Week 3 Questions

In this document, I try to give some answers to the questions that were asked in the IVLE survey. Feel free to come chat during office hours if you want to talk more! The questions may be slightly modified for clarity.

Question 1 *For the General / Metric / Euclidean Steiner tree problems, do we assume that the set of Steiner points S is finite? If S is finite, can we easily reduce the Euclidean Steiner tree problem to the Metric Steiner tree problem trivially by stating that euclidean distance is a metric? If S is infinite, can we simply define S by specifying a space in which S resides (e.g., a Euclidean space, or some other geometric space) and using that to implicitly define S? (For example, you could define the space of all strings, and let the distance metric be the edit distance.)*

Typically, we think of S as being finite (or at least countable) in the case of the General and Metric Steiner tree problems. Therefore it is not immediate to reduce the Euclidean Steiner Tree problem to the Metric Steiner Tree problem (even though Euclidean space has a valid distance metric).

Recall, however, that we can prove some facts about the optimal Steiner points in the Euclidean Steiner Tree problem, i.e., they have degree 3 and have 120 degree angles. Hence if we look at the $\Theta(n^3)$ possible triplets of required nodes, this yields a set of possible Steiner points. But it does not generate *all* the possible sets of Steiner points: for example, a triplet of the newly created Steiner points may also generate a new Steiner point! However, we also know that there are at most $n-2$ Steiner points in total, and hence we can generate an exponential sized set of possible Steiner points. This is quite an inefficient way to solve the Euclidean Steiner tree problem!

We could certainly define variants of the Euclidean Steiner tree problem on other geometric spaces. For example, it is natural to consider d-dimensional Euclidean spaces—and any space isomorphic to a d-dimensional Euclidean space. It seems perfectly plausible to generalize to other spaces the support a distance metric as well. If you think of any that are interesting and yield nice results, let me know!

Question 2 *Is there a way to use linear programming to achieve a good approximation ratio for the maximum independent set problem?*

The maximum independent set problem is NP-hard to approximate within any polynomial factor, i.e., it is one of the hardest problems we know of to approximate. As long as $P \neq NP$, we will not be able to come up with a good approximation ratio for the maximum independent set problem (using linear programming or otherwise).

Question 3 *How do you do time complexity analysis for parameterized complexity? For $T(k, m)$ in lecture notes, if you take into consideration the fact that the number of edges is reduced in recursive calls, can you still solve it (find a tighter bound)?*

One way is to set up a recurrence where k is a parameter, and hence as you recurse the k decreases. You can look at the version in the lecture notes, and also the version on the problem set (for the KirkPatrick-Seidel algorithm). However, there are many different techniques that depend on the precise problem.

More generally, you ask whether we can get a tighter analysis for the vertex cover problem. In this case, the 2^k term really dominates. The analysis seems a bit loose, and you can probably improve the analysis to approximately $O(2^k+m)$ with some care. Essentially, the 2^k term dominates (in almost any reasonable case), and the other terms end up being negligible. In part, it will depend on the precise way that the graph is being stored. Imagine it is being stored as an adjacency list. In that case, when you recurse removing node v, you have to update all the neighbors of v—in the entire execution, each edge is examined this way at most 2, and adds an additive $2m$ cost. Similarly, to find the next node to remove, we simply iterate through the remaining edges, which adds only $O(1)$ cost for each recursion call. In this case, at least, it simply requires a little bit of care to bound the amount of work being done in each recursion call.

Question 4 *In Week 3 lecture, you covered set cover. I was wondering how much the amortized runtime will increase (or not) if we do some pre-processing before running the greedy algorithm. Consider the following algorithm:*

1. *Find every item that is in only one set. Add that set to S.*
2. *Mark all the items covered by the sets in S.*
3. *Proceed to run the greedy algorithm to cover the remaining items*

We can implement this in $O(SV^2)$, where S is the number of sets and V is the number of vertices in the worst-case. But it in some cases it does a lot better. How does the average/amortized runtime compare with the greedy algorithm's? Or do we not care about amortized runtime?

First, I'm not sure what exactly you mean by "amortized" runtime here. Typically, we think of amortized runtime when we have a sequence of operations $op_1, op_2, op_3, \ldots$ where some may be expensive and some cheap, but the average or amortized runtime might be good.

Second, heuristics like this are certainly interesting to look at. In many real-world cases it will lead to significant improvements in running time (and sometimes in approximation factor). Often it is hard to state a general theorem, though, as it depends significantly on the input—and for the example you gave, I do not know a simple characterization of when it works well. (For example, if the sets added during the pre-processing phase cover at least an α fraction of the elements, it would seem like it should speed things up by an α factor.)

Third, it is a very good observation that this simple greedy algorithm can be quite slow when run on large data. And there is a very simple observation (that in some ways generalizes your proposal) that leads to faster implementation: if instead of choosing the set with the *most* uncovered elements, you choose a set with at least an α-fraction of the most uncovered elements, then you get an

algorithm that still guarantees a $\Theta(\log n/\alpha)$ approximation. But now it is much faster to implement! For more details on how this works (and the resulting performance improvement) see the paper (I'll add a link on the website):

> *Set Cover Algorithms for Very Large Data Sets by Cormode, Karloff, and Wirth*

Question 5 *What is LP standard form used for?*

It simply provides a standardized way to write an LP. This is useful when we want to state theorems about LPs, or when we need standard notation to refer to LPs. For example, I can say: Assume we have a linear program: $\max(c^T x)$ such that $Ax \leq b$. Then such-in-such is always true. Since you know that you can transform any LP into this form, my theorem will hold for all LPs. Good notation makes math a lot easier.

Question 6 *It seems that the set cover is very similar to the vertex cover problem. Therefore, is it possible to reduce it to VC? If it is possible, how can we show it?*

Yes! As we saw during tutorial, you can reduce the problem of vertex cover to set cover: given a vertex cover instance $G = (V, E)$, you can generate a set cover instance where the elements are the edges and the sets are derived from the vertices. By contrast, you cannot reduce set cover to vertex cover (unless $P = NP$) in an approximation-preserving fashion, since we know that vertex cover is easily 2-approximated, while set cover cannot be 2-approximated. (Notice that this argument does not exclude a non-approximation-preserving-reduction: see the relationship between vertex cover and maximum independent set.)

Question 7 *Are there general approaches to approximately solutions of ILP with LP ones?*

There are many, many techniques that have been developed. (And for some problems, it cannot be done, depending on the integrality gap.) In general, the idea is:

- Formulate the problem as an ILP.
- Relax the ILP to a linear program by allowing non-integral solution.
- Round the resulting solution to an integral one.
- Relate the integral solution back to the non-integral LP solution.

For vertex cover, we saw an example of this, where rounding was particularly simple: if the variable x_j was $\geq 1/2$, then we rounded it to 1, otherwise we rounded it to 0. For other problems, we must use randomized rounding, treating the variable x_j as a probability and setting it to 1 with probability x_j. And there are many other rounding techniques.

Question 8 *Problem set 2 Advanced Problems.* and
Discuss the routine/advanced problems.

Postponed. (Some were discussed. Some we will come back to. Some you will have to come to office hours for.)

Question 9 *Can we discuss problem 1 from problem set 1 and also problem 3, part c?*

Of note, during the discussion in tutorial we discovered some more efficient solutions to the k-blue edge spanning tree problem!

Question 10 *How to tell whether a algorithm is randomised or not? When we should use the expectation to decide the α value for α-approximation?*

An algorithm is randomized if at any point during the algorithm it uses a random number generator, i.e., if it makes a random step. Randomized algorithms can be random in several different ways:

- The random choices may affect the running time, i.e., the running time is a random variable.
- The random choices may affect the correctness, i.e., the algorithm returns a correct answer with some probability (and an incorrect answer otherwise).
- The random choices may affect the approximation factor, i.e., the approximation factor is a random variable.

If the randomness affects only the running time (and not the correctness), we say that it is a Las Vegas algorithm. If the randomness affects the correctness (but not the running time) then we say that it is a Monte Carlo algorithm. (Why? I have no idea.) The approximation factor is usually seen as an aspect of correctness.

If the quantity being calculated is a random variable, then you should either refer to its expected value, or state the probability that it satisfies your claim. For example, I might say:

> The expected running time of Algorithm Foo is $O(n^2)$.

Or, I might instead say:

> The running time of Algorithm Foo is $O(n^2)$ with probability at least $1 - 1/\sqrt{n}$.

A special category of the latter is when you say something is satisfied "with high probability" which means that the statement is true with probability at least $1 - 1/n^c$, where n is some reasonable notion of the size of the problem and c is a constant that (by modifying the algorithm) you can make as large as you want.

Question 11 *Given a LP with a strict inequality $ax < b$, how do you generate a linear program with all non-strict inequalities (i.e., standard form) where $ax \leq b$.*

You cannot! A linear program must consist of all non-strict inequalities, i.e., $\leq$, $\geq$, or $=$. Each constraint must define a closed half-space. Otherwise, the problem becomes much harder to solve.

Question 12 *Are there approximation algorithms for Steiner trees which makes use of Steiner nodes to do better than a 2-approximation?*

You *can* do better than a 2-approximation. For the Euclidean version, you can actually find a $(1 + \epsilon)$-approximation, for an $\epsilon > 0$. That is, you can approximate it as well as you'd like! The running time, however, will depend on ϵ, and the closer to optimal you try to get, the longer it will take. For the metric and general versions, we know that you can achieve a $\ln 4 + \epsilon = 1.386$ approximation. And we know that you cannot do better than a 96/95-approximation (unless $P = NP$). Closing that gap remains an open question!

Question 13 *How does the Simplex algorithm finds neighbors exactly?*

We discussed this in tutorial. In general, the idea is that if we have n variables and m constraints, then any set of n constraints can be used to generate a potential vertex. Two vertices are neighbors if they differ in only one constraint. (Think of that constraint as defining the edge between the two vertices.)

If you are at a vertex define by a set of constraints $(c_1, c_2, \ldots, c_n)$, then you can drop a constraint (e.g., c_j) and try replacing it with each of the other $m - n$ constraints; each such combination yields a potential neighboring vertex. If you try replacing all the n constraints with each of the $m - n$ other constraints, you will examine all the neighboring constraints. If you find one that is better, move there.

Of course, this is a horribly inefficient way of implementing Simplex. In practice, you can get much more efficient solutions by using standard linear algebraic tricks. If you would like more details on this, e-mail me and I can point you to some notes online. (Alternatively, just try google!)