6.046/18.410 Problem Set 4

Yijun Jiang
Collaborator: Hengyun Zhou, Eric Lau

October 9, 2015

1 Learn to Fuel Wisely

1.1 Part (a)

Description: First run APSP on the graph $G = (V, E)$ with edge weights $l(u, v)$. Construct a new unweighted and undirected graph $G' = (V, E')$, using all original vertices but newly defined edges, as follows:

Check $\delta(u, v)$ for all pairs. If $\delta(u, v) \leqslant K$, create an edge (u, v). By this construction, all the island pairs that can be reached within one fill are connected.

Run APSP on the new graph G'. Then pick out the longest one among all-pair shortest paths in G'. This gives the value t we want.

The pseudocode is shown below.

Algorithm 1 Finding the smallest refilling time that guarantees travelling between any vertex pair

```
procedure FuelWisely(G)
    δ = APSP(G)
    Create a new unweighted and undirected graph G' = (V, E'), where E' = ∅
    for u in V do
        for v in V do
            if u ≠ v and δ(u, v) ⩽ K then
                Add edge (u, v) to G'
    δ' = APSP(G')
    A double for loop to find t = max(δ'(u, v))
    return t
```

Correctness: We prove that in order to get from any vertex u to any distinct vertex v in the original graph G, the smallest number of refills, denoted here by t_{uv}, equals the length of the shortest path, $\delta'(u, v)$, in G'. Then $t = \max\limits_{u,v}(t_{uv}) = \max\limits_{u,v}(\delta'(u, v))$ is the smallest number of refills that guarantees travelling between any vertex pair.

Given a shortest path p'_{uv} from u to v in G', we can use the following refilling strategy: start by filling at u, then until reaching v, we always go to the subsequent vertex in p'_{uv} and refill there. This can always be done since, by the construction of G', we can reach from a vertex in p'_{uv} to its subsequent vertex in one fill. This strategy thus involves $\delta'(u, v)$ refills. Since t_{uv} is the smallest number of refills from u to v, $t_{uv} \leqslant \delta'(u, v)$.

On the other hand, if a refilling strategy involves filling at u and refilling at some (or maybe no) intermediate vertices, then going from one (re)filling vertex to the next requires only one fill. If this strategy involves t_{uv} refills, then there is a path in G' connecting u and v with length t_{uv}. It cannot be shorter than the shortest path from u to v in G'. So $t_{uv} \geqslant \delta'(u, v)$.

In conclusion, $t_{uv} = \delta'(u, v)$. Then the correctness of this algorithm follows.

Runtime: Notice that G does not contain negative weighted edges. So we can run the first APSP on G by $|V|$ times Dijkstra, which costs $O(V^2 \log V + VE)$ time.

The construction of G' is of $O(V^2)$ time because all the vertex pairs need to be checked.

Since G' is unweighted, $|V|$ times BFS is enough for the second APSP, which costs $O(V^2 + VE')$ time. In the worst case, G' is dense and this time becomes $O(V^3)$.

Finally, finding the maximum among all $\delta'(u, v)$ is of $O(V^2)$ time.

In all, this algorithm runs in $O(V^3)$ time.

1.2 Part (b)

Description: First calculate the shortest path between this new island L (denoted by s_L as a vertex) and other previously existing islands (denoted by $u, v, \cdots$ as vertices). Assign $\delta(s_L, s_L) = 0$ and $\delta(s_L, u) = \infty$ for all previously existing vertices u. Then for all u directly connected to s_L and all v, use $w(s_L, u) + \delta(u, v)$ to relax $\delta(s_L, v)$. Since the graph is undirected, we also get $\delta(v, s_L)$ for all v.

Then update the shortest path between the previously existing islands. For each vertex pair u, v, relax $\delta(u, v)$ by $\delta(u, s_L) + \delta(s_L, v)$.

The pseudocode is shown below.

Algorithm 2 Updating APSP after adding another vertex

```
1:  procedure UpdateASAP(G, s_L, A_s2G)
2:      ▷ Assume G = (V, E) is the old graph; A_s2G stores all u ∈ V that are connected to s_L, as well as w(s_L, u)
3:      δ(s_L, s_L) ← 0
4:      for u in V do
5:          δ(s_L, u) ← ∞
6:      for u in A_s2G do
7:          for v in V do
8:              if w(s_L, u) + δ(u, v) < δ(s_L, v) then
9:                  δ(s_L, v) ← w(s_L, u) + δ(u, v), δ(v, s_L) ← δ(s_L, v)
10:     for u in V do
11:         for v in V do
12:             if δ(u, s_L) + δ(s_L, v) < δ(u, v) then
13:                 δ(u, v) ← δ(u, s_L) + δ(s_L, v)
```

Correctness: (1) [Calculation of $\delta(s_L, v)$] The shortest path from s_L to a previously existing vertex v can only visit s_L once, otherwise there will be a cycle and the path becomes longer (unless it is a zero weight cycle, then s_L can be but does not need to be visited multiple times; for practical reasons, fuel consumption should be positive anyway). Thus, this path first contains (s_L, u) for some vertex $u \in V$ and then stays inside the old graph $G = (V, E)$: $p_{s_L,v} = \{(s_L, u)\} \cup p^G_{u,v}$. Therefore, when relaxing $\delta(s_L, v)$ by some u directly connected to s_L, we can use $\delta(u, v)$ of the old graph. Since we loop over all such u for relaxation, for each $v \in V$, the optimal u is selected such that $w(s_L, u) + \delta(u, v)$ is minimized. Then this must be the shortest path from s_L to v.

(2) [Calculation of $\delta(u, v)$] The only possibility to reduce $\delta(u, v)$ is to use s_L as an intermediate vertex. Therefore, by trying to relax $\delta(u, v)$ using $\delta(u, s_L) + \delta(s_L, v)$ (both are calculated previously), we can decide whether using s_L as an intermediate vertex shortens the path. Then we are guaranteed to get the shortest paths between all previously existing vertex pairs.

Runtime: Both $\delta(s_L, v)$ and $\delta(u, v)$ require doubly nested loops. Thus the total runtime is $O(V^2)$.

2 Lazy Random Homework Solving

2.1 Part (a)

Proof: By induction on k. When $k = 1$, suppose that there are r_1 friends working on this problem. $r_1 \geqslant r$. The assignment becomes invalid when all these r_1 friends are assigned to TA Nirvan or TA Kelly. The possibility is $P[1, \text{invalid}] = 2 \times 2^{-r_1} \leqslant 2^{1-r} = k2^{1-r}$. So the statement is true for the base case.

When $k > 1$, suppose the statement holds for $k - 1$. Therefore, $P[k-1, \text{invalid}] \leqslant (1-k)2^{1-r}$. In other words, $P[k-1, \text{valid}] \geqslant 1 - (k-1)2^{1-r}$. When we add the k-th problem, we have

$$P[k, \text{valid}] = P[k-1, \text{valid}]P[1, \text{valid}|\text{assignment valid for previous } k-1 \text{ problems}]$$

where the second factor is a conditional probability: the probability for the assignment to be valid for the one-problem case (the k-th problem), under the condition that the assignment is valid for the $k-1$-problem case (the privious $k-1$ problems). If we use the unconditional probability, the equality becomes an inequality because the unconditional probability is always no larger.

$$P[k, \text{valid}] \geqslant P[k-1, \text{valid}]P[1, \text{valid}]$$

The inequality can be further relaxed,

$$\begin{aligned} P[k, \text{valid}] &\geqslant (1 - (k-1)2^{1-r})(1 - 2^{1-r}) \\ &= 1 - k2^{1-r} + (k-1)2^{2-2r} \\ &\geqslant 1 - k2^{1-r} \end{aligned}$$

which means $P[k, \text{invalid}] \leqslant k2^{1-r}$. Therefore, by induction, the statements is true for all k. Larry fails to choose a valid assignment with probability at most $k2^{1-r}$.

2.2 Part (b)

We need $k2^{1-r} < 1$, which means $k < 2^{r-1}$. This guarantees that valid assignments exist. However, this bound can be relaxed, since the inequality $P[k, \text{invalid}] \leqslant k2^{1-r}$ derived in the previous part is quite loose. I feel that the tightest (highest) bound is $k < \binom{2r-1}{r}$. I cannot prove this. However, I can show that if $k \geqslant \binom{2r-1}{r}$, there is a choice of Larry's friends (who does which problem) that makes it impossible for a valid assignment to exist. TODO

2.3 Part (c)

Suppose $k \leqslant 2^{r-2}$, then $P[k, \text{invalid}] \leqslant k2^{1-r} \leqslant \frac{1}{2}$. The chance for a random assignment to be valid is more than a half. We can thus use the following Las Vegas algorithm:

Description: We use two arrays of length k, TA_0 and TA_1, to store if Larry can get feedback from either TA on each problem. For example, if he gets feedback on problem i from TA Nirvan but not from TA Kelly, then $TA_0[i] = 1$ and $TA_1[i] = 0$. We use two integers $count_0$ and $count_1$ to store the sum of all elements in TA_0 and TA_1, respectively. If eventually $count_0 = count_1 = k$, then the assignment is valid. Also, we use an array of length n, called A, to record the assignment.

We begin a loop by randomly assign friends to TAs. This is done by generating a random bit for each friend, with equal probability being 0 or 1. Store this bit in the corresponding entry of A. If a friend is assigned 0, then check his/her done problems in TA_0. If the corresponding entry is 0, set it to 1 and update $count_0$ by adding 1. If a friend is assigned 1, then do the same thing to TA_1 and $count_1$.

When all friends are processed, if $count_0 = count_1 = k$, the assignment is valid and we jump out of the loop. Output A as the assignment. Otherwise, the assignment is not valid. Go the the start of the loop and try another random assignment.

The pesudocode is shown below.

Algorithm 3 Finding valid assignment

```
1: procedure ValidAssignment(F, n, k)
2:     ▷ Assume F is an array of length n that stores the problems done by each friend in F[i].problems
3:     while true do
4:         Initialize TA_0 and TA_1 to length-k arrays of 0
5:         Initialize A to length-n array of 0
6:         count_0 ← 0, count_1 ← 0
7:         for i = 0 : n − 1 do
8:             A[i] ← RandBit()
9:             if A[i] = 0 then
10:                for p in F[i].problems do
11:                    if TA_0[p] = 0 then
12:                        TA_0[p] ← 1
13:                        count_0 ← count_0 + 1
14:            else
15:                for p in F[i].problems do
16:                    if TA_1[p] = 0 then
17:                        TA_1[p] ← 1
18:                        count_1 ← count_1 + 1
19:        if count_0 = k and count_1 = k then
20:            break
21:    return A
```

Correctness: Unless a valid assignment is obtained, the procedure will not break the while loop. Therefore, if it terminates, it must give the correct output. If $k \leqslant 2^{r-2}$, then the expectation of number of while loops is no larger than 2. So the algorithm is expected to terminate in finite time.

Runtime: In each while loop, every friend is assigned a random bit, which costs $O(n)$. Moreover, for each friend, all the problems that he/she does are visited. The total number of problems being considered is $\Theta(kr)$, since $\Theta(r)$ friends are doing each of the k problems. So altogether, the for loop in the main while loop costs $O(n + kr)$. The rest part of the while loop costs constant time.

The while loop is expected to be executed twice. Therefore, the overall expected runtime is $O(n + kr)$, which is linear in terms of the input size (notice that $size(F) = O(n + kr)$).

2.4 Part (d)

Randomly find a friend f and let him/her try all the shoes. By doing this we find the correct shoe s for f, and we can also partition all the shoes around s. Then for every other friend, let him/her try s. By doing this we partition them around f. Then we have a subset of shoes smaller than s and a subset of friends whose feet are smaller than f, as well as a subset of shoes larger than s and a subset of friends whose feet are larger than f. We recurse on both the smaller subset and the larger subset.

The pseudocode is shown below.

Correctness: Say friend f has shoe s. By partitioning the shoes around f, all the shoes smaller than s are picked out as S_1. By partitioning the friends around s, all the friends whose feet are smaller than f are picked out as F_1. Shoes smaller than s must belong to friends whose feet are smaller than f. The same is true on the larger side. Therefore, matches are constrained within either the smaller side or the larger side. So recursively calling QuickShoeMatching finds all the matches.

Runtime: Say partition of F and S happens at the rank-k element, where k is uniformly distributed. Then

$$T(n) = T(k-1) + T(n-k) + O(n)$$

Algorithm 4 Matching shoes with friends

```
1: procedure QuickShoeMatching(F, S)
2:     if F.length = 1 then
3:         F[0].shoe = S[0]
4:     else
5:         Randomly pick pivot i in 0, 1, ··· , F.length − 1
6:         Partition S by F[i].size: S = [S_1, s, S_2], where s.size = F[i].size
7:         Partition F by s.size: F = [F_1, F[i], F_2]
8:                                  ▷ It is guaranteed that S_1.length = F_1.length and S_2.length = F_2.length
9:         QuickShoeMatching(F_1, S_1)
10:        QuickShoeMatching(F_2, S_2)
```

This is the same recursion as Quicksort. Therefore, the expected runtime of QuickShoeMatching is $O(n \log n)$.