CS4234: Optimization Algorithms

Week 5 Questions: answered!

Week 5 Questions

Here 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.

Travelling Salesman Problem and Related

Question 1 *For Steiner tree and TSP problems, what do we do if the graph is not complete? We can no longer use the "short-circuit" technique because the direct edge might not exist.*

Exactly! In the general or graph versions of these problems, some edges may not exist. For the General Steiner Tree problem, our definition already accomodates this issue: a Steiner tree is a tree on the input graph (and any algorithm has to find such a tree consisting only of edges in the original graph). For TSP, we defined the problem in terms of a distance function. However, as we discussed in Tutorial, you could also define the problem for a graph: given a graph, find a tour that visits every node in the graph.

Algorithmically, we showed how to cope with this in the Steiner Tree case by first calculating the metric completion, then using the Metric Steiner Tree algorithm to find a tree, and then reconstructing a tree (from the shortest paths) in the original graph. See the reduction from General Steiner Tree to Metric Steiner Tree for details. The same idea works for TSP. If you begin with a general graph, you can calculate the metric completion, solve the problem, and reconstruct a tour of the same (or lesser) cost in the original graph.

In general, if you have a graph, it is often (but not always) useful to think about the metric completion, because then you can imagine nodes connected by a distance metric that satisfies the triangle inequality and connects all pairs. (But beware: when you reconstruct paths in the original graph, you have to be careful. In this case, you get repeats in your TSP tour.)

Question 2 *In the general TSP problem, must the distance d be defined between every pair of two points in X? (I.e. must the graph be complete?) If it must, can the cost be infinity?*

See above. We defined TSP in terms of a complete distance function d, but it is easy to derive a version that takes a graph as an input. For our purposes, there seems to be little harm in allowing infinite cost edges, but in general one should be careful with that as it can occasionally cause problems. Often, it is a shorthand for saying, "some value that is sufficiently large as to be irrelevant," and sometimes that can cause issues. Here, though, it is not an issue.

Question 3 *In Christofides Algorithm, consider step 3 and 4:*

3. *Let O be the nodes in T with odd degree. Notice that $|O|$ is even.*
4. *Let M be the minimum cost perfect matching for O.*

There is no definition for the edge between O, so what does it mean when mentioning minimum cost perfect matching for O?

Recall that for TSP, we assume that part of the input is a general distance function d that gives us a distance between between any pair of points (u, v). So if we have $u \in O$ and $v \in O$, then we can find $d(u, v)$. A related question is what happens if we do not know d, i.e., the input to TSP is a graph (rather than a set of points and a distance function). In that case, see the previous question.

Question 4 *In the matching part of the 1.5-approximate algorithm (Christofides Algorithm) of TSP, is the matching bipartite matching?*

No, it is *not* a bipartite matching. The graph on which we running the matching algorithm is the complete graph on the odd-degree nodes. (A complete graph, i.e., a clique, is not bipartite.)

Question 5 *For the TSP problem, can the distance function d really be any general function? What happens when d is not symmetric? Will the approximation algorithms covered in class still work even though the metric completion is not well defined in such a case?*

We discussed in Tutorial and in class how to cope with asymmetric distances. (The approximation algorithms for symmetric TSP do not work.)

Question 6 *Are there any difference between TSP-metric in two dimensions and higher dimensions?*

If the distance function satisfies the triangle inequality, then we have already seen how to find a good (1.5) approximation. If the distance function is Euclidean and d-dimensional, we can find an arbitrarily good $(1+\epsilon)$-approximation for any $\epsilon > 0$, but the running time depends exponentially on $1/\epsilon$ and d. If the distance function is not Euclidean, then there are a few different definitions of "dimension" that you might use. In general, if you can embed your points into some low-dimensional space (for a variety of definitions of dimension), then you can often find a good approximation.

Question 7 *A simpler version of TSP might requires the salesman travel in one direction (left to right) then another (right to left). This should could be solved in polynomial time. Is this some approximation to the original problem?*

The version you describe is known as *bitonic TSP*, and as you say, it is solvable in polynomial time. (We talked about a simpler version in Tutorial where the salesman can only travel one direction, and then return to the start.) As we saw in tutorial, this is *not* necessarily a good approximation. But there may be situations where it provides a good enough answer anyways. It would be interesting if we could classify the instances when bitonic TSP *is* a good approximation. (For example, if all the points lie on a circle, I bet bitonic TSP is a good approximation. Is there any way to generalize this?)

Question 8 *How do we solve TSP approximately with linear programming?*

First, let us focus on the metric version of TSP. (If it is non-metric, and no repeats, there will be no good approximation unless $P = NP$.) The most famous linear programming approach is known as the Held-Karp Relaxation. Assume a graph $G = (V, E)$ with n nodes, m edges, and distance function d. The variables here are x_e for every $e \in E$. (There will be an exponential number of constraints.) We use the notation $\delta(S)$ to indicate the set of edges that cross from inside the set S to outside the set S, i.e., the edges $e = (u, v) \in E$ such that $u \in S$ and $v \notin S$.

$$\begin{aligned}
\min \sum_{e \in E} d(e) \cdot x_e & \\
\text{such that} & \\
\sum_{e \in \delta(S)} x_e &\geq 2 \quad \forall S \subseteq V,\ S \neq \emptyset,\ S \neq V \\
\sum_{e \in \delta(\{v\})} x_e &= 2 \quad \forall v \in V \\
x_e &\geq 0 \quad \forall e \in E \\
x_e &\leq 1 \quad \forall e \in E
\end{aligned}$$

The basic idea here is that if the x_e represent a valid tour of the graph, then each node should have exactly degree 2: one incoming edge and one outgoing edge along the tour. Similarly, for any subset $S \subseteq E$, there should be at least one tour edge entering the set and at least one leaving it (unless $S = V$). If $x_e \in \{0, 1\}$, this ensures that the resulting ILP yields an optimal solution to the travelling salesman problem (i.e., the resulting edges yield a valid tour).

The Held-Karp relaxation is known to have an integrality of at least 4/3 (i.e., it will not yield better than a 4/3-approximation), and an integrality gap of at most 3/2, so it can be used to find a 1.5-approximation (much like the Christofides approach). If you could prove that the Held-Karp relaxation really had an integrality gap of 4/3, then we would have a better approximation algorithm for the metric TSP problem—and that would be really exci

The details of how to round this linear program are not simple, but it follow a familiar basic strategy:

1. If the LP computes variables x, use the value of x to compute a distribution, and randomly sample from that distribution. This yields some collection of edges T.
2. Let O be the odd degree vertices in T. Find a minimum cost matching M among these edges.
3. Return $T \cup M$.

In general, given the relationship between the Held-Karp relaxation and the linear-programming formulation for minimum spanning trees, it perhaps makes sense that we follow a similar idea to Christofides algorithm here.

Question 9 *For problem set 4, exercise 2, how is the time complexity of traveling saleman $O(2^n n^3)$.*

There are several ways to get a dynamic program here (each with different running time bounds). One interesting idea is to imagine, for every subset S of points X, building optimal paths connecting every pair of points (u, v) while visiting every node in S. That is, for every set S where $u, v \in S$, define:

$$TSP(u, v, S) = \text{the cheapest path from } u \text{ to } v \text{ that visits every node in } S$$

First, we notice that if we can solve this problem, then we can solve TSP: for every pair (u, v), we simply examine the optimal solution for $TSP(u, v, X) + d(u, v)$, and take the smallest. The optimal solution has to be one of these options.

Second, we notice that for $|S| = 2$, the problem is trivially solved by the path connecting u and v (which are the only two nodes in S).

Third, we observe that we can solve this problem inductively. Imagine we are given $TSP(u, v, S)$, and we have already solved the problem for all sets S' that are smaller than S. The optimal path from u to v in S must consist of an optimal path from u to w in $S \setminus \{v\}$, along with the edge to v. That is:

$$TSP(u, v, S) = \min_{w \in S} [TSP(u, w, S \setminus \{v\}) = d(w, v)]$$

It remains to calculate the cost of this solution: there are 2^n possible sets $S \subseteq X$, and for each set S there are $\binom{|S|}{2} \leq n^2$ pairs of nodes. Thus in total, we need to calculate $O(2^n n^2)$ combinations of (u, v, S). Each calculation takes time $O(n)$ and hence we get a total cost of $O(2^n n^3)$.

A simple optimization reduces the total cost to $O(2^n n^2)$ by observing that we do not really need to calculate every pair (u, v) of every set: the optimal solution must contain node x_1, and so we can consider only sets S containing x_1, and only paths that begin at $u = x_1$. This reduces the cost to $O(2^n n^2)$.

Integer Linear Programming, Linear Programming, and Relaxation

Question 10 *Is there any example of integer linear programming (ILP) problem that can be solved in polynomial time?*

In general, solving an ILP is NP-hard. This is easy to see, as there are many NP-complete problems that can be easily expressed as an ILP. However, it is also easy to represent easy problems (that are in the complexity class P) as ILPs. For example, see the exercises on problem set 4 for an example of an ILP that finds a minimum spanning tree. (And you already know how to solve the MST problem in polynomial time.)

The more interesting answer to the question is that for some integer linear programs, the vertices of the convex space defined by the constraints are all integral. If the vertices are all integral, then you can simply relax the ILP to a linear program and solve it in polynomial time. This is a standard technique for solving an ILP efficiently: show that the vertices are all integers. This is the case for the Max Flow problem (and for matching problems), and that is why these problems can be solved efficiently, while very similar and closely related problems cannot!

Question 11 *Why is it that when the integrality gap is large, there is no way to round the linear program to find a good integral solution?*

Imagine we are solving an optimization problem with objection function f. Assume that the best integral solution for an ILP is x_I with value $f(x_I)$ and the best general solution to the LP relaxation isx_{lp} with value $f(x_{lp})$. Typically, the goal of rounding an LP is to use the non-integral solution x_{lp} to find a solution x that has a value $f(x)$ very close to the value $f(x_{lp})$. More generally, we want to let the solution x_{lp} guide us to help find a similar integral solution.

For example, given solution x_{lp}, we might somehow find a solution x' such that $f(x') \leq \alpha f(x_{lp})$. Since we know that $f(x_{lp}) \leq f(x_I)$, we conclude that x' gives us an α-approximation of optimal.

If the integrality gap is large, however, then there are no integral solutions anywhere close to x_{lp}. We will not (for example) be able to find a solution x' where $f(x') \leq \alpha f(x_{lp})$. Thus it is very hard to imagine any way you can use the information from x_{lp} to find the best integral solution—because there is no real relationship between the integral and non-integral solutions.

The independent set problem is a good example of this (which is why it was mentioned in tutorial the other week). Consider the natural LP for maximum independent set:

$$\begin{aligned} \max & \sum_{j=1}^{n} x_j \\ & \text{such that} \\ \forall (u,v) \in E \quad & x_u + x_v \quad \leq 1 \\ \forall u \in V \quad & x_u \leq 1 \\ \forall u \in V \quad & x_u \geq 0 \end{aligned}$$

We showed in Tutorial that the integrality gap is large: there exist solutions to this LP that are very large (e.g., as big as $n/2$), while the best integral solution is 1. For example, consider a clique.

In fact, for every graph G, no matter what the topology, I can give you a solution to the linear program of value at least $n/2$: set each variable $x_v = 1/2$. Notice in this case, the constraints are always satisfied, no matter what graph you choose.

And yet different graphs have very sized independent sets. Since I can give you the solution x to the LP without even seeing your graph, how could it possibly provide any useful information for finding the maximum sized independent set? The fact that $x_v = 1/2$ gives you *zero information* on whether v should be in the independent set.

Thus if the integrality gap is small, we can hope to use the relaxation to find a good integral solution. If the integrality gap is large, then

General Questions on Approximation Algorithms

Question 12 *Are there real life examples of multiple concepts being used to optimise solutions to the same problem? For example, is there a problem that requires the use of vertex covers, TSP and Steiner Trees.*

I am sure there are! Often one optimization problem becomes a building block for another (e.g., in the way that finding an MST, finding an Eulerian tour and finding a perfect matching become building blocks for Christofides algorithm).

It is not that hard to invent such problems! For example, imagine a variant of TSP where instead of visiting every location, we want to visit either every location or its neighbor. (Perhaps we are putting up posters to announce an event, and if two noticeboards are close together, we only need to place our bulletin on one of the two boards.) This problem combines dominating set (which is related to vertex cover) and TSP.

Question 13 *Other than giving a bound to probably reduce the search space, do approximation algorithms have any use in a real-world application where optimality is needed?*

If you really need an optimal answer, and the problem really is NP-hard, and the instances of the problem you are interested in are hard instances that are very big, then you are in trouble! In that case, we do not know of any good solutions to your problem. As you say, if we really need an optimal solution, perhaps an approximation algorithm can be used to limit the search space, but it will not guarantee an optimal solution.

In many cases, a good approximation algorithm may provide you with a good enough solution. (In fact, it often will find a solution that is even closer to optimal than the stated guarantee.) And in other cases, a good approximation algorithm may provide you with a good starting point to run a heuristic (e.g., local search) that will find an even better solution. And a good approximation algorithm may provide insight that helps to implement a useful heuristics (e.g., branch-and-bound).

So I would conclude that in practice, approximation algorithms are often very useful in practice. But they do not guarantee an optimal solution—and for NP-hard problems, no known technique does!

Question 14 *Will the fact that we are using a pseudorandom number generator affect the approximation of random algorithms? It is possible to construct a a bad input, if we know the sequence that a pseudorandom number generator outputs.*

Yes! This is a very important (and interesting) issue. As you point out, if the pseudorandom sequence is known in advance, then we can construct bad inputs (even if they are rare). Essentially, your "randomized" algorithm is deterministic in that case!

Hence randomized algorithms are only useful if you believe there is *some* true randomness in the world (or if you believe that the pseudorandomness is sufficiently disguised by apparent randomness that it is computationally hard to find the underlying determinism).

For the purpose of analysis (and for this class), we tend to assume true randomness. We imagine that we have a good source of random bits. However, there is a lot of really interesting research looking at the question of:

- How little *true randomness* do you need? Imagine that true randomness is hard and expensive to come by (e.g., you need to query a piece of hardware measuring radioactive decays of isotopes in a little box attached to your computer). In that case, we want to minimize the number of random bits that we need to ensure a good outcome.
- Can we take a small amount of real randomness and turn it into enough almost-randomness to solve our problems? It turns out that we can use some beautiful graph theory (based on graph expansion) to take a small number of completely independent (real) random bits and generate many more "sufficiently random" bits.
- How random do our random bits need to be? For example, for many applications, we do not need perfectly (mutually) independent random bits, but instead we only need pairwise independent (or k-wise independence). In other cases, we only need a sequence to be sufficiently random that a computationally bounded adversary cannot decode it, using past information to predict the future. There is a lot of work trying to understand, for different problems and settings, how random the random bits need to be.

For this class, though, we will tend to assume an arbitrarily large supply of trully independent random bits.

Question 15 *For the set cover and the greedy algorithm to solve set cover, the proof to show it is a $O(\log n)$-approximation is very interesting. Are there other cases which uses similar proof techniques? What are some of the other NP-hard problems that have $O(\log n)$-approximation algorithms.*

In Tutorial, we saw the example of vertex cover on a hypergraph: this has an $O(\log n)$ approximation, and if you prove it directly (instead of by reduction), it will have a very similar proof to set cover. We also saw in class that asymmetric-TSP has an $O(\log n)$ approximation, but the technique is entirely different. In general, there are many problems that have $O(\log n)$ approximations; a few involve similar proof ideas—particularly those that have a greedy algorithm. (Some of the knapsack approximation algorithms have a similar flavor, perhaps.)

Question 16 *What is the general techniques or common practise to prove the lower bound for an approximation ratio?*

In general, showing that something is inapproximable is difficult. The simplest approach is via a gap-preserving reduction. Imagine you are trying to solve a problem A, and you want to reduce from set cover. Then what you need to show is the following:

- Given any set cover instance x, you can construct an instance $f(x)$ to your problem.
- Given a solution s to the problem $f(x)$, you can construct a valid solution $g(s)$ to the set cover problem.
- If the solution s is an α-approximation to the optimal for your problem $A(f(x))$, then $g(s)$ is an α-approximation to the set cover instance x.

In that case, you can conclude that if you have an α-approximation algorithm for A, then you can build an α-approximation algorithm for set cover. We can then conclude that, for $\alpha = \log(n)$, there is no $\log n$-approximation algorithm for A.

Sometimes the approximation gaps do not line up perfectly: you can show that an α-approximate solution to your algorithm would yield a β-approximation to set cover. In that case, the inapproximability depends on the ration of α to β.

Question 17 *Do you know any interesting problems that can be reduced to metric TSP in a gap-preserving fashion, e.g., so we can use Christofides algorithm to solve them.*

That's a great question, and I do not have any good (non-trivial) examples off the top of my head.

Knapsack

Question 18 *I come across the Knapsack problem which is similar to PS4 S-2 problem, but with an extra factor: the profit. This problem can also be solved by a greedy algorithm as well. Can you show the proof of this?*

The problem on the problem set is a simplified version of the Knapsack problem. In the more typical version, each item i has a weight w_i and a value v_i. Given a maximum weight W, the goal is to find a subset of the items with maximum value that do not exceed the weight limit, i.e., output a set S such that $\sum_{j=1}^{|S|} w_j \leq W$ and maximizing $\sum_{j=1}^{|S|} v_i$.

In this case, you want to sort the elements by (v_i/w_i), i.e., their value per weight. From now on, assume the elements are sorted in this order (i.e., item 1 has the largest value per weight, and item n has the smallest). You want to consider two possible solutions:

- Add elements in sorted order from largest to smallest until there is no room left. Assume that this algorithm adds the first k elements.
- Add only element $k+1$.

The most elegant proof that this is a 2-approximation is to write down the linear programming relaxation for the knapsack problem, and observe that the sum of these two solutions is always larger than the fractional solution (which is always larger than OPT). I will leave that as an exercise.

The same style of dynamic programming solution yields an algorithm that will find an optimal solution in running time that depends on W or on the maximum value. If you use the same type of scaling trick that we used in the context of maximum-flow, scaling the values down to something smaller and then back up, you can get a $(1+\epsilon)$-approximation algorithm that runs in polynomial time (independent of W and the values).

Question 19 *Can we solve knapsack using linear programming?*

We cannot find an optimal solution (as it is NP-hard), but it provides useful insight into the simple 2-approximation algorithm. We can, of course, solve a fractional version of the knapsack problem using linear programming—but I'll leave that as an exercise.

Miscellaneous

Question 20 *Why does each Steiner point in an optimal solution have degree 3?*

Geometry. Imagine you have a Steiner point u with some degree $d > 3$. Then it must have two outgoing edges that form an angle < 120 degrees. Assume these edges connect to nodes v and w. I claim that you can create a new point z in the middle of the triangle (u, v, w) such that $d(u, z) + d(z, v) + d(z, w)$ is less then $d(u, v) + d(u, w)$.

Question 21 *Is min-cut the dual of max-flow problem? If yes, how can we find the dual given a problem?*

Yes. We will come back to this soon when we talk about duality.

Question 22 *We have seen that the Vertex Cover problem can be reduced into a Set Cover problem. Can the same be said for Hypergraphs?*

Yes! We discussed this in tutorial.

Question 23 *There were a variety of questions regarding the problem set. For regular problems, see the solutions. For advanced problems, try to solve them and then meet with me. For problem set 4, 2.c, I have no idea why I suggested there might be an $O(m \log n)$ solution—I suspect that the last time I asked this question, someone came up with such a solution!*