INTRO TO APPROXIMATION – HW1

TSP and friends

Every task is worth two points. Deadline: **16. 11. 2016 23:59** via email. Please send PDFs only. High-quality scans (with high-quality handwriting) are acceptable. Do not be afraid to email me if any task is unclear to you.

EXERCISE ONE 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, i.e. $\sum_{e \in E'} c(e)$. Your task is to design a 2-approximation algorithm.

Hint: The graph does not need to satisfy the triangle inequality. First, think about the case when it does (it should be easy then). To solve the general case, try to use some of the techniques from the TSP approximation.

Solution. A natural idea is the following: take the graph G, restrict it only to the vertices S, and find a minimum spanning tree. This would be very silly for the general metric (because there, the prices among S can be high while detours through the rest of G can be cheap) but with triangle inequality, there is hope that this should be a good approximation.

Having found the $MST(S)$, we need to bound it by OPT somehow. First of all, we note that OPT is also a tree (as the title hints), but it contains more vertices on G. We will transform it into a tree T' on $V(S)$ and then claim that $MST(S)$ is shorter than T'.

To do this, we take the DFS-traversal of OPT. In it, every vertex is visited once (but internal vertices are visited twice). Clearly, the length of such a DFS-traversal is at most $2OPT$. Then, we look at where the vertices of S are in this traversal and use shortcutting to get a path on $V(S)$ – from a vertex s_{i-1} we go directly to s_i, instead of through the DFS traversal. Since we only used shortcutting, the new path is of length at most $2OPT$, and, by minimality, $MST(S) \leq 2OPT$ as well.

As almost all of you guessed correctly, to solve the general problem withou triangle inequality, we can move to the *shortest path metric* of the input graph G. When we expand the solution from the metric case to the non-metric case, we can save some edge cost by making sure the final graph G is a tree – but similarly to the “shortcutting” in TSP, doing this will not affect the approximation ratio.

EXERCISE TWO Consider the problem MAXSAT – given a CNF Boolean formula with clauses of any size, we need to find an assignment that satisfies as many clauses as possible (even if the full formula is unsatisfiable).

We will analyze the following algorithm:

“Try setting all variables to 0 and compute how many clauses we have satisfied. Then, try setting all variables to 1 and compute how many clauses we have satisfied. Take the bigger of the two values and return it as the approximation.”

1. Thoroughly analyze the approximation ratio of the algorithm; that is, prove that it is an r-approximation algorithm and also prove that it is not an r'-approximation for any $r' < r$.
2. Let us consider any *constant-testing* algorithm – such an algorithm does the same thing as the one described above, but instead of two assignments, it tests c pre-selected assignments, where c is a constant not dependent on the input. (An assignment is any infinite sequence of 0/1 values, where we assign the first value to x_1, the second value to x_2 and so on, until we run out of variables.)

   What is the tight approximation ratio of any constant-testing algorithm? Again, you need to find a number r_2 such that some constant-testing algorithm is an r_2-approximation and prove that *no* constant-testing algorithm is strictly better than an r_2-approximation.

Solution. The first subitem was not a problem for most of you; testing all zeroes or all ones leads to a 1/2-approximation algorithm, and a very easy tight example is a formula $(x_1) \wedge (\neg x_2)$.

The second part was slightly more difficult and had a tricky bit inside that was unintentional.

First the intended solution: if we take c assignments, we can look at the first $2^c + 1$ values. We will define *a column* as the vector of c assignments for some specific variable x_i that the algorithm will check. A column is a 0/1 vector of length c.

The first 2^c columns could in theory be pairwise different, but with at least $2^c + 1$ vectors, we know that at least two columns are the same. In fact, we can strengthen it and say that if we take all (infinitely many) columns, at least one column is repeating infinitely often.

Since one column repeats infinitely often, we know that the algorithm, even though it has c assignments, will behave on the column variables (say) $x_{10000}, x_{20000}, \ldots$ as if it were the 0/1 tester from Point 1!

We therefore first create a huge clause of all variables $(x_1 \vee x_2 \vee \ldots \vee x_B)$ and then add positive and negative mono-clauses as before, i.e. $(x_{10000}) \wedge (\neg x_{20000}) \wedge \ldots$.

From this we learn that any c-assignment-testing algorithm is a 1/2-approximation in the limit.

Now, let me elaborate about the unintended trickiness. Some of you claimed the following:

"Taking $2^c + 1$ columns, we find two variables x_{10000} and x_{20000} where the algorithm test two zeroes or two ones much like in Point 1. Therefore, $(x_{10000}) \wedge (\neg x_{20000})$ is an input where the algorithm is a 1/2-approximation."

This seems correct (and it definitely uses the correct approach, as we did) but it claims the algorithm is always a 1/2-approximation, and not only in the limit.

Unfortunately, $(x_{10000}) \wedge (\neg x_{20000})$ is not a good input format for the algorithm. You can see why by noting it has only two variables, but it *forces* the algorithm to use some very distant assignment for each of those, ignoring 19998 assignments that the algorithm has in its store.

In fact, if we accepted this form of input as a valid one for formulas, we can even claim the algorithm does not perform in polynomial time! Indeed, if the input is just the formula $(x_{2^{54}})$, we only need about 2×54 bits for the input, but the algorithm really needs to go through 2^{54} positions in the assignments to even find what value $x_{2^{54}}$ should get.

To patch this problem, we could claim the actual input looks like this:

$$\{x_1, x_2, x_3, \ldots, x_{20000}\}, (x_{10000}) \wedge (\neg x_{20000}).$$

This is very similar to how we receive graphs on input: first the list of vertices, then the structure (list of edges).

However, doing it like this requires a notion of "variable of degree zero" – a variable that is present in the formula, but its number of occurences is zero. Plus we would need that $(x_{10000}) \wedge (\neg x_{20000})$ is a formula with 20000 variables, which is a somewhat untraditional view. The most unfortunate thing about this input format is that it allows formulas of two variables which are impossible in other input formats.

We should contrast it with the "blackboard" input format for formulas, which is the one we often use when we draw them on the board: a formula is just given as

$$(A \vee B) \wedge (C \vee \neg D)$$

and we compute the number of variables from the input itself (in this case, it is four). In this format, it is impossible to create a formula of length 2 where the algorithm would be a 1/2-approximation. Many of you have noted that – that is why you claimed the 1/2-approximation is only in the limit.

To conclude, we should not say the blackboard format is strictly better than the graph-like input format. Sometimes the graph-like format may be much more useful, as it doesn't require the algorithm to compute the number of variables (and it forces one specific ordering). However, we have learned that even such a simple thing as input format can have fairly non-trivial impact on our claims – it can change a 1/2-approximation algorithm into a 1/2-approximation only in the limit.

EXERCISE THREE Given a directed graph $\vec{G}$ with distance function $d\colon \vec{E} \to \mathbb{R}^+$, design and analyze a polynomial-time algorithm for finding the directed circuit which is shortest *on average* – it is a circuit minimizing $\frac{\sum_{\vec{e}\in\vec{C}} d(\vec{e})}{|\vec{C}|}$.

Keep in mind that the shortest averaged circuit can often be longer and have more edges than the shortest circuit overall.

Solution. We use a dynamic program for computing the values $d^k(x,y)$, which will be equal to *the length of the shortest k-vertex walk between x and y.* Note that we allow vertices and edges to be repeated in this walk.

To compute $d^1(x,y)$ is easy – we just set it to be the distance $d(x,y)$. To compute it for larger k, we can use the recursion $d^k(x,y) = \min_z d^{k-1}(x,z) + d^1(z,y)$. It looks deceptively simple, but it actually can be computed this way. (Think for a moment why we cannot solve the Hamiltonian circuit problem in arbitrary directed graphs using this recursion.)

Alright, so what if we find some minimum $d^k(x,x)/k$ and the walk minimizing $d^k(x,x)$ is not actually a circuit? Here comes the trick: *if a walk attains the minimum average value but it is not a circuit, then it contains a shorter circuit that attains at most the same average value.*

This follows from the fact that if the closed walk W can be split into two closed walks A and B (which it always can if it is not a circuit itself), then $E[W] = E[A] + E[B]$ by linearity of expectation.

Therefore, our polynomial-time algorithm computes the table $d^k(x,x)$ and then finds the minimum value in it, going from smaller k to larger. The first occurence of the minimum value is exactly the value of the smallest circuit. To find the circuit itself is done by the standard dynamic programming argument.

EXERCISE FOUR Assume there is a polynomial-time algorithm for finding the shortest averaged circuit (see above).

Consider the following algorithm for asymmetric TSP on n elements with the asymmetric function $d\colon \{1,\dots,n\} \times \{1,\dots,n\} \to \mathbb{R}^+$ which satisfies the triangle inequality:

1. We find a directed circuit $\vec{C}$ in the metric which minimizes $\frac{\sum_{\vec{e}\in\vec{C}} d(\vec{e})}{|\vec{C}|}$.
2. We add all the edges $\vec{E}(\vec{C})$ to the solution.
3. We remove all vertices of $\vec{C}$ except one. We continue recursively until only one vertex remains.
4. We use shortcutting on the Eulerian walk and return a directed Hamiltonian circuit.

Prove that the previous algorithm is an $\mathcal{O}(\log n)$-approximation for asymmetric TSP.

Solution.

First of all, because of our problem setting, we know that OPT (the optimum solution of the asymmetric TSP) is a Hamiltonian circuit. Using the triangle inequality (which is necessary!), we can show that if we remove any vertices from the original graph G and form a graph G' with a new optimum Ham. circuit OPT', it holds that $d(OPT') \le d(OPT)$.

Interestingly, we cannot use a stronger inequality here – for example, it does **not** hold that $\frac{d(OPT')}{n'} \le \frac{d(OPT)}{n}$ in a general step; while the optimum circuit must get shorter, the optimum average length can both increase and decrease.

Now, to the algorithm. In the first step, we know that $d(\vec{C_1})/|\vec{C_1}| \le d(OPT)/n$ from the definition of $\vec{C_1}$. After deleting the vertices of $\vec{C_1}$ except one, we want to bound $d(\vec{C_2})/n_2$ by some function of OPT, where $n_2 = n - |\vec{C_1}| + 1$. Again, because of our setting, we can use the fact that the remaining (not-yet deleted) edges in $OPT \setminus V(\vec{C_1})$ can be completed into a Hamiltonian circuit OPT' on the smaller graph $\vec{G} \setminus V(\vec{C_1})$ by shortcutting. We get that $d(OPT') \le d(OPT)$ from the triangle inequality.

As OPT' is a valid circuit, the minimal choice of $\vec{C_2}$ gives us that $d(\vec{C_2})/|\vec{C_2}| \le d(OPT')/n_2 \le d(OPT)/n_2$. The same will hold for $\vec{C_3}, \vec{C_4}$ and so on, with $n = n_1 > n_2 > n_3 > n_4 > n_k > 1$ being the sequence of the remaining vertices in the graph $\vec{G}$, as we remove one circuit after another.

We now proceed to bound $d(T)$, the total distance of the ATSP tour T of our algorithm. The first circuit was of $d(\vec{C_1})$, which we can express as $d(\vec{C_1}) = |\vec{C_1}| \cdot \frac{d(\vec{C_1})}{|\vec{C_1}|} \le |\vec{C_1}| \cdot \frac{d(OPT)}{n}$. Similarly, $d(\vec{C_i}) \le |\vec{C_i}| \cdot \frac{d(OPT)}{n_i}$ for $i \ge 2$. Putting the bounds together, we see our total cost is

$$d(T) \le \sum_{i=1}^{k} |\vec{C_i}| \cdot \frac{d(OPT)}{n_i} = d(OPT) \cdot \left(\sum_{i=1}^{k} \frac{|\vec{C_i}|}{n_i} \right).$$

We will be done when we show that the sum on the right is upper bounded by $\mathcal{O}(\log n)$. This is an easy exercise in combinatorics, but we include it here for completeness.

We first split the sum into two parts:

$$\left(\sum_{i=1}^{k} \frac{|\vec{C_i}|}{n_i} \right) = \left(\sum_{i=1}^{k} \frac{1}{n_i} \right) + \left(\sum_{i=1}^{k} \frac{|\vec{C_i}| - 1}{n_i} \right)$$

The first part is clearly bounded from above by $\sum_{i=0}^{n-1} \frac{1}{n-i} \le \log n$.

The second part seems to be more difficult to bound, because it contains $|\vec{C_i}| - 1$ in the numerator. Suppose now for a second that $|\vec{C_1}| = 4$ and $|\vec{C_2}| = 3$. This means that $n_2 = n - 3$. In this case, the initial part of the sum can be written as:

$$\frac{3}{n} + \frac{2}{n-3} + \ldots = \frac{1}{n} + \frac{1}{n} + \frac{1}{n} + \frac{1}{n-3} + \frac{1}{n-3} + \ldots$$

The right-hand side of the last equation is bounded from above by $\sum_{i=0}^{n-1} \frac{1}{n-i}$, which is again at most $\log n$, and we have our total bound of $\mathcal{O}(\log n)$.

Bonus exercise

We know from the lecture that the Christofides algorithm satisfies ALG $\le \frac{3}{2}$OPT, where ALG is the value of the solution for the algorithm and OPT is the value of the minimum/optimum solution.

Let us refresh linear programming by proving that for Christofides algorithm, it is also true that ALG $\le \frac{3}{2}\mathrm{OPT}_{\mathrm{LP}}$, where $\mathrm{OPT}_{\mathrm{LP}}$ is the optimum value of the following linear relaxation:

$$\begin{aligned}
(P):& \qquad \min \sum_{e \in E} c_e x_e \\
\forall v \in V:& \qquad \sum_{e=vx} x_e = 2 \\
\forall S \subsetneq V, S \neq \emptyset:& \qquad \sum_{e \in E(S, V \setminus S)} x_e \geq 2 \\
\forall e \in E:& \qquad 0 \leq x_e \leq 1
\end{aligned}$$

The battle plan is as follows:

1. First verify that ALG $\leq \frac{3}{2}\mathrm{OPT}_{\mathrm{LP}}$ implies the original claim of ALG $\leq \frac{3}{2}$OPT.
2. Next, prove that for an optimum solution x^* of the LP (P) (that is precisely the point of value $\mathrm{OPT}_{\mathrm{LP}}$) it holds that $\frac{n-1}{n}x^*$ is a point inside the spanning tree polytope for the same graph.
3. Finally, use point 2 and finish the claim that ALG $\leq \frac{3}{2}\mathrm{OPT}_{\mathrm{LP}}$.

If you do not remember, the *spanning tree polytope* is the polytope given by these linear constraints:

$$\begin{aligned}
& \qquad \sum_{e \in E} x_e = n - 1 \\
\forall S \subsetneq V, S \neq \emptyset:& \qquad \sum_{e \in E(S, V \setminus S)} x_e \geq 1 \\
\forall e \in E:& \qquad x_e \geq 0
\end{aligned}$$

The *matching polytope* looks like this:

$$\begin{aligned}
\forall v \in V:& \qquad \sum_{e=vx} x_e \leq 1 \\
\forall S \subsetneq V, S \neq \emptyset, |S| \text{odd}:& \qquad \sum_{e \in E(S, V \setminus S)} x_e \geq 1 \\
\forall e \in E:& \qquad x_e \geq 0
\end{aligned}$$

Solution.

1. The first point is just the classic inequality $OPT_{LP} \leq OPT_{ILP}$; in other words, solution of a linear program is always better than the integer program.
2. All we need to do is paste $\frac{n-1}{n}x^*$ (the optimum fractional solution of the TSP polytope) into the inequalities of the spanning tree polytope and make sure that it satisfies them.
   The equality $\sum_{e \in E} x_e = n - 1$ is true because the original $\sum_{e \in E} x_e^* = n$ – every feasible point in the polytope must have sum around the vertices equal to 2, from which we get that the sum of all edges must be n (same as in a Hamiltonian cycle).
   The constraint $\forall S \subsetneq V, S \neq \emptyset\colon \sum_{e \in E(S, V \setminus S)} x_e \geq 1$ is true because we are changing x_e by $\frac{n-1}{n}$ – for $n = 2$ we are multiplying by $1/2$, for $n = 3$ by $2/3$, and then by even higher numbers. In all cases, if the sum was originally, $\forall S \subsetneq V, S \neq \emptyset\colon \sum_{e \in E(S, V \setminus S)} x_e^* \geq 2$, by multiplying all numbers on the left by $1/2$ or more we still get that $\forall S \subsetneq V, S \neq \emptyset\colon \sum_{e \in E(S, V \setminus S)} x_e \geq 1$.
3. Finally, we use a similar argument to claim that $0.5x^*$ is a valid point in the matching polytope. Christofides' algorithm finds the minimum spanning tree and then a perfect matching of minimum cost (on a subgraph), so in total the algorithm finds a tour of length

$$\mathrm{MST} + \mathrm{MATCH} \leq \frac{n-1}{n} \cdot x^* + \frac{1}{2}x^* \leq \frac{3}{2}OPT_{LP}.$$