INTRO TO APX. – SOME SOLUTIONS FOR HW1

A credit for a part of them goes to Martin Böhm

Note: some parts of the solutions are quite detailed which is usually not needed to get all points, but appreciated (however, all important observations must be stated).

HOMEWORK ONE — Steiner tree — [5 points]

In the *Steiner tree problem* we get on input a connected undirected graph $G = (V, E)$, an edge cost function $c : E \to \mathbb{R}^+$, and finally a list of *terminals* $S \subseteq V$. A feasible solution to our problem is any subset of edges $E' \subseteq E$ so that the graph $G' = (V, E')$ has all the terminals in one connected component. We aim to minimize the cost of E', i.e. $\sum_{e \in E'} c(e)$. Your task is to design a 2-approximation algorithm.

Hint: Start with the case in which edges satisfy triangle inequality (that is, the shortest path between vertices connected by an edge e is this edge e).

Solution. We first observe that an optimum Steiner tree is always a tree, as there is no reason to consider cycles or other graphs other than trees.

After learning this, our first idea should be some sort of spanning tree – either on all vertices or just on the terminal vertices. Clearly, the former idea is a bad one, as there can be a very distant vertex that is not part of the terminals and any optimum avoids it.

We now claim that the minimum spanning tree on the subgraph induced by terminal vertices is a 2-approximation in the metric case — in particular, we assume that the graph is complete and that the edges satisfy triangle inequality.

Okay, suppose that we have some optimal tree OPT that also includes some other vertices. We wish to bound our cost by $2c(OPT)$, but we will do it in an indirect way. Consider a DFS traversal of the tree OPT. From the DFS traversal, we create a list L of terminals by adding every terminal when we first encounter it.

Now, we think of L as a path P_L on the terminals – if the list is $[t_1, t_2, t_3]$, we go from t_1 to t_2 and from t_2 to t_3. We claim that $c(P_L) \leq 2c(OPT)$. This follows from the fact that a DFS traversal visits each edge of the tree OPT at most twice, and our shortcuts $t_1 - t_2$ and $t_2 - t_3$ are shorter than the paths in OPT by triangle inequality.

To finish the metric case, we just note that P_L is a valid spanning tree on the terminals, and so the minimum spanning tree has to be even shorter.

The general case works basically the same way, we just need to work with the *shortest path metric*. After finding a solution T of the metric case we then replace each edge of the tree T by the shortest path between endpoints of this edge (which gave length for the metric).

HOMEWORK TWO — Scheduling on machines with speeds — [6 points]

We have m machines with speeds $s_1 \geq s_2 \geq \cdots \geq s_m$ on which we want to schedule n jobs (so that one machine processes at most one job at a time). Now, processing the j-th job on machine i takes p_j/s_i units of time, where p_j is the length of the job. (The schedule starts at time 0 and two jobs cannot of course run at the same time on one machine.)

We say that a polynomial-time algorithm is a ρ-relaxed decision procedure if it gets a number D in addition to the intput and either (I) creates a schedule so that all jobs are finished till time ρD, or (II) outputs that there is no schedule that can finish all jobs till time D.

a) First show that one can use a ρ-relaxed decision procedure to find a ρ-approximation algorithm.
b) Then prove that the following algorithm is a 2-relaxed decision procedure: First for each job j we find its type i which is the number of the slowest machine which finishes job j till time D (that

is, maximal i such that $p_j/s_i \leq D$). If for some job j there is no machine which finishes it till time D (and thus $p_j/s_1 > D$), the algorithm immediately stops outputting (II).
After assigning types to jobs, each machine i starts processing jobs of type i so that after finishing a job j machine i continues as follows:

- If job j finishes at time D or later, the machine stops and processes nothing from now on.
- If there is no available job of the same type as job j (that is, not finished and not started) the machine starts processing a job of the next type (for which there is an available job). Machine i thus first processes jobs of type i, then of type $i+1, i+2, \ldots$
- Otherwise the machine starts a job of the same type as job j.

If not all n jobs are processed by this procedure, the algorithm stops outputting (II), otherwise it outputs the created schedule containing all jobs.

Solution.

a) The right idea was to apply binary search. However, there was an unintended catch: what if the processing times or speeds are rational / real numbers? We first deal with the case in which p_j's and s_i's are integers (which should have been written in the homework) and then show how to extend it to rational numbers. In the general case of real numbers (if we somehow got them) it is not clear when to stop the binary search.
Let $\mathcal{R}$ be the ρ-relaxed decision procedure. We apply binary search for the smallest D such that $\mathcal{R}$ returns a schedule. In the binary search we need to start with an interval containing the correct D. A correct upper bound may be simply $\sum p_j/s_1$ (sum of processing times divided by the speed of the fastest machine) and for a lower bound one can simply take 0 (of course, better bounds may be devised simply). In each iteration if $\mathcal{R}$ returns a schedule we modify the upper bound to the current D and if $\mathcal{R}$ returns (II) we set the lower bound to D. The new D is in the middle of the interval and we round it to the closest multiple of $1/s_i$ for some i (any schedule has length that is multiple of $1/s_i$ for some i as p_j's are integers).
Why the number of iterations is polynomial and when to actually stop it? We now use that all numbers are integers. We stop the binary search when $U - L < 1/(s_1 \cdot s_2)$ where U and L are the current upper and lower bounds — the reason is that the optimum length of the schedule is a multiple of $1/s_i$, since all processing times are integers and the difference of the length of two schedules is thus at most $1/(s_i \cdot s_j)$; the smallest possible difference is then $1/(s_1 \cdot s_2)$. Thus the number of iterations is at most

$$\log\left(\frac{\sum p_j}{s_1} \cdot s_1 \cdot s_2\right) \leq \mathcal{O}\left(\sum \log p_j + \log s_2\right)$$

which is less than the size of the input in binary.
Clearly, such a binary search will find D such that for any $D' < D$ which is a multiple of $1/s_i$ for some i there exists no schedule of length D'. Thus OPT $\geq D$ and our algorithm returns a schedule of length at most 2OPT,
If p_j's or s_i's are not integers, but we still get their binary encoding on input (so they are rational), then we multiply them by the maximal number of decimal digits and the number of iterations will be rational. This causes no problems in the analysis as multiplying all p_j's and/or all speeds does not change the behavior of the algorithm or the optimal schedule except scaling its length. (Multiplying rational numbers by the lowest common denominator works similarly if we got rational numbers on input as binary encodings of the numerator and of the denominator.)

b) If there is some job j, which even the fastest machine cannot complete in time D (i.e., $p_j/s_1 > D$), then outputting (II) is clearly correct.
Otherwise, first suppose that the algorithm returns a schedule. We just need to show that its length is at most $2D$. This is easy as any job is started before time D and each machine is processing only jobs of its type or of higher types, but jobs of higher types can be processed in time D even on a slower machine. Thus the machine processes any job for at most D time units

and our schedule is of length at most $2D$.

Finally, the main part of the solution was to show that when there is an unprocessed job j, there exists no schedule of length D and hence returning (II) is correct. Let i be the type of job j. The key observation is that all i fastest machines $1, \ldots, i$ run only jobs of types $\leq i$ — indeed, if any such machine is about to schedule a job of type $> i$, then it would prefer scheduling j instead. It remains to observe that all jobs of types $\leq i$ (including j) cannot be processed in any schedule of length D. Any machine $\leq i$ cannot be idle before time D (otherwise it would schedule j) and thus the total processing time of jobs of types $\leq i$ is at least $\sum_{k=1}^{i} s_k D + p_j$. However, any machine $> i$ cannot process job j or any job of type $\leq i$ and thus a schedule of length D can only process $\sum_{k=1}^{i} s_k D$ of volume of jobs of type $\leq i$. This concludes the proof.

HOMEWORK THREE — **Looking for TSP instances** — **[6 points]**

a) Find an infinite class of graphs showing that the Christofides' algorithm for metric TSP is no better than a 3/2-approximation.

More precisely, for infinitely many n's construct a graph G_n with n vertices so that

$$\frac{\mathrm{ALG}(G_n)}{\mathrm{OPT}(G_n)} \to \frac{3}{2}$$

for $n \to \infty$ where $\mathrm{ALG}(G_n)$ is the cost of the algorithm's solution and $\mathrm{OPT}(G_n)$ is the optimum cost. Similarly as in the class, you can choose the behavior of the algorithm if there are more possibilities (for example, if the graph has more minimum spanning trees).

b) In *asymmetric TSP* our task is to find a minimum-length walk that visits each vertex in a directed graph with edge lengths. While triangle inequality still holds, symmetry $d(u,v) = d(v,u)$ may not hold. Show that a spanning tree tree algorithm cannot be used for any good approximation (constant-, or even $\mathcal{O}(\log n)$-approximation). In this algorithm minimum spanning tree is found in the graph in which we forget edge orientations.

Hint: Find an example of a directed graph with k really long edges (and the rest of edges with length 1). An optimum solution uses only one long edge, while a DFS traversal of a minimum spanning tree (and taking shortcuts) uses all k long edges. How long can these long edges be? (Of course, you can solve this exercise using a different approach.)

Solution. Our main gadget will be *a wheel* – n vertices $c_1, c_2, \ldots c_n$ connected in a cycle C and an extra vertex s (the center) with additional edges (called *spokes*) $sc_1, sc_2, \ldots, sc_n$.

In our example for Christofides, we assume n is even, set the edges of C to be of length 1 and the spokes to be of length $1 - \varepsilon$ for some $0 < \varepsilon < 1/n$. We choose the minimum spanning tree which takes all the spokes — of length $n - \varepsilon n$.

One of the minimum-cost matchings for the odd-degree vertices of our spanning tree is the following: we connect every odd-numbered vertex c_{2i-1} with the even-numbered vertex c_{2i}.

Except for the shortcutting, our TSP tour is now of length $n + n/2 - \varepsilon n$, which is okay as the optimal TSP tour is of length $n + 1 - 2\varepsilon$ (we could just traverse the cycle C of the wheel and visit s last). Thus the ratio tends to 3/2 as $n \to \infty$.

If we had a good order on the vertices (good for the algorithm), we could visit s, c_1, c_2 and follow this by c_3, which we could shortcut and decrease the cost. Therefore, we need to argue that our traversal visits some other vertex c_j afterwards. But we can choose the order of the vertices (the algorithm does not specify it), so we just choose a bad one.

An example for asymmetric TSP can be similar: we orient edges of the cycle in one direction (so edge $c_i c_{i+1}$ leads from c_i to c_{i+1}). Spokes will be oriented in both direction with very different lengths — there will be an edge sc_i of length $1 - \epsilon$ and a long edge $c_i s$ of length L for some big L. Note that L

can be arbitrarily large. Now an optimal solution is roughly $n + L$, but the algorithm may be forced to use all n long edges, giving ratio n (the spanning tree is the same as in the previous example). The rest of the solution in this case is not hard and we leave it as an easy exercise.