CS5800 Algorithms *Out: October 01, 2019*

Solutions for Problem Set 3

Name: Instructor's solutions *Due: October 08, 2019*

Problem 1 (Super or normal) 20

You have one supercomputer and n normal computers on which you need to run n jobs. Each job i first spends s_i time on the supercomputer and then n_i time on the normal computer. A job can only start running on any of the normal computers after it has finished on the supercomputer. However, as soon as any job finishes on the supercomputer, it can immediately start on one of the free normal computers.

The goal is to finish running all the jobs as soon as possible. Note that since there is only one supercomputer, you'll always have to wait $\sum_{i=1}^{i=n} s_i$ time so that the jobs finish running on the supercomputer. However, you can optimize when you run the jobs on the normal computers to try to finish running all the jobs as soon as possible.

Show the following schedule is optimal: execute jobs after sorting them in *decreasing order* by n_i.

Solution:

Let the jobs be indexed by k_i for $1 \leq i \leq n$. Now suppose there's an optimal schedule $S = [n_{k_1}, n_{k_2}, ..., n_{k_n}]$ that doesn't sort the job by decreasing order of n_i. Then we have some inverted pair n_{k_i}, n_{k_j} such that $n_{k_i} < n_{k_j}$ for some $i < j$. If there is any inversion, it can be shown that there is an adjacent inversion. See recitation 4, problem 1 of the extras: there is some $i \leq l \leq j$ such that $n_{k_l} < n_{k_{l+1}}$.

For schedule S let t_i denote the time it takes for a job i to finish.

$$
\begin{aligned}
t_1 &= s_{k_1} + n_{k_1} \\
t_2 &= s_{k_1} + s_{k_2} + n_{k_2} \\
&\vdots \\
t_n &= s_{k_1} + s_{k_2} + ... + s_{k_n} + n_{k_n}
\end{aligned}
$$

The time it takes to finish every job is equal to $\max(t_1, t_2, ..., t_n)$. Now let's create the new schedule $S' = [n_{k_1}, n_{k_2}, ..., n_{k_{l-1}}, n_{k_{l+1}}, n_{k_l}, n_{k_n}]$ by swapping the adjacent inversion at $l, l+1$. Let's compare t_l, t_{l+1} in schedule S and t'_l, t'_{l+1} in schedule S'.

$$
\begin{aligned}
t_l &= s_{k_1} + s_{k_2} + ... + s_{k_{l-1}} + s_{k_l} + n_{k_l} \\
t_{l+1} &= s_{k_1} + s_{k_2} + ... + s_{k_{l-1}} + s_{k_l} + s_{k_{l+1}} + n_{k_{l+1}} \\
t'_l &= s_{k_1} + s_{k_2} + ... + s_{k_{l-1}} + s_{k_{l+1}} + n_{k_{l+1}} \\
t'_{l+1} &= s_{k_1} + s_{k_2} + ... + s_{k_{l-1}} + s_{k_{l+1}} + s_{k_l} + n_{k_l}
\end{aligned}
$$

Note that $t_l < t_{l+1}$ and $t'_{l+1} < t_{l+1}$, and $t'_l < t_{l+1}$. So t_{l+1} is the 'weakest link' (it it always larger than the other elements), so we get that $\max(t'_l, t'_{l+1}) < \max(t_l, t_{l+1})$ (strict inequality assuming all the $s_i > 0$). Thus the schedule S' is better than S, which is a contradiction as S' is supposed to be optimal. ■

Problem 2 (Medians all the way down) 5+10

Given a list of n numbers A we can choose one of the numbers $a \in A$ and in linear time use it as a pivot to split the remaining numbers into two halves; the left one consisting of everything less than (or equal to) a and the right half consisting of everything greater than a. Using this, the Median of Medians algorithm solves the k-selection problem by finding a "good enough" pivot element p. For the k-selection problem a "good enough" pivot element is any $p \in A$ that ensures that a constant fraction of the n elements in A get discarded before making each recursive call. As long as the recursive calls only needs to deal with a smaller fraction of the original input, the final recurrence solution will be linear.

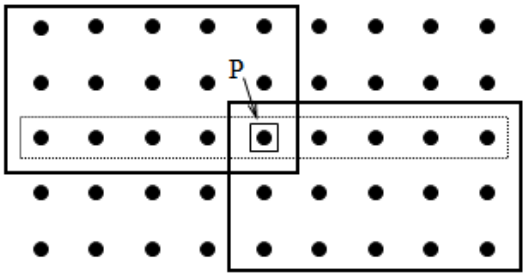

Figure 1: The Median of Medians argument when using group of 5.

The standard MoM algorithm works by first grouping all input elements A in groups of 5 and then finding the median of each such group (either sorting each group or just by brute force). Let these medians be denoted by B. The pivot p is then the median of the $\lceil n/5 \rceil$ elements of B, which is found using a recursive call to the MoM algorithm. We then perform the k-selection algorithm using p as the pivot (with $k = n/2$ if we're finding the median). We can argue that p is "good enough", more precisely that the MoM will ensure that at most $\lceil 7n/10 \rceil$ elements remain on the larger side when A is split at p. For visualization, use Figure 1 where each column is a group of 5 and the middle row consists of the elements of B.

1. Show that the Median of Medians algorithm is not linear time if we use groups of 3. Come up with the correct recurrence relation for this algorithm and using that show that this variant of MoM runs in $\Theta(n \log n)$ time.

2. Now consider a variant of the Median of Medians algorithm called the Median of Medians of Medians algorithm. At a high level MoMoM can be described as follows:

   - Divide A in groups of 3, find the medians of each group (via sorting/brute-force). Let these medians be B.
   - Now divide B into groups of 3 and again find the median of each group (via sorting/brute-force). Let these medians be C.
   - Recursively find the median of C using a call to MoMoM variant. Let this median be p.
   - Use p as the pivot to perform k-selection (with $k = n/2$ to find the median).

Intuitively, in MoMoM instead of grouping into 3 once and then making the recursive call, we group into 3 twice before recursing. Come up with the correct recurrence relation for the MoMoM variant and show that MoMoM runs in linear time.

Hint: First consider the recurrence for the MoM problem when using groups of 5, and understand where each term is coming from and how the terms would change for both the variants above.

Solution:

1. If we divide the array into groups of 3, there are obviously $\lfloor n/3 \rfloor$ groups. So the first part of the recursion is finding the median of the $\lfloor n/3 \rfloor$ medians of each group, which contributes a $T(\lfloor n/3 \rfloor)$. Now once we have the median of the $\lfloor n/3 \rfloor$ medians, we know half the groups will have two elements larger ($\lfloor n/3 \rfloor \cdot 1/2 \cdot 2 \approx n/3$ elements that we can eliminate). So we can recurse on the $n - n/3 = 2n/3$ elements remaining, and that'll be the second term in the recursion. Finally, finding the median of each group will take linear time. So our final recursion is

$$T(n) = T(n/3) + T(2n/3) + O(n)$$

   We can show that this recurrence is $\Theta(n \log n)$ using the recursion tree. At the first level we have that it does $O(n)$ work by definition. At the next level if will do $n/3 + 2n/3 = n$ work, at level 3 it'll do $n/9 + 2n/9 + 2n/9 + 4n/9 = 9n/9 = n$ and so on. At each level n work is being done and there will be $O(\log n)$ levels. Thus, the recursions solution will be $T(n) = \Theta(n \log n)$.

2. So A has $\lfloor n/3 \rfloor$ groups of 3, and then B has $\lfloor n/9 \rfloor$ groups of 3. Note that we're not finding the median of the medians of B, but only the median of medians of C. This gives us a recursive term of $T(n/9)$ since there are $\lfloor n/9 \rfloor$ medians in C If we were finding the median of B as well, then we get a recursive term of $T(n/3) + T(n/9)$ which would be too much in our context.

   Now we calculate how many elements are being eliminated based on finding the median of C. We know that 2 elements in half of the $\lfloor n/9 \rfloor$ groups of C are going to be larger than the median of medians of C. But since these were also the medians of B, we know that for each median in C there's an element in B that is larger. So in total we get at least $2n/9$ elements that are larger than the median of medians of C.

   As before, finding the median of each group of B is linear, and finding the median of each group of C is linear. So our final recursion is

$$T(n) = T(n/9) + T(7n/9) + O(n)$$

   We can say the linear term is bounded by c_2. We guess $T(n) \leq c_1 n$ for some constant c_1.

$$T(n) \leq c_1 n/9 + c_1 7n/9 + c_2 n \leq$$
$$8c_1 n/9 + c_2 n$$

   Remember, we want to show $T(n) \leq c_2 n$ so we need

$$8c_1 n/9 + c_2 n \leq c_1 n$$
$$9c_2 < c_1$$

   Since c_2 is a constant, we can set c_1 to be any value satisfying this inequality and we are done.

■

Problem 3 (Median-Heap) 30

Given a set of numbers, its *median*, informally, is the "halfway point" of the set. When the set's size n is odd, the median is unique, occurring index $i = (n+1)/2$. When n is even, there are two medians, occurring at $i = n/2$ and $i = n/2 + 1$, which are called the "lower median" and "upper median" respectively. Regardless of the parity of n, medians occur at $i = \lfloor (n+1)/2 \rfloor$ (the lower median) and $i = \lceil (n+1)/2 \rceil$ (the upper median). For simplicity in this question, we use the phrase "the median" to refer to the lower median. So we care about the $i = \lfloor (n+1)/2 \rfloor$ position.

Design and implement a data structure $\texttt{Median} - \texttt{Heap}$ to maintain a collection of numbers S that supports $\texttt{Build}(S)$, $\texttt{Insert}(x)$, $\texttt{Extract}()$ and $\texttt{Peek}()$ operations, defined as follows:

- $\texttt{Build}(S)$: Produces, in linear time, a data structure $\texttt{Median} - \texttt{Heap}$ from an unordered input array S. For implementing $\texttt{Build}(S)$, you can assume access to the Median-of-Medians procedure $\texttt{Find} - \texttt{Median}(S)$, which finds the median of S in linear time.
- $\texttt{Insert}(x)$: Insert element x into $\texttt{Median} - \texttt{Heap}$ in $O(\log n)$ time.
- $\texttt{Peek}()$: Returns, in $O(1)$ time, the value of the median of $\texttt{Median} - \texttt{Heap}$.
- $\texttt{Extract}()$: Remove and return, in $O(\log n)$ time, the value of the median element in $\texttt{Median} - \texttt{Heap}$.

Solution:

We can solve this problem by creating two heaps. One will be a min heap $minH$ that contains all the elements greater than or equal to the median. The second heap will be a max heap $maxH$ that'll contain all the elements less than or equal to median. We will place the median at the root of $maxH$. If there are an odd number of elements, then there'll be one more element in $maxH$ (this corresponds to the median being in $maxH$). If there's an even number of elements our setup matches the assumption that the median is the smaller element.

First we construct $\texttt{Build}(S)$ in linear time. We can get the median m of S in linear time by calling $\texttt{Find} - \texttt{Median}(S)$. Then we partition all the elements less than s into one set S_1 and all the elements more than s into a second set S_2. If there are multiple elements equal to s, we evenly split them between S_1 and S_2. Then we can construct $maxH$ from S_1 and $minH$ from S_2 in linear time.

This can be shown by noting that the heapify operation takes constant time at the leaves, and $O(\log n)$ time if we heapify at the root. The cost as we are building the heap is the following and the $\sum_{i=0}^{\infty} \frac{n}{2^{h+1}}$ term can be shown to converge to a constant value.

$$\sum_{i=0}^{\log n} O(h)\frac{n}{2^{h+1}} = O(n)\sum_{i=0}^{\log n} \frac{n}{2^{h+1}}$$

To insert an element, we compare to the median. If its less, we push it into $maxH$, otherwise we push it into $minH$. We might need to rebalance our two heaps, to maintain the property that they either have the same number of elements, or that $maxH$ has one more element than $minH$ (if the total number is odd). If there are too many elements in $minH$, then we can extract the root of $maxH$ and insert it into $minH$, or vice versa. All this can be done in time $O(\log n)$.

We can peek by looking at the root of $maxH$ in constant time. And to extract we pop the root of $maxH$, and then rebalance if needed. Note that throughout this, we kept reusing the properties of the heap just right so that we got the desired properties.

■

Problem 4 (Programming: Spy Escape) 30

Jim Bone, the Husky Dog secret agent, has finished his secret mission in the Kat Kingdom capital, and now has to return to the Dog District. However, the Kitties Guarding Boarders know that Jim Bone is in their country, and are attempting to stop him by spreading his picture to the kitty police at their train stations.

You must help Jim Bone escape the kitty capital by determining which trains he should take to leave the country. However, because the police are actively looking for him, Jim has decided that he cannot sit still at any one train station for too long. Thus you need to determine which trains he should take so that his layover is never too long.

Input Format:

Line 1 : N M T

- N is the max label of any train station (1 is the starting point, N is the destination outside the country). Jim Bone always starts at station 1 at time 0.
- M is the number of train rides which appear on the schedule.
- T is the max time Jim can spend at any one station without being caught by the Kitties Guarding Borders. Note that he is allowed to spend exactly T time steps at a station, but not a moment more.

Next M Lines: x y t_1 t_2.

- x is the station label where a train will be leaving from
- y is the station label where this train will arrive
- t_1 is the time the train will leave station x.
- t_2 is the time the train will arrive at station y.

Output Format:

"NO" if, no matter what trains Jim takes, he will be caught before reaching station N.
"YES T_{min}" if he can escape , where T_{min} is the earliest possible time that he arrives at station N.
At the very start, Jim is at station 1, at time 0. Suppose that Jim arrives at station x at time t. He can move to station y if and only if:

- There is a train going from x to y at time t_1, where $t_1 > t$, but $t_1 - t \leq T$, the max time he can spend at the station.
- If there is such a train ride, labeled as x, y, t_1, t_2, then after taking this train, he will be at station y at time t_2. Furthermore, he will have spent $t_1 - t$ time units waiting at station x.

Solution:

One way to approach this is to think of every train as a node. There's an edge between i and j if

1. i ends at a station from which j starts (and j leaves after i arrives)
2. If the spy can afford to stay at some the common station and wait for j (layover condition).

Once you set this up YES or NO is simply figuring out if the end station is reachable from the start (BFS/DFS). The min time is shortest path (Dijkstra which subsumes the DFS/BFS). Note that you need to account for the time you wait at a station either in Dijkstra or in the graph. The graph option might be easy to do when you setup the edges while following the above two conditions.

```
#include <bits/stdc++.h>

using namespace std;

int spyEscape(vector <vector <int> >graph[], int n, int time) {
    int ans = 0;
    int timeEscape = -1;
    vector <vector <int> > q;
    int currentLocation = 1;
    int currentTime = 0;
    vector <int> current;
    current.push_back(currentLocation);
    current.push_back(currentTime);
    q.push_back(current);
    while(q.size() > 0) {
        vector <int> temp = q.back();
        q.pop_back();
        if (temp[0] == n && (temp[1] < timeEscape || timeEscape == -1)) {
            timeEscape = temp[1];
        }
        else {
            for (int i = 0; i <graph[temp[0]].size();i++) {
                if (graph[temp[0]][i][3] == 0) {
                    if (graph[temp[0]][i][1] - temp[1] >= 0 && graph[temp[0]][i][1] - temp
    [1] <= time) {
                        vector <int> flag;
                        flag.push_back(graph[temp[0]][i][0]);
                        flag.push_back(graph[temp[0]][i][2]);
                        q.push_back(flag);
                        graph[temp[0]][i][3] = 1;
                    }
                }
            }
        }
    }

    if (timeEscape == -1) {
        cout<<"NO";
    }
    else
    {
        cout<<"YES "<<timeEscape;
    }

    return ans;
}

int main () {

    int n, m, t;
    cin>>n>>m>>t;

```

```
    int x, y, t1, t2;

    vector <vector <int> > graph[200001];

    for (int i = 0;i < m; i++) {
        cin>>x>>y>>t1>>t2;
        vector <int> a;
        a.push_back(y);
        a.push_back(t1);
        a.push_back(t2);
        a.push_back(0);
        graph[x].push_back(a);
    }

    spyEscape(graph, n, t);



    return 0;
}
```

Listing 1: C/C++ solution

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Scanner;

public class SolutionSpy3 {

  public static void main(String[] args) {
    int m, n, t;
    Scanner sc = new Scanner(System.in);
    String[] initialInputs = sc.nextLine().split("\\s");
    n = Integer.parseInt(initialInputs[0]);
    m = Integer.parseInt(initialInputs[1]);
    t = Integer.parseInt(initialInputs[2]);
    Map<Integer, List<Node>> graph = new HashMap<>(m);
    for (int i = 0; i < m; i++) {
      String[] inputLine = sc.nextLine().split("\\s");
      int x = Integer.parseInt(inputLine[0]);
      int y = Integer.parseInt(inputLine[1]);
      int t1 = Integer.parseInt(inputLine[2]);
      int t2 = Integer.parseInt(inputLine[3]);
      graph.computeIfAbsent(x, k -> new ArrayList<>());
      graph.computeIfPresent(x, (k, v) -> {
        v.add(new Node(y, t1, t2, false));
        return v;
      });
    }
    solveGraph(graph, n, t);
  }

  private static void solveGraph(Map<Integer, List<Node>> graph, int n, int t) {
    int start = 1;
    int end = n;
    List<QueueNode> queue = new ArrayList<>();
```

```
    queue.add(new QueueNode(0, start));
    int min_time_escaped = -1;
    while (queue.size() != 0) {
      QueueNode first = queue.get(0);
      queue.remove(0);
      int time = first.getT();
      int node = first.getStation();
      if (node == end && (time < min_time_escaped || min_time_escaped == -1)) {
        min_time_escaped = time;
      } else {
        for (Node node1 : graph.getOrDefault(node, new ArrayList<>())) {
          int dest = node1.getDest();
          int time_leave = node1.getT1();
          int time_arrive = node1.getT2();
          boolean seen = node1.isVisited();

          if (!seen) {
            if (time_leave - time >= 0 && time_leave - time <= t) {
              queue.add(new QueueNode(time_arrive, dest));
              node1.setVisited();
            }
          }
        }
      }
    }
    if (min_time_escaped == -1) {
      System.out.println("NO");
    } else {
      System.out.println("YES " + min_time_escaped);
    }
  }

  static class QueueNode {

    int t, station;

    QueueNode(int t, int station) {
      this.t = t;
      this.station = station;
    }

    int getT() {
      return t;
    }

    int getStation() {
      return station;
    }
  }

  static class Node {

    int dest, t1, t2;
    boolean visited;

    Node(int dest, int t1, int t2, boolean visited) {
      this.dest = dest;
```

```
      this.t1 = t1;
      this.t2 = t2;
      this.visited = visited;
    }

    int getDest() {
      return dest;
    }

    int getT1() {
      return t1;
    }

    int getT2() {
      return t2;
    }

    boolean isVisited() {
      return visited;
    }

    void setVisited() {
      this.visited = true;
    }
  }
}
```

Listing 2: Java solution

```
from collections import defaultdict

def solve(graph, N, T):
    start = 1
    end = N
    node_queue = [(0,start)]
    min_time_escaped = -1
    while node_queue:
        time, node = node_queue.pop(0)
        if node == end and (time < min_time_escaped or min_time_escaped == -1):
            min_time_escaped = time
        else:
            for i in range(len(graph[node])):
                dest, time_leave, time_arrive, seen = graph[node][i]
                if seen:
                    continue
                elif time_leave - time >= 0 and time_leave - time <= T:
                    node_queue.append((time_arrive, dest))
                    graph[node][i][3] = True

    if min_time_escaped == -1:
        return "NO"
    return "YES " + str(min_time_escaped)

if __name__ == "__main__":
    N, M, T = map(int, raw_input().split())
    graph = defaultdict(lambda: [])
    for _ in range(M):
        x, y, t1, t2 = map(int, raw_input().split())
```

```
30          # all edges are a tuple of the form (destination, time_leaves, time_arrives, seen)
31          graph[x].append([y, t1, t2, False])
32
33      print solve(graph, N, T)
```

Listing 3: Python solution

■