CS 124 Data Structures and Algorithms — Spring 2015
Problem Set 3 solutions

Problem 1

In the Minimum Spanning Tree (MST) problem we would like to find a spanning tree whose *sum* of weights is minimum when compared to all other spanning trees. An Internet Service Provider (ISP) might use the solution to such a problem to determine the network of least cost (and therefore most profit) where each edge cost is measured in dollars ($). Suppose instead that the ISP wishes to maximize their profit in another way. The cost of each edge now measures the bandwidth (in Mb/s), and the ISP wishes to find a spanning tree (of its network) whose *maximum* bandwidth is minimum, thereby guaranteeing the smallest possible maximum transfer speed on the network.
In the following three problem parts, you may assume you have access to a "selection algorithm" $T(A, k)$ such that, given a length n array A of numbers and an integer $1 \leq k \leq n$, $T(A, k)$ outputs the kth smallest number in A (breaking ties arbitrarily). For example $T([6, 10, 5, 1], 2)$ would return 5. $T([6, 10, 5, 5], k)$ would return 5 both for $k = 1$ and $k = 2$ and would return 6 for $k = 3$. The running time of T is $\Theta(n)$.

(a) Before hiring you, the ISP wants to make sure you know your stuff. Show that any MST is a solution to the ISPs new problem.

(b) Great! However, the ISP still isn't satisfied. Show that the ISPs new problem can be solved in time $O(n + m)$ when all bandwidths are distinct in an undirected network with n customers and m links.

(c) For the final touch (before you get the job!), modify your algorithm from (b) to handle the case when bandwidths might be equal.

Solution The problem of finding a spanning tree whose *maximum* weight is minimum is better known as the Minimum Bottleneck Spanning Tree (MBST) problem.

(a) What the question is asking us to show is that if T is an MST then T is an MBST. We can proof this by contradiction. We will make the assumption that there exists some tree T which is an MST but is not and MBST, and then show that this allows us to create a new tree T'' whose weight is less than that of T. Clearly, this is a contradiction on the fact that T is an MST.

Suppose we have an MST T which is not also an MBST. Then there exists and edge e in T of weight strictly larger than the max weight in some MBST T'. That is, $w(e) > w(m)$ where m is the maximum weight edge in T'. If we remove the edge e from T and then bridge the cut using some edge e' in T' (which must exist because T'

is a spanning tree. See Figure 1), we have a new spanning tree, T''. The weight of this new spanning tree is given by

$$\begin{aligned} w(T'') &= w(T) - w(e) + w(e'') && \text{(by construction)} \\ &\leq w(T) - w(e) + w(m) && (w(e'') \leq w(m) \text{ since m is the maximum edge}) \\ &< w(T) && (w(e) > w(m)) \end{aligned}$$

This contradicts our assumption that T is an MST because we have just constructed a new tree T with lower weight.

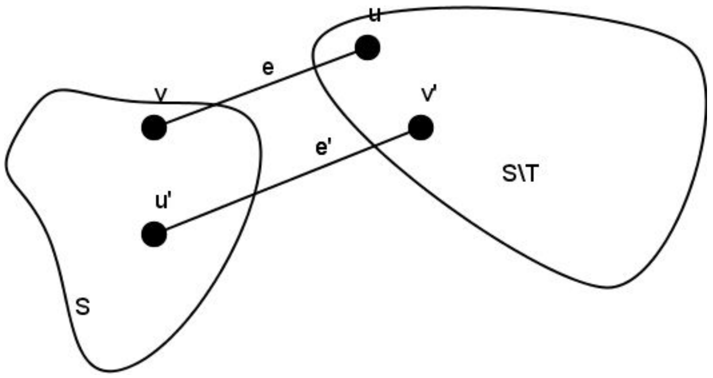


Figure 1: The ST cut is bridged by edge $e \in T$ and edge $e' \in T'$ where $e \neq e'$.

(b) We're told to find a linear time algorithm for finding an MBST of a graph G, which at first sight, appears rather difficult. However, we begin by taking a similar approach as the one we took when constructing MSTs.

First, is there anything similar to the cut property of MSTs for MBSTs? Intuitively, we took advantage of the fact that the edge of least weight is always in an MST. In MBST, we can similarly reduce our candidate edges. For example, take the graph G and remove the edge of maximum weight, e. If removing this edge leaves a connected graph, then we know that e is not in any MBST. However, if removing this edge disconnects the graph, then we know that e must be in some MBST. This approach of removing one candidate edge at a time is rather slow (we'd do it $O(m)$ times). How can we remove more candidates?

We can take a divide and conquer approach to the problem of reducing the number of edges we process on each step. Let $G = (V, E)$ be our input graph. On each step, we can partition E into two sets, L (the small edges) and M (the large edges) of roughly equal size. We do this by finding the median edge weight, m, which we can do in linear time using the Select algorithm with input $\frac{|V|}{2}$. Then $e \in L \iff w(e) \leq m$ and $e \in M \iff w(e) > m$. At this point, note that if $G' = (V, L)$ is connected, then the MBST of G' is the MBST of G (this is a generalization of the above argument). Therefore, we can simply recurse on $G' = (V, L)$ to find the MBST of G.

In the case where $G' = (V, L)$ is disconnected, we require some edges from M in order to complete a spanning tree of G. However, note that for each connected component of $G' = (V, L)$, any spanning tree of that connected component will belong to an MBST because it won't affect the maximum weight edge of the MBST (the maximum edge will come from the set M). Therefore, we can just take any such spanning tree (which we can find using DFS) and mark these edges as part of our solution MBST. After marking these edges, we can collapse the input graph G into a new graph G'' where each connected component of G' becomes a single super-vertex and the only edges of G'' are those from M. Then simply recurse on this new graph to find its MBST. Finally, the base case for the recursion occurs when the set E contains a single edge.

The above leads to the following algorithm:

```
procedure MBST (V,E)
if |E| = 1 then return E else
  L = edges whose weight are no more than the median
  M = E - L
  F = forest of G = (V,E)
  if F is a spanning tree then
    return MBST(V,L)
  else
    return MBST(V',E')
```

where $G = (V', E')$ is the graph composed of super-vertices by considering each connected component of G as a vertex.

The running time of the algorithm can be defined recursively. On each call, we first find the median, which can be done in $O(m)$ using the Select algorithm. We then find a forest of G which can be done with DFS or BFS in $O(m + n)$. Therefore, there is a linear amount of work on each iteration, and the work is cut in half on each call. This gives a recurrence of $T(n + m) = T((n + m)/2) + O(n + m)$ which we've seen to be linear, or $O(n + m)$.

Note: For a detailed depiction (with figures), check the Wikipedia page on Camerini's algorithm for undirected graphs:
`http://en.wikipedia.org/wiki/Minimum_bottleneck_spanning_tree`.

(c) The problem with the algorithm from (*b*) when edge weights could be equal is that each recursive call doesn't necessarily reduce the problem size. In fact, $|L|$ might not decrease (edges all of weight 1). To fix this, if there's one connected component we also see if there's only one when taking the largest edge weight strictly below the median weight. If there is, we use this restricted partition of edges in our recursive call.

If there's not, we can just take all the edges tied with the median into our solution set. This will result in a set of edges which do not form a spanning tree, but all of which belong in an MBST. Therefore, we can simply use DFS to filter out cycles, adding linear time complexity to our algorithm.

Problem 2

As you'll recall from lecture, the number of spanning trees of a graph G can grow rather quickly (there are n^{n-2} for the complete graph). A Civil Engineer is currently considering a road network for his hometown and, as is natural, wishes to minimize the cost of building such network. The road network can be thought of as a graph $G = (V, E)$ where each $e \in E$ is a road segment connecting two intersections, $u, v \in V$. Suppose the Civil Engineer has access to the SpanTree software suite (through his job). SpanTree works as follows: when given a weighted connected undirected graph G on n vertices, SpanTree(G) returns the number of spanning trees of G modulo 2015 in time $T(n)$. Note the lack of the word "minimum". Using the SpanTree software as a black box, the Civil Engineer wants to figure out just how many possible road networks fit his criteria, modulo 2015. Show that the number of minimum cost road networks of G modulo 2015 can be computed in time $O(nT(n) + m \log n)$. You can assume $T(\cdot)$ is a monotonically increasing function, and for the purposes of the story, will be small enough for the Civil Engineer to be home in time for dinner.

Solution The key to the problem is to recall that any MST must contain the edge of least weight. Given a graph $G = (V, E)$, consider the graph on V consisting of only all edges with the minimum weight. Then note that there exists a MST of G, say T, such that any spanning tree of each connected component of the graph will form a subtree of T. We give a construction of T: first we collapse all connected components into single vertices, find the MST of the resultant graph, and expand collapsed vertices back into their spanning trees. This is clearly an MST of the entire graph G.

The above leads to an outline for an appropriate algorithm.

Given $G = (V, E)$, first sort E by weight in increasing order. For each cost c starting with the smallest one, consider $G' = (V, E')$ where $e \in E' \iff w(e) = c$. Then for each connected component of G', C, count the number of spanning trees t_C of C using SpanTree. By the above, we know each spanning tree forms part of some MST of G, so to count the total number of MSTs, we can multiply all these t_C values together modulo 2015. However, we know that an MST must span the entire graph, not just subgraphs. Therefore, to compute the other possible MSTs, we merge all connected components into single vertices. This gives us a new graph, G_α with one fewer edge weight (and at least one less vertex). Then the total number of MSTs mod 2015 of G is given by the number of MSTs of $G_\alpha \times \prod_C t_C$ mod 2015, so we can simply recurse.

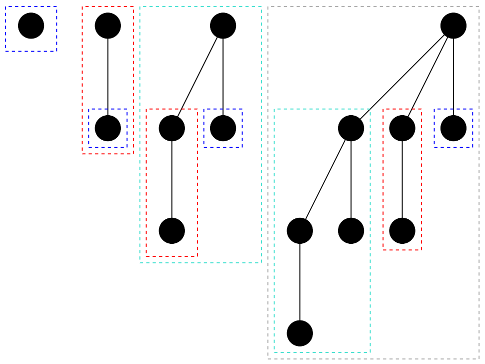
To show that the exact count is correct, we also need to prove that any MST of G must contain a spanning tree of each connected component (this is the other direction from proving that any spanning tree of each connected component forms part of some MST of G). The proof is by contradiction: suppose a connected component connects vertices V' and there exists some MST of G, T, such that the subtree of T connecting vertices V' (let this be S) does not only use minimum weight edges (of weight w). Take the maximum weight edge (of weight w') in subtree connecting V' of T (let this be S') and replace it with an edge in S that crosses the cut. Since we know that $w < w'$, we have found another MST T' such that $w(T') < w(T)$, a contradiction.

Since $T(\cdot)$ is monotonically increasing, we can evaluate SpanTree on any C in time at most

$T(n)$. Every time we make a call to SpanTree we collapse a connected component to a single vertex, decreasing n by at least 1. Thus this happens at most $n-1$ times, giving $O(nT(n))$ work total for all such calls combined. Sorting the edges, which we only need to do at the beginning, takes $O(m \log n)$.

Problem 3

In class we saw that with the disjoint forests data structure, if you use both "union by rank" and "path compression", a sequence of n make-set and $m \geq n$ find and union operations takes time $O(m \log^* n)$. As you probably noticed, the proof for that bound is non-trivial, which might leave you wondering, could I possibly simplify the data structure and still achieve this bound? Along that train of thought, suppose we use only one of "union by rank" or "path compression", but not both. Show that if we only use union by rank, there is an infinite family of operation sequences (with m, n going to ∞) such that the total runtime to serve a sequence in this family is $\Omega(m \log n)$. Similarly, show that if we only use path compression, we can also make the total time $\Omega(m \log n)$. In both of your example sequences, m should be $\Theta(n)$. **Hint**: you may find the following *sequence* of trees useful to think about:



Solution If we only use union by rank without path compression, we can give a sequence of n make-set and $m \geq n$ find and union operations that takes time $\Omega(m \log n)$. At a high level, because there is no path compression, we can build up a deep tree and continue to call find on the deepest node. How deep can we build the tree? The union by rank heuristic only increases the depth of the tree if two trees of the same rank are unioned together, and our first goal is to maximize this height. We will construct a maximal depth tree of our original n singleton sets (note that we've already performed n makeset operations) by successively calling union on trees of the same rank. Another way to see this is that we will first union pairs of singletons, then pairs of those pairs, and so forth. How many times do we call union in total?
We call $\frac{n}{2}$ unions on pairs of trees that have rank 0. Next, we call $\frac{n}{4}$ unions on these resulting trees that have rank 1. Next, we call $\frac{n}{8}$. We proceed in this fashion for $\log n$ levels, giving

us the following sum:

$$\begin{aligned}\sum_{k=0}^{\log n-1} \frac{n}{2^{k+1}} &= n \sum_{k=0}^{\log n-1} \frac{1}{2^{k+1}} \\ &= n\left(\frac{1-\frac{1}{2^{\log n}}}{1-\frac{1}{2}}\right) \\ &= n-1\end{aligned}$$

Ultimately after $n-1$ unions we will have one tree of depth $\log n$. Now call find on the deepest node in this tree n times. How long does this operation take? Because the depth is $\log n$ and we're not using the path compression heuristic, this operation will take $\log n$ time on each call. Ultimately, we have $m = 2n - 1 = \Theta(n)$ operations that takes $\Theta(n \log n) = \Theta(m \log n)$ time to complete.
If we only use path compression without union by rank, we can also give a sequence of n make-set and $m \geq n$ find and union operations that takes time $\Omega(m \log n)$. We are going to have to approach this part differently because successive calls to find will decrease in cost due to the path compression. We will have to leverage the lack of union by rank to maintain the cost of the find operations. Take $\frac{n}{2}$ of our original singleton sets and union them together in successive pairs as was described in the previous part. There really isn't a notion of "rank" anymore because we are not using a union by rank heuristic, but we have built up something called a binomial tree with depth $\log(n) - 1$ (just as we did in the previous part). This kind of tree has some pretty neat properties due to the way it was constructed! Specifically, this type of tree is preserved with path compression (See Figure 2 for a concrete example, and convince yourself this holds true). Take one of the singleton sets we still have left and union our binomial tree to this node (the singleton node is the parent). Now call find on the deepest node in the binomial tree. Because it has depth $\log n$, this operation will take $\log n$ time and will do path compression. After this sequence of a union and a find, we are left with a new binomial tree of the same $\log(n) - 1$ depth simply with one extra child of the root node. Repeat this sequence of operations $\frac{n}{2}$ times (for each of the remaining singletons). Ultimately, we have done $n/2 - 1$ union operations to build the binomial tree along with $\frac{n}{2}$ singleton unions and $\frac{n}{2}$ finds. The cost of the finds alone is $\frac{n}{2}\log(n) - \;= \Theta(m \log n)$. In order to prove the claim that binomial trees are preserved under a singleton union and a find, we can look at the recursive structure of the tree (formally, you can prove this using induction on the height of the tree, but a thorough explanation (or images) of what happens is enough).
By drawing some pictures, it is easy to show that this is true. To elaborate, take a look at the process of union with a singleton and running find on the node surrounded by the red outline.

Problem 4

You know what can get tough when running an automated factory? Checking that each robot is up and running! Suppose each robot has a set schedule on which it runs. For

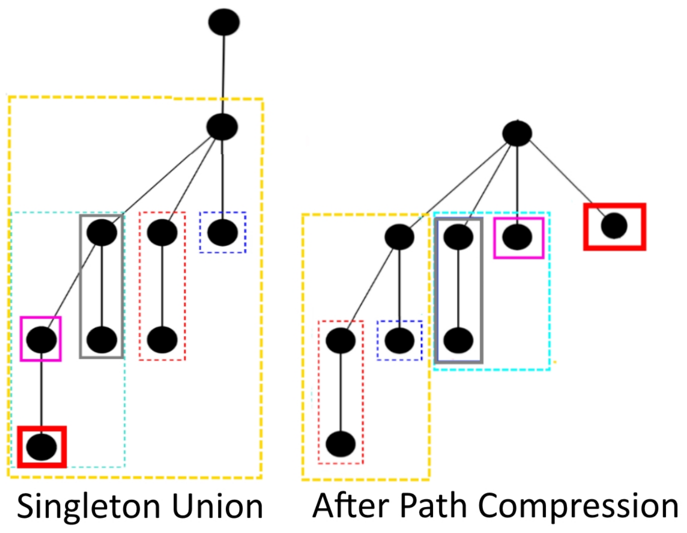


Figure 2: To the left, we have a binomial tree that has been unioned to a singleton, with the singleton becoming the new parent. After performing a find operation on the deepest node (surrounded in red), we have the tree shown on the right. Note that, ignoring the singleton we called find on, the tree is a binomial tree of the same height as our original tree (surrounded in yellow in the left picture).

simplicity, a robot can only run once and must run continuously until some end time. We can then represent such a schedule as n horizontal line segments in the plane. The ith segment has some height h_i (which may be negative and represents the equity on robot i) and runs from the start time, $x = a_i$, to the end time, $x = b_i$ ($a_i < b_i$). For some bizarre reason, the line segments are half-open: they contain their end time, but not the start time. For simplicity, "checking in" at the factory can be represented by a vertical line (note *line* and not *line segment*), and can occur at non-integer times. The only limitation that exists is that each robot (horizontal line segment) is checked on (intersected by a vertical line) at least once.

(a) Help the manager solve this problem! Give a greedy algorithm which minimizes the total number of "check-ins" and prove its correctness. The running time should be $O(n \log n)$.

(b) However, the robots actually form part of a union, and are rather wary of being constantly checked on. Suppose the objective was instead to minimize the maximum number of times that any robot is checked on. That is, each robot should be checked on at least once, and *at most* z times such that z is minimized. Show that if the optimal solution for a given instance achieves a value of z_{opt}, then there is a greedy solution which is optimal for part (a) while achieving $z \leq z_{opt} + 1$ for this new objective.

(c) Well, the robots really *do* insist that you try your best to absolutely achieve z_{opt}.

Assuming that the statement of (b) is true, derive an algorithm which achieves the objective of (b) but with $z = z_{opt}$. Any algorithm running in polynomial time (i.e. $O(n^c)$ for some constant c) will receive full credit because the robots are on the verge of mutiny.

Solution The solutions have figures, and we encourage you to use them to convince yourself of the arguments made.

(a) The running time and the fact that we want a greedy algorithm gives us an idea of what we might do. We're attempting to minimize the total number of check-ins, which intuitively, we can do greedily by attempting to "check-in" on the fewest robots. For concreteness, we'll refer to Figure 3.

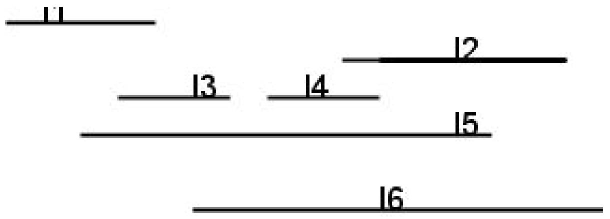


Figure 3: A sample schedule of robots laid out in a 2D space for simplicity. We have Robots labeled I_i for $i \in \{1, 2, \cdots, 6\}$.

How might we do this? As a manager, our first idea would be to check-in as soon as possible, but this isn't really optimal (imagine a robot schedule where every robot starts $\epsilon > 0$ seconds after the previous robot but they all end at the same time). Another idea, which mitigates this, is to check-in on a robot at the latest possible time! More precisely, for any robot we haven't checked-in on yet, we check-in at the very last second of their shift. Algorithmically, we can do this as follows:

First, let each schedule be given by $I_i = (a_i, b_i]$ and sort them by their endpoint such that $b_0 \le \cdots \le b_{i-1} \le b_i \le b_{i+1} \le \cdots \le b_n$.

Then we can process the schedules in this sorted order, and greedily check-in on schedule $I_i = (a_i, b_i]$ at $x_k = b_i$ if and only if I_i has not already been cut. We can easily determine if I_i has been checked on by checking if $x_{k-1} > a_i$ (if the previous check-in occurred after the robot began his shift). For Figure 3, we can see that our described algorithm will select to check in on I_1 and I_4 (and thereby also checking in on all the other robots). We can see the result of this in Figure 4.

To prove correctness, consider the L robots which the above algorithm selects to check on. Then by construction, for the L intervals describing their schedules, we have $a_0 < b_0 \le \cdots \le a_i < b_i \le \cdots \le a_L < b_L$ (in layman's terms, the intervals don't overlap at all/are disjoint). Looking at Figure 4, we can verify this. Then $L \le OPT$ because the optimal solution must check-in on the L robots with non-overlapping schedules. Therefore the algorithm we provide achieves OPT (by definition, OPT is the best

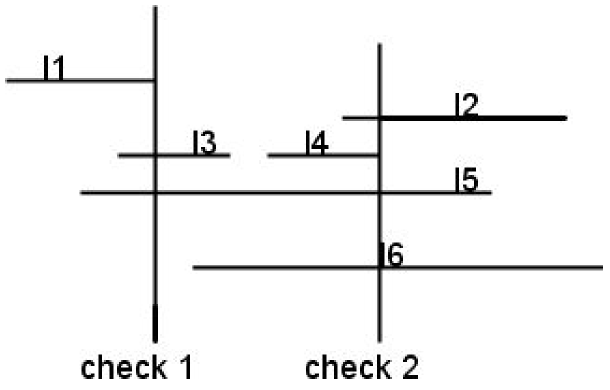


Figure 4: The robot schedule with check-in occurring at the end of the shift for Robot I_1 and I_4. Check-ins are drawn as vertical lines so we can easily tell all the robots that are checked during that check-in.

solution possible, and we have shown above that our solution is less than our equal to the best possible, therefore it is the best possible).

The running time of the algorithm is dominated by the sorting, which is $O(n \log n)$. The processing of each sorted robot schedule can be done in $O(1)$ time. We simply need to check if we have already checked that robot, which can easily be done by seeing if the start of his shift occurs after or before the last time we checked-in. If it occurs after, we don't need to do anything explicit to mark the other robots we check (we'll deal with them when we get to their schedules), we just need to remember the endpoint of this robot's schedule.

(b) The question is a bit long, but let's break it down. We're asked to prove the existence of a greedy solution for part (a) which also has a z, the maximum number of times a robot is checked on, which is no greater than $z_{opt} + 1$. In other words, we want to show that a greedy solution not only solves part (a), but comes really close at minimizing the number of times a specific robot is checked.

Not surprisingly, the greedy solution satisfying the constraints is the same greedy solution as that in (a). To prove this, suppose the algorithm from (a) checks in on some robot working schedule $\ell = (a, b]$ a total of $z > 0$ times (in other words, find the grumpiest robot!). The first of these check-ins must have occurred either at $x = b$ (in this case, $z = 1$ and there are no further check-ins to consider), or at $a < x < b$. For concreteness, in Figure 4, we have $z = 2$ for I_5. Then note that the last $z - 1$ check-ins must have been made to check-in on some robot working schedule $\ell' = (c_i, d_i]$ such that $x \leq c_i < d_i \leq b$ for all i. For Figure 4, the witness schedule is I_4 which is fully contained within schedule I_5. This containment for the last $z - 1$ intervals holds in general. To see this, suppose the offending schedule started before ℓ. Then $c_i < x$, and the robot would have been checked on during the first check-in at x and would not have been processed by greedy (see I_3 in Figure 4). On the other hand, if the schedule

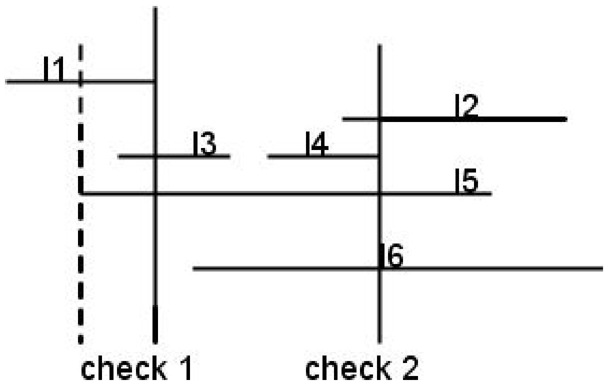


Figure 5: The dashed line shows the improved check-in 1.

starts after ℓ, then $d_i > b$, and the check-in would not intersect ℓ and is therefore not one of the $z-1$ check-ins.

Next, note that the $z-1$ schedules must be non-overlapping. By (a), the greedy algorithm used the minimum number of check-ins required to check on these $z-1$ non-overlapping robots. Therefore, even in the best possible scenario, robot ℓ must be checked in on at least $z-1$ times. We then have $z-1 \leq z_{opt} \implies z \leq z_{opt}+1$ as desired.

(c) As per the suggestion in the question statement, we use the above result. We first run greedy from (a) to get some value z. Then either $OPT = z$ or $OPT = z-1$. We have a solution achieving z, so we want to know if $z-1$ is possible.

We also know that for a robot with schedule ℓ that is checked on z times, an improvement can only occur if we check in earlier on the first robot that lead to the first check-in. More concretely, looking at Figure 5, we can reduce the number of check-ins on I_5 if we check-in on I_1 before it overlaps with I_5.

More generally, we can take a segment $(a, b]$ which was cut z times and attempt to cut it only $z-1$ times. The first cut was to satisfy some interval $(c, d]$. Replace $(c, d]$ in the input with $(c, a]$, which removes the possibility of the first cut, and rerun greedy. If we find a solution to this new problem with cost $z-1$, then the same solution applies to the original problem and has cost $z-1$. If instead we have some interval $(c, r]$ for $r \leq c$, then we know $z = OPT$ because the first cut of the segment was caused by an interval completely contained within it. Otherwise, we continue for each segment with z cuts until we either improve all of them, or find one which cannot be improved on.

The running time of the above algorithms is $O(n^2 \log n)$ because there can be up to $O(n)$ robots checked on z times (imagine $n-2$ schedules all the same and two schedules completely contained with-in them) and each run of greedy takes $O(n \log n)$ time as per (a).

Problem 5

Solve "Problem A - Game" on the programming server; see the "Problem Sets" part of the course web page for the link.

Solution

Let $m = M + L$, the total number of rounds and donations that occur and n be the number of people playing the game. We can implement an $O(mn)$ algorithm by naively simulating the game and updating the scores of all the players as soon as their corporation wins.

However, this isn't optimal. Instead, we can think of each round of the game as the *union* of two companies, and the donations performed by the people as a $find$ operation which looks up the score of that particular person. We can therefore use the disjoint forest data structure with union-by-rank and path compression to simulate the game efficiently, giving a running time of $O(m \log^* n)$ (note that we accepted solutions with running time $\Theta(m \log n)$ which did not use union by rank).

To implement the game, construct an array of length N where $\mathbf{parent}[x]$ denotes the **parent** of item x in its tree in the disjoint forest data structure. The array is initialized to the identity mapping. Let $score(x)$ denote the score of player x at some point during the game. We store an array $\mathbf{diff}[1 \ldots N]$ where $\mathbf{diff}[x] = score(x) - score(\mathbf{parent}[x])$ (unless x is the root of its tree, in which case we store $\mathbf{diff}[x] = score(x)$). The root x of every tree also stores $\mathbf{value}[x]$, which is the size value of the corporation rooted at x. Now, each person starts at his or her own singleton set. When a match between person x and person y occurs, we just union(x, y). This causes some root a to become the child of some other root b. Then $\mathbf{value}[b] \leftarrow \mathbf{value}[a] + \mathbf{value}[b]$ (update value of the corporation rooted at b)and **diff** $[b] \leftarrow score(a) + score(b)$ (update scores), and also $\mathbf{diff}[a] \leftarrow \mathbf{diff}$[a] -$\mathbf{diff}[b]$ (maintaining invariant). Then to calculate the score for person x, we do a find on x and add up all **diff** values between all nodes from x to its root, which telescopes to equal $score(x)$, i.e.

$$\mathbf{diff}[x] + \mathbf{diff}[\mathbf{parent}[x]] + \mathbf{diff}[\mathbf{parent}[\mathbf{parent}[x]]] + \ldots + \mathbf{diff}[root[(x)]] =$$
$$(score(x) - score(\mathbf{parent}[x]]) + (score(\mathbf{parent}[x]] - score(\mathbf{parent}[\mathbf{parent}[x]]]) + \ldots + score(root[x]) =$$
$$score(x)$$

For all the nodes compressed along the path, we also change their **diff** values since their parent is now changed to the root.