graph. Justify the correctness of your algorithm. (**Hint**: Use another algorithm from class to begin.)

Solution:

1 GENERALIZATION(G)
2 calculate the strongly connected parts(scc) of this graph G
3 **if** any part of these scc has both a cycle and weight variation in this cycle,**then return** ∞
4 **else** think of the scc as a vertice, the weight of this vertice should be the same
with the vertices inside this scc
5 use the algorithm in part (a) to calculate the maximum cost of this newly built graph
6 **return** the maximum of M

Correctness: It is obvious, if a graph contains a cycle and contains weight variation in this cycle, the final cost should be infty. Because the maximum path should contain this infty loop of this cycle.

If these strongly connected parts do not have weight variation in the cycle. Then it is easy too. We will think of this part to be a single vertice, with its weight equal to the weight of vertices inside this part. Then this problem has been turned into the same problem in part a. Then we can just use the algorithm in part a to solve it. □

CSCI-GA.1170-001/002 Fundamental Algorithms December 5, 2019

Solutions to Problem 3 of Homework 11 (7 points)

Name: jingshuai jiang (jj2903) *Due: Thursday, 5 pm on December 5*

Collaborators: NetID1, NetID2

For purposes of this question we have a connected, undirected graph $G = (V, E)$ such that $|E| = m, |V| = n$. Each edge has an assigned weight.

(a) (3 points) Assume that all edge weights are equal to the same number w. Design the fastest algorithm you can to compute the MST of G. Argue the correctness of the algorithm and state its run-time. Is it faster than the standard $O(m + n \log n)$ run-time of Prim?

Solution:

```
1   for each v in G.v
2       v.π = NIL and add v to set C
3   add startpoint s into Q and remove s from C
4   while Q is not empty
5       u = Q.dequeue()
6       for each v in u.adjlist
7           if v is in C
8               Remove v from C
9               and add v to Q
10              v.π = u
```

This code is actually using a variate of BFS. Since all the edges have the same weight, we do not need to think about how to minimize the whole weight. We just need to add all nodes into our tree for one time.

Before we start, we built a set of C, which is a copy of all the nodes in G.

Then We start from the startpoint s. Firstly we will remove s from C. And search its adjcent list. If the neighbour is still in C, then we add it to the queue and remove it from C and keep searching.

In this algorithm each node will be add into the queue for only one time. And the edge will be searching for one time. Then the total runtime is $O(m + n)$.

It is faster than the standard $O(m + nlogn)$ □

(b) (4 points) Now assume the all the edge weights are equal to w, except for a single edge $e' = (u', v')$ whose weight is w' (note, w' might be either larger or smaller than w). Show how to modify your solution in part (a) to compute the MST of G. What is the running

time of your algorithm and how does it compare to the run-time you obtained in part (a) (or standard Prim)? (**Hint**: Consider different cases, first based on whether e' is needed in the MST or not. Then, based on if $w' > w$ or not.)

Solution: Firstly we should decide whether we need this edge. By scanning all the edges we find this special edge e'. If any of the two vertices connected by this edge can only be reached by this edge, then this edge is obviously needed no matter whewher it is bigger than others. Then we just need to ran the algorithm in Part 1.

Secondly if this edges is not a necessary, then we should compare its weight w' to w if it is bigger than the w, then we just delete this edge from both of its edge's adjcentlist. And run the algorithm from part1.

Thirdly if the weight of this edges w' is smaller than w. Then we choose one of its vertices to be the start point and use the part1 solution.

Runtime:In our solution, the process of find this special edge takes time $O(|E|)$, and decide whether we need it takes $O(1)$ time. The process of deleting this bigger edge from the adjcentlist takes the worst time $O(|E|)$. And the process of part 1 takes time $O(|E| + |V|)$ The total runtime will be $O(3|E| + |V|) = O(|E| + |V|)$. It is better than standard Prim. But it should be worse than algorithm in part 1 because part1 is one part in part2. But in asymptotic meaning they are in the same level. □

CSCI-GA.1170-001/002 Fundamental Algorithms December 5, 2019

Solutions to Problem 4 of Homework 11 (14 points)

Name: jingshuai jiang (jj2903) *Due: Thursday, 5 pm on December 5*

Collaborators: NetID1, NetID2

For purposes of this question we have a connected, undirected graph $G = (V, E)$ such that $|E| = m, |V| = n$. Assume all edge weights in G are integers from 1 to w.

(a) (3 points) Show how to modify Prim's algorithm to achieve running time $O(m+nw)$. Hence, if $w = O(1)$, you get optimal time $O(m+n)$. Recall that Prim's algorithm makes n EXTRACT-MIN operation and m DECREASE-KEY operation. Clearly, explain how your modification would implement EXTRACT-MIN and DECREASE-KEY and what the runtime is. (**Hint**: Priority queues are typically implemented using a heap. However, in this example use a doubly linked list as the underlying data structure. How many such lists do you need?)

Solution: In this problem we will have w doubly linked list for values between 1 to w. The ith list will store nodes whose value is i.

For ExtractMin: We check if the 1st list is empty. If not,extract one from the list. If it is empty, we check the 2nd list. Then untill the wth list. This runtime will be $O(w)$

For DecreaseKey: We remove the node from its orginal list and add it to its new list.

Runtime: The ExtractMin takes time $O(w)$, the DecreaseKey takes time $O(1)$, Since the whole time of prim is $O(n \cdot T_{ExtractMin} + m \cdot T_{DecreaseKey})$, then it is $O(m + nw)$. □

(b) (6 points) Show how to modify the above algorithm further to achieve a running time of $O(m+n \log w)$. Clearly, explain how your modification would implement EXTRACT-MIN and DECREASE-KEY and what the runtime is. (**Hint**: Implement the priority queue using the linked lists but by storing them in an efficient fashion. You may find it useful to employ a Fibonacci Heap in a black box fashion. You may use the Wikipedia page of Fibonacci Heap to retrieve the running time of the various operations.)

Solution: In this problem we will have w linked list for values between 1 to w. The ith list will store nodes whose value is i. And the head node of these w list will be store in a Fibonacci heap

For ExtractMin: We use the heap to do the extract thing. Extract the minimum value in a W nodes heap takes time $O(logw)$.

For DecreaseKey: We remove the node from its orginal list and add it to its new list.

Runtime: The ExtractMin takes time $O(logw)$, the DecreaseKey takes time $O(1)$, Since the whole time of prim is $O(n \cdot T_{ExtractMin} + m \cdot T_{DecreaseKey})$, then it is $O(m + nlogw)$. □

Recall that the runtime of Kruskal's Algorithm is $O(m \log m + m \cdot T_{find} + n \cdot T_{union})$. There are set-disjoint data structures, as discussed in class, where $T_{find} = T_{union} = \alpha(n)$ where $\alpha(n)$ is a certain function much smaller than $\log n$.

(c) (3 points) Now assume $w = n$, so that the previous solution in part (a) is no longer faster than standard. Show how to modify Kruskal's algorithm, so that your modified algorithm now takes time $O(m \cdot \alpha(n))$, instead of $O(m \log n)$.

Solution:

□

(d) (2 points) What is the largest w, as a function of $\alpha(n)$, for which you can still maintain the run-time in part c?

Solution:

□