SHANGHAI JIAOTONG UNIVERSITIY

Assignment 2 Answer

Course: *AI2615 - Algorithm Design and Analysis*
Due date: *Nov 15th, 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.

Answer Referred by Huangji Wang This algorithm is not perfectly true. Counterexample:

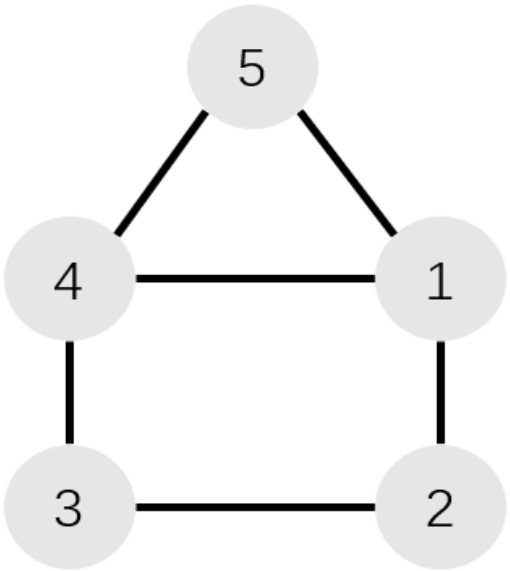

Figure 1: Counterexample

We can always find one path like this picture above. If we do the DFS from the initial vertex $v(1)$, we will form the path as:

- initialize ans = $+\infty$

- (1)
- (1) -> (2)
- (1) -> (2) -> (3)
- (1) -> (2) -> (3) -> (4)
- (1) -> (2) -> (3) -> (4) -> (1) **ans = min(ans, level[4] - level[1] + 1) = 4**
- (1) -> (2) -> (3) -> (4)
- (1) -> (2) -> (3) -> (4) -> (5)
- (1) -> (2) -> (3) -> (4) -> (5) -> (1) **ans = min(ans, level[5] - level[1] + 1) = 4**
- (1) -> (2) -> (3) -> (4) -> (5)
- (1) -> (2) -> (3) -> (4)
- (1) -> (2) -> (3)
- (1) -> (2)
- (1)
- **ans = 4**

However, if we look at the graph, we can easily find that the smallest cycle is $v(5) -> v(1) -> v(4)$ and $ans = 3$. Thus, the algorithm may miss some important cycles, leading to the wrong answer.

My suggestion on how to improve this algorithm is to do DFS without visited arrays. Though it will cost much time than we think, it can find every cycle in the graph and it cannot miss any cycle.

The best way to handle with this problem is to use **union-find disjoint sets** which is very useful to find cycles in (unweighted) undirected graph.

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.

Answer Referred by YiChen Tao. The reliability of a path $v_1 - v_2 - \ldots - v_k$ can be expressed as $\prod_{i=2}^{k} p(v_{i-1}, v_i) = \exp[\sum_{i=2}^{k}(\ln p(v_{i-1}, v_i))]$. Hence, to maximize the reliability of a path is to maximize the value of $\sum_{i=2}^{k}(\ln p(v_{i-1}, v_i))$, which is equivalent to minimizing $\sum_{i=2}^{k}(-\ln p(v_{i-1}, v_i))$. So the problem converts into a shortest path problem, where we take $-\ln p(u, v)$ as the weight of the edge (u, v). Since $p(u, v) \in [0, 1]$ holds for all u, v, the edge weight $-\ln p(u, v)$ is always positive, so the problem can be addressed using the Dijkstra algorithm.

Here is a brief explanation of the algorithm:

1. Initialize
   - $w(u, v) = -\ln p(u, v)$ for all (u, v).
   - $T = \{s\}$.
   - $tdist[s] \leftarrow 0$, $tdist[v] \leftarrow \infty$ for all v other than s.
   - $pred[s] \leftarrow -1$, $pred[v] \leftarrow 0$ for all v other than s.
   - $tdist[v] \leftarrow w(s, u)$, $pred[v] \leftarrow s$ for all $(s, v) \in E$.
2. Explore and Update
   - Find $v \notin T$ with smallest $tdist[v]$.
   - $T \leftarrow T + \{v\}$
   - For all $(v, u) \in E$, if $tdist[v] + w(v, u) < tdest[u]$,then $tdist[u] \rightarrow tdist[v] + w(v, u)$ & $pred[u] \rightarrow v$.
   - Repeat the steps above until t is added to T.
3. Result: Form the path from s to t by backtracking the *pred* array.

Answer Referred by Hanmo Zheng. Since weight $p(u, v)$ in range [0,1] represents the probability that the channel will not fail. The higher $p(u, v)$ is, the lower is the probability path from u to v failed. Thus, we need to find the way from s to t with the highest $p(s, t)$, ($p(s, t) = p(s, a_1) * p(a_1, a_2) * \cdots * p(a_k, t)$, where $s, a_1, a_2, \cdots, a_k, t$ is the way from s to t). Then we design an algorithm with reference to Dijkstra Algorithm as follows:

1. **Initialize**
   - $T = \{s\}$,
   - $tprob[s] = 1, tprob[v] \leftarrow 0$ for all v other than s
   - $tprob[v] \leftarrow w(s, v)$ for all $(s, v) \in E$
2. **Explore**
   - Find $v \notin T$ with biggest $tprob[v]$
   - $T \leftarrow T + \{v\}$
3. Update tprob[u]
   - $tprob[u] = max\{tprob[u], tprob[v] * w(v, u)\}$ for all $(v, u) \in E$

Now we show by the steps above, we can get the path with the minimized failure probability. Firstly, since there is a path from s to t and the algorithm will explore all the nodes connected from s, thus $t \in T$ in the end. Besides, if there is another path from s to v and it's failure probability is less than the path we get.

Suppose the way we find is $s, a_1, a_2, \cdots, a_k, t$, and the other path is $s, b_1, b_2, \cdots, b_l, t$,

$$p(s,a_1) * p(a_1,a_2) * \cdots * p(a_k,t) < p(s,b_1) * p(b_1,b_2) * \cdots * p(b_l,t) \tag{1}$$

. However, by the algorithm we explore the node which hasn't been explored with the biggest tprob. Then since p in range [0,1], we have

$$\begin{aligned} p(s,b_1) * p(b_1,b_2) * \cdots * p(b_l,t) &\leq p(s,b_1) * p(b_1,b_2) * \cdots * p(b_{l-1},b_l) \\ p(s,b_1) * p(b_1,b_2) * \cdots * p(b_{l-1},b_l) &\leq p(s,b_1) * p(b_1,b_2) * \cdots * p(b_{l-2},b_{l-1}) \\ &\cdots \\ p(s,b_1) &\leq p(s,b_1) * p(b_1,b_2) \end{aligned} \tag{2}$$

After $a_1, a_2, \cdots, a_k$ has been explored, suppose i is the smallest number that $b_1, b_2, \cdots, b_{i-1}$ has been explored but b_i isn't. Then by inequality (2) and the update method, we can see

$$\begin{aligned} tprob[t] &\leq p(s,a_1) * p(a_1,a_2) * \cdots * p(a_k,t) \\ &< p(s,b_1) * p(b_1,b_2) * \cdots * p(b_l,t) \\ &< p(s,b_1) * p(b_1,b_2) * \cdots * p(b_{i-1},b_i) \leq tprob[b_i] \end{aligned} \tag{3}$$

, then we will explore b_i before explore t by the algorithm. Thus $b_1, b_2, \cdots, b_l$ will all been explored before we explore t. After b_l was explored, tprob[t] will be update by the update method and inequality (1). And it is updated by the latter path, which leads to a contradiction. Thus the algorithm above is true.

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 breadth-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).

Answer Referred by Yichen Tao. We prove the conclusion by two steps:

1. G does not contain any cycles, *i.e.* G is a tree rooted at u.
2. $G = T$.

G ***does not contain any cycles.*** We prove this by contradiction. Assume that there is a cycle $v_1 - v_2 - \ldots - v_k$ in the graph G. Denote the DFS tree by T_D and the BFS tree by T_B.

Suppose that the first node of the cycle to be visited when we DFS the graph is v_f. Then all the nodes in the cycle will be visited when we explore v_f, indicating that $v_1, v_2, \ldots, v_k$ will be on the same path from the root to a leaf.

However, for BFS tree, the case is different. Again we suppose the first node visited in the cycle is v_f, then v_{f-1} and v_{f+1} will be on the same level of the BFS tree. So the DFS tree and BFS tree cannot be the same, which contradicts to the given condition, and the assumption does not hold.

G=T. Now that we already have that G is a tree, the conclusion is rather obvious. If G contains an edge that does not belong to T, in other words, T has at least one edge less than G, then T cannot be a connected graph. So it's impossible that G has any edges that do not belong to T, and our conclusion is proved.

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

$$r(c) = \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}} \tag{4}$$

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.

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

(*b*) Based on the guessing approach, given a deisred 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 $|V|, \epsilon$ and $R = \max_{(u,v)\in E}(p_u/c_{uv})$.

Answer Referred by HaoXuan Xu.

(*a*) First, we are going to assign weight to each edge in E when given the guess ratio r, which will be helpful for out later analysis.

$w(u, v) = r \cdot c_{uv} - p_v$.

Then, we will prove two lemmas.

Lemma 1. If there is a cycle of negative weight, then $r < r^*$.

Proof. Suppose the cycle of negative weight is C, then we have $\sum_{(u,v)\in C} r \cdot c_{uv} - p_v < 0$, which implies $r < \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}}$. Hence, r is smaller than the profit-to-cost ratio of a cycle, which must be no greater than the maximum, and we get $r < r^*$.

Lemma 2. If all cycles in the graph have strictly positive weight then $r > r^*$.

Proof. As all cycles have strictly positive weight, then for all cycles $C \in G$, $\sum_{(u,v)\in C} r \cdot c_{uv} - p_v > 0$, which implies $r > \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}}$ holds for all cycles, including the cycle with maximum profit-to-cost ratio. Hence, $r > r^*$.

With two lemmas above, we can detect whether the graph has a negative cycle to to determine whether $r < r^*$. If there is, then $r < r^*$. After, we have $r \geq r^*$, and we can detect whether the graph has a zero cycle to determine whether $r > r^*$. If there is, then r cannot be strictly greater than r^*, which means $r = r^*$, if there isn't, $r > r^*$.

And now, all we need to do is implement the algorithm to find if there is a negative cycle or zero cycle in a given graph.

The first task can be done with Bellman-Ford algorithm, which has been introduced in class, just run the algorithm for $|V|$ rounds, find that if there is still a vertex updating, if so, there is a negative. And the time complexity is $O(|V||E|)$. We will not show the algorithm here as it is clearly explained in slides. After running the Bellman-Ford algorithm, we can get that if there is a negative cycle and the shortest distance from s if not conerning the negative cycles, which is useful to find that if there is a zero cycle.

Then we introduce the following algorithm to dectect zero cycles. And we will prove another lemma to prove its correctness.

Algorithm 1 Zero cycle detection algorithm

Input: $G = (V, E)$, s, shortest distance array $dist$

Construct a new graph $G^0 = (V^0, E^0)$, $V^0 = V$, $E^0 = \emptyset$.
for all $(u, v) \in E$ **do**
  if $dist[v] = dist[u] + w(u, v)$ **then**
    Insert (u, v) to E^0.
  end if
end for
if cycle C identified in G^0 **then**
  return True and C
else
  return False
end if

Output: True and zero cycle C if it exists in G, False if there is no zero cycle in G.

In the above algorithm, we can use DFS to determine if there is a cycle in G^0, and the time complexity of DFS is $O(|V| + |E|)$, the time complexity of G^0 construction is $O(|E|)$, so the time complexity for the zero cycle detection algorithm is $O(|V| + |E|)$.

Correctness of it comes from the following lemma.

Lemma 3. If G doesn't have negative cycle, and G^0 has a cycle then G must has a zero cycle.

Proof. Obviously, if there is a cycle C in G^0, then it must also exist in G. And because G doesn't have negative cycle, C can be zero or positive. And now, we'll prove that if C is positive in G, it will not exist in G^0.

For the cycle C, $\sum_{(u,v)\in C} w(u, v) = \sum_{(u,v)\in C} dist[u] + w(u, v) - dist[v] > 0$. As the sum is positive, there must be one edge (u, v) such that $dist[u] + w(u, v) - dist[v] > 0$, which will not be inserted to E^0. Hence the positive cycle will not appear in G^0. So, if there is a cycle C in G^0, it must be zero cycle in G.

Therefore, we can detect whether a cycle exists in G^0 to determine whether a zero cycle exists in G.

In conclusion, we can use the negative-cycle-detect algorithm and zero-cycle-detect algorithm to compare r and r^*.

(b) With the above method to compare r and r^* and given the initial range $(0, R)$, we can use binary search to continuously compress the range of r^*.

Algorithm 2 Max profit-to-cost ratio found algorithm

Input: $G = (V, E)$, desired accuracy ϵ, initial upper bound R

$lb \leftarrow 0, rb \leftarrow R, r \leftarrow \frac{lb+rb}{2}$.
repeat
  $tempr \leftarrow r$.
  for $(u, v) \in E$ **do**
    $w(u, v) \leftarrow tempr \cdot c_{uv} - p_v$.
  end for
  if NegativeCycleDetect(G, W) **then**
    $lb \leftarrow tempr$.
  else
    if ZeroCycleDetect(G, W) **then**
      return tempr and zero cycle C.
    else
      $rb \leftarrow tempr$.
    end if
  end if
  $r \leftarrow \frac{lb+rb}{2}$.
until $|tempr - r| > \epsilon$

Output: r^* and its corresponding cycle C.

Correctness. Obviously, $r^* \leq max_{(u,v)\in E}(p_u/c_{uv}$, hence the initial upperbound is larger enough. And we have proved the correctness of two algorithm to compare r and r^*. So the binary search algorithm correctness is obvious, it will give us r^* with acceptable accuracy.

Time complexity ayalysis. We do $log\frac{R}{\epsilon}$ iterations at most. The time complexity of each iteration is $O(|V||E|) + O(|E|) = O(|V||E|)$. So, the total time complexity is $O(log\frac{max_{(u,v)\in E}(p_u/c_{uv})}{\epsilon} \cdot |V||E|)$.

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.

(a) Show how to make Dijkstra run in $O(C|V| + |E|)$.

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

Answer Referred by Haoxuan Xu.

(*a*) If we want to reduce the time complexity from $O(|V|^2 + |E|)$ to $O(C|V| + |E|)$, we need to make the unexplored vertex with smallest distance in $O(C)$ in average instead of $O(|V|)$. As each edge weight is bounded by C, the longest distance is bounded by $C|V|$ when two vertices are connected. Besides, as we are maintaining a SPT during the Dijkstra algorithm, the distance of vertex added to explored set is non-decreasing, we can use an array of doubly linked list *bucket* to store vertices of distance, which means $bucket[dist] = \{\text{vertices whose distance to source is } dist\}$, and the array size is $|C|V$. We need the data structure of $bucket[dist]$ has following properties:

(a) Check if it's empty in $O(1)$.

(b) Delete/Add an element in $O(1)$.

So, we can choose doubly linked list or any data structure sastisfying beyond properties.

Algorithm 3 Improved Dijkstraj algorithm

Input: G = (V, E), s, $1 \leq$ w(e) $\leq C$ for all e $\in$ E.

```
1: Initialize dist(v) = ∞, pre[v] = nil for all v ∈ V. bucket ← An array of size C|V|,
   whose element data structure is doubly linked list.
2: dist(s) = 0, bucket[0] = { s }, index = 0.
3: while index != C|V| do
4:   if dist[index] is empty then
5:     index ← index + 1.
6:     continue.
7:   end if
8:   u ← bucket[index] top element, and remove it from bucket[index]
9:   for all (u, v) ∈ E do
10:    preDist ← dist[v].
11:    if dist[v] >dist[u] + w(u, v) then
12:      dist[v] ← dist[u] + w(u, v).
13:      pre(v) = u.
14:      Add v into bucket[dist[v]]
15:      if preDist ≠ ∞ then
16:        remove v from bucket[preDist]
17:      end if
18:    end if
19:  end for
20: end while
```

Hence, we traverse index from 0 to $C|V|$, the internal time complexity of each loop except the internal for loop is $O(1)$, and visit all edges through the internal for loop only once. So, the time complexity of the improved algorithm is $O(C|V| + |E|)$.

(*b*) As the hint says, we need to improve the dijkstra algorithm with heap by reducing the heap size from $|V|$ to C. We use the idea inspired by the algorithm in **(a)**, let the elements of heap be buckets, which is are sets containing all vertices has the

same distance to s. And now, we are going to prove that, when we choose an unexplored vertex from a new $bucket[d]$, there will be at most C buckets in use, which are $bucket[d], \ldots bucket[d + C - 1]$.

We will prove it by induction.

(a) **Base case.** The initial state is that there is only one bucket in use, $bucket[0]$, obviously satisfying the statement.

(b) **Induction.** Suppose we are choosing from a new $bucket[d]$, there are C buckets in use, which are $bucket[d], \ldots bucket[d + C - 1]$. And now, we will choose a vertex from $bucket[d]$, then the distance of updated adjacent vertices must lie in $(d + 1, d + C)$, which means there will be a new $bucket[d + C]$. Then, we choose vertices from $bucket[d]$ contiguously until is empty. Then, we will choose a vertex from a new bucket $bucket[d + 1]$, and now, there are still C buckets in use, $bucket[d + 1], \ldots bucket[d + C]$. And now, we finish the induction.

Hence, we have proven the previous statement, and it's easy to see that there are at most $C + 1$ buckets in use with that. And now, we have bounded the size of the heap to $C + 1$, the time complexity for finding minimum will be $O(|V|log(C + 1)) = O(|V|logC)$, the time complexity for updating will be $O(|E|log(C + 1)) = O(|E|logC)$, and the total time complexity for the improved Dijkstraj algorithm will be $O(logC(|E| + |V|))$.