CS2040S Final Exam Notes

Question 10

Discussion

Part of the goal of this question was to be able to read a question and figure out the best way to use algorithmic tools to solve it.

The question is essentially a shortest path problem, where the goal is to make a round-the-world trip as quickly as possible. The natural way to model the question is as a graph G where each city is a node and each (directed) edge is a flight connecting two cities, where the weight is the flight duration.

Note that there are two ways of reading the question: (1) you are only allowed to circumnavigate the globe once, or (2) you can make as many loops around the earth as you want, as long as you only land in Singapore once. We accepted both, as long as you explained yourself.

For grading purposes, the problem was divided into two parts: (a) - (c) were Part I, and were allocated 7 marks, while (d) - (h) were Part II and were allocated 9 marks. The most common scores were (6 or 7) points earned for Part I and (8 or 9) points earned for Part II, since in general solutions were either correct, close, or didn't work.

Part I

Part I was a basic shortest path question. Given the graph of the airline network, remove all the east-to-west edges and find the shortest cycle that includes Singapore. There were two ways to interpret the question.

Interpretation (1) implied the graph is a DAG (as no cycles are possible going only from west to east in one circumnavigation of the globe), and so we were looking for DAG-shortest-path solutions, e.g., perform a topological sort of the graph containing only west-to-east edges (via post-order DFS) and then relax the outgoing edges of each node in toposort order. This led to a very simple O(n+m) solution.

Interpretation (2) implied the graph might not be a DAG (e.g., you might fly around the world from New York to Tokyo to New York to Tokyo as many times as you like). In this case, Dijkstra's

Algorithm was the right solution.

General notes (for both interpretations):

- Ideally, you pointed out that the solution was a DAG or was not a DAG.
- For the DAG solution, we wanted you to mention the shortest path / edge relaxation part. Just saying that you were performing a topological sort was not enough.
- For the DAG solution, many students suggested first doing a topological sort, and then doing "one iteration of Bellman-Ford." This isn't quite right, since Bellman-Ford is *unordered*, i.e., it relaxes edges in an arbitrary order. For a correct shortest path algorithm, you need to relax edges in topological order, not arbitrary Bellman-Ford.
- For either solution, we wanted some explanation of how you were getting from a shortest path algorithm to a shortest cycle. For example, you could split Singapore into two nodes (a source and a destination), or you could first run the shortest path algorithm, and then examine each neighbor of Singapore, etc. (The picture here sometimes helped.)
- Bellman-Ford was not as good a solution as Dijkstra or DAG-SSSP, because it is less efficient, and there is really no reasonable way to think about flights having negative duration. Even if time zones cause confusion, the flight time is still positive. Some partial credit was usually given.
- We wanted you to actually give an algorithm. Some people just stated "use an SSSP algorithm." Which one? Some are better than others!
- Some people had over complicated solutions, duplicating the graph an unnecessary number of times (sometimes twice, sometimes n times,). If the solution seemed like it worked (just inefficiently), some partial credit was given. If the solution seemed like it just didn't work, then none.

Notes on incorrect solutions:

- Some solutions said you could stop running Dijkstra when you "reached" Singapore again. What does this mean? If you implied that it meant you could stop as soon as you relaxed an edge (u,SG) that was an incoming edge to Singapore, then that is incorrect. (If you said that you could stop when you extracted SG from the priority queue, that is correct, assuming SG is in the priority queue and hasn't already been removed on Stop 1 of the algorithm.)
- Some people tried to use Prim's or Kruskal's algorithm to build an MST. This clearly will be wrong because the graph is directed, and Prim's and Kruskal's do not work on directed graphs.
- Some people tried to use directed-MST algorithms. That's better than using Prim's or Kruskal's, but still the question clearly cannot be solved by finding an MST. An MST does not help with shortest paths.
- Similarly, BFS and DFS are not helpful with shortest paths in weighted graphs.
- A few people seemed to interpret the question as requiring a visit to every city in the world. Beyond the plain reading of the question, there are two reasons this should have seemed unlikely: (i) it is plainly impossible to visit every city in the world on a round-the-world trip only

going east, and (ii) visiting every node in a graph is an instance of the traveling salesman problem (in this case, the asymmetric version), which you may recall from the last problem set is NP-hard. It is unlikely that I would ask you for an optimal solution to an NP-hard problem on an exam. (The example graph in part (d) also might have been a hint.)

Part II

This second part was a much trickier question. It should have reminded you of the wall-breaking maze question on Problem Set 7. And it might have reminded you of the lazy prize collector's problem.

I was fairly strict in that I only gave credit, generally, for algorithms that seemed likely to work. If there were various pieces of a correct solution but not enough to actually describe an algorithm that worked, it did not get credit. As above, there were times when a solution had some good ideas and may have been on the right track, but there was not enough to get credit.

In general, points were given for a correct description of the entire algorithm. For example, I did not give credit for later parts if the algorithm itself was wrong. (E.g., if you said that it could be solved using QuickSort and then told me that QuickSort had expected running time of O(n log n), you did not get partial credit for correctly identifying the running time.). The goal of the problem is to design an algorithm, and so you got credit for how close it was to a successful design of a solution or the problem.

Before outlining solutions, a few general notes:

- The problem asked for an algorithm with running time as a function of n and m. We didn't want solutions based on other parameters (like the number of one way edges), though we did not penalize much for this.
- Some students assumed that the number of wrong-way edges was a constant. This is also incorrect. Just because it is not a parameter does not make it a constant. In particular, you have to think about the worst-case: in the worst-case, the number of wrong-way edges could be m.
- A few students assumed T was a constant. Same point as above: T is not a constant. Worse, there is no real upper bound on T. Maybe T = n^{100}. It is not a good idea to have the running time of your algorithm depend on T.
- A few students assumed that all the edge weights and T were integers. If you want to make that assumption, please state it. That is not always true, and there is no real reason for it to be true when we are talking about flight times! In this case, it's not a great assumption.
- Some students gave exponential time solutions. Any solution that ran in exponential time in the worst case did not get credit.

- In general, I was fairly harsh about partial credit: if the algorithm did not work, I did not give credit. (It is too hard to try to decide "how close is an algorithm to working" or "how many good ideas did it have.") So your solution may have had several good ideas, but if there wasn't enough there to explain to me how it worked, it didn't get credit.
- And some solutions definitely felt like you were close to a correct answer, but not quite there, not close enough to get credit. (E.g., solutions that said things like, "It's kind of like the maze problem from PS7, but substitute wrong-way edges for walls, and then adapt the relax function kinda to prioritize things differently.") I could often see that you were on the right track! But I needed more detail to give credit.
- A lot of students were not very good at illustrating their algorithms (perhaps for lack of time), basically just drawing a graph with the solution. Drawing the solution doesn't help! Try to illustrate the key aspects of the algorithm so I can understand how it works.
- A lot of students were not very good at explaining why their algorithms worked (perhaps due to lack of time). Many just asserted, "it works because it finds the best path" or some such assertion that provides little extra information. Try to work on explaining why your algorithms work. (You will get more practice in CS3230!)
- And I'll say it again: a graph may have an *exponential* number of paths connecting any two points. Don't explore all the paths.

Let me outline some possible solutions.

1. The cleanest solution involved graph duplication. Any round-trip route will contain at most n edges (since there are no cycles in a shortest path). Therefore we can make n+1 copies of the graph. Within each copy, remove all the wrong-way edges, and connect consecutive graphs with wrong-way edges. (E.g., if (u,v) is a wrong way edge, then create edge (u*i, v*{i+1}) to connect u in graph i and v in graph i+1, for all i.).

   Notice that the way the graph is constructed, any path that starts in Singapore in Graph 1 and ends in Singapore in graph j must have crossed *exactly* j-1 wrong-way edges.

   At this point, you can run a shortest path algorithm from Singapore in the first graph, and find the shortest path to each of the Singapore nodes in each of the graphs (using a track as in Part I to find the distance to Singapore in the first graph). The first Singapore (in the smallest graph copy) that has time < T yields the solution.

   Note that you only need to run a single shortest-path algorithm (not once for each possible number of wrong-way edges), since it finds the shortest path from Singapore to all of the graphs in one execution.

If you think about what is really happening, the shortest-path algorithm is simultaneously finding the shortest path to Singapore with 1, 2, 3, ..., n wrong-way edges. Neat, right?

You could use either DAG-SSSP (if you believe the graph is a DAG) or Dijkstra (otherwise), either works. If you recognize that the graph is a DAG, it runs in time O(nm), which is surprisingly fast, given how much work it has to do. I don't know of a faster solution than this (though I also don't know that it is impossible).

2. A closely related solution did not actually describe the graph duplication, but instead just stored n different estimates at each node. Each estimate represented a pair (w, d) where w is the number of wrong-way edges and d is the distance. Each node stores one such estimate for each value of w from 1..n. (In effect, each node is *simulating* the n copies from the duplicates graphs above.)

   NOTE: it is critical that you store all n estimates. Any solution that didn't explicitly explain that you were storing all the estimates, I assumed was wrong.

   Another critical aspect of this solution was that you described how the edges were relaxed. For example, if you relax node u, do you relax *all* the estimates? Or do you just relax the smallest estimate? Or some other rule? Without a clear explanation of the relax function, it was impossible to tell if you had the right idea.

   And for the DAG solution, you had to specify in what order you were relaxing nodes. In this case, if you assumed the underlying graph is a DAG, you actually have a lot of flexibility. You can actually relax the edges in toposort order, all the estimates of each node at once. (There are some other orders that work to.). Prove for yourself that this works!

3. The same approach can work for Dijkstra, but you need to be a bit more careful about relaxation. In Dijkstra, you aren't allowed to specify which thing to relax. You have to follow the algorithms. And some key notes about Dijkstra:

- It stores nodes in the priority queue, not edges.
- When you extract a node from the priority queue, you relax all its outgoing edges.

   A lot of students wanted a version of Dijkstra that stored edges in the PQ, not nodes. That is a potentially tricky modification that needs to be described (as it can change when edges are relaxed).

   So for Dijkstra to work, you need to treat each (w, d) estimate as a separate entry in the priority queue. Again, I needed you to state this explicitly. In general, it *does not* work to simple relax all the outgoing edges and all the estimates for a node at once. It is very important that you relax

relax all the edges for estimates (w1, d1) before estimates for (w2, d2) if w1 < w2.

One possible way to achieve this is to modify Dijkstra to store edges instead of nodes in the PQ and then prioritize by wrong-way count AND insert n entires for each edge into the priority queue. Your solution had to explicitly explain that you were putting more than one entry for an edge into the priority queue.)

As a result, the running time was going to be a factor of n slower. (If your solution was O(m log n), then it was clear that the solution missed some of the details of the solution.)

You will notice that describing this solution properly and clearly is a lot longer and a lot harder than the graph duplication solution! A lot of solutions clearly had some subset of these ideas, but very few were able to explain all the pieces correctly.

4. There is also a Bellman-Ford solution along these lines. It is a bit easier to describe, since just like original Bellman-Ford, all you have to do is first establish n different estimates for each node, and then go and relax edges for different estimates. (You had to explain that you were relaxing n different estimates per node, though.)

   The interesting thing is that, since any path in the end is going to be length n, you still only have to do n-1 iterations of Bellman-Ford. But each iteration now has cost nm, since you are relaxing m edges each of which has n estimates to relax across it. So the total running time is O(n^2 m). This is slower (and so less credit) but a valid solution.

5. There are also valid solutions based on dynamic programming. Many of these are really just another way of describing the solutions above, e.g., they either end up looking like the Bellman-Ford solution or the DAG-toposort solution.

   The basic approach is to define subproblems like P[v, w] = the shortest path from Singapore using at most w wrong-way edges (which looks a lot like an estimate, right?), or P[v,w] = the shortest path to Singapore using at most w wrong-way edges.

   A second key requirement for getting credit for a DP solution was explaining how to solve the subproblems, e.g., the minimum among the incoming (or outgoing) edges of ....

   A third key requirement for getting credit for a DP solution was explaining the order in solving the subproblems.

   If you recognized that the graph was a DAG, for example, you might suggest that the right order

was forward or reverse topological order. Saying something like "moving backward from Singapore" without mentioning topological order was unclear, especially if you hadn't specified it was a DAG. What does backwards mean? Just taking any edge doesn't work.

If you didn't think the graph was a DAG, then you could try to solve the subproblems iteratively, and you end up with something like Bellman-Ford that requires more iterations.

Some notes on incorrect solutions:

1. The most common class of wrong solutions included the words "explore all the paths in the graph from Singapore to Singapore" or something similar. Let me repeat myself again: a graph (even a DAG) has an *exponential* number of paths connecting two points. So any such solution will be exponential, no matter how clever the rest of it is.

   So for example, any solution that used DFS (modified) or BFS (modified) to explore every cycle of length < T is going to be exponential time. There is no DFS (or BFS) related solution that is going to work efficiently.

   A lot of these solutions looked something like recursive backtracking: follow a path for a while, and when it exceed T, backtrack and try again. This is, of course, exponential time because of all that backtracking.

2. Some students suggested trying all the possible subsets of wrong-way edges and finding the shortest path. That too is inherently exponential, since there may be m wrong-way edges, so the number of possible combinations can be as bad as 2^m.

3. Many students tried to use binary search on the number of wrong-way edges. I like this instinct! That's the right way to think about problems. It's very good intuition: divide the problem into two parts, a binary search and a part that finds the shortest path with exactly k wrong-way edges. Unfortunately, you don't need the binary search since just one shortest-path execution is good enough.

4. Some of the binary search solutions asserted that you could solve the subproblems greedily. E.g., there is some set S of wrong-way edges that you keep in sorted order, and your binary search is going to test for paths of length < T with different subsets of S of size exactly k.

   I don't know any way to make this work directly. You can't simply add edges in order from smallest to largest. You can't simply add a prefix of the edges in S. Etc. Unfortunately, you

might need a strange subset of the edges in S to find the best solution.

The only thing that I know that works is to try all the subsets of S of size k, and that is exponential time.

3. Some solutions talked about graph duplication, but in an exponential way, not in a way that involved pre-building a good graph and using it. The typical incorrect solution here said something like, "Whenever you cross a wrong-way edge, make a copy of the graph." The problem with that is that you end up with an exponential sized graph, as you will cross the same wrong-way edge in many different copies, making many more copies. (From a DP perspective, it is a failure to memoize: each copy is unique and not overlapping, when what you want is the same copy for everyone that crosses the ith wrong-way edge.) Sometimes I couldn't tell and had to guess. If your picture showed the entire graph repeated 3 times with edges linking them up in order, I tended to assume you had the right idea. If your picture involved a new piece of the graph replicated after each wrong-way edge, I assumed it was exponential. When constructing a new graph, it is important to explain it well, either with words or pictures or both.

4. There were several solutions that made TWO copies of the graph, e.g., one for the right way edges and one for the wrong way edges. There were various strategies for connecting these graphs, and various discussion of counters to count how many times you crossed from one to the other. I don't think any of these worked. Often, the resulting graph was disconnected, or didn't have a path containing the correct solution. Sometimes, the resulting graph (in the picture) looked like it worked, but would fail if T=12, i.e., if the path needed to cross TWO wrong-way edges. In general, making two copies of the graph was not sufficient.

5. A common set of solutions used Dijkstra but "prioritized wrong-way counts" in the priority queue. Without other features of a correct solution, this does not work. There were several issues here. First, often this assumed the priority queue contained edges, not nodes. Second, it almost always ignored the fact that each node needed to store more than one estimate. Third, it rarely included the fact that each edge was inserted in the priority queue multiple times, once for each value of w.

   Various solutions included some subset of the above, but they were not correct unless they included all the needed components.

   One hint that a solution was not correct (or did not understand a key aspect) was if it seemed like each edge was only relaxed once. If each edge is relaxed only once, it does not work, and so implies something critical is missing. (This was often clear from the explanation of the running time.)

6. A lot of solutions focused on "aborting" when a path reached length T. There's nothing wrong with that, but it generally didn't help any. As described above, you generally want to keep all the different estimates anyways (for different wrong-way counts) and so keeping the extra estimates doesn't hurt much. And I think it confused a lot of people, because when you try the example in the problem, if you abort relaxing edge (C,B), you miss out on the fact that node B might need to be relaxed twice. (Think about the case where the edge weight for (C,B) is small enough that you can't abort, but still large enough that the path through C will eventually exceed T, at some point later in the graph.)

7. A few people tried DP solutions that depended on T, e.g., P[v, t] = minimum number of wrong-way edges to arrive in time t (or in time T-t). It is possible to formulate a correct DP this way. However, the running time will now depend on T, which is unfortunate (as discussed above), e.g., because T = n^{100} is possible, and maybe T isn't even an integer.

8. At least one clever solution tried to replace each edge of duration d with d-1 edges of duration 0 and 1 edge of duration 1. Now the time requirement of T reduces to a hop limit of T, and maybe you can make Bellman-Ford work properly! It seems complicated, and now the running time and space depend on T, but there might be a way to make it work.

9. Clearly any MST related solution is not going to work, because we are looking for short paths, not spanning trees.

Comments on some other questions

Question 8B

I think this was one of the most subtle questions on the exam. In general, for most questions like this, the answer is that it does not work: for Dijkstra's Algorithm, you can have lots of different values in the priority queue because the estimate depends on the distance, not the edge value. So specialized priority queues don't usually help Dijkstra.

But this problem had a special restriction: all the edge weights are 0 and 2. Draw an example and try it! What you'll realize is that (given the way Dijkstra works) at any given time there are only two values in the priority queue: x and x+2. Dijkstra first extracts all the x values from the PQ, and relaxes edges to add values that are equal to x and x+2. It keeps doing this until all the x values are removed, and it moves on to the x+2 values. At that instant, there are ONLY x+2

values in the PQ. And you will never get anything smaller than x+2 in the PQ again. And so it moves on to a new phase where the only two values are x+2 and x+4.

I think that's pretty neat, and a fact I didn't know before I wrote this exam!

Question 8D

Modified BFS basically never works. Without even reading the question, you should have initially been very suspicious. Trying to modify BFS or DFS where things get reinserted into the graph is almost always a mistake. Usually, it means you want something liked Bellman-Ford of Dijkstra.

However, just being suspicious is probably not quite enough to answer this question correctly. Maybe you thought I was asking a trick question. It is possible to come up with things that look a lot like BFS and do something that is reasonable.

There was a second key hint though: I claimed that the goal of the algorithm was to find all the possible distances between two nodes. And it wants to store these set of distances as estimates.

As soon as you saw that, you could stop reading the question. You didn't need to read the pseudocode. You didn't need to read anything else. The algorithm has to be exponential time.

Remember, there are an exponential number of paths connecting two points in a graph. And it isn't that hard to design such exponential set of paths to each have a different distance. (Imagine that each edge weight is a unique power of 2, so each unique path has a unique length.)

So the running time has to be exponential. The only thing left to check, if you want to be extra careful, is that the algorithm terminates. And it does, because the graph is a DAG.

Question 9

Note this is just InsertionSort, backwards.