Solutions to Homework 5

Debasish Das
EECS Department, Northwestern University
ddas@northwestern.edu

1 Problem 4.17

We want to run Dijkstra algorithm whose edge weights are integers in the range 0,1,...,W where W is a relatively small number.
a)Using a bucket implementation (also known as Dial's implementation) Dijkstra algorithm can be made to run in O($W|V| + |E|$). Idea is to find an upper bound on shortest distance labels. Since the maximum edge weight is W and a vertex can be updated at most $|V| - 1$ times, we get a bound of O($W|V|$) on shortest path distances. Using the property of Dijkstra algorithm that the distance labels that are designated permanent are non-decreasing we can present the following algorithm

```
procedure Efficient-Dijkstra1(G,l,W,s)
Input:  Graph G=(V, E), edge weights l_e,
    Max edge weight W, vertex s ∈ V
Output:  Shortest path distance labels
for all vertices v ∈ V:
  dist(v) = ∞
  prev(v) = nil
  bucket(v) = nil
Create an array B of size W|V|:
  B[i] keep vertex of distance label i
B[0] = s,dist(s)=0,bucket(s)=0
index = 0
while index ≠ W|V|
  Increment index:
    B[index] ≠ ∅
  u = B[index]
  for all edges (u,v) ∈ E:
    temp = dist(v)
    if dist(v) > dist(u) + l(u,v):
       dist(v) = dist(u)+l(u,v)
       prev(v) = u
       B[dist(v)] = v
       if temp ≠ ∞:
          Remove v from B[temp]
```

Doubly linked list should be used for array B, which allows us to do the following operations in O(1) time:
(1) Checking whether a bucket is empty or nonempty
(2) Deleting an element from a bucket
(3) Adding an element to the bucket
(b)The algorithm is same as the algorithm shown on Page 110. We claim the following lemma

Lemma 1 *If edge weights are integers in the range of 0,1,...,W, where W is a relatively small number then at each iteration of dijkstra algorithm, there can be at most $|W|$ elements in the heap*

Proof: We use induction to prove our claim. We refer to the minimum key of the heap as min-k and the maximum key of the heap as max-k.
Initialization: On the first iteration of Dijkstra's algorithm we can add at most W elements in the heap as number of edges can be at most W. Therefore max-k − min-k = W
Hypothesis: For i-th iteration of Dijkstra's algorithm we have the following max-k^i − min-k^i ≥ W.
Induction: During the i+1-th iteration, we choose the minimum key from the heap which is min-k^i. Suppose the respective vertex is v. In worst case we will update all the vertices connected to v and update the keys in the heap. Since edge weights are bounded by $|W|$, new min-k and max-k of the heap at i+1-th iteration is given by

$$\begin{aligned} \text{max-k}^{i+1} &= max(\text{min-k}^i + W, \text{max-k}^i) \\ \text{min-k}^{i+1} &\geq \text{min-k}^i \end{aligned}$$

From the above equations we can establish that

$$\begin{aligned} \text{max-k}^{i+1} - \text{min-k}^{i+1} \leq W \text{ or} \\ \text{max-k}^{i+1} - \text{min-k}^{i+1} \leq \text{max-k}^i - \text{min-k}^i \leq W \end{aligned}$$

which validates our claim. □
Now since at every iteration of Dijkstra's algorithm there can be at most $|W|$ elements in the heap, O((|V|+|E|) log V) bound changes to O((|V|+|E|) log W). Note that this is the property of this problem which leads to an efficient bound on the complexity.

2 Problem 4.22

Given a directed graph G=(V,E) whose nodes are ports, and which has edges between each pair of ports. For any cycle C in this graph, the profit-to-cost ratio is

$$r(C) = \frac{\sum_{(i,j)\in C} p_j}{\sum_{(i,j)\in C} c_{ij}} \tag{1}$$

The maximum ratio achievable over all cycles is called r^*. Given each edge (i,j) we assign a weight of $w_{ij} = r \cdot c_{ij} - p_j$.
(a)

Lemma 2 *Show that if there is a cycle of negative weight, then $r < r^*$*

Proof: Suppose the graph G contains a cycle of negative weight. For that cycle $\sum_{(i,j)\in C}(r \cdot c_{ij} - p_j) < 0$ which implies that

$$r < \frac{\sum_{(i,j)\in C} p_j}{\sum_{(i,j)\in C} c_{ij}} \tag{2}$$

Therefore the r we choose is a lower bound of r^*. □
(b)

Lemma 3 *Show that if all cycles in the graph have strictly positive weight then $r > r^*$*

Proof: If G contains no negative cycle then for every directed cycle C, $\sum_{(i,j)\in C}(r \cdot c_{ij} - p_j) \geq 0$ which implies that

$$r \geq \frac{\sum_{(i,j)\in C} p_j}{\sum_{(i,j)\in C} c_{ij}} \tag{3}$$

Therefore in this case the r we choose is a upper bound of r^* □

Corollary 0.1 *If G contains a zero weight cycle, then* $r = r^*$

(c)The algorithm we present for this part is a binary search algorithm. Since we know the upper bound and lower bounds on r^* we choose a r from the interval where r is feasible. Upper bound of r is given by R where as lower bound of r is 0. The lower bound is crude and better bound can be obtained. Therefore r is feasible in interval $(0, R)$.

```
procedure max-pcr-cycle(G,C,P,ϵ)
Input:  G = (V,E), Transportation cost C = {c_ij : (i,j) ∈ E}
    Profit P = {p_j : j ∈ V},accuracy ϵ
Output:  r* and corresponding cycle
lb = 0, ub = R
r = (lb+ub)/2
do
temp-r = r
for each edge (i,j) ∈ E
  w_ij = temp − r · c_ij − p_j
flag1 = NegativeCycleDetect(G,W):
  W={w_ij : (i,j) ∈ E}
if flag1 = true:
  Update the lb:
    lb = temp-r
else
  flag2 = ZeroCycleDetect(G,W,Cycle)
  if flag2 = true:
    return temp-r and C
  else Update the ub:
    ub = temp-r
r = (lb+ub)/2
while(|temp-r − r| > ϵ)
```

NegativeCycleDetect(G,W) is straightforward. See Section 4.6.2 for its detail. The idea is to do one more iteration in Bellman-Ford algorithm and keep track whether any shortest path distance labels change in the final iteration or not. If they change, then we have found a negative cycle. If NegativeCycleDetect terminates with true, it implies that shortest distance labels have been correctly obtained. ZeroCycleDetect then checks whether the resultant graph has any zero cycle length cycle or not. If not then we upate upper bound. If there is a zero length cycle then we return the maximum profit-to-cost ratio and corresponding cycle.

```
procedure ZeroCycleDetect(G,W,Cycle)
Input:  G = (V,E) with shortest path distance on vertex,
    W={w_ij : (i,j) ∈ E}
Output:  true if zero weight cycle exists:
       Cycle stores one such cycle
  false if no zero weight cycle exists
Construct a graph G^0 = (V^0, E^0):
  G^0 = V^0
  for each (i,j) ∈ E
    if dist(i) + l(i,j) = dist(j):
       insert (i,j) to E^0
if cycle identified in G^0:
  return true and Cycle
else
  return false
```

Correctness of ZeroCycleDetect comes from the following lemma

Lemma 4 *If G has a zero length cycle then G^0 also has a cycle*

Proof: Let W be a zero length cycle in G. Then

$$\sum_{(i,j)\in W} w_{ij} = 0 = \sum_{(i,j)\in W} dist(i) + w_{ij} - dist(j) \tag{4}$$

Note that over a cycle, dist(i) and dist(j) pairs will cancel each other. But $dist(i) + w_{ij} \geq dist(j)$ (since they are shortest path labels). Therefore $dist(i) + w_{ij} - dist(j) \geq 0$. Therefore each of the terms $dist(i) + w_{ij} - dist(j)$ should be 0 to satisfy Equation 4. Since we kept only those edges in G^0 whose distance label satisfies the above condition, G^0 must have cycle W. □

Runtime analysis of the algorithm: Each iteration of the binary search reduces the feasible search space for r^* by half. Let the number of iterations be n. We have $\frac{R}{2^n} = \epsilon$. Therefore $n = \log \frac{R}{\epsilon}$. Negative cycle detection algorithm has a complexity of O($|V||E|$) where as zero cycle detection takes O($|V| + |E|$). Overall complexity of the algorithm is given by O($|V||E| \log \frac{R}{\epsilon}$)

3 Problem 5.5

(a)

Lemma 5 *If each edge weight is increased by 1, the minimum spanning tree doesn't change*

Proof: Suppose initially the minimum spanning tree was T. After each edge weight is increased by 1, the minimum spanning tree changes to $\hat{T}$. Therefore there will be at least an edge $(u, v) \in T$ but $(u, v) \notin \hat{T}$. Suppose we add edge (u, v) to the tree $\hat{T}$. $T^* = \hat{T} + (u, v)$. Since (u, v) was not in $\hat{T}$ therefore (u, v) must be the longest edge in the cycle C formed in T^*. But since (u, v) is the longest edge it cannot be in the MST $\hat{T}$(We prove this lemma in Problem 5.22). (u, v) is the longest edge and therefore when we decrease each edge weight by 1, (u, v) will still be the longest edge in cycle C formed in T. But longest edge (u, v) can not be contained in MST T. Therefore $(u, v) \notin T$ which is a contradiction. It implies that trees T and $\hat{T}$ are same. □

(b)We show an example in Figure 1 where the shortest path can change due to increase in weight by 1. Before increasing the edge weights, shortest path from vertex 1 to 4 was through 2 and 3 but after increasing

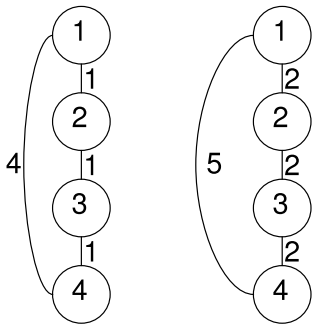

Figure 1: Counterexample for Shortest Path Tree

the edge weights shortest path to 4 is from vertex 1.

4 Problem 5.8

If the graph is directed it is possible for a tree of shortest paths from s and a minimum spanning tree in G not to share any edges. A counterexample is shown in Figure 2 (taken from Xi Chen's solution). MST will

have the edges {(3,1),(3,2),(2,4)}. Shortest path tree rooted at vertex 1 has the edges {(1,4),(1,2),(4,3)}. However if the graph is undirected, by the cut property minimum cut edge is in MST. According to Dijkstra

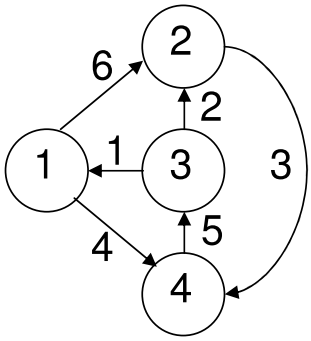

Figure 2: Counterexample for MST and Shortest Path Tree

algorithm, minimum cut edge must be in shortest path tree.

5 Problem 5.22

This problem presents an algorithm for finding minimum spanning trees. The algorithm is based on the following property

Lemma 6 *Pick any cycle C in the graph and let e be the heaviest edge in that cycle. Then there is a minimum spanning tree that does not contain e.*

(a)**Proof:** Suppose there is a minimum spanning tree which contains e. If we add one more edge to the spanning tree we will create a cycle. Suppose we add edge $\hat{e}$ to the spanning tree which generated cycle C. We can reduce the cost of the minimum spanning tree if we choose an edge other than e from C for removal which implies that e must not be in minimum spanning tree and we get a contradiction □
(b)Correctness of the algorithm follows from the lemma. Since we are looking at all the edges in decreasing weight, we are choosing the heaviest edge first. That edge must not be in the MST if it is part of a cycle C. Presented algorithm checks for a cycle and remove the edge from the graph if it is part of a cycle.
(c)Linear time algorithm to check whether there is a cycle containing a specific edge e: Let $e = (u, v)$. Start a DFS from u and exclude edge e while considering outgoing edges from u. Check if e is a back-edge in resultant DFS tree
(d)Complexity Analysis: Total number of edges in a MST is given by $|V| - 1$. Therefore on the worst case the algorithm will remove $E - |V| + 1$ edges from G to obtain the MST. At each iteration of the algorithm, cycle detection takes $O(V + E)$. Therefore total running time is given by $O((|E| - |V| + 1) \cdot (V + E))$.

6 Problem 5.30

We present the algorithm first followed by correctness proof

```
procedure ternary-huffman(f)
Input:  An array f[1...n] of frequencies
Output:  An encoding tree:
    n leaves if n is odd
    n+1 leaves if n is even
if n is even:
  add a new element f[n+1] = 0
```

```
let H be a priority queue of integers ordered by f
for i = 1 to size(f):  insert(H,f[i])
for l = size(f) to 2n-3:
  i = deletemin(H), j = deletemin(H), k = deletemin(H)
  create a node numbered l with children i,j,k:
    f[l] = f[i]+f[j]+f[k]
    insert(H,l)
```

Ternary Huffman Algorithm in essence is similar to binary huffman but there are some distinct differences. We add a node of zero frequency if n is even. This is necessary as we want to form a complete ternary tree. Objective function of ternary huffman algorithm is given by

$$\sum_{a \in \sigma} h(a) \cdot f(a) \quad \text{h(a),f(a) are height and frequency of a} \tag{5}$$

The following three lemmas are essential for proof of correctness

Lemma 7 *There is always an optimal tree such that three letters of minimum frequency must be combined together*

Lemma 8 *After the combination with the sum of frequency as the frequency for the new tree, the problem becomes same problem of smaller size*

Lemma 9 *If at any point of time three letters of minimum frequency cannot be combined then the resultant tree is not optimal*

Proof of lemma 7-8 are already presented in class. I am not repeating it again. If you are not clear about it, drop by during my office hours and I can explain it to you.

Proof: Consider at some point of time we cannot find 3 letters of minimum frequency to combine. Without any loss of generality assume we are left with nodes i and j with frequency $f(i)$ and $f(j)$. If we combine them the cost of the tree formed will be

$$h(i)f(i) + h(j)f(j) \tag{6}$$

Consider any children c_i of i. $h(c_i) = 1 + h(i)$. Therefore cost of the tree considering children of i can be written as

$$(1 + h(i))(f(c_i) + k) + h(j)f(j) \quad \text{where} \quad k = f(i) - f(c_i) \tag{7}$$

Now we can reduce the cost of the tree if we bring c_i to the height of j

$$(1 + h(i))(f(c_i) + k) + h(j)f(j) > (1 + h(i))k + h(j)f(j) + h(j)f(c_i) \tag{8}$$

as h(i) is equal to h(j). □

Optimality condition is violated if at any point of time we cannot combine three letters of minimum frequency. To satisfy this constraint we add a additional letter of 0 frequency when n is even as it is evident from Equation 8 that

$$(1 + h(i))(f(c_i) + k) + h(j)f(j) = (1 + h(i))k + h(j)f(j) + h(j)f(c_i) \quad \text{when} \quad f(c_i) = 0 \tag{9}$$