Homework 3

Simon Zheng
260744353

November 3rd, 2017

1

First, include all leaves as no 2 leaves can be adjacent in a graph with more than 2 vertices (cases with a graph containing less than 3 vertices are trivial). Then, with the remaining vertices, 2-colour them and keep the biggest set.

For $n > 2$, every leaf will only have an internal vertex by the property of being in a tree. Between two adjacent vertices where one is a leaf (single neighbour) and one is an internal vertex (more than 1 neighbour), choosing the leaf allows us to also choose any of the internal vertex's neighbours, while choosing the internal vertex removes the possibility of choosing the leaf or any of it's neighbours. Thus, along with the fact that no two leaves can be adjacent for $n > 2$, choosing the leaf is more advantageous and will allow for a bigger set.

2

For any order of processing chosen, pick 2 successive jobs. The finishing time of each is the processing time of the previous job plus its own. Thus if the processing time of the first one is t_1, both finishing times must be at least as long as t_1. So if $t_1 > t_2$, then both finishing times must be as long as the longest processing time. But if we reverse the job order (so where $t_1 > t_2$ still applies), then both jobs' finishing times must only be as long as the shorter one (t_2) and t_2's finishing time will be shorter than the first case. Therefore, it is always more optimal to put the shorter job first. Now, applies this to every successive pair of jobs from (t_1, t_2), (t_2, t_3), ... , (t_{n-1}, t_n) and we will get the optimal solution, which is exactly putting them in increasing order according to their processing time.

3

Using a proof by induction, we will show that the algorithm works with 2 processors with the conditions in the question and thus show it works for $n + 1$ processors (our base case is one processor which is proven in the previous question).

Beginning with both processors free and a set of jobs, pick two jobs and give them to the processors. For this basic case it is obvious that picking the shortest job is optimal as the average finishing time is $\frac{t_1+t_2}{2}$. It is a similar reasoning to the previous question.

Now, for the next job, if the two previous jobs had the same length then we are back to the previous case. Otherwise we have a processor that finished first, and therefore was available sooner, and the other one that finished after, and therefore had a longer job.

If we give the next job (that takes time t_i) to the processor that finished sooner (with, say, finishing time f_1), then the total finishing time is now $f_1 + t_i$. If we give it to the processor that finished later at time f_2 then it is $f_2 + t_i$. Thus, we get a more optimal average finishing time if we give it to the processor that finished sooner, since $t_1 < t_2$ such that $\frac{f_1+(t_i+f_1)+f_2}{2} < \frac{f_1+f_2+(t_i+f_2)}{2}$.

This next job must also be picked according to its processing time, choosing the shortest, for reasons explained earlier (and in the previous question).

By doing this, we know that at every step where we choose the next job, the jobs picked and processed previously were done in the most optimal way. And if we pick the most efficient job and give it to the first processor available again, then at the next case we will also have the most optimal way by induction.

If we look at everything that was done, it is exactly how the greedy algorithm does it: picking the first processor available, or rather immediately feeding a processor a new job when it becomes available, and picking the job that is the shortest.

4

No, it is not.

Counter-example: construct a tree equivalent to this:

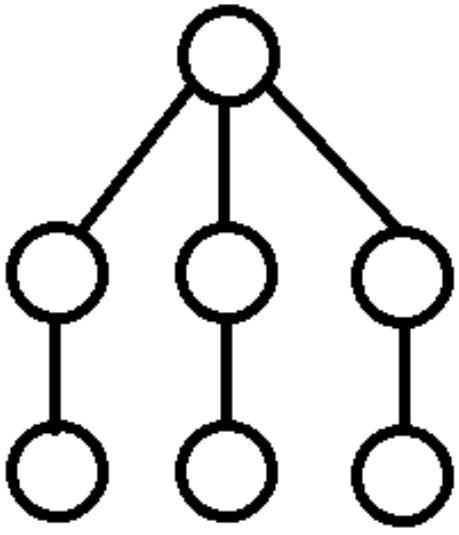

Figure 1: Counter-example tree

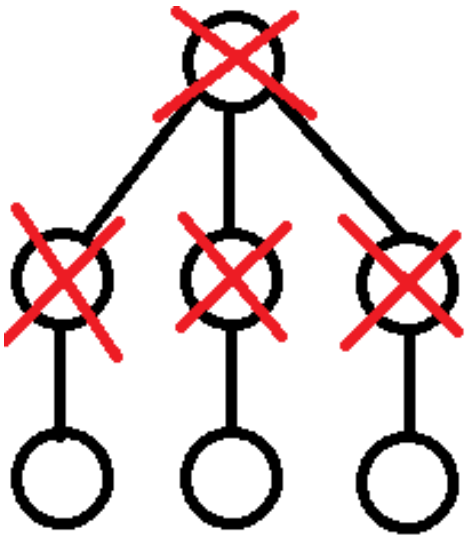

Figure 2: Greedy algorithm result

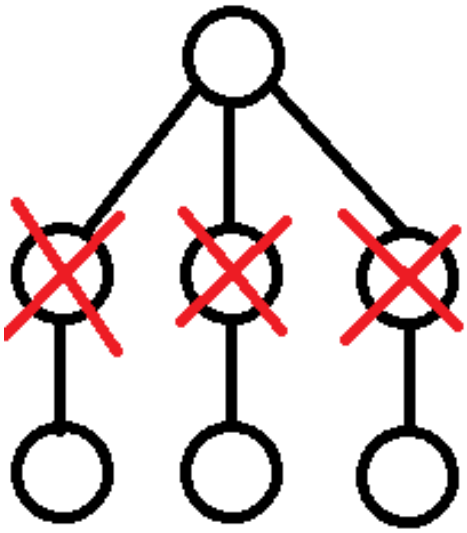

Figure 3: Optimal result

The greedy algorithm will start with the vertex at the top, but if we look at the optimal way, it is useless a adds an extra vertex to delete.

5

Graph, edge weight between two points (vertices) is distance, put edges in graph matrix and sort.

6

We use the reverse-deletion algorithm to find a spanning tree, except that we force it to keep edges in F.

Begin with putting all of the edges in a set. Remove one by one the most costly edge left every time but without removing edges in F.

We know reverse-deletion gives us the minimum spanning tree. Since in this case, all we are doing is keeping *more* edges (that do not form cycles), then either we end up with what the normal algorithm gives plus extra edges in F, or we delete more (the next most costly edges) than what it would normally give, to avoid cycles created by edges in F that we wouldn't have included in the normal algorithm

Thus we still get the minimum spanning tree, but at most as optimal as the normal algorithm's result but only when we are forced to.