COMP 251 - Assignment 3

Jeremie Poisson - McGill ID 260627104

McGill School of Computer Science – November 17th, 2015

1 Sorting

We already know that operations in red-black trees all take time $O(\log n)$, where n is the number of elements in the tree. Therefore, one could use the construction of a red-black tree to store every *distinct* elements and simply keep track of the elements' count as satellite data in the node itself. Hence, every operations in the tree would take $O(\log \log n)$ time in the worst case –as we are only storing the distinct elements in the tree. The construction of such a tree requires us to either insert or update n nodes in the tree, all of which is taking $O(n \log \log n)$ worst-case time as we have a total of n elements to sort. Finally, we simply have to in-order walk the tree to output the n elements in a sorted order, making sure that the in-order walk procedure takes into account the count of the nodes stored in the red-black tree (stored as satellite data).

Algorithm 1 Sorting

```
1:  procedure SORT(x)
2:      tree = (empty red-black tree)
3:      X = (collection of integers to sort)
4:      for each node x ∈ X do
5:          node = rb_search (x, tree)
6:          if node == NIL then
7:              nNode = new node to insert initialized for x
8:              nNode.count = 1
9:              rb_insert nNode in the red-black tree tree
10:         else
11:             node.count++
12:             rb_update node in red-black tree tree
13:         end if
14:     end for
15:     rb_inorder_walk (tree)
16: end procedure
```

For the construction of the red-black tree *tree*, we are using *rb_search*, *rb_insert* and *rb_update* on *tree*. All of these operations are $O(\log n)$ time.

As we are only storing the *distinct elements* as nodes in the red-black tree, these operations are then $O(\log \log n)$ time in the worst case. In addition, the *rb_inorder_walk* is also a red-black tree operation that takes $O(\log n)$ time. As we are sorting a total of n element, this means that this algorithm is $O(n \log \log n)$ time.

2 Interval Partitioning

It is possible to design such an algorithm that schedules n lectures using a minimum number k of lecture rooms in $O(n \log k)$ worst case time. To accomplish this task, it is crucial to use a priority queue that is implemented using a binary heap data-structure, allowing us to keep track of the different classrooms that are scheduled throughout the algorithm and the end-time of the currently scheduled lecture.

Algorithm 2 Scheduling n lectures using a minimum of k classrooms

```
1:  procedure SCHEDULE(lectures)
2:      numClassrooms = 0
3:      schedule = (empty classroom priority queue with endTime as the key)
4:      for lec ∈ lectures do
5:          lastLec = schedule.peek()
6:          if (lastLec != NIL) and (lastLec.endTime ≤ lec.startTime) then
7:              schedule.pop()
8:          else
9:              numClassrooms++
10:         end if
11:         schedule.push(lec)
12:     end for
13: end procedure
```

Should the priority queue be implemented using a binary heap data-structure, *pop*() and *push*() operations are $O(\log n)$ worst case time. In addition, the *peek*() operation is $O(1)$. The priority queue is used to assign the different k classrooms to multiple lectures where the *end time* of the lectures is used in ascending order for the comparison in the priority queue (the lecture scheduled with the earliest end time would be the first out).

Therefore, *pop*() and *push*() are $O(\log k)$. As we are scheduling a total of n lectures using those *pop*(), *push*() and *peek*() operations, it follows that the algorithm is $O(n \log k)$.

3 Competition Scheduling

As the contestants are required to complete each legs of the triathlon in a specific ordering, the pool acts as a bottleneck that is delaying the competition finish time for all contestants. Therefore, there is a simple greedy rule in order to produce an optimal scheduling of the contestants so that the competition is taking the least possible time in the end.

Observation : The pool is acting as a bottleneck that is delaying the finish time for every following contestants.
It is not possible to save time using the pool as there can be only one person that is in the pool at any given time and that every contestants must be following the same ordering. Hence, all contestants must proceed in the pool anyway. In the end, the total swimming time summed over all contestants will be exactly the same regardless of the order on which the contestants were sent.

1. *Send the contestants whose (projected biking + projected running time) in decreasing order*
   The only way to save time on the finish time of the competition is to take advantage of the fact that the biking and running legs of the triathlon can both be accomplished asynchronously. Therefore, sending those whose projected biking + running time are the largest first will take full advantage over the fact that those activities can be both accomplished at the same time by all contestants (sending the slowest ones first so that they can finish as fast as possible).

Algorithm 3 Produce optimal schedule minimizing competition finish time

```
1: procedure SCHEDULE_COMPETITION(contestants)
2:     finalOrdering = (empty priority queue by (biking + running time desc))
3:     for i = 0; i ≤ n; i++ do
4:         biking = contestants[i].biking_time
5:         running = contestants[i].running_time
6:         finalOrdering.push(contestants[i], biking + running)
7:     end for
8: end procedure
```

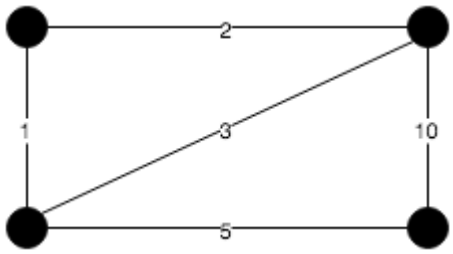
The above algorithm is fairly trivial and is only using the priority queue to sort constestants using their $biking + running$ time. The priority queue uses these sums to order the contestants in its internal structure so that they are ordered in a descending fashion (based on $biking + running$ time).

Depending on how the priority queue is implemented, the worst case running time of this algorithm will certainly change. As an example, using a priority queue implemented using a binary-heap data-structure would yield a $O(n \log n)$ worst case running time.

4 Minimum-bottleneck spanning tree

(a) *This is clearly false.*

Consider the following connected graph, containing 4 vertices and 5 edges. One must use a minimum of 3 edges in order to connect the 4 vertices and therefore create a spanning tree.



In this case, the following options (in terms of weight) to choose from when creating a spanning tree are :

$$\{1; 2; 3; 5; 10\}$$

Even though we have to choose a total of 3 edges to create a spanning tree over the above set, the minimum bottleneck value here would be 5 –picking 3 as a bottleneck value requires the edges 2 and 1 to be choosen, creating a cycle (not allowed in spanning trees). Hence, picking the following edges would yeild to a minimum-bottleneck tree:

$$\{2; 3; 5\}$$

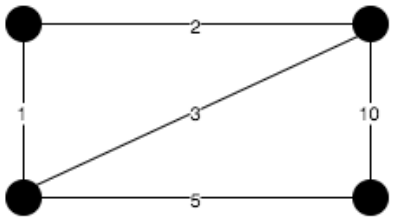
yielding a total weight of 10.

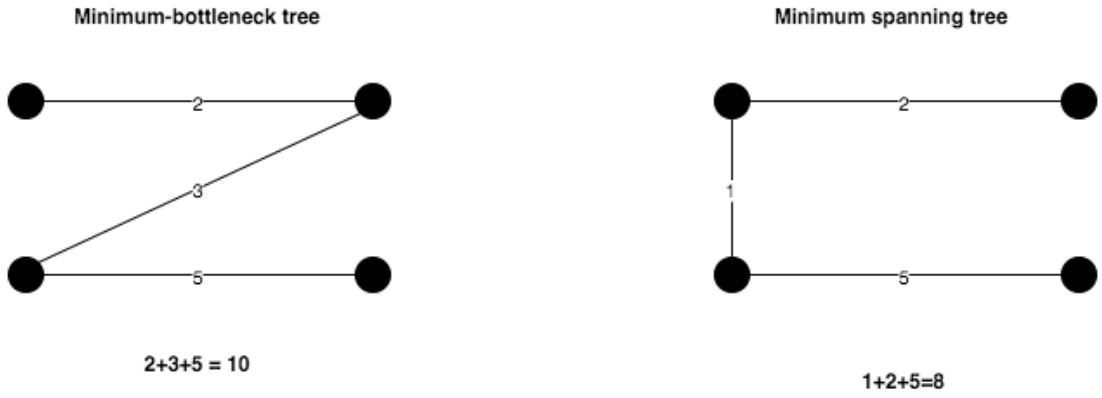
This is still a totally valid minimum-bottleneck tree as 5 is the cheapest maximum value possible.

However, the minimum spanning tree associated with this graph is given by choosing the following edges :

$$\{1; 2; 5\}$$

yielding a total weight of 8.





Hence, we have shown as a counterexample that is it possible to have a minimum-bottleneck tree that is not minimizing the total weight of the resulting spanning tree. **Therefore, a minimum-bottleneck tree is not necessarily a minimum spanning tree.**

(b) *This is true.*

Assuming that T is a minimum spanning tree of G;
Assuming that T' is a minimum-bottleneck tree of G;

- This means that T has an edge that is heavier than any other edges in T'
- Suppose that edge e is from (x, y) in T

- Since T' is also a spanning tree, there exists a path from x to y. Assuming that path is $(x, p_1, ..., p_k, y)$
- Clearly the weight of edge $e(x, y)$ from T is heavier than the edge (p_k, y) in T'
- In T, insert the edge (p_k, y) and remove the edge $e(x, y)$. Clearly, we still have a path from x to y as there is a path from x to p_k in T.
- As (p_k, y) was part of the minimum-bottleneck spanning tree, its weight must be less than (x, y). So we must have decreased the total weight of T.

However, T was defined as being the *minimum spanning tree* of G. This contradicts the definition of a minimum spanning tree. Thus, any minimum spanning tree must be a minimum-bottleneck tree as well.

5 Photocopying service

We want to find a set of greedy rules that allows the photocopying business to minimize the weighted sum of the completion times. That sum is dependent of two parameters, (i) the time in the sequence at which the job is executed and (ii) the importance of a customer i in their business.

A simple greedy rule that could provide us an optimal scheduling of jobs to minimize the weighted sum is to calculate some metric that is indicative of the impact that client i with job j have on the weighted sum. As the weighted sum is also dependent on the time of the sequence at which the jobs are scheduled, *it is essential that the metric value is being expressed on the same unitary time basis for all jobs*. This allows us to scale the impact of each and every jobs to be scheduled and therefore order each of them accordingly.

To produce the optimal scheduling of a pair of client i and job j, we simply have to calculate to following metric and order them in a decreasing fashion.

$$I_{(i,j)} = \frac{importance_i}{time_j}$$

We are first ordering those whose impact are bigger as the impact they will have on the weighted sum is also dependent of the completion time of their job j in the sequence of jobs. The more quickly you process their job, the less impact they will have in the end on the weighted sum we are trying to minimize.

Algorithm 4 Produce optimal schedule minimizing competition times

```
1: procedure SCHEDULE_JOBS(numJobs, time, importance)
2:     impacts = (priority queue of job indexes based on desc. impact order)
3:     for i = 0; i < numJobs; i++ do
4:         impact_val = importance[i] / time[i]
5:         impact.push(i, impact_val)
6:     end for
7:     finalOrdering[0..n-1] = (empty array)
8:     for i = 0; i < numJobs; i++ do
9:         finalOrdering[i] = impact.pop()
10:    end for
11: end procedure
```

We first calculate an impact value for each of the n jobs to be scheduled. This impact value is calculated by the formula above. We then use that impact value to push the job index j in a priority queue that is ordering the element in descending order of impact value (the biggest impact value will be popped out first). Finally, we simply pop all of the elements pushed in the priority queue to get them in descending order of impact value in our final ordering.

Depending on which data-structure is used to implement this algorithm in practice, it is possible to say that this algorithm is $\Omega(2n)$ at least.

6 Proofs

(a) For each edge $e(x, y)$ in G, there exists a path P_{xy} from x to $y \in H$ such that $\sum_{i \in P_{xy}} l_i < 3l_e$. *This is necessarily true*, if not, e would have been rejected and therefore H wouldn't be a proper spanning tree.

Let's call Q the set of edges on the path $P_{xy} \in H$.

It is also true that for all edges $j(s, t) \in Q$ that $l_j < 3l_{P_{st}}$ in G – since all edges j has been choosen in a previous iteration of the algorithm (they are in the tree H produced by the algorithm).

Therefore, for the edge-to-edge length of path $P_{xy} \in H$, we are effectively summing over all the length of edges in Q (that are also at most 3 times their respective path length in G).

(b) Observing the general behaviour of the algorithm, we find that the tree H produced by the algorithm cannot have any cycle having a length ≤ 4. This is mainly because of the fact that we are not adding any edges whose length is not ≥ 3 times the current shortest path in H.

Having a cycle of length > 4 also means that every nodes within a cycle has a degree of at most $\sqrt{n}$, where n is the number of nodes within that cycle. Having n nodes whose maximum degree is $\sqrt{n}$, this means that

there is at most $n^{3/2}$ edges in H as well. We have found $f(n) = n^{3/2}$, being the function of the maximum number of edges in H.

$$\lim_{n\to\infty} \frac{f(n)}{n^2} = \lim_{n\to\infty} \frac{n^{3/2}}{n^2} = \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$$

7 CluNet

It is possible to create such an algorithm that outputs a spanning tree with exactly k X-labeled edges using *the cut property of spanning trees*. However, we must make sure that such a tree even exists.

First of all, we make use of a minimum spanning-tree algorithm to generate a spanning tree containing *a minimum* of X-labeled edges. Let's call that tree X and the number of X-labeled edges (minimized) x.

We also use that same algorithm to generate a spanning tree containing *a maximum* of X-labeled edges.
Let's call that tree Y and the number of X-labeled edges (maximized) y.

It is possible to make the following observations :

- *If ($x = k$) or ($y = k$) :* X or Y are already spanning tree having exactly k X-labeled edges !
- *If ($x > k$) :* We have too much X-labeled edges –remember that we tried to minimize the number of X edges... Therefore there wont be enough Y-labeled edges to fill in the resulting tree. Hence, it is not possible to build such a spanning tree.
- *If ($y < k$) :* We don't have enough X-labeled edges –remember that we tried to maximize the number of X edges...
- *If ($x < k < y$) :* We are in a situation where we are sure that we have a sufficent number of both X and Y edges to create a spanning tree having exactly k X-labeled edges and $(n-k-1)$ Y-labeled edges. However, neither the generated tree X nor Y are properly configured to satisfy the desired properties. At this point, we can use the *cut property of the spanning trees* to generate a series of trees where the number of X-labeled edges goes from x to y. Hence, there is one tree in that series that is suitable as a resulting spanning tree containing exactly k X-labeled edges.

As expected, this algorithm is clearly $O(n^2)$ worst running time.

Algorithm 5 Generate a spanning tree containing k X-labeled edges

```
1: procedure GENERATE_SPANNING_TREE(labeled_graph, k)
2:     X = (generate MST minimizing X-labeled edges using MST alg.)
3:     x = (calculate the number of X-labeled edges in X)
4:     Y = (generate MST maximizing X-labeled edges using MST alg.)
5:     y = (calculate the number of X-labeled edges in Y)
6:     if (x > k) or (y < k) then
7:         (return failure - no such tree)
8:     else if (x = k) then
9:         (return tree X)
10:    else if (y = k) then
11:        (return tree Y)
12:    else
13:        P = (path from X - Y)
14:        n_k = x
15:        Z = X
16:        while p ∈ P and n_k ≠ k do
17:            (add p in Z)
18:            n_k + +
19:            while there is a cycle in Z do
20:                (remove a Y-labeled edge from Z to break the cycle)
21:            end while
22:        end while
23:    end if
24: end procedure
```