CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Problem Set 5

Harvard SEAS - Fall 2024 *Due: Wed Oct. 23, 2024 (11:59pm)*

Your name: Ethan Veghte
Collaborators:
No. of late days used on previous psets: 7
No. of late days used after including this pset: 0

Remember to mark your pages on Gradescope properly, or points will be taken off. Additionally, if your handwriting isn't particularly neat please submit written proofs, which are much easier to grade for the TFs.

1. (Solving Games) Consider playing a solo game on an $n \times n$ chessboard. You have one piece, a chess knight, which starts in the lower-left corner, and your goal is to reach any of the other three corners in as few moves as possible. Like a usual chess knight, in one move, you can move to any position that is two squares away in a horizontal direction and one square away in a vertical direction, or two squares away in a vertical direction and one square away in a horizontal direction. There is a catch, however: some squares have visible landmines, so you cannot move to them (since you do not want set off an explosion).

   (a) Give an algorithm that achieves the above goal in time $O(n^2)$ by reduction to either the ShortestWalks problem or the SingleSourceShortestPaths problem. The algorithm should output $\perp$ if no sequence of moves can take you to any of the other three corners when started in the lower-left corner. (Note: if reducing to SingleSourceShortestPaths, we haven't defined abstractly what it means to reduce a computational problem to a data structure problem, but you may construct and use a SingleSourceShortestPaths data structure on an appropriately constructed graph.)

   The way to solve this problem in $O(n^2)$ time is to set up a SingleSourceShortestPaths problem from a starting square to any of the 3 other corners. The key is the graph set up.

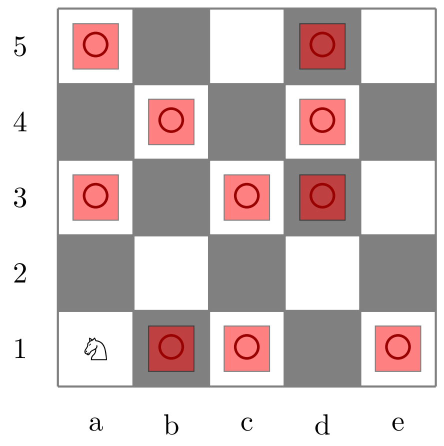
   Graph setup: In order to properly represent the grid in a graph, you would have every square be a node, and every edge connect nodes that can be reached by a knight in one move, but that also do not have a mine. This means that for example, in the grid below (2b), b3-a1 will have an edge, but b3-c1 will not. This is because c1 holds a mine and thus even though c1 is a square that generally a knight could travel to (2 away vertically and 1 horizontally), there is a mine that prevents the traversal. Setting up this graph takes $O(n^2)$ because it is a $n \times n$ grid thus there are n^2 nodes to create.

   Solution: To solve, you would use a reduction to SingleSourceShortestPaths which can use BFS as an oracle to solve. BFS is efficient on unweighted graphs and can solve in $O(n^2)$ time. All the nodes (n^2) and edges ($O(n^2)$ because a node has at most 8 edges). Furthermore, once BFS completes you check to see if any of the 3 corners have been *visited. If the distances to all the corners is infinite then the algorithm can return $\perp$, because it means there is not a shortest path.*

(b) Carry out your algorithm on the 5×5 board shown below, listing both the frontier vertices and the predecessor relationships at each stage of BFS (the red square symbols are landmines).

i. *Initialize:* $F_{frontier} = \{\}$, $S_{startingpoint} = a1$, $V_{visited} = \{\}$
ii. *Starting point:* $C_{current} = a1$, $F = \{b3, c2\}$ $V = \{a1\}$
iii. $C = b3$: $F = \{c2, d1, c5\}$ *(a1 already visited)*, $V = \{a1, b3\}$
iv. $C = c2$: $F = \{d1, c5, e3\}$ *(a1 already visited)*, $V = \{a1, b3, c2\}$
v. $C = d1$: $F = \{c5, e3, c4, e4\}$ *(b3 already visited)*, $V = \{a1, b3, c2, d1\}$
vi. $C = c5$: $F = \{e3, c4, e4, a4\}$, $V = \{a1, b3, c2, d1, c5\}$
vii. $C = e3$: $F = \{c4, e4, a4\}$, $V = \{a1, b3, c2, d1, c5, e3\}$ *(all neighbors are in frontier or have been visited)*
viii. $C = c4$: $F = \{e4, a4, e5\}$, $V = \{a1, b3, c2, d1, c5, e3, c4\}$
ix. $C = e4$: $F = \{a4, e5, d2\}$, $V = \{a1, b3, c2, d1, c5, e3, c4, e4\}$
x. $C = a4$: $F = \{e5, d2, b2\}$, $V = \{a1, b3, c2, d1, c5, e3, c4, e4, b2\}$
xi. $C = e5$: $F = \{d2, b2\}$, $V = \{a1, b3, c2, d1, c5, e3, c4, e4, a4, e5\}$
xii. $C = d2$: $F = \{b2\}$, $V = \{a1, b3, c2, d1, c5, e3, c4, e4, a4, e5, d2\}$
xiii. $C = b2$: $F = \{\}$, $V = \{a1, b3, c2, d1, c5, e3, c4, e4, a4, e5, d2, b2\}$
xiv. *Finish: All of frontier has been visited, thus now we check V to see if any of the corner nodes are there, and we see it is successful because e5 is there.*

This completes the BFS algorithm as there are no longer any nodes in the array to visit (there are others like a2 in grid that do not have landmines, but cannot be reached because its neighbors have landmines). At this point we can search through the visited nodes and see that e5, a corner node, was visited and thus show a successful path from start to a corner node. Were we to need the actual path, you would also store the parent nodes (as value of key-value pair) whenever you add a node to the frontier so that you can traverse backward from the final node. It would have looked too messy in the steps above so I omitted it. We can go back through and check where each node was introduced into the frontier to see its parent node. e5 was introduced into F by c4, which was introduced by d1, which was introduced by b3, which was introduced from starting node a1. So to traverse the path backward, $e5 \rightarrow c4 \rightarrow d1 \rightarrow b3 \rightarrow a1$. *Or forward,* $a1 \rightarrow b3 \rightarrow d1 \rightarrow c4 \rightarrow e5$. *The reason this is the shortest path is because the only way to have a node be the current node is to have already visited the nodes that are of lower level (i.e. fewer number of steps from starting point). This is because every time a node is added to the frontier, it is added to back of set, so that it is reached only once nodes of lower level have been added to visited set (FIFO). Also once a node has been added to frontier or visited set it will not be added again, so that you can ensure that it is the shortest path to that node.*



2. (Maximal Independent Sets) Let $G = (V, E)$ be a graph. A set $S \subseteq V$ is a *maximal* independent set if we cannot add any vertices to S while it remains an independent set. That is, for every vertex $v \in V \setminus S$, $S \cup \{v\}$ is not an independent set.

   (a) Show that given a graph G, a maximal independent set can be found in time $O(n+m)$. Note that this is in sharp contrast to *maximum-size* independent sets, for which we do not know any subexponential-time algorithms. (Hint: be greedy.)

   A maximal independent set can be found in time $O(n + m)$ with a greedy algorithm, because you can start at any node, and find a maximal independent set from there. Given a graph, put all of the nodes in a set G, and choose one randomly. Make that the first entry in the independent set. Remove all of the nodes in G which share an edge with that node. Then go back to G and choose another node, and remove its neighbors. Once G becomes empty, it means there is not another node that is in the graph that is not connected to any of the nodes in the independent set. All of the nodes removed either are put into the independent set, or share an edge with a node in the independent set, thus cannot be added to the set without breaking the independent set principle. This then confirms the output as a maximal independent set.
   Runtime: First you must create G which means to traverse every node n and every edge m to ensure that you know who all of the neighbors are of every node. The creation of this set which also holds information on the set of neighbors for each node requires traversing every node and edge. To go through all of the nodes is $O(n)$ and all of the edges is $O(m)$ $\Rightarrow O(n + m)$. Actually creating/finding a maximal independent set entails, at a worse case level, traversing through every node (no edges between any) ($O(n)$), or every node shares edges with every other one and you cut all the nodes after first iteration because every other node is touching via an edge. Either case, the set up is the dominant factor thus overall this is $O(n + m)$.

   (b) Show that if G is 3-colorable, then it has a 3-coloring f in which the set of vertices of color 2 (i.e. $f^{-1}(2)$) is a maximal independent set.
   If G is 3-colorable, the first recognition is that all vertices of color 2 do not share edges with each other (definition of being same color in graph coloring), making them an independent set. The reason, there must be a 3-coloring f in which the set of vertices of color

2 is a maximal independent set is a function of the possibility of setting up color 1 and 3 so that each vertex in the two colors shares an edge into colr 2. It is possible to find a 3-coloring, such that every vertex in color 2 set, is either neighbors with a vertex in color 1 or color 3 (see figure below), and that there are no vertices in color 1 or color 3 that have an edge only to the opposite (color 1 to 3, or 3 to 1). Both colorings below are valid, but using the right one, it becomes a maximal independent set. You manipulate the coloring on the left so that node 6 is color 2, making it so there is no longer a path from color 1 to color 3 from a node (or two nodes) that do not connect to color 2. If there are, those vertices can be freely moved to color 2 without jeopardizing the independence of the colors. In setting up a coloring in this manner, there will be no vertex in color 1 or color 3 that is not connected to color 2 in at least 1 edge. This means that any vertex in color 1 or 3 cannot be added to color 2 without ruining the independent set principle. This then shows that there is a possible coloring for any 3-colorable graph such that color 2 is a maximal independent set.

left figure: C1: {1, 2} C2: {3, 4} C3: {5, 6}
right figure: C1: {1, 2} C2: {3, 4, 6} C3: {5}

```
  /----------\                /----------\
 1     3 ---- 5              1     3 ---- 5
   \        /          to      \        /
     \    /                      \    /
 2 --- 4      6              2 --- 4
  \__________/                 \
                                 \
                                   6
```

(c) It is known that every graph G has at most $3^{n/3}$ maximal independent sets, and there is an algorithm (the Bron-Kerbosch algorithm) that enumerates all of the maximal independent sets in time $O(3^{n/3})$. Use this fact to conclude that 3-coloring can be solved in time $O((n+m)\cdot 3^{n/3}) = O(1.44^n)$, improving the runtime of $O(1.89^n)$ from SRE4.

The reason 3-coloring can be solved in time $O((n+m)\cdot 3^{n/3})$ is because any color in a 3-coloring can be completely removed and the resulting vertices/graph is a bipartite graph (2-colorable). Thus, since we know enumerating/finding all of the maximal independent sets in $O(3^{n/3})$, we must iterate across these independent sets until we find a valid 3-coloring. As demonstrated in 2b, any 3-colorable graph will have a set of vertices that is a maximal independent set, thus by using the maximal independent sets found with the Bron-Kerbosch algorithm, we can assume that one of them will lead to a tripartite graph coloring (i.e. is a independent set that is a color in that graph). For every maximal independent set enumerated, you have to remove all of those vertices from G (original set of vertices), and check if remaining graph G' is bipartite. The first step, removing vertices, can take O(n+m) because, worse case, you have to traverse over every vertex and edge to ensure they are all properly removed. Then, as from lecture we know that bipartite graphs can be checked for colorability *in $O(n+m)$ using BFS. If it can be 2-colored, then it is a valid 3-coloring.*

The result means that you remove vertices and edges in $O(n+m)$, check for bipartite coloring on leftover nodes $O(n+m)$, and potentially iterate over all maximal independent sets in $O((n+m)\cdot 3^{n/3})$, producing runtime of $O((n+m)\cdot 3^{n/3} * (n+m))$

3. (Exponential-Time Coloring) In the Github repository for PS5, we have given you basic data structures for graphs (in adjacency list representation) and colorings, an implementation of the Exhaustive-Search k-Coloring algorithm, an implementation of the Bron-Kerbosch algorithm, and a variety of test cases (graphs) for coloring algorithms.

   (a) Implement the $O(n+m)$-time algorithm for 2-coloring that we covered in class in the function `bfs_2_coloring`, verifying its correctness by running `python3 -m ps5_tests 2`. Your implementation of BFS should follow the presentation and notation that we used in class (with the loop over distance d and the sets F and S), which may be different than presentation of BFS in other sources (online or in the optional textbooks).

   (b) Implement the $O((n+m)\cdot 3^{n/3})$-time algorithm for 3-coloring (MaximalIS + BFS) from Problem 2 above in the function `iset_bfs_3_coloring`, also verifying its correctness by running `python3 -m ps5_tests 3`.

   (c) Compare the efficiency of Exhaustive-Search 3-coloring and the $O((n+m)\cdot 3^{n/3})$-time algorithm. Specifically, identify and write down the largest instance size n each algorithm is able to solve (within a time limit you specify, e.g. 1 second) and the smallest instance size n each algorithm is unable to solve (again within that same time limit).

   In addition to these numeric values, please provide a brief explanation of why these results make sense, based on your knowledge of both the algorithms' runtime and how each algorithm goes about finding a coloring. For this part, there is no need to go through every combination of parameters; feel free to give just the largest and smallest instances each algorithm can solve and speak generally as to why one algorithm performs better than the other. More instructions can be found in `ps5_experiments`.

It is pretty clear from running `ps5_experiments` that Exhaustive-Search 3-coloring algorithm is less efficient than the one we implemented. For the ring-test in the experiments, I had to reduce n to 15 for Exhaustive-Search to work in < 1 second, and when it iterated with $n = 18$ it timed out. For the other algorithm though, I could get it to $n = 1500$, but it would error (not timeout) at $n = 2000$.
The other experiment with clusters, Exhaustive-Search timed out with $n =$ 30 although some cluster numbers it succeeded higher, and others lowers. This was generally the threshold line though. On the other hand, the algorithm we implemented succeeded up to around $n = 50$ but failed first at $n = 72$ and higher.
These numbers make sense, because as n (nodes) increases, it follows that the number of coloring permutations increases. Although this likely means that the number of valid colorings increases, so too do the invalid colorings. Thus for exhaustive-search which goes through, potentially all in worse-case situation, coloring permutations, it will obviously take longer in general and thus when timeout maximum is low like 1 second it will generally time out for pretty low n values. The other implemented algorithm, is more powerful, but it is difficult to make it so efficient that even a very large n value does not time it out. This is because the

operations within the algorithm take time. Generally these are O(1) operations, like coloring a node or adding it to frontier etc., but as nodes increase so to do the number of O(1) operations thus making it more likely to time out.

4. (Reflection): Take one homework problem you have worked on this semester that you struggled to understand and solve, and explain how the struggle itself was valuable. In the context of this question, describe the struggle and how you overcame the struggle. You might also discuss whether struggling built aspects of character in you (e.g. endurance, self-confidence, competence to solve new problems) and how these virtues might benefit you in later ventures.

   Note: As with the previous psets, you may include your answer in your PDF submission, but the answer should ultimately go into a separate Gradescope submission form.

   Quick note on grading: Good responses are usually about a paragraph, with something like 7 or 8 sentences. Most importantly, please make sure your answer is specific to this class and your experiences in it. If your answer could have been edited lightly to apply to another class at Harvard, points will be taken off.

   One question I struggled with was question 3 on PS3. This was the problem about Ram simulation of Word-RAM. Even after the lectures on the topic, I really struggled to wrap my brain around, one, what was the purpose of even talking about these models, and two, how to follow the weird repetition of variables and semantics. Nonetheless, I found this question, to be the reason I feel much more confident about the topic now. I, frankly, after reading question, had no idea what to do when. I remember going by OH briefly after class one day, and asking a question about the problem, and it becoming much more clear to me. I definitely struggled through it at first, but once I understood what the question was asking it was much easier to come up with an answer. The struggle was valuable, because it shows, that in the end these are not that complex of topics, but there is definitely a barrier to understanding. You cannot understand the RAM model, by understanding the semantics. You only understand it when you realize the foundation and purpose of it as an abstract framework representation of the simple operations a computer breaks complex commands down into. These topics require work to understand, but they are generally not tricky or counter-intuitive. I think that is an important virtue to recognize every time I get stuck: break it into the fundamentals, understand it from the bottom up, so that I can apply the concepts to problems.

5. Once you're done with this problem set, please fill out this survey so that we can gather students' thoughts on the problem set, and the class in general. It's not required, but we really appreciate all responses!