Homework2

Jiang Shuyang (519030910043)

November 10, 2021

PROBLEM 1. Here is a proposal to find the length of the shortest cycle in an unweighted undirected graph:
DFS the graph, when there is a back edge (v, u), it forms a cycle from u to v, and the length is $level[v] - level[u] + 1$, where the level of a vertex is its distance in the DFS tree from the root. This suggests the following algorithm:

- Do a DFS and keep tracking the level.
- Each time we find a back edge, compute the cycle length, and update the smallest length.

Please justify the correctness of the algorithm, prove it or provide a counterexample.

SOLUTION. Examining the algorithm, the first step has no mistakes.
Suppose after the whole algorithm, we find a shortest cycle denoted as p^* with each node denoted as $u_1, \cdots, u_{n-1}, u_1$, whose length is n. And we divide the possible shorter path as two conditions:

- the shorter path begins with another node
- the shorter path also begins with u_1.

First we suppose the algorithm is able to find all cycles. But this assumption is incorrect for the reason that we cannot find all cycles by just detecting the back edge. Consider the following circumstance:

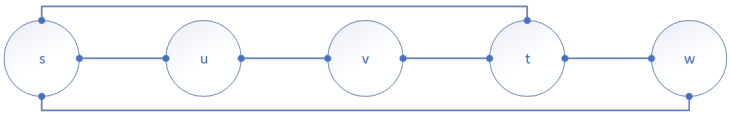

Figure 1: Counter example

We choose s as the source point for DFS, which will detect (s, u, v, t, s), (s, u, v, t, w, s) cycles, which both start at s. However, following the algorithm, it cannot detect the cycle (s, t, w, s), which is the shortest cycle in this graph. As a result, this algorithm is incorrect.

PROBLEM 2. Given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has a weight $p(u, v)$ in range $[0, 1]$, that represents the reliability. We can view each edge as a channel, and $p(u, v)$ is the probability that the channel from u to v will not fail. We assume all these probabilities are independent. Give an efficient algorithm to find the most reliable path from two given vertices s and t. Hint: it makes a path failed if any channel on the path fails, and we want to find a path with minimized failure probability

SOLUTION. This is another form of longest path problem. I propose an algorithm illustrated as follows:

Algorithm 1 Maximum likelihood of a channel

Input:
The nodes of the graph $\{V_i | i = 1, 2, \cdots, |V|\}$;
The edges of the graph $\{E_i | i = 1, 2, \cdots |E|\}$;
The probability for each edge $\{p_i | i = 1, 2, \cdots |E|\}$;
The source vertex and the target vertex s and t, respectively;
Output:
The most reliable path from s to t denoted as a list of vertices $s, a, \cdots, t$.

Maintaining a cumulative probability from each node u to source node s, denoted as $AP[u]$;
For a neighbor node v for u, $AP[v] = AP[u] \times p(u, v)$;
Maintaining a Fibonacci max heap for $AP[u]$.
Maintaining a previous vertex for each vertex $PrevNode[u]$.
$CurNode = s$
while $CurNode! = t$ **do**
  Fetch one node v with maximum $AP[v]$
  For its neighbor v_{nei}, $AP[v_{nei}] = \min\{AP[v_{nei}], AP[v] \times p(v, v_{nei})\}$
  if any neighbor v_{nei}'s information has been updated **then**
    $PrevNode[v_{nei}] = v$
  end if
  $CurNode = v$
end while
The maximum probability can be fetched from $AP[t]$
The path with maximum probability can be obtained from the $PrevNode$,

denoted as *path*
16: **return** *path*;

The correctness of it can be guaranteed from Dijkstra algorithm. I only change the cumulative addition operation into the cumulative multiplication operation.

PROBLEM 3. We have a connected undirected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a T that includes all nodes of G. Suppose we then compute a breath-first search tree rooted at u, and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a DFS tree and a BFS tree rooted at u, then G cannot contain any edges that do not belong to T.)

SOLUTION. Assume that if T is both the DFS tree and BFS tree of one node, and G contains another edge that does not appear in T. Suppose T is as follows:

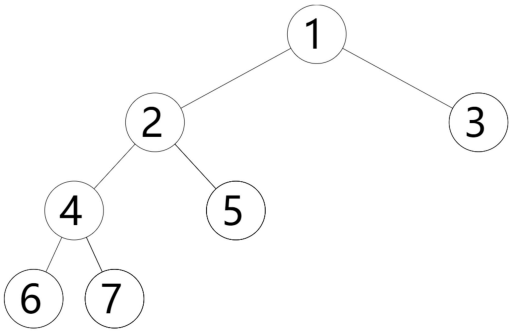

Figure 2: Illustration of T

The order of DFS tree is $1, 2, 4, 6, 7, 5, 3$ while that of BFS tree is $1, 2, 3, 4, 5, 6, 7$. The difference between them determines that onece there is a cycle in a graph, the DFS tree will become too deep while the BFS tree will maintain its original shape to a great extent. Suppose there is an edge e in G while it does not appear in T, shown as follows:

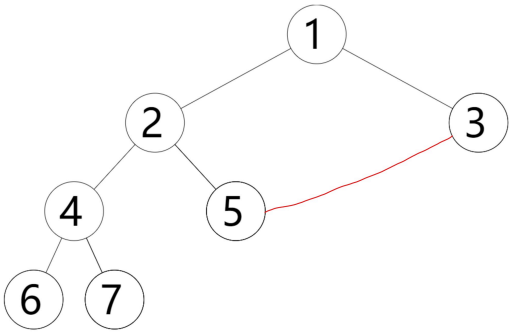

Figure 3: Illustration of G

Currently, the DFS tree will add $(5, 3)$ while BFS tree will not. **More generally**, if there is a cycle in G, denoted as $s, u, \cdots, v, s$, there must exist a path $s, u, \cdots, v$ in DFS tree while there must exist an edge (s, v) in BFS tree. The difference is determined by the properties of these two different traverse methods. Therefore, there occurs a contradiction between the condition and the conclusion, which proves that there cannot exist an edge in G such that it does not exist in T.

PROBLEM 4. Given a directed graph $G(V, E)$ where each vertex can be viewed as a port. Consider that you are a salesman, and you plan to travel the graph. Whenever you reach a port v, it earns you a profit of p_v dollars, and it cost you c_{uv} if you travel from u to v. For any directed cycle in the graph, we can define a profit-to-cost ratio to be

$$r(C) = \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}}$$

As a salesman, you want to design an algorithm to find the best cycle to travel with the largest profit-to-cost ratio. Let r^* be the maximum profit-to-cost ratio in the graph.

1. If we guess a ratio r, can we determine whether $r^* > r$ or $r < r^*$ efficiently?

2. Based the guessing approach, given a desired accuracy $\epsilon > 0$, design an efficient algorithm to output a good-enough cycle, where $r(C) \geq r - \epsilon$. Justify the correctness and analyze the running time in terms of $|V|$, ϵ, and $R = \max(u, v) \in E(p_u/c_{uv})$.

SOLUTION.

1.

$$r < r^* \tag{1}$$

$$r < \frac{\sum p_v}{\sum c_{uv}} \tag{2}$$

$$\sum_{uv} rc_{uv} - p_v < 0 \tag{3}$$

If we assign a weight $w_{uv} = rc_{uv} - p_v$ for each c_{uv}, then the formula means that if we find a cycle with gross weight less than 0, we can make sure that r is less than r^*, namely $r < r^*$. The opposite is also correct. If we cannot find a cycle with gross weight less than 0, we can make sure than r is greater than r^*, namely $r > r^*$

2. I first hand out the designed algorithm as follows and illustrate the time complexity as well as the correctness in the end.

Algorithm 2 good-enough cycle

Input:
The nodes of the graph $\{V_i | i = 1, 2, \cdots, |V|\}$;
Output:
The good enough cycle C.

initialize an array $dist[u]$ of size |V| + 1, representing the minimum distance from u to s.
$lb = 0, rb = R, r = 0, r^* = \frac{lb+ub}{2}$;
while $|r - r^*| >= \epsilon$ **do**
  $r = \frac{lb+ub}{2}$
  $W[v] = rc_{uv} - p_v$ for each v in V
  flag, dist=3
  if flag= True **then**
    $lb = r$;
  else
    flag, C=4
    if flag=True **then**
      return C
    end if
    $ub = r$
  end if
  $r^* = \frac{lb+ub}{2}$
end while
return C;

Algorithm 3 NegativeCycleDetection

Input:
The nodes of the graph $\{V_i | i = 1, 2, \cdots, |V|\}$;
The weights of the graph $\{w_{uv}\}$;
The distance array $dist$

Output:
the flag representing whether there is a negative cycle and the $dist$ array.

```
for each node v do
    dist[v] = ∞
end for
dist[0] = 0 // 0 is a super node connecting all nodes of V
for i in range(0, V + 1) do
    for each (u, v) ∈ E do
        if dist[v] < dist[u] + w_uv then
            dist[v] = dist[u] + w_uv
            flag = (i == |V|)
        end if
    end for
end for
return flag, dist;
```

Algorithm 4 ZeroCycleDetection

Input:
The nodes of the graph $\{V_i | i = 1, 2, \cdots, |V|\}$;
The distance array $dist$

Output:
The flag representing whether there is a zero cycle and the cycle C.

```
Initialize a new graph G'with the same nodes of V but no edges at first.
for each edge e_uv do
    if dist[u] = w_uv + dist[v] then
        G'.insert(u, v, w_uv)
    end if
end for
initialize start[v] array with ∞ for each vertex.
initialize C
```

9: run 5 with (0, $time$, C, $start$) and get C.
10: **return** C;

Algorithm 5 DFS

Input:
The current node u;
Initial $time = 0$;
The cycle array C;
The start time for each node $start[u]$
Output:
The cycle C.
1: $start[u] = time++$
2: $u.isVisited = True$
3: $C.append(u)$
4: **for** each edge e_{uv} **do**
5: **if** $v.isVisited = false$ and $start[v] < start[u]$ **then**
6: run 5 with (v, $time$, C, $start$)
7: **end if**
8: **if** $start[v] > start[u]$ **then**
9: return C
10: **end if**
11: **end for**
12: **return** C;

The exterior loop experiences $\log \frac{R}{\epsilon}$ times. The interior part is divided into two parts: one for 3 and the other for 4. The first algorithm is $Bellman - Ford$ whose time complexity is $O(|V||E|)$. The second algorithm is a simple DFS with a loop over all E, whose time complexity is $O(|V| + |E| + |E|) = O(|V| + |E|)$. Therefore, the overall time complexity is $\log \frac{R}{\epsilon}(|V||E| + |V| + |E|)$.
The correctness is based on the first question. We know that the cycle is good enough if and only if $r = r(C)$. Therefore, we need to select an optimum r s.t. $r(C) \geq r - \epsilon$. If $r < r^*$, we know that we need to increase r, and vice versa. Furthermore, the search range is bound in $(0, R)$. Compared to linear search, the binary search is more excellent. Consequently, the exterior loop is correct.
To detect the negative cycle, I refer the slides of $Bellman - Ford$, which can detect the negative cycle if we increment the loop number by 1. The correctness is proved several years ago.

To detect the zero cycle, consider the following formula:

$$\sum_{uv} w_{uv} = 0 \text{ iff } \sum_{uv} dist[u] + w_{uv} = \sum_{uv} dist[v] \tag{4}$$

This formula satisfies in a cycle. So we just need to select those edges satisfying the right formula and connect those nodes with DFS, after which we can get one of the optimum cycles.

PROBLEM 5. Consider if we want to run Dijkstra on a bounded weight graph $G = (V, E)$ such that each edge weight is integer and in the range from 1 to C, where C is a relatively small constant.

1. Show how to make Dijkstra run in $O(C|V| + |E|)$.
2. Show how to make Dijkstra run in $O(\log C(|V| + |E|))$. Hint: Can we use a binary heap with size C but not $|V|$?

SOLUTION.

1. **Algorithm 6** Dijkstra in $O(C|V| + |E|)$

Input:
The nodes of the graph $\{V_i | i = 1, 2, \cdots, |V|\}$;
The edges of the graph $\{E_i | i = 1, 2, \cdots |E|\}$;
The source vertex s;
Output:
The minimum distance from source node s.

initialize an array $dist[u]$ representing the minimum distance from u to s.
initialize a bucket array B with size $C|V|$, whose entry is a doubly linked list connecting nodes with the same $dist$ value.
for each node u, $dist[u] = \infty$, and $B[u] = NULL$
$dist[s] = 0, B[0] = s$
while 1 **do**
  idx = the first index of B s.t. $B[idx]! = NULL$
  if $idx = C|V|$ **then**
    break
  end if
  $CurNode = B[idx].first$
  delete $B[idx].first$
  for each edge e_i of $CurNode$ **do**
    if $dist[CurNode] + e_i.weight < dist[e_i.end]$ **then**
      if $dist[e_i.end]! = \infty$ **then**

15: delete the corresponding node in B
16: **end if**
17: $dist[e_i.end] = dist[CurNode] + e_i.weight$
18: $B[dist[e_i.end]].insertFront(e_i.end)$
19: **end if**
20: **end for**
21: **end while**
22: **return** *dist*;

The algorithm is similar to Dijkstra algorithm. The update of the *dist* for each node can be done in $O(1)$ while the search range for nodes expand to $O(C|V|)$, thus resulting in the final time complexity. Why this algorithm is still correct? Because in each loop what we do is select the node with the minimum *dist* value. Based on this node, we update its neighbor nodes *dist* just as what we do in Dijkstra. We discard the use of heap but utilize the doubly linked list to help find the corresponding vertex. The correctness is guaranteed by Dijkstra algorithm.

---

2. **Algorithm 7** Dijkstra in $O(\log C|V| + |E|)$

Input:
The nodes of the graph $\{V_i|i = 1, 2, \cdots, |V|\}$;
The edges of the graph $\{E_i|i = 1, 2, \cdots |E|\}$;
The source vertex s;

Output:
The minimum distance from source node s.

1: initialize an array $dist[u]$ representing the minimum distance from u to s.
2: initialize a bucket array B with size $C|V|$, whose entry is a doubly linked list connecting nodes with the same *dist* value.
3: initialize a priority queue q whose comparison rule is based on the *dist* value of each node
4: for each node u, $dist[u] = \infty$
5: $B[0].insertEnd(0)$
6: $dist[s] = 0$, $q.push(\{B[0], 0\})$
7: **while** $!q.empty()$ **do**
8: **if** $q.top() == NULL$ **then**
9: $q.pop()$;
10: continue
11: **end if**
12: $head = q.top().first$

```
13:    delete q.top().first
14:    if q.top() == NULL then
15:      q.pop()
16:    end if
17:    for each e_i of head do
18:      if dist[head] + e_i.weight < dist[e_i.end] then
19:        if dist[e_i.end]! = ∞ then
20:          delete e_i from B[dist[e_i.end]]
21:        end if
22:        dist[e_i.end] = dist[head] + e_i.weight
23:        B[dist[e_i.end]].insertEnd(e_i.end)
24:        if B[dist[e_i.end]].size() == 1 then
25:          q.push({B[dist[e_i.end]], dist[e_i.end]})
26:        end if
27:      end if
28:    end for
29:  end while
30:  return dist;
```

The necessary discussion is that the max number of items in the priority queue is no more than C. Each time I just push the bucket pointer to be heap once so that even if there are new items appended to the bucket or there are some items deleted from the bucket, I can still find the elements of the bucket. Each time I select the first element of the doubly linked list because all the elements share the same priority and popping the first element only costs $O(1)$ time complexity. Therefore, except for the size of the priority queue, the other part is the same as that in normal Dijkstra algorithm. The overall time complexity is $O(\log C(|V| + |E|))$.