Homework Assignment #5

Due: Noon, 10th Dec, 2018
Submit solutions via eClass

CMPUT 204
Department of Computing Science
University of Alberta

Note: All logs are base-2 unless stated otherwise.
You should answer all questions. Your max-grade is 100.

Exercise I. Prove that any n-node tree must have at least two leafs (nodes with degree$=1$).
Answer. The solution is based on the handshake lemma. A n-node tree has $n-1$ edges, hence the sum of all degrees of all nodes in the tree is $2n-2$. Denote the number of nodes of degree 1 as x, hence $n-x$ nodes have degree ≥ 2. We get that

$$2n-2=\sum_{v}\deg(v)=\left(\sum_{v:\ \deg(v)=1}1\right)+\left(\sum_{v:\ \deg(v)\geq 2}\deg(v)\right)\geq x\cdot 1+(n-x)\cdot 2=2n-x \qquad \Rightarrow \qquad x\geq 2$$

Exercise II.

(a) Prove that in any graph with edge-weights that are distinct (no pair of edges exists with the same weight), then both Prim and Kruskal *must* output the same MST, regardless of the starting node.

Answer. If all edge-weights are distinct, the MST is unique. Since there's a single MST, both Prim and Kruskal must output the only MST of the graph regardless of the starting node.

(b) Give an example of a graph where the edge-weights aren't distinct, when Prim, Kruskal and Dijkstra *must* output the same tree regarding of the starting node.

Answer. Let G be a graph which is a tree to begin with (and make all edges have the same weight to satisfy the required...) Therefore, there's a unique spanning tree, let alone a unique MST and a unique Shortest-Paths tree. All algorithms must produce this tree.

Hint: Each question can be answered with a 1-sentence-long answer.

Problem 1. (30 pts)

(a) (15 pts) Rather then the usual (8×8)-chess board, you are given a non-standard $(t \times t)$-chess board. Still, the knight chess-piece moves on this board in the usual fashion — two squares in some direction and one square in the perpendicular direction (see Figure 1 for an example.)

Describe an algorithm that takes as input such a chess board with $(t \times t)$ squares, a start position $s = (i, j)$ and a finish position $f = (i', j')$, and finds the shortest sequence of knight-steps from s to f. Your algorithm must run in $O(t^2)$ time. Explain why it is correct and why its runtime is $O(t^2)$.

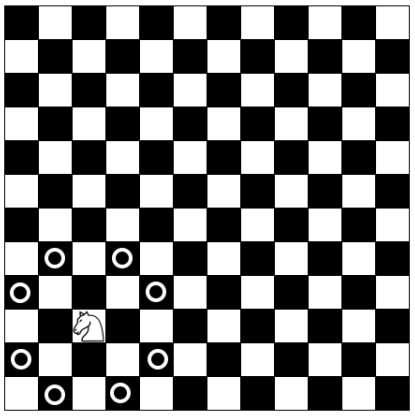

Figure 1: A large chess board. In a single move, the knight piece can move into any of the squares marked with a white cycle.

Answer. We model this problem as a graph.

- For each square (i, j), there will be a node in the graph hence the total number of nodes is t^2.
- for each (i, j) and (i', j') such that the knight is able to move from (i, j) to (i', j') in a single move, we will place an edge connecting the two squares.

Note that since for each (i, j) there could be no more than 8 possible moves. This means that the degree of each node in this graph will be ≤ 8, making the number of edges in this graph $\leq 4t^2$ via the handshake lemma.

Now, we run BFS from the start node s until we reach f. By the property of the BFS, it assigns f the *dist* field which is the min-number of edges connecting s and f. In other words, the min-number of knight moves connecting s to f. Moreover, traversing the *prdec* field of f we find a shortest-path connecting s to f, namely, a shortest sequence of instructions taking the knight from s to f.

Lastly, BFS rund in time which is linear in the number of nodes and the number of edges. There are t^2 nodes and $\leq 4t^2$ edges, so the runtime of the BFS algorithm is $O(t^2)$. Traversing the BFS-tree to find a shortest path connecting s and f is also linear in the size of the tree, $O(t^2)$. All in all our algorithm runs in time $O(t^2)$.

(b) (15 pts) You are given 3 weirdly shaped bottles, $bottle_1$ with a capacity of k liters, $bottle_2$ of capacity of l liters and $bottle_3$ of capacity of m liters. Assume $k \leq l \leq m$, all integers. In addition, next to the bottles there is a large barrel filled with water (much more than $k+l+m$ liters of water). You are allowed to fill each bottle from the barrel or empty each bottle into the barrel, and also allowed to pour water from $bottle_i$ into $bottle_j$ until either $bottle_i$ is empty or $bottle_j$ is filled to the brim.

Give an algorithm that takes as an input a, b, c such that $a \leq k, b \leq l$ and $c \leq m$, and the finds the shortest sequence of fill/empty/pour instructions that starts with all 3 bottles being empty and ends with a liters of water in $bottle_1$, b liters of water in $bottle_2$ and c liters of water in bottle 3. What is the runtime of your algorithm?

E.g., suppose $k = 4, l = 5$ and $m = 9$. In order to get to a state where we have 4 liters of water in $bottle_1$, 0 liters of water in $bottle_2$ and 6 liters of water in $bottle_3$ here is a possible sequence of instructions:
fill $bottle_2$; pour $bottle_2$ to $bottle_1$ (now $bottle_2$ has 1 liters of water); empty $bottle_1$; pour $bottle_2$ to $bottle_1$ (now $bottle_1$ has 1 liters of water); fill $bottle_3$; pour $bottle_3$ to $bottle_1$ (now $bottle_1$ has 4 liters and $bottle_3$ has 6 liters).

Answer. Again, we model the problem as a graph.

- For each $0 \leq k' \leq k, 0 \leq l' \leq l, 0 \leq m' \leq m$ we put a node indicating a state where $bottle_1$ has k' liters, $bottle_2$ has l' liters and $bottle_3$ has m' liters. Thus the graph has $(k+1)(l+1)(m+1)$ many nodes.

- In every state (k', l', m') there are 12 possible moves we can make (fill each of the 3 bottles, empty each of the 3 bottle, pour from i to j). For each state (k', l', m') and each move we check if by following this move we reach a new state $(k", l", m")$ and if so — we put a *directed* edge from (k', l', m') to $(k", l", m")$.

Note that is it crucial for the edges to be directed, since (as opposed to the knight moves) it doesn't necessarily mean we can move from state $(k", l", m")$ back to (k', l', m'). (E.g., $k = 10$ and $k' = 8$. By pouring k' liters back to the barrel it doesn't mean we can directly fill back 8 liters into the first bottle...) Also note that there are nodes and moves where we do not get into a new state (e.g. $k' = 0$ and we empty $bottle_1$). However, the out-degree of each node is at most 12 making the number of edges in the graph at most $12(k+1)(l+1)(m+1)$.

Lastly, we run BFS from the starting state ($k' = l' = m' = 0$) to the required end state. If the end-state is reachable, the BFS tour sets that node's *dist* field to the shortest sequence of moves that takes us from empty bottles to the desired state (a, b, c). BFS takes time linear in the number of nodes and edges, so this takes $O(klm)$. Lastly, traversing the BFS-tree to find a shortest sequence of moves that gets us to the state (a, b, c) also takes linear time in the tree-size, so the overall runtime is $O(klm)$.

Problem 2. (10 pts)

(a) (4 pts) Show that for any $n \geq 3$ there exists a connected graph with n edges where all nodes are not an articulation point; yet all connected graphs with $\leq n-1$ edges must have at least 1 articulation point.

Answer. The requires graph is a cycle. In a cycle with n nodes, removing any one node leaves the remaining $n-1$ nodes with a path connecting them, hence no node is an articulation point.

Any connected graph over n nodes with $\leq n-1$ edges must be a tree with exactly $n-1$ edges. (A tree is minimal in regard to connectivity, removing any edge must disconnect the graph.) Any tree must have a node of degree ≥ 2. That is due to the handshake lemma: the sum of degrees is $2(n-1) = 2n-2$ whereas all nodes of degree 1 contribute at most n to this sum, and we have that $n < 2n-2$ since $n \geq 3$. The removal of this node of degree ≥ 2 leaves its neighbors disconnected of one another and so this node is an articulation point.

(b) (4 pts) Give an example of an undirected graph with n nodes and $m = \Theta(n)$ edges (you can have $m = 2n$) where the number of cycles is exponential in n.

Answer.

This is basically a graph composed of a double cycle. Assuming n is even, we denote $n = 2t$ and have the $2t$ nodes $(1,1),(1,2),(2,1),(2,2),(3,1),(3,2),(4,1),(4,2),....,(t,1),(t,2)$. In each pair of nodes $\{(i,1),(i,2)\}$ we connect both nodes to the following pair $\{(i+1,1),(i+1,2)\}$, and of course we connect each node in the last pair $\{(t,1),(t,2)\}$ to each node in the first pair $\{(1,1),(1,2)\}$.

Since all $2t$ nodes come in pairs, and each node is connected the two nodes in the preceding pair and in the following pair, then each node in this graph has degree 4. Applying the handshake lemma we have that the total number of edges is $\frac{4n}{2} = 2m$. An example of a such a construction of a graph with $n = 14$ (so $t = 7$) nodes in given in Figure 2.

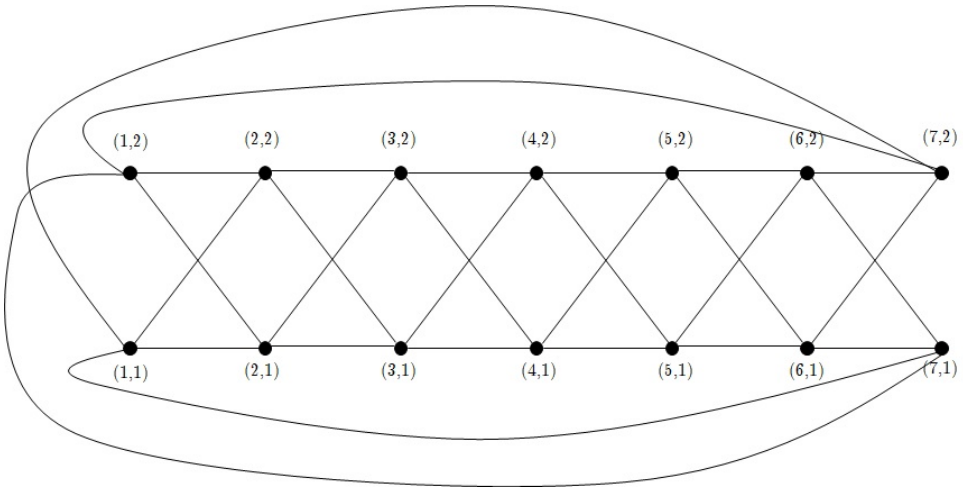

Figure 2: An example of a graph with n nodes and $2n$ edges

This graph has many many cycles. But let's just look at a subset of these cycles: cycles of length t that involve one node in each pair. For each pair i between 1 and t we can pick one of the two nodes in the i-th pair to participate in this cycle, and so we have just 2^t possible cycles just of this type. Thus, the total number of cycles on this n-node $(2n)$-edges graph is at least $2^t = 2^{n/2}$.

(c) (2 pts) Given a undirected graph $G = (V, E)$, an *orientation* of G is a *directed* graph $G' = (V, E')$ over the same set of nodes, such that for each undirected edge $e = \{u, v\}$ in G we place exactly one directed edge in G' — either the directed edge (u, v) or the directed edge (v, u).

Prove that for any undirected graph G there exists an orientation G' which is acyclic.

Answer. Basically, we force a topological order. We number the nodes of the graph from 1 to n and

orient each edge $\{u, v\}$ so that it leaves the smaller of the two nodes and enters and larger of the two nodes. Since all edges respect this topological order, the graph has no cycles.

Problem 3. (20 pts) I am taking my jeep to and going to tour the desert. On a full tank of gas, it can drive a distance of d kms. Of course, each stop for gas is time wasted as I am not able to concentrate of the view, so I want to minimize the number of gas stops I make along the way.

(a) (8 pts) First I'll be traveling along the only highway crossing the Australian outback, from the north-end of this high-way to its south-end. My plan is to stop at the furthest possible gas station which is still within distance $\leq d$ from the last stop. Will this plan minimize the number of stops along the way?
If your answer is 'yes' – prove it. If your answer is 'no' – give a counter example and propose a different approach for minimizing the number of gas stops.
Answer. Yes, this algorithm, that works by sorting the gas-stations according to location along the highway, and picks the furthers station which is still reachable from the current location.

```
Procedue Travel(A) **A is the array of gas-stations locations
    Sort(A)
    loc ← 0
    stops ← 0
    i ← 0
    while (i ≤ n)
        while (i < n and A[i + 1] ≤ loc + d)    ** As long as the next station is reachable
            i ← i + 1
        loc ← A[i]
        stops ← stops + 1
```

First, sorting A takes $O(n\log(n))$ time using, say, MergeSort. Then it is evident that for any time we increment i we only do $O(1)$-work, so the rest of the algorithm takes $O(n)$. This results in an overall runtime of $O(n\log(n))$.
As for correctness:
The optimal substructure property is simple to see. Whichever station we pick as first, we better make the minimal number of stops on the remaining waterholes in order to minimize the number of stops we make overall.
The substitution property is also quite simple. Let $A[i_1], ..., A[i_t]$ be a sequence of gas-stations such that $A[i_1] \leq d$ (we can reach the first stop from the origin) and such that any consecutive two stations are of distance $\leq d$. We show that we can replace the first stop with the furthest waterhole from the start. If $A[i_1]$ isn't the furthest waterholes with distance $\leq d$ then replace it with the $A[furthest]$ and remove from the sequence any stop that comes before $A[furthest]$. The new sequence, $A[furthest], A[i_j], ..., A[i_t]$ is still a solution to the problem — the new first stop is within distance d of the origin, and $A[i_j] - A[furthest] \leq A[i_j] - A[i_{j-1}] \leq d$ so any pair of consecutive stops is within distance d.

(b) (12 pts) Then I will be traveling through the Sahara, from its North-West end in Morocco to its South-East end in Sudan. The desert is scattered with many villages throughout (each with its local gas station), and it doesn't really matter to me which ones I'll visit along the way. While I do plan on always driving in a direction somewhere between south and east, my plan is the same as before: to stop at the furthest possible gas station which is still within distance $\leq d$ from the last stop. Will this plan minimize the number of stops along the way?
If your answer is 'yes' – prove it. If your answer is 'no' – give a counter example and propose a different approach for minimizing the number of gas stops.
Answer. Unfortunately, in this case the greedy strategy may result is a sequence of stations which is strictly larger than the optimal sequence. A counter example is given in Figure **??**. We start at the top-left point and conclude at the bottom-right point. The greedy approach causes us to south (it is the furthest point from the start point which is still within distance d from the start point), and then, heading east, we traverse a sequence of points each within distance, say, of $\frac{3d}{4}$ of one another. In contrast, the optimal

tour heads east from the starting point — where distances, except for the very first pair, are exactly d — and finally takes one hop south towards the end point.

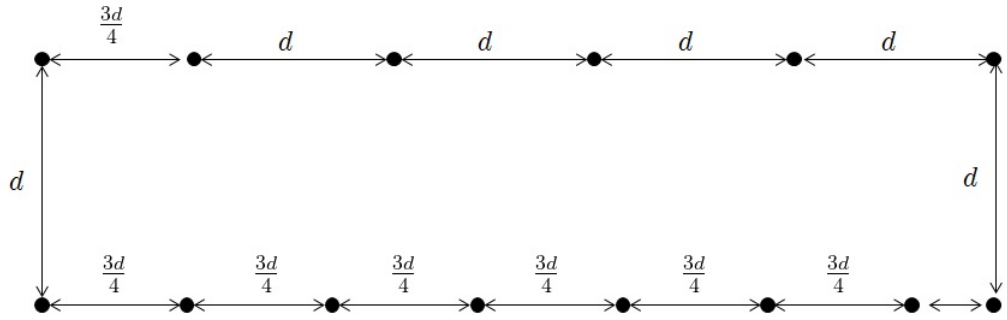

Figure 3: An example of an input where the greedy approach fails.

What does work in this case? Modeling this problem as a graph. The nodes are the start-point, end-point and all gas-stations/villages in the desert. For each pair of nodes we check is their distance is $\leq d$ and if so, add an edge to the graph. Finally, we do a BFS on this graph and we are guaranteed to find a path of fewest possible stops from the start point to the end point.

The runtime of this algorithm is $O(n^2)$ for constructing this graph and then $O(n+m)$ for the BFS. Since $m \leq \binom{n}{2}$ then the runtime of the BFS algorithm is $O(n^2)$ as well. So all in all, in time $O(n^2)$ we find the shortest sequence of gas-stations that takes from the start-point to the end-point.

Problem 4. (20 pts) A d-dimensional box is specified by a d-tuple $B = (l_1, l_2, .., l_d)$ where l_j is the length of the box along axis j. It is possible to nest box $B_1 = (l_1, .., l_d)$ inside box $B_2 = (m_1, m_2, ..., m_d)$ if we can rotate the d-axes so that under the rotation B_2 is wider than B_1 along all axes. I.e. if there exists a permutation π on the d coordinates such that $l_{\pi(1)} < m_1$, $l_{\pi(2)} < m_2$, ..., $l_{\pi(d)} < m_d$.

(a) (10 pts) Even though there are $d!$ possible permutations, give an efficient algorithm to determine if it is possible to nest box B_1 into box B_2.

Answer. The algorithm for discerning whether B_1 can be nested into B_2 is as follows:

```
Sort the axes-lengths of B_1 from smallest to largest.
   Denote those under the sorted order as l_1, l_2, ..., l_d.
Sort the axes-lengths of B_2 from smallest to largest.
   Denote those under the sorted order as m_1, m_2, ..., m_d.
for (i from 1 to d) do
   if (l_i ≥ m_i) do
      return FALSE
return TRUE
```

First, it is clear that this algorithm runs in time $\Theta(d \log(d))$ as it does two sorting operations over d elements and then iterates $O(d)$ times through a loop that requires $O(1)$ work per iteration.

Secondly, it is clear that if the algorithm returns `TRUE` then we can nest B_1 inside B_2: under the permuted order of the axes for each box we have asserted that $\forall i, \;\; l_i < m_i$.

But why is it the case that if B_1 can be nested into B_2 then our algorithm returns `TRUE`? We argue that if a permutation π exists such that $\forall i, \;\; l_{\pi(i)} < m_i$ then, under the ordering where $l_1 \leq l_2 \leq l_3 \leq ... \leq l_d$ and $m_1 \leq m_2 \leq ... \leq m_d$ we also have that for all i it holds that $l_i < m_i$. The proof is via induction on d. The base case is easy, as $d = 1$ where the claim vacuously holds (as ever, if there's just one axis, then l_1 is the smallest element by definition and so it m_1).

Fix $d \geq 2$. Assuming the claim holds for $d-1$ axes, we show it also holds for $d+1$ axis. Take l_d, the largest axis of the box B_1. If $\pi(d) = d$ then we are done — since it means that the first (the smallest) $d-1$ axes of B_1 can be nested into the first $d-1$ (the smallest) $d-1$ axes of B_2 and by I.H. we have that thus $l_i < m_i$ for each $i \leq d-1$ and $l_d = l_{\pi(d)} < m_d$ as required. So assume $\pi(d) \neq d$. All we do is to construct a different permutation π' that shows that B_1 can be nested into B_2 and under which $\pi'(d) = d$.

So take π, a permutation that shows B_1 can be nested into B_2, where $\pi(d) = i \neq d$. This means that $l_i = l_{\pi(d)} < m_d$. Moreover, since π is a permutation it also means that for some $j \neq d$ we have $\pi(j) = d$, thus $l_d = l_{\pi(j)} < m_j$. Note that since m_d is the largest axis of B_2 then $m_j \leq m_d$ and so $l_d < m_j \leq m_d$, and since l_d is the largest axis of B_1 then $l_i \leq l_d < m_j$. So this the alteration we make to transition from π to π': we set $\pi'(d) = d$ and $\pi'(j) = i$, and all other coordinates remain the same and so any $k \neq j, d$ we still have $l_{\pi'(k)} = l_{\pi(k)} < m_k$. This is the required π' and we are done.

(b) (10 pts) Give an efficient algorithm that takes as input n such d-dimensional boxes $B_1, ..., B_n$, and returns the longest possible sequence of boxes $B_{i_1}, B_{i_2}, ..., B_{i_t}$ which can be nest inside one another sequentially. Namely, B_{i_1} can be nested inside B_{i_2}, B_{i_2} can be nested inside B_{i_3} and so on, so all t boxes will stored nicely in your (d-dimensional) cupboard.

Answer. We set it as a graph problem. The n nodes of the graph are boxes and we put a directed from B_i to B_j if B_i can be nested inside B_j. Writing down this graph takes $O(\binom{n}{2} \cdot d \log(d)) = O(n^2 d \log(d))$.

We argue that this graph is a DAG. Why is that? The reason is that nesting is a transitive relation: if B_1 can be nested into B_2 using the permutation π and B_2 can be nested into B_3 using the permutation σ, then the composition of the two permutations, $\sigma \circ \pi$ shows that B_1 can be nested into B_3. So now, if there's a cycle in this graph, this means a box B can be nested into itself, but since the relation uses strong inequalities it is easy to see that a box cannot be nested into itself. (If you truly wish to be formal: we cannot have a permutation π such that $l_{\pi(i)} < l_i$ for all i since otherwise: $\sum_i l_i = \sum_i l_{\pi(i)} < \sum_i l_i$).

Having established this graph is a DAG, we apply the algorithm that finds the longest path in a DAG. Each path corresponds to a sequence of nested boxes and a longest path is clearly a longest sequence of nested boxes. This algorithm runs in time $O(n + n^2) = O(n^2)$ (since there are at most $n(n-1)$ edges in this graph. Overall, our algorithm runs in time $O(n^2 d \log(d))$.

Problem 5. (20 pts) Consider the following algorithm, whose input is a connected undirected graph G with a non-negative weight function w on its edges.

```
procedure SomeTree(G, w)
Run DFS on G starting with some node.
Let T be the DFS tree.  Let B be all the back-edges and denote b = |B|.
foreach (e = (u, v) ∈ B in an arbitrary order) do
   find e', heaviest edge on the path in T connecting u to v.
   if (w(e') > w(e)) then
      update T ← T \ {e'} ∪ {e}
return(T)
```

Does this algorithm return a MST? (Prove or give a counter example.) Regardless of correctness – for what values of b is the WC-runtime of the `SomeTree` algorithm faster than the WC-runtime of Kruskal's algorithm?

Hint: First prove the following claim, and then use it to make your argument about the output of `SomeTree`. Claim: For any $0 \leq i \leq b$ let T_i denote the tree that `SomeTree` maintains after iteration i of the `foreach`-loop. Fix any two distinct vertices $u \neq v$ in the graph and let $f_{u,v}(i)$ denote the weight of the heaviest edge on the tree path in T_i that connects u and v. Then $f_{u,v}(i)$ is a monotone non-increasing function of i (namely, for any u, v and for all i we have $f_{u,v}(i) \geq f_{u,v}(i+1)$).

Answer. Let us first prove the claim given in the hint.

Fix two distinct vertices u and v and let i be some iteration of the `foreach`-loop. We show that $f_{u,v}(i) \geq f_{u,v}(i+1)$ which proves that $f_{u,v}$ is a monotone non-increasing function of i.

At the following iteration, there could be two cases: either the $u \to v$ path on T_i was altered or not. If it wasn't altered at all, then by definition $f_{u,v}(i) = f_{u,v}(i+1)$. Otherwise, this path was altered: we removed some edge e' from this path and replaced it with a new edge e. Note that by removing e' we disconnect u from v and so it must be the case that the new tree-path from u to v uses e. Note that the new path $u \to v$ might use a very different set of edges, but here is what we do know. Let us denote $e' = (u', v')$ and C as the edges of the cycle on T_i the e closes (the cycle from which we have omitted e'). Then there's a new $u \to v$ path that goes from $v \to v'$, from $v' \to u'$ through the edges of the cycle C, and then from $v'tov$ through the edges of T_i. This path isn't necessarily a simple path (we might traverse a few edges back and forth), but what we do know is that the new $u \to v$ path on T_{i+1} uses a subset of edges from the old $u \to v$ path on T_i and the edges of the cycle C. This is illustrated in Figure 4

Now, let e^* be the heaviest on the $u \to v$ path on T_i, the one where $w(e^*) = f_{u,v}(i)$. For all edges on the $u \to v$ path on T_i, by definition, their weight is $\leq w(e^*)$. For all edges on the cycle C we have that their weight is at most $w(e')$ as this was the heaviest edge on this cycle, and since e' belongs to the $u \to v$ path on T_i, these edges too have weight $\leq w(e^*)$. Therefore, all edges on the new path from u to v on T_{i+1}, which – as we established above – are contained in the set of edge of the $u \to v$ path T_i and the edges of the cycle, are of weight $\leq w(e^*) = f_{u,v}(i)$. It follows that the heaviest edge on the new $u \to v$ path in T_{i+1} is also $\leq f_{u,v}(i)$, and so $f_{u,v}(i+1) \leq f_{u,v}(i)$. This concludes the proof of the claim.

Now, with this claim, let us prove that `SomeTree` produces a MST, using the characterization that for any edge that doesn't belong to T_b, the output of `SomeTree`, that edge must be the heaviest edge along the cycle it creates. (T_b is the output tree, the one we get after all b iterations of the `SomeTree`-loop.) So take any edge $e = (u, v)$ that doesn't belong to T_b. There are two possible cases, both are quite simple once the claim has been proven:

- Case 1: e never belonged to any of $T_0, T_1, T_2, ..., T_b$.

  This means that $e \notin T_0$, the original DFS tree, and so $e \in B$. In particular, at some iteration i we considered the edge e and decided not to update the tree and place e in the tree. This means

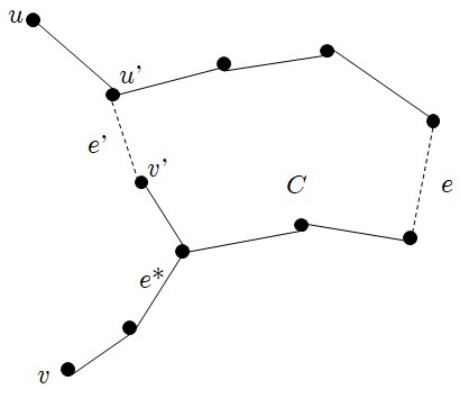

Figure 4: An update of T_i: the edge e closes the cycle C with the edges of T_i, out of which e' is the heaviest edge and so we replace e' with e and as a result update the $u \to v$ path.

e was the heaviest edge along the cycle it closed with T_i, and so $w(e) \geq f_{u,v}(i)$ and by the claim, $f_{u,v}(i) \geq f_{u,v}(b)$. Hence, $w(e)$ is still the largest on the cycle it closes with T_b.

- Case 2: e did belong to some T_i but at a later iteration j we took out e.

  If $e \in T_i$ then the $u \to v$ path on T_i was composed of the single edge e. Thus $f_{u,v}(i) = w(e)$. That is enough to conclude this case, because the monotonicity claim above states that $f_{u,v}(b) \leq f_{u,v}(i)$ and so e is the heaviest edge on the cycle it closes with T_b.

This proves that the output of `SomeTree` is a MST.

What about the runtime of this algorithm? Assume the input graph has n nodes and $m = (n-1) + b$ edges. Finding the DFS tree and the set B takes $O(n+m) = O(n+b)$ time. For each $e \in B$ we find the connecting path on the tree using a graph traversal over the tree (with $n-1$ edges), so this takes $O(n)$ time per traversal, so, all in all, the WC-runtime of this algorithm is $\Theta(n + b + b \cdot n) = \Theta(bn)$. Solving

$$bn < (n+b)\log(n) = n\log(n) + b\log(n) \qquad \Rightarrow \qquad b < \frac{n\log(n)}{n - \log(n)} \overset{0 \leq \log(n) \leq \frac{n}{2}}{\approx} \frac{n\log(n)}{n} = \log(n)$$

we see that if $b = o(\log(n))$ then the WC-runtime of `SomeTree` is asymptotically smaller than the WC-runtime of Kruskal.