CS2040S Final Exam Discussion

May 10, 2022

Overview

In this document, I'm going to go over some of the problems on the CS2040S final exam, discussing why certain solutions are (or are not) correct. I will not discuss all the question, but focus on those that seem most interesting (or that seemed most challenging). Feel free to raise questions on others on the forum.

1 Question 1

For this question, parts (b), (c), and (e) appear to have been the most challenging. The overall average for this question was 5/7 points.

(a) The point here is that binary search works even when the array has repeated elements. (That is unlike peak finding and other search methods that can struggle with repeated elements.)

(b) Function $f + g$ has more than one maximum value, so peak finding won't guarantee that you find the maximum.

(c) This is just solving a recurrence. (I find that drawing a recursion tree here is the easiest way to observe that at each level of recursion, the work halves.)

(d) This is not as obvious as it may initially seem, because the new items can be either to the right or the left of where they belong.

(e) This is the definition of amortized time. (This question appears to have been very tricky, as a large number of people got it wrong. I conclude that the definition of amortized running time was not very clear to a lot of people.)

(f) This is not the invariant for Bellman-Ford.

(g) Longest path is hard, but aside from that, if you negate a graph with positive weights, then it has negative weights and you can't use Dijkstra in a graph with negative weights.

2 Question 2

Overall, most people did well on these questions (scoring on average 5/6), with large majorities answering each question correctly. The hardest one appears to have been 2C (perhaps due to some confusion as to what it meant to remove the sorting step from Kruskal's).

(a) DFS does not find shortest paths. (BFS does, as long as the graph is unweighted.)

(b) For a sparse graph, an adjacency matrix is generally fairly inefficient for most applications.

(c) If you do not sort the edges by weight, then Kruskal's does not work properly. (For example, imagine the edges happened to be listed from largest to smallest in terms of weight. Then it would find a maximum-spanning tree!)

(d) BFS is faster than Bellman-Ford, but it only finds shortest paths if the graph has no edge weights or all the edge weights the same.

(e) These are two different ways to find a topological order, and they are roughly equivalent in terms of performance.

(f) If you replace a priority queue with a queue, Prim's will not find a minimum spanning tree.

3 Question 3

For this question, 3E was by far the hardest. Either the question was hard to understand, or the early-stopping condition for Dijkstra was not very clear. The other questions were relatively easy (it seemed) with an average score of 4/5.

(a) I think it is an interesting fact that each estimate (in both Bellman-Ford and Dijkstra) represents the distance on some path from the source. This should *not* be confused with the idea that Dijkstra (or Bellman-Ford) is constructing a shortest path one edge at a time. (This leads to fallacious arguments such as assuming Dijkstra can look at the current shortest path and backtrack or change it's construction on the fly.) Remember, Dijkstra is building a shortest path tree, and most of the edges you add will not actually be part of the shortest path from a specified source to a specified destination.

(b) This is the key property of Dijkstra that makes it work!

(c) This is a key invariant of basically every shortest path algorithm we looked at.

(d) And this is fundamentally not true of any shortest path algorithm we looked at. (Do you think we could build shortest path algorithms based on this invariant instead? We would be stretching edges instead of relaxing them them? Could we still use the triangle inequality to bound the distance?)

(e) A node is only done (for Dijkstra) when it is removed from the priority queue. For Bellman-Ford, you have to have an entire sequence of relaxations without any edge weight changing. (If no edge weight changes on the current shortest path from the source to the destination during Bellman-Ford, is that good enough to stop early?)

4 Question 4

The goal of this question was to think about a broken implementation of BFS, specifically in a DAG. The key problem with this version of BFS is that it did not check whether or not a node had been visited. Thus whenever a node enumerates its outgoing edges, it adds them all to the queue and then (later) visits them all.

Overall, this question appears to have been fairly tricky, with an average score of around 6.5/11. In particular 4E and 4F appear to have been the hardest, so I assume that it was challenging to figure out what the algorithm was really doing (and to realize that it was really exploring all the paths in the DAG from the source to every node). Beyond that, there appears to have been some confusion about whether $2^m = O(2^{2m})$ (which perhaps mirrors some confusion seen on the midterm about asymptotic analysis).

(a) If the graph has any cycles, this algorithm will never terminate. It will continue to explore around the cycle forever. However, since the graph is a DAG, it will terminate! Each node will be added to the queue at most a finite number of times, and so it will eventually terminate. At this point, you might wonder whether this broken version of BFS works acceptably well in a DAG? To figure that out, we have to explore ruther.

(b) The first observation is that it really is worse than regular BFS. Even in a DAG, an item most be added to (and removed from) the queue more than once. For example, if there are two paths from the source to a node, it will be added twice.

(c) In fact, it's even worse than that. You might have a node in the queue more than once at the same time! Imagine a node u at level-3 of the BFS tree is connected to two different nodes at level-2 of the BFS tree. Then both of those level-2 nodes will add node u to the queue.

(d) In fact, it's even worse than that. You might hope that you could limit the number of times a node is added to the queue by the number of incoming edges. It would be nice of each adjacent edge only added a node once. Alas, that's not the case. Imagine that node u is added by some nodes v and w (and those are the only two incoming edges). Well, each time v or w is visited, then node u is added to the queue. So if v and w are themselves added more than once, then u is added more than twice. And it continues growing exponentially as you work your way up the tree. All of which is suppsoed to help you with the next part.

(e) We have already seen that it terminates. The question is, what is the estimate when it terminates? (This is similar, but not identical, to asking how often a node is added to the queue.) Think about what happens in terms of "frontiers," as when we analyze a correct BFS. In the first frontier F_1, we visit all nodes that have a 1-hop path to the source. In the second frontier F_2, we visit all ndoes that have a 2-hop path to the source. (Notice this is different than BFS, where we do not visit *all* such nodes, but only nodes that have a 2-hop path *but no 1-hop path*.) In frontier F_k, we visit *every* node with a $k-$-hop path to the source. Next, think about the estimate: if the source starts out with estimate 0, then the estimate after visting is always equal to the frontier number! Frontier 1 gets estimate 1 (i.e., 0+1). Frontier 2 gets estimate 2 (i.e., 1+2). Etc. (Prove it by induction!) Finally, think about the last time a node u is visited. Imagine that it is visited in Frontier k. That means that it has a k hop path to the source, but no longer path to the source, i.e., k is the length of its longest path to the source. And if it is visited in frontier k, then it has estimate k after being visited. (Alas, the running time will thus be exponential.)

You might also observe that the algorithm is broken in a second way: we never explicitly initialize the estimate. So if the estimate is not initially 0 (e.g., initialized by the memory allocator), then the estimate may be nonsense (and so we accepted none-of-the-above as a correct answer as well).

(f) In order to find the longest path, you will notice that the algorithm is exploring every possible path! A node u will be visited (at least) once for every different path length it has to the source. And in fact, it is visited once for every possible path to the source. (We could count the number of paths to the source, if we wanted to!) Since the algorithm is exploring every path from the source to every node, it is visiting an exponential number of paths (in the worst-case), and so the running time is going to be exponential in m. There are at most 2^m paths in the graph, and all the other overhead is certainly at worst polynomial. Thus, 2^{2m} is the tightest possible answer. (Admittedly, it is significantly bigger than 2^m, but it is the tightest possible answer available and a correct asymptotic upper bound.)

5 Question 5

The goal of this question was simply to implement Dijkstra's Algorithm. Students often asked during lecture (e.g., on Archipelago) when the CS2040S practical exam would be—this would be the closest we would come! Luckily, almost everyone got these right with a score of 8.4/9. The hardest part seems to have been 5I, where you had to think about how to build the shortest path tree—the key point was that you want to update the parent pointer whenever you decrease the estimate. When a node's estimate decreases, that means it has found a new candidate for the shortest path to the source.

(The algorithm itself is a fairly standard implementation of Dijkstra's, so I'm not going to go through each part. Fill in the blanks with the correct answers and you should be able to show that it is a reasonable implementation of Dijkstra's.)

6 Question 6

This question was mostly about analyzing shortest path algorithms, and appears to have been very tricky, likely for two reasons: first, it involved being able to tweak and twist the algorithms and recompute the running time (which required really understanding how the analysis works), and second, it required some comfort with asymptotic notation. The average was 3/6 points.

Almost everyone got questino 6B wrong, and the most common answer was $O(m \log n)$. So it appears that a very large number of students did not realize that using an adjacency matrix would change the running time for Prim's

Algorithm. Interestingly, it appears that a lot of people who realized that BFS was slower with an adjacency matrix did not think that Prim's was any slower with an adjacency matrix.

(a) The challenge here was thinking about how BFS would be implemented with an adjacency matrix. Whenever you visit a node, you need to add all of its neighbors to the queue. To do that, you need to scan an entire row of the adjacency matrix. So the cost for a given node is $O(n)$, regardless of how many neighbors it has. In addition, each edge is traversed once (when a neighbor is found in the adjacency matrix). Since there are n nodes (and each node is visited once), the total cost is $n^2 + m = O(n^2)$. (It is, of course, always true that $m \leq n^2$, since we can only have two edges for every pair of nodes: from u to v and from v to u.)

(b) Similarly, for Prim's Algorithm, whenever a node is removed from the priority queue and checks all of its outgoing edges, it needs to traverse the entire row of the adjacency matrix. So there is an $O(n)$ cost per node, i.e., $O(n^2)$ total. However, there is still also the cost of using the priority queue. We still need to add and remove the nodes from the priority queue, at a cost of $O(n \log n)$ total. And we still need to perform `decreaseKey` whenever we explore an edge, which adds cost $O(m \log n)$. So the total cost here is $O(m \log n + n^2 + n \log n)$. Notice that this is not $O(n^2)$: in a clique $m = \Theta(n^2)$ so it is $\Theta(n^2 \log n)$ in that case. Thus the tightest answer available is $O(mn)$. (This is a tighter answer than $O(n^3)$ because sometimes $m < n^2$.)

You might observe that it is possible to implement Prim's with an adjacency matrix in $O(n^2)$ by using a different priority queue (i.e., not an AVL tree). Notably, imagine that you use a simple array as a priority queue. Now, decreaseKey is $O(1)$: simply lookup the node and update it. However, findMin is now $O(n)$. So the cost of using the priority queue is now $O(n^2)$ for adding and removing nodes from the priority queue, and $O(m)$ for decreaseKey. Hence the total cost of the algorithm is $O(n^2 + m)$. (However, the question explicitly insisted that you use an AVL tree for the priority queue.)

Surprisingly, the most common answer here was $O(m \log n)$, implying that an adjacency matrix has the same cost as an adjacency list for running Prim's. I suspect that means that we didn't well explain how Prim's works at a step-by-step level. (I wonder whether the answers would have been different if we asked about Dijkstra?)

(c) Recall that Dijkstra adds and removes each node once, and performs a decreaseKey once per edge.

7 Question 7

This question was an attempt to think about algorithm design. The goal was to think about which algorithm was best for a given real-world situation. That made the questions inherently a little bit less precise (and requiring a little more interpretation) than the rest of the exam, because you had to try to think about the constraints of a real-world setting (e.g., road distances aren't negative!). The difficulty seems to have been mixed, with some questions easy and some quite hard. Thus the overall average was about 5/10. Being able to answer questions like these is, I think, one of the most important goals of a class like CS2040S (i.e., being able to apply the algorithms to problems that don't exactly match the spec from class).

(a) This question was a classic network design question. The subway tunnels are, effectively, edges and the goal is to minimize the total cost of the network. Hence this was an MST problem. (You might allow cycles in your subway network, but in that case it is not minimal cost, which is the explicit goal.) This is not a shortest path question, since shortest paths do not yield the cheapest network. The travelling salesman approximation would have yielded a shortest (approximate) tour to visit all the stations, but again that was not the goal.

(b) The puzzle in question does not really matter. For answering this question, all you needed to know was that we have a puzzle with some state space, there is a winning state, and from each state there are a small number of moves we can make to move to a new state. And we want to find the shortest path. So, this is a shortest path problem (much like the Rubik's Cube or the 9-tile puzzle). The graph is unweighted, as each move has the same cost: 1. So BFS is the best solution. (Alas, solving such puzzles is exponential time.)

(c) This question was about finding the diameter of the graph. For any pair of points, the taxi route is the shortest (i.e., cheapest) path. You want to find the pair with the longest shortest path, i.e., the diameter. The graph is weighted (because every road segment does not have the same cost). Thus, there are two natural approaches for finding the diameter: Floyd-Warshall All-Pairs Shortest Path and run SSSP from each node as the source. Floyd-Warshall has cost $O(n^3)$. For SSSP, we can use Dijkstra (since all the costs are positive), and so the cost is $O(mn \log n)$. The problem specifies that the road network is sparse, i.e., $m = O(n)$, so the cost for n iterations of Dijkstra is $O(n^2 \log n)$. We conclude that Dijkstra is better than Floyd-Warshall in this case.

(d) This was a longest path question. You can imagine that there are two Singapore nodes: departure-Singapore and arrival-Singapore, and you want to find the longest path from departure-Singapore to arrival-Singapore. Unfortunately, longest path cannot be solved efficiently. The graph may have positive weight cycles, and that makes it impossible.

(e) This is *still* a longest path question. In fact, this is identical to the previous question, except that it assumes no negative weight edges. Alas, even not havig negative weight edges does not make the problem feasible. Finding longest paths is still hard.

The last two questions about longest path were tricky and many people got them wrong. There seem to have been several possible reasons to be confused.

For Question 7D, the most common wrong answers (by far) were Dijkstra's Algorithm and Prim's Algorithm. Since the problem explicitly contained negative weight edges, I'm not sure why Dijstra's was such a popular answer. I suspect Prim's was popular because it looks a little bit like a network design problem, designing a set of routes. For Question 7E, Dijkstra's and Prim's were still the most common wrong answers, along with DAG-shortest-paths. (Perhaps there was the assumption that if all edges are positive, then the graph is a DAG? It isn't.)

One possible confusion was what it means for it to be impossible to solve a problem efficiently. In essence, these problems *have no good solution*. Because there are positive weight cycles, the reason you can't solve them efficiently is not because we haven't invented clever algorithms but because there is no solution to find (short of an infinite cyclic path). Perhaps there was a better way I could have worded the problem (though I still believe that given the available options, "cannot be solved" is the most reasonable).

A few students seem to have interpreted the question from a true software engineer perspective: given this (unsolvable) problem, what is the best we can do? At least one student interpreted that to mean finding a path from Singapore to Singapore that traversed a cycle. Other students interpreted that to mean checking whether or not a positive weight cycle existed. I find this a laudable attitude, and exactly the right sort of "can do" attitude I'd like everyone in the class to have. Faced with an insoluble problem, don't give up—find something that works.

Alas, I don't think those are reasonable answers to this specific exam question, as written (and only a small minority of students appeared to have this interpretation, so I don't think it was implied by the question in some way that I am missing). Especially for 7E, since we are guaranteed to have positive weight cycles (as every edge is positive), these interpretations yield fairly trivial problems. (Still, I applaud the students who chose this interpretation.)

One final aspect to the confusion here, I think, was that in class we talked about (interchangably) two different types of longest path problems: longest *simple* path (where you can't repeat a node) and longest (any?) path. (In fact, you'll find them treated interchangeably on the web as well.) Longest path in a graph with positive weight cycles has no solution. Longest simple path has a solution, but it is NP-hard to compute. I sometimes may have discussed these interchangeably in class, because they are both infeasible. However, they are infeasible for different reasons, and so in the future I will be more precise about that distinction.

8 Question 8

This question was basic graph properties and executing graph algorithms. Mostly these were questions very similar to lecture trainings (e.g., executing an algorithm and tell me the fifth node seen, etc.). Overall, people did pretty well, with an average of 14.5/18 points. The hardest appear to have been 8A and 8C. Perhaps people didn't remember the definition of strogly connected component? I'm not sure why DFS was difficult.

(For parts (C) and (D), it was unclear whether the adjacency lists were sorted by weight or id, so I accepted both. For Part (E), it was unclear whether examining a node occurred when the edge was relaxed or when it was extracted from the priority queue, so I accepted both.)

(a) Definition of strongly connected components.

(b) Graphs with cycles don't have a topological order.

(c) Execute a DFS.

(d) Execute a BFS.

(e) Execute Dijkstra's.

(f) Execute Kruskal's.

(g) Execute Prim's.

(h) Cut rule of MSTs.

(i) False cut rule of MSTs.

(j) False cut rule of MSTs.

(k) Cycle rule of MSTs.

9 Question 9

These questions mainly involved executing the specified hashing algorithm. The average was 8/10. The first two parts were simply running the algorithm, finding the length of the chains and/or the locations of the elements in the array, and counting.

Parts (C), (D), and (E) were about properties of chaining and open addressing. The key point for (C) is that chaining will always work, no matter how full the table (but it's performance will degrade slowly as the table getsoverfull). Open addressing will fail catastrophically when the table is full, and it's performance degrades badly too.

For (D) and (E), the point is that neither hashing approach has worst-case efficient searching. (For that you need Cuckoo hashing!) But both yield good efficient performance when the table is not too full.

Finally, Part (F) asked you to do some computation comparing the two approaches for a given load. Basically, the load was just a little bit less than 1: $1/(1 + 1/\log n)$. So for chaining, we have approximately $O(1)$ operations. However, for open addressing we know that as the load approaches 1, the cost goes up fast! The cost per operation is $1/(1 - \alpha)$, and so that yields a cost a little bit more than $\log n$.

10 Question 10

The problem can be summarized as follows: given a directed, weighted, connected graph $G = (V, E)$, a source A and a destination Z, find a shortest path from A to Z where the number of hops on the path is divisible by 3. (Later we asked you to generalize this to paths divisible by k.)

The main part of the solution was to come up with a graph transformation that would allow you to efficiently solve the problem (and later generalize it). The second key component of a solution was choosing the right algorithm to run on the transformed graph. The third part was some analysis and generalization.

From a grading perspective there are two important things to remember. First, while the exams have three separate sub-marks, the question was really being evaluated as a whole, looking at the quality of the solution. The main goal was to assign points (out of 15) as follows:

- Basically right solution that yields an efficient algorithm (for the basic and generalized versions): 13-15 points.

- Inefficient solution, or solution with a (minor) error: 4-8 points. This included transformations that yielded slow algorithms (see below) or incorrect choice of algorithms (e.g., choosing Bellman-Ford instead of Dijkstra).
- Solution that does not find the right answer or is excessively slow: 0-3 points.

The second aspect is that the problem was (in some sense) cumulative: it was hard to get later parts right if you got earlier parts wrong (even if you wrote some of the correct words). For example, if the graph transformation did not work, then writing "Dijkstra" for the algorithm would not be a correct answer. Similarly, if you had an inefficient/slow algorithm for the first part, then you probably did not get points for the later parts. See the above philosophy about evaluating the solution as a whole, not as a set of pieces. The average for the question was about 8/15 points. (Hence, a little more than one third fell in the first category, with a little less than a third in each of the other two categories.)

Here, I will summarize the aspects of a model solution, and then discuss a few common wrong solutions.

10.1 Model Solution

The "model" solution involved replicating the graph 3 times. The key insight is that you want each node in your transformed graph to capture the *state of the system*, i.e., in this case both the location and who is driving.

Thus, we can define the nodes in the new graph as follows: each node u in the original graph is mapped to (u, j) in the new graph where j is an integer from $\{0, 1, 2\}$, where 0, 1, and 2 refer to the drivers Alice, Bob, and Carol.

The edges connect nodes in one graph to nodes in the next, using the same weights as the original graph. Thus, if there is an edge of weight w connecting (u, v) in the original graph, then there is an edge of weight w from (u, i) to $(u, i + 1 \bmod 3)$ in the new graph for $i = \{0, 1, 2\}$.

Ideally, a picture should illustrate this structure. Typically, people drew this solution as three interconnected graphs: A, B, and C. Key points to look for in the diagram:

- It should be clear that the graph contains three copies of each node. (You do not need four.)
- If the graphs are divided up into three (graph A, graph B, and graph C), there should be a cycle: A should connect to B should connect to C should connect to A. (Many people forgot to connect the third graph back to the first.)
- There should be no edges internal to a copy of a graph. That is, there should be no edges connecting an A node with another A node or a B node with another B node. All edges should be between copies.
- The edges should have the proper weights. (There should not be 0-weight edges added between graphs.)
- There is no requirement to remove the destination Z node from any graph. You can cycle around Z if you need to. (There is no stipulation that the trip ends as soon as you arrive at Z, though we accepted this interpretation.)
- There is no need to create a super-node for the start, as there is only one fixed start node (A in Alice's graph).

This graph is easy to build $O(n+m)$ tie using a simple graph traversal, either BFS or DFS. The new graph contains $3n$ nodes and $3m$ edges. The new graph may have cycles (i.e., it is not a DAG), but it has no negative weights (because roads are positive length). Thus, we can run Dijkstra's algorithm on the new graph to solve the problem is $O(m \log n)$ time.

(Once the graph was trasnformed, some students suggested using Bellman-Ford. This find the right answer but slower. Other students suggested using DAG-SSSP or BFS as the shortest path algorithm. These are simply incorrect as the graph is weighted and not a DAG. Some students suggested using DFS, which does not find shortest paths.)

A common mistake was misidentifying the destination. Some students chose a destination in Graph 3. (That would not yield the right divisibility.) Some students checked all three destinations (i.e., Z in each graph copy) and chose the shortest. Again, not the right multiplicity.

The generalized version asks for a shortest path where the number of hops is divisible by k, and the same solution works, except repeating each node k times insteads of 3. The running time, though, is larger: $O(km \log(kn))$. (The parameter k should not be treated like a constant, as we expicitly asked for the running time in terms of k.)

The DAG version asks how you would solve the problem more efficiently if there are no cycles in the graph. In that case, you can use the DAG_SSSP (toposort+relax edges) instead of Dijkstra. The running times is then $O(km + kn)$ since there are (still) km edges and kn nodes. (Many students only partly explained what to do, suggesting that you should perform a topological sort, but forgetting the other important half of the algorithm: relaxing the edges in the proper order.)

10.2 Some Less Correct Solutions

Expensive graph transformation 1. There were several different ways to transform the graph that could lead to a correct solution, but were a lot more expensive. Typically these solutions were also much more complicated to implement and did not generalize as well.

For example, one approach was to keep the same set of nodes, but to only connect a pair of nodes (u, v) if there was a three hop path connecting them. Thus each edge in the new graph represents a 3-edge path in the original graph and has an equivalent cost. Given such a graph, you can correctly find a shortest path divisible by 3.

Unfortunately, there are two problems here. First, the new graph may have a lot more edges than the old graph. Remember, the new graph has one edge for every 3-edge path in the original graph. If the original graph is a clique, there are approximatley n^3 such 3-edge paths. (Perhaps you only want to include one edge for every pair (u, v), but you may still need to generate them all to find the one that is the lowest weight.) Or we can easily construct a graph with only $n^{3/2}$ edges where the transformation generates a clique. (Think about the birthday paradox!) So you can no longer assume the new graph has only m edges. If you prune edges, you should probably assume that the new graph has n^2 edges. (Tighter bounds seem difficult!)

The second problem is that such a graph may be expensive to build. If you build it by simply exploring all the 3-hop paths, then there may be n^3 such paths and so it may take n^3 time. You *cannot* build such a graph by running BFS or DFS, because BFS and DFS do not visit every path in a graph. You also cannot simply use 3 iterations of normal Bellman-Ford, because Bellman-Ford finds shortest paths—it does not keep track of paths at a given number of hops. If you have a path of length 1 and a path of length to reach a node, it will happily return the path of length 1 (if it is shorter). What is the easiest way to efficiently construct the desired graph? Well, via graph transformation: replicate the graph 3 times and run a shortest path algorithm. (Does that sound familiar?)

So this type of solution inherently had at least an n^3 or nm cost (unless you talked about using a replication approach to efficiently compute it), and produces a new graph with up to n^2 edges. Dijkstra's algorithm on the new graph will therefore run in time $O(n^2 \log n)$, yielding (potentially) a running time of $O(nm + n^2 \log n)$.

This algorithm also generalizes poorly to the k-divisible version. While it is at least polynomial time to find 3-hop paths, it is potentially n^k time to find k hop paths. (Again, remember, BFS and DFS do not work, and Bellman-Ford likely needs to be modified in a way that would make it have a similar cost.) In general, the running times achievable for the k-divisible generalization were too slow to receive credit for the generalization.

Expensive graph transformation 2. A similar approach (with similar issues) build a graph that tried to replace each path of length three with a single node. (This was often described as "compressing" the graph.) These nodes were then (somehow, often not really explained) connected up to form a correct sequence. Again, this can (probably) be made to work. However, as above, it suffers the problems of expensive construction and expensive transformation.

Since the graph contains one node for each sequence of three nodes in the original graph, it may contain n^3 nodes. (Think of a clique.) And it may similarly contain at least n^3 edges, likely more (e.g., $n^2 m$). Thus the construction will be at least n^3 time, and Dijkstra will take $O(n^3 \log n)$ or worse.

And again, it generalizes quite poorly to the k-divisible case, yielding running times and size exponential in k.

Expensive graph transformation 3. A different approach was to construct a DAG, as in the prize collecting problem. That is, the graph is replicated a large number of times, and you only move from graph i to graph $i + 1$. How many times does the graph need to be replicated? (And it is important to specify—you cannot just choose a variable and leave it unclear.) In fact, the graph has to be replicated at least $3n$ times. It turns out that n times or m times is not enough, since you may have to go around a cycle multiple times to fix the "driver parity." (Remember, you can essentially be in any of three states when you enter a node: Alice, Bob, or Carol driving.)

Once you have constructed a DAG, you can use the DAG-shortest-path algorithm, instead of Dijkstra. You can then check the destination in graphs that are divisible by 3.

In this case, you end up with a graph with $3n^2$ nodes and $3nm$ edges, and it takes at least $O(n^2 + nm)$ time to construct. DAG-shortest paths runs in time $O(n^2 + nm)$ as well. Again, this is a slower solution, but it can work.

When you want divisble by k paths, it is now possible that the shortest path has length kn. So you need to create kn copies of the original graph (since you can now be in any of k states when you enter a node), resulting in a new graph of kn^2 nodes and knm edges. (Very few solutions correctly identified the number of copies you need here.)

Exponential graph transformation. A variant of the "build a DAG" approach was to "build a tree." Some solutions attempted to build a tree where each path corresponded to one path in the graph. (Some phrased this as graph replication, where at every step you make a copy of the graph. Some tried to integrate this into modified Dijkstra, where at every step you made a copy of the graph.) Unfortunately, at best any such construction is exponential: there are an exponential number of paths in a graph. In fact, really such a construction yields an infinite tree, since there are cycles in the original graph. (See below for cycle elimination.)

Cycle elimination. Related to "build a DAG" and "build a tree", some solutions wanted to remove cycles from the graph. Sometimes this involved simply deleting problematic edges. Other times, the idea was to make a new copy of the graph whenever a cycle was discovered. (Sometimes this was supposed to be done as part of modified Dijstra.)

In general, removing cycles from a graph is hard because there are an exponential number of cycles in a graph. (Note that any solution which specifies making one copy of the graph for each cycle is an exponential solution.) Think about a clique: any set of $n/2$ nodes forms a cycle, so there at least $n^{n/2}$ cycles in a clique!

Most such solutions were not specified in enough detail to see if they would work. How do you decide which edges to remove? (Again, think about a clique.) How many edges? (To make a clique into a tree, you have to remove $\Theta(n^2)$ edges.) How much do all those copies cost? Have you changed the cost by deleting those edges? (If you copy the graph and delete an edge on the cycle, does the new graph contain that edge? If so, the new graph has a cycle. If not, the new graph may not have the shortest path.)

Of all the ways of making the graph into a DAG or a tree, the only one that really makes sense is the first, i.e., copy the graph once per time step.

Iterated Dijkstra. Some studetns came up with versions that ran Dijkstra iteratively, modifying the graph after each execution. For example, you might run Dijkstra, find the shortest path, and if the shortest path isn't divisible by 3, then (somehow) compute a "bad" edge and delete it. In general, these didn't work. I don't think there is an easy way to figure out which edges are safe to eliminate (and none of the solutions I saw explained such a rule). Moreover, even if it did work, the solution would be fairly slow (as it requires re-running Dijkstra many times).

Modifying Dijkstra. Some students came up with versions that modified Dijkstra in some complicated way. These generally didn't work, and they also didn't satisfy the goal of the questions (i.e., giving a graph transformation that would make the problem easy to solve).

For example, some solutions discussed prioritizing certain nodes in the priority queue when running Dijkstra. Or backtracking and not following a certain edge if Dijkstra discovered a bad property. Some solutions insisted that Dijkstra not follow 0-weight edges at some times, and only follow them at other times. Some solutions suggested jumping to a new graph (or creating a new graph copy) while Dijkstra was running based on certain conditions.

Most of these types of changes don't make sense and can't be implemented (correctly) in the context of Dijstra. Remember, Dijkstra is not following a shortest path. It is not traversing a path. It doesn't know (at any given time) what the shortest path is, or how it is being extended. It is building a shortest-path-tree, and it is maintaining estimates for shortest paths along the way. Moreover, most modifications like those described break the required invariants of Dijkstra's Algorithm, and hence wuold result in either incorrect solutions or unknown performance.

(Let me be careful to not say that any such change is *impossible*. There are complicated ways you can do some of the things. And I do know of at least one "modify Dijkstra" solution to this problem, though it did not appear on any exam, and would have been an incorrect answer anyways, given the question asked for a graph transformation. So there are possible solutions, but they are much, much harder.)

DAG's are not trivial. Some solutions assumed that once the graph was a DAG (whether via transformation, cycle elimination, whatever), then no more work was needed: simply toposort the graph and take the nodes divisible by 3 in the toposort order. This does not work: the toposort order is not necessasrily the order you visit nodes, and hence whether or not the toposort order is divisible by 3 does not matter. You still need to do some extra work (e.g., a graph transformation) to find paths of length divisible by 3.

DFS. Some people suggested performing a DFS, keeping track of the hop count. This does not work because DFS does not visit all paths in a graph (and if it did, it would be exponential), and DFS does not find shortest paths at all.