CSOR 4246 Algorithms for Data Science (Fall 2022) Problem Set #2t

Xinhao Li - xl2778@columbia.edu

2022/10/18

Problem 1

Intro

In this question, we are asked to compute the number of shortest $s - t$ paths in G. In order to do so, we will first assign the distance[] list to be ∞ except the source node which is 0. Then we are going to run the BFS on the source node. Each time when we enqueue v where $(u, v) \in E$ into the queue, there are two cases.

- If $dist[v] > dist[u] + 1$, this means that there is a shorter path from s to node v than the current path ($dist[v]$). So we will assign the $dist[v]$ to be $dist[u] + 1$ (since v is a child node of u). And the number of shortest paths from s to v will just be the same as the number of shortest paths from s to u.
- If $dist[v] = dist[u] + 1$, this means that the number of shortest paths from s to u should be added to the number of shortest paths from s to v.

Pseudocode

```
def compute_num_shortest_paths(G = (V, E), s, t):
  array discovered[V] initialized to 0
  array dist[V] initialized to infinity
  array parent[V] initialized to NIL
  array num_paths[V] initialized to 0
  queue q
  discovered[s] = 1
  dist[s] = 0
  parent[s] = NIL
  num_paths[s] = 1
  enqueue(q, s)
  while size(q) > 0:
    u = dequeue(q)
    for (u, v) belongs to E:
      if discovered[v] == 0:
        enqueue(q, v)
        discovered[v] = 1
      if dist[v] > dist[u] + 1:
        dist[v] = dist[u] + 1
        num_paths[v] = num_paths[u]
      if dist[v] = dist[u] + 1:
        num_paths[v] = num_paths[v] + num_paths[u]
  return num_paths[t]
```

Proof of correctness

Let's say that we have a node v and would like to determine the number of shortest paths from s to v. The parent of node v is u which means that $(u, v) \in E$. Based on claim 1 introduced in the lectures, if u and v are in layers L_i and L_j, respectively, then i and j differ by at most 1. Assume that we have already successfully determined the number of shortest paths from s to u. To determine the number of shortest paths from s to v, then there are three possibilities.

- If v is discovered after u, then v is a child of u, then the shortest paths from s to v will just be the shortest paths from s to u plus edge (u, v). Therefore, the number of shortest paths from s to v should be the same as the number of shortest paths from s to u.
- If v was discovered before u is explored, v should appear in the tree at some layer L_i with $j \leq i + 1$. Since $j \geq i$, so $i = j$. This means that v and u are at the same layer. Then this means if there are n shortest paths from s to u, it should be also n shortest paths from s to v plus m shortest paths belonging to v only.

Running time

The running time of this algorithm is the same as the BFS algorithm. The new added if statement will still take $deg(u)O(1)$ for fixed u and $\sum_{u \in V} deg(u)O(1) = O(m)$ for all u. And the initialization steps take O(n) time. So in total, the running time of this algorithm is $O(n + m)$.

Problem 2

(a)

Intro

In this question, we are asked to give a linear-time algorithm that works when the input is a Directed Acyclic Graph (DAG) to fill in the entire array cost. Basically, to design this algorithm, we first need to know the connectivity between all notes since the cost of one node u is defined by the price of the cheapest node reachable from node u (including u itself). Just knowing the connectivity itself is not enough, we also need to know about the order of the connectivity (sequence of visiting). For example, since the graph is a DAG, then if we have A → B → C, then the cost(A) should be:

$$cost(A) = \min_{A,B,C}(price_i) \tag{1}$$

However, the cost(B) should be:

$$cost(B) = \min_{B,C}(price_i) \tag{2}$$

In order to know the sequence of visiting, DFS algorithm will be used here since it can give us information about the staring and finish times which can be used as an efficient parameter to understand the sequence of visiting. At the same time as running the DFS, we need to initialize another array to store the nodes on the basis of the sequence that pops out from the stack. In other words, all nodes will be added to the array in the order of increasing finish time. The node with smallest finish time simply means that it is the last node in the path. Therefore, it is safe to compute the cost of this node first (just itself) and then consequently compute the cost of nodes visited before the last node (the prices of all child nodes are considered but without any parental nodes). In this order, the cost for all nodes can be computed efficiently.

Pseudocode

```
def cost_DAG(G = (V, E), price):
 seq_visit = []
 cost = {}
 for u belongs to V:
  explored[u] = 0
 for u belongs to V:
  if explored[u] == 0:
   Search(u, seq_visit)
 # Now the nodes are in increasing order of finish time
 for node in seq_visit:
  cost[node] = price[node]
  for (node, v) belongs to E:
   if cost(v) < cost(node):
    cost[node] = cost[v]
 return cost

def Search(u, seq_visit):
 previsit(u)
 explored[u] = 1
 for (u, v) belongs to E:
  if explored[v] == 0:
   Search(v, seq_visit)
 postvisit(u)
 seq_visit.append(u)
```

Proof of correctness

There is no need to proof the correctness of DFS algorithm since it has been proved in the lectures, so we can trust that the finish times returned by running DFS are correct. As implemented in the DFS, we know that the finish time is calculated when *postvist*(u) is executed This means once *postvist*(u) is executed, u is popped from the stack and all the neighbors of u have been explored. In the new designed algorithm, the append/add operation (*seq_visit.append*(u)) is just after the *postvisit*$()$. This means the node will only be added to the *seq_visit* array when the finish time of this node has been finalized. The last thing we need to prove here is the sequence of visit or adding nodes into the array. Based on the definition of DFS, it will start from a node s, explore the graph as deeply as possible, then backtrack. So only when the DFS reaches the dead end, that node will be popped from the stack, so the sequence of adding node into the *seq_visit* array should just follow the excution of *postvisit*$()$. After all nodes have been visited, the *seq_visit* array must be in the order that the finish time is sorted from small to large. The node a with smallest finish time means that it is the first dead end reached by DFS and there is no edges from a to other edges. Therefore:

$$cost_a = price_a \tag{3}$$

Then the node b with the second smallest finish time means that it is the parental node of a. Since the cost of a has been determined, so the cost of b is simply just compare the price of b with the cost of a.
If $price[b] > cost[a]$, then:

$$cost[b] = cost[a] \tag{4}$$

If $price[b] < cost[a]$, then:

$$cost[b] = price[b] \tag{5}$$

The same analysis can be extended to all nodes. Since the nodes in the seq_visit are in the order of finish time from small to large, it can be made sure that the costs of parental nodes will be determined by themselves and the min cost of their children.

Running time

This new designed algorithm is still based on the DFS algorithm.

(b)

Intro

In this question, we are asked to extend to a linear time algorithm that works for all directed graphs. As the hint given in the question, for this question, we can consider to find the SCCs of the graph. After finding the SCCs, each SCC can be represented by a "super" node and the directed graphs can simply be converted to a DAG which each node is a SCC. We know that based on the definition of SCC, the cost of any nodes in SCC should be:

$$cost_{u \in SCC} = \min_{i \in SCC} price_i \tag{6}$$

This means that the cost of any nodes in SCC should be the smallest price among all nodes. Therefore, the cost for all nodes can be calculated on the basis of the new constructed DAG.

Pseudocode

```
def cost_directed_graph(G = (V, E), price):
 Find all SCCs using find_SCC(G = (V, E))
 Build meta-graph of all SCCs of G and call it G'
  - Make a super vertex for every SCC and each super vertex has
    the cost min_price(scc)
  - Add a super edge from SCC Ci to SCC Cj
 cost = cost_DAG(G', price)
 return cost

def find_SCC(G = (V, E)):
 Compute G_r
 Run DFS(G_r) # compute finish(u) for all u
 Run DFS(G) in decreasing order of finish(u)
 Output the vertices of each tree in the DFS forest of line 3
 as an SCC
```

Proof of correctness

As been proved in the lecture, the way to find all SCCs is correct. The way to compute the costs of all nodes has been proved in question 2(a). Therefore, this algorithm is correct.

Running time

From the lecture notes, we know that the running time for finding all SCCs is $O(n + m)$. Running the cost_DAG to calculate the cost of the meta-graph G' also take $O(n + m)$ time. So overall, the running time is: $O(n + m)$.

Problem 3

Intro

In this question, we are asked to compute the minimum weight of any path from s to u (s is given as the source). Basically, this question is similar to what have been shown in the lectures: "Dijkstra algorithm". First of all, at all times we need to maintain a set of S of nodes for which the distance from s has been determined. Initially, we have:

$$dist[s] = 0, S = s \tag{7}$$

Each time, add to S the node $v \in V - S$ that:

- has an edge from some node in S;

- minimizes the following quantity among all nodes $v \in V - S$

$$d(v) = \min_{u \in S, (u,v) \in E} [dist[u] + w_{uv} + \mathbf{c_v} \tag{8}$$

- Set $prev[v] = u$

The difference between the new developed algorithm and the "Dijkstra algorithm" is just the non-negative vertex costs c_v have also been considered. And also at the initialization step, the $dist[source]$ has to be initialized as:

$$dist[s] = c_s \tag{9}$$

Pseudocode

```
def Initialize(G, s, c):
  for v in V:
    dist[v] = INF  # infinite large
    prev[v] = NIL
  dist[s] = c[s] # cost of source
  return dist, prev

def Update(Q, u, v):
 new_dist = dist[u] + w_uv + c[v]  # cost of v is considered
 if dist[v] > new_dist:
  dist[v] = new_dist
  Decreasekey(Q, dist[v], new_dist)
  prev[v] = u
 return Q, dist

def min_weight(G=(V, E, w, c), s: belongs to V):
 dist, prev = Initialize(G, s, c)
 Q = BuildQueue({V: dist})
 S = []
 while Q != 0:
  u = ExtractMin(Q)
  S += u
  for (u, v) belongs to E:
   Q, dist = Update(Q, u, v)
 return dist
```

Proof of correctness

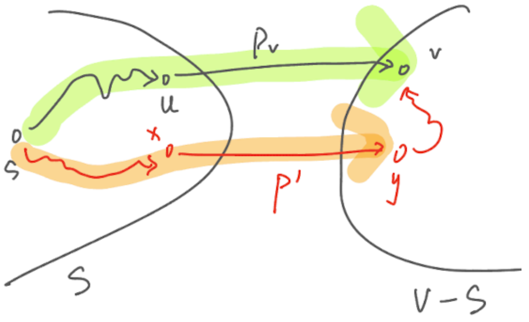
As I mentioned before, only really small modifications have been made to the new algorithm. So the proof of correctness should be the same as the proof of correctness for "Dijkstra algorithm" introduced in the lectures. The new algorithm is like to add the cost of each of vertex to the weight of out-degree path. For example, if there is a path (u, v) ∈ E, then the new updated w_{uv} is the previous path weight plus the cost of u:

$$w_{uv} = w_{uv} + c_u \tag{10}$$

Since the objective is still to minimize the distance, the same proof of correctness introduced in the lectures can be used here.
By induction on the size of S.

- Base case: $|S| = 1,\ dist(s) = 0$

- Hypothesis: Assume that it also holds true for $|S| = k$.

- To prove it also holds true for $|S| = k + 1$.
  - Let v be the k + 1st node added to S
  - 

  - Assume $w(P'_{s\to v}) < w(P_{s\to v})$
  - $w(P'_{s\to v}) \geq d(y) \geq d(v) = w(P_v)$
  - $w(P'_{s\to y}) + w(P'_{y\to v}) \geq w(P'_{s\to y}) \geq w(P_{s\to v})$
  - $w(P'_{s\to v}) \geq w(P_{s\to v})$
  - This is in contradiction with our assumption, so $w(P_{s\to v}) \leq w(P'_{s\to v})$

Running time

In the above algorithm, we used a priority queue implemented as a binary min-heap: store vertex u with key $dist[u]$. The things we slightly modified based on the "Dijkstra algorithm" do not affect the total running time. Therefore, as discussed in the lectures, the total running time for new developed min_weight to calculate both the weights of paths and costs of vertices is:

$$O(nlogn + mlogn) = O(mlogn),\ n = |V|\ and\ m = |E| \tag{11}$$

Problem 4

Intro - subproblem

In this question, we are asked to return if a string is able to be reconstituted as a sequence of valid works. If the algorithm answers yes, then output the corresponding sequence of words. So here we will use the dynamic programming to solve the problem. The basic idea behind the dynamic programming is to find the smaller subproblems, solve them, and finally use the subproblems to build up the final solution. In our case, the subproblem should be whether the string $s_1, ..., s_i$ in s only contain valid words in dictionary $dict$. So here, we will define an index i to loop over all elements in string s and correspondingly build s_i as $s[: i]$.

Recurrence

After defining the subproblem, we can start working on the recurrence. So basically recurrence will introduce another index j to loop over from 0 to i-1 and check if the words in s_i are all valid by having $s_i[j + 1, i]$ and $s_i[: j]$ are all in $dict$. Here, we need a memorization array M to store if the words are valid. After fill the memorization array, we can start to look at if the string can be reconstructed as a sentence. So, basically the last element should be 1 if the string can be reconstructed and 0 if not. If the string can be reconstructed, we will loop over the string from n-1 to the first. Once $M[i] == 1$, then we will check if string from i+1 position to the last is a valid word in dict, if it is, we will append the word in the sentence list and remove this work from the tail of the string. Finally, we should be able to print the whole sentence.

Boundary conditions

The boundary condition here is that when we have an empty string, we will consider it as a valid work. Therefore: $M[0] = 1$.

Pseudocode

```
def get_valid_words(s: str, word_dict: dict):
 n = len(s)
 M = [0 for x in range(n+1)] # assign a n+1 memorization array
  for i in range(1, n):
   for j in range(0, i-1):
    if M[j] == 1 and s[j+1:i] in word_dict:
     M[i] = 1
     break
  if M[n] == 0:
   return False
  print_sentence(M, s)

def print_sentence(M, s):
 n = len(M)
 sentence = []
 for i in range (n-1, -1, -1): # loop M reversely
  if M[i] == 1 and s[i+1:] in dict:
   sentence.append(s[i+1:])
   s = s[:i]
 return sentence.reverse() # reverse the list
```

Running time

Here we have two nested *for* loop, so the total running time would be $O(n^2)$.

Space

As mentioned before, the memorization array is used and it has a size $n + 1$. So the space required is: $O(n)$.