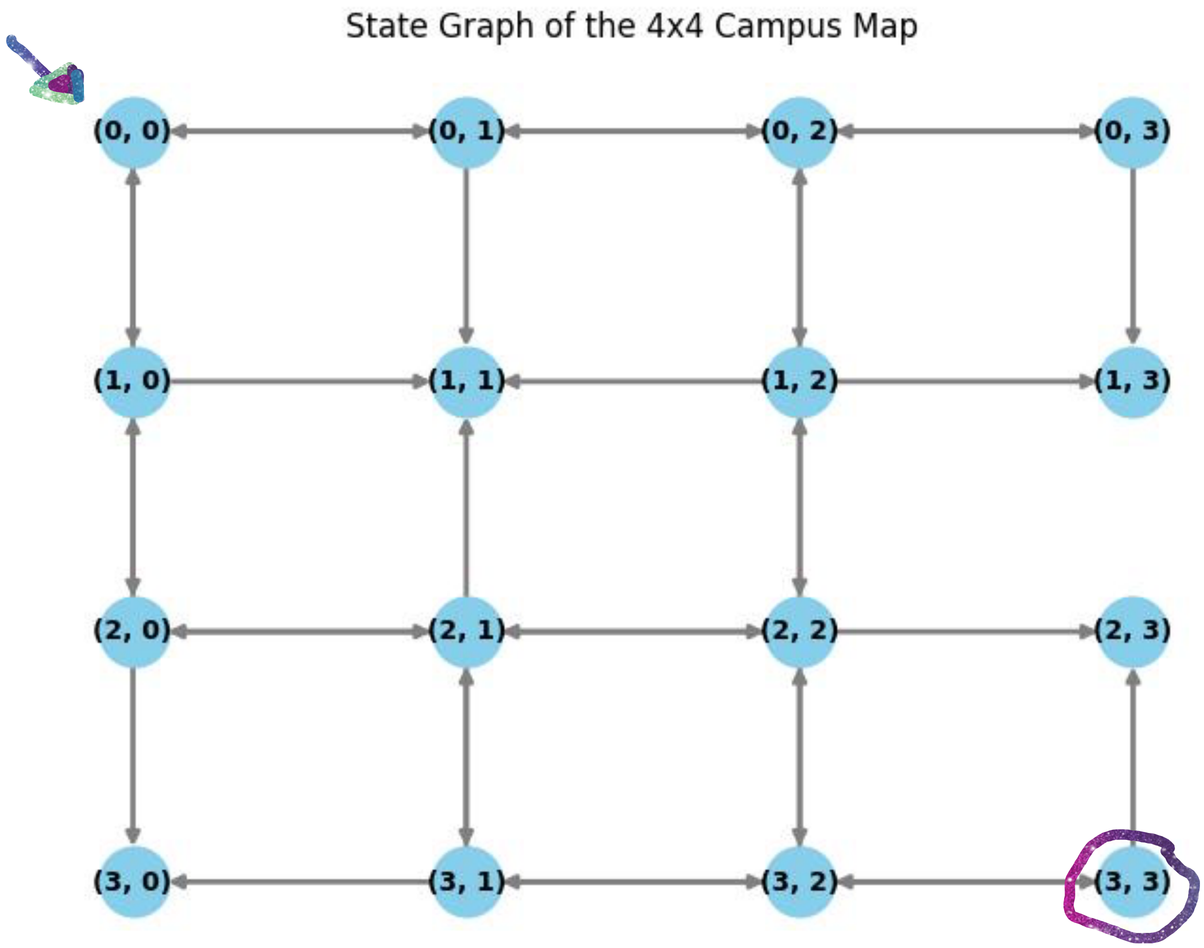
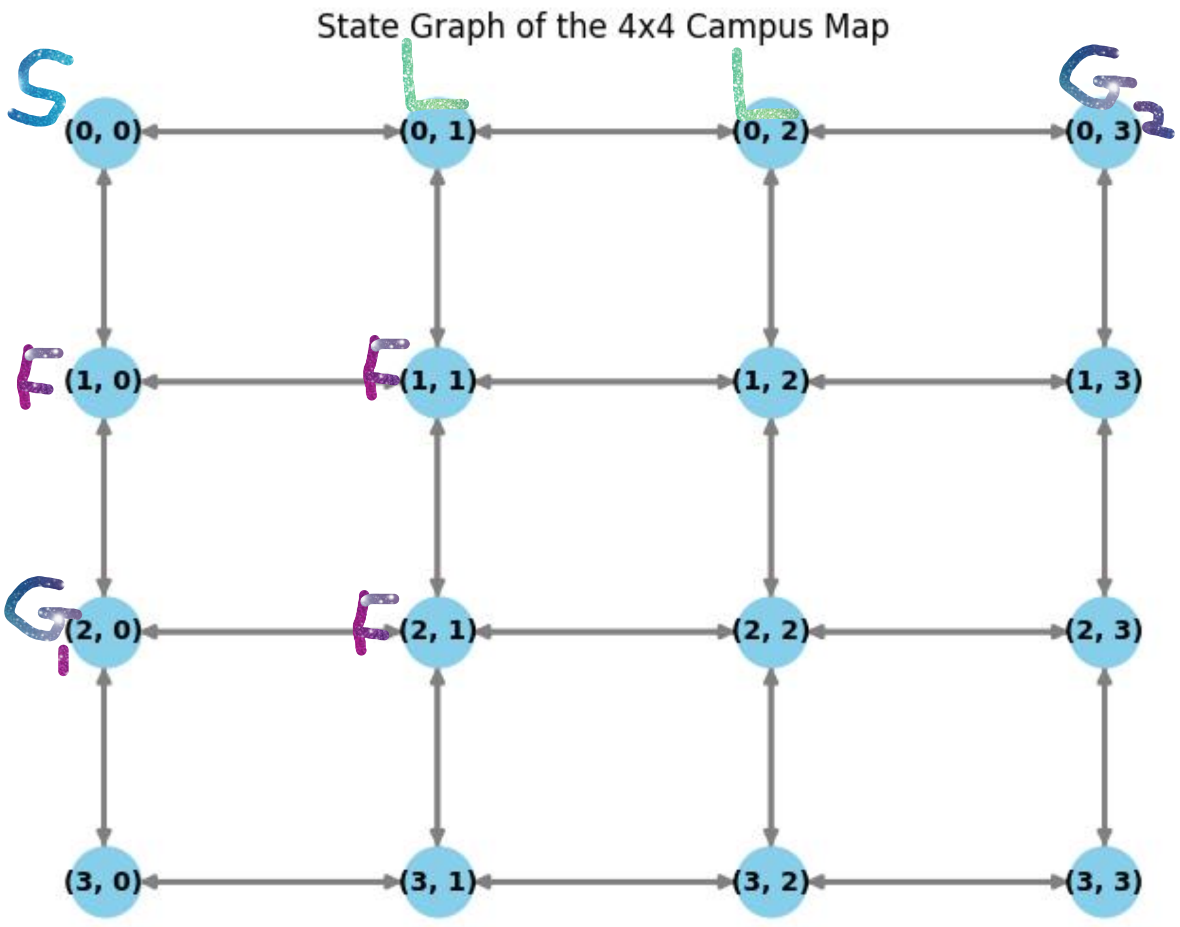
Q1:

1. CODE
2. S= {(cost$_i$, state$_i$, terminate$_i$)$_i$ | i є {0,..,63}}
   O={DOWN =0,RIGHT =1,UP = 2, LEFT = 3}
   I = {s$_0$}
   G = {s$_{63}$}
3. |S| = 64.
4. Domain(Down = 0) = {s$_i$ | i < 56}
5. Succ(s$_0$) = {s$_1$,s$_8$}
6. Yes, lets say s$_8$ is not a hole so s$_8$ is part of succ(s$_0$) and vise versa.
7. 4



8.
9. Because there are circles in the graph, for a random agent, the worst case scenario is entering such a circle and not leaving thus the number of actions for him is unbounded. As for the best case scenario it is lower bounded by the width and height of the map as the agent needs to perform atleast width times RIGHT and height times DOWN to reach the end goal (such a route exists in the 8x8 map so that is the the minimum number of actions in that map.)

10. We will show on a 4x4 map with 2 goals that the cheapest route from start tht finishes in a goal doesn't necessarily mean the goal is closest in the manhatten distance system:



as can be seen in this example, the cheapest route would be from S to G_2 (total cost of 3) and not G_1 (total cost of 11), while the Manhatten distance would be 3 to G_2, and only 2 to G_1.

Q2:

1) BFS finds all the connected vertices so if G is connected to S then BFS will find a route from S to G and if not the it will return there is no route. Since this is the requirement for an algorithm to solve the problem, we can say it is complete. As the costs can vary from node to node the shortest path is not guaranteed to be the fastest one so it is not "kavil".
2) The condition needs to be that the search-graph will not contain a circle as the tree-BFS doesn't not take into account the possibility of encountering a node twice, which is a possibility in a search graph.
3) We will create a function T:G->G' that will take graph G and for every node v and v' which are connected by an edge, we will delete this edge and replace it with a path from v to v' that has as many vertices as the cost of moving from v to v' is. Thus every path in G is also in G' and vise-versa but the length of the path in G' is equal to the cost of the path is G thus

running bfs-g on G' will give the shortest path available from completeness which is equal to the cheapest path in G if we take the vertices that are both in G and G'.

4) As we start the upper left corner and want to reach the lower right corner and we can move in left right up down fashion, bfs-g will open nodes along the contra-main diagonal resulting in Sum of (1 to n) plus sum of (n-1 to 1) = n^2 openings, but because in the last diagonal the node that is will be created is the Goal, the algorithm will stop in one of the last diagonal's nodes resulting in the other not opening the goal itself not opening so the amount that will actually be opened is n^2-2. the amount created will be n^2.

Q3)

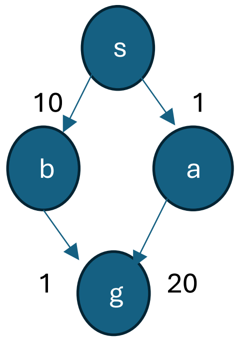
1) CODE
2) Yes. DFS-G holds 2 queues so no to enter a node twice eliminating the problem caused by circles in the graph and since the depth is therefor bounded it will return a path if it exists making it complete. It is not optimal.
3) DFS for trees is sensitive to circles thus will be susceptible to endless runs making it not complete and hence not optimal in the given graph.
4) 1. Assuming the order of opening is down, right, up, left, DFS-G will open N nodes to the down direction and N-1 nodes to the right direction (as there is no down in the last row of the board). Because the expanding of DFS-G creates nodes for all available connections before entering the next level, the number of Nodes that will be created is 3(N-2) (each node on the left wall that is not a corner has down,right,up) + 2*2 (the upper corner and the lower corner of the map have only down, right and up, right) + 3(N-2) (the lower boarder for the same reasons as the left border) = 2(3(N-2)+2*2) = 2(3N-2) = 6N-4. The Number of Nodes that will be opened is the route taken which is down all the way and right all the way which is n+n-1 = 2n-1 as DFS-G opens also the goal itself.
   2. Assuming the order of opening is down, right, up, left, DFS-G will open N nodes to the down direction and N-1 nodes to the right direction (as there is no down in the last row of the board). Because the DFS-G backtracking creates nodes lazily, thee number of Nodes that will be created is N+(N-1) = 2N-1. This is the same as the number that will be opened as DFS-G checks for solution only after opening. The backtracking doesn't really happen as the first try will be a valid route so the numbers turn out the same.
   The obvious advantage of the backtracking method is that it cuts on nodes that might not be needed to open thus freeing more space or more precisely not using unnecessary space.
5) 1. We will use the fact that d = d/2 + d/2 and so we will design the search problem to be $(S,O,I,G)_1$ like the original only that that G = {s | $d(s_0,s)$ = d/2} where the function d(x,y) returns the relative depth of y to x, and also we will design a new search problem $(S,O,I,G)_2$ where $I_2 = G_1$. This way Amy can run DFS-L on every I in I_1 (s_0) and get the group G_1 and run DFS-L again on every s in I_2 to get the shortest route from each s in I_2 to the original goal. Now Amy is guaranteed to have a route as if it is given that the depth of the end goal was d there has to be a node in depth d/2 that is on the path to d.
   2. No it hasn't changed.
   3. For DFS-L with L = d, the complexity of time and space is $O(b^d)$, $O(bd)$ respectively. For our algo the complexity of time and space is $O(b^{d/2})$ + h $O(b^{d/2})$, $O(bd/2)$ + h $O(bd/2)$ where h is the number of nodes that sit in the depth d/2 giving us a complexity of $O(hb^{d/2})$,

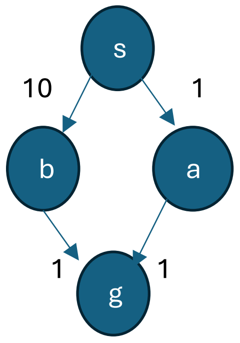
$O(hbd/2)$. We can bound h to be the number of nodes in the d/2 level thus we get that $h < b^{d/2}$ and thus our algorithm is guaranteed to be as fast as DFS-L regular d, if not faster.
4. Lets assume there is only one node in depth d/2 that is passable, our new algorithm will be both faster and is expected to be more space efficient than the regular than the DFS-L for L =d as given by the analysis above. given $h = b^{d/2}$ we exceed the space complexity of the regular DFS-L and given but are still expected to reach the result in the same time.

Q4

1) CODE
2) Yes the algorithm is complete and is opt as the cost function is positive (x > 1 = delta) so from class.
3) For cases when the cost is uniform for each step the UCS and the BFS-G algorithm will return the same paths as UCS works on the search frontier in the order of lowest cost -> first revealed so if the lower cost constraint is the same it will work on the first revealed which is the same for a BFS-G run.
4) in example a UCS will open a before b as the cost of a is lower than b, then it will create g realize it is the goal and return 21 even though it is not the cheapest route.

a.





In example b the algorithm will return the cheapest route even though it is flawed.

Q5

1) 1. h is not acceptable. Suppose h_1 and h_2 are both the optimal heuristic then they are both acceptable as required and $h > h_*$ thus h is not acceptable.
2. h is acceptable. $0 <= (h_1 + h_2)/2 <= (h_* + h_*)/2 = h_*$.
2) 1. lets say there is a s in S such that s' is in succ(s) so that $h_2(s) - h_2(s') = cost(s,s')$ and $h_1(s) - h_1(s') = cost(s,s')$ then $h_1(s)+h_2(s) – (h_1(s')+h_2(s')) = 2cost(s,s')$ so it is not עקבית.
2. $(h_1(s)+h_2(s))/2 – (h_1(s')+h_2(s'))/2 = (h_1(s) - h_1(s'))/2 + (h_2(s) - h_2(s'))/2 <= cost(s,s')/2 + cost(s,s')/2 = cost(s,s')$ so it is עקבית.
3) BLANK QUESTION
4) $h_{campus} >= 0$ as the Manhattan norm is bigger or equal to 0. If the minimal cost path to g goes through a portal then $h_{campus} <= h_*$ as h_{campus} will result in the cost of the portal and only it, elsewise, lets assume there are no portals in the route, and the route is cheaper then a portal (because if not then again h_{campus} is bouded by it and will be smaller than h_*) the Manhattan counts the minimum number of steps needed to reach the goal as we cant move

in crosses, as the cost of each step is lower bound by 1 it is then can be proven by induction that the Manhattan norm lower bounds the cost of the path thus h_{campus} <= h_* for every s.

5) if the step taken from s to s' is left or up $h_{campus}(s) - h_{campus}(s') = -1$ or 0 as we add 1 to the Manhatten norm of the node to the goal either on the x axis or the y axis (if it is minimized by C then they are both C and thus 0 in this case they don't enter a portal). if the step taken from s to s' is right or down $h_{campus}(s) - h_{campus}(s') = 1$ or 0 for the same reasons only this time we decrease the Manhatten norm. in the case we enter a portal from s to s' then, $h_{campus}(s) - h_{campus}(s') = C - h_{campus}(s')$. $0 <= h_{campus}(s') <= C$ thus $0 <= C - h_{campus}(s') <= C$ thus $h_{campus}(s) - h_{campus}(s') <= Cost(s,s')$. since $1<=Cost(s,s')$ holds for every case we have shown that $h_{campus}(s) - h_{campus}(s') <= Cost(s,s')$.

Q6

1) yes the graph is finite and connected so Greedy Best First Search is complete. No, as the campus norm is not directly tied to the cost of the path.
2) Both are from the Best First Search family and thus share many properties and thus by using the same heuristics we advance in the same manner. The Advantage of the beam over the greedy is in it time and space complexity as it limits the number of opened nodes held by the algorithm, the disadvantage is that it limits the search frontier and thus may result in an incomplete algorithm.

Q7

1) CODE
2) Y = 2x is a linear one to one function thus for every f'(s) we can translate it back to f(s) and also it saves the property of size i.e if f(s) < f(s') then f'(s) = 2f(s) < 2f(s') = f'(s') so the algorithm will open exactly the same nodes and return the same path under f'(s) that it does under f(s).
3) CODE
4) Algorithm IDA* uses DFS-L in each iteration resulting in an optimized memory usage. Although it does so it does not store previous knowledge on nodes it visited and thus it can run several times on the same nodes which is not preferable. We would use IDA* on problems where the search field is to big to be stored. Notice that some researchers created A*+IDA* that runs A* until memory exhaustion and then switches to IDA* to get the best of both worlds. (A*+IDA*: A Simple Hybrid Search Algorithm Zhaoxing Bu and Richard E. Korf)
5) Algorithm A*-epsilon allows us to add preferences to our search such as time complexity and more this allows us to get sub-optimal results up to an upper limit in better time that A* does. We would choose such an algorithm if our system is required to make decisions fast because we want the optimal solution but we care more for the time it takes to achieve it. A* ensures we get the optimal solution always so we would use such an algorithm when time is of limited value compared to the correctness of the decision given.

Q8

Explanation of Results:

DFS-G (Depth-First Search with Graph Search):

- Cost: Generally higher than UCS and A*. DFS-G often does not find the shortest path due to its nature of exploring deep first.

- Expanded Nodes: Fewer than UCS but more than A* in some cases. Inefficient exploration can lead to unnecessary expansions.

- Path and Cost: Does not guarantee the shortest path, leading to higher costs.

UCS (Uniform Cost Search):

- Cost: Consistently optimal and lower than DFS-G. UCS finds the shortest path by exploring nodes based on the lowest cumulative cost.

- Expanded Nodes: More than A*. UCS explores nodes exhaustively without heuristic guidance.

- Path and Cost: Guarantees the optimal path and cost.

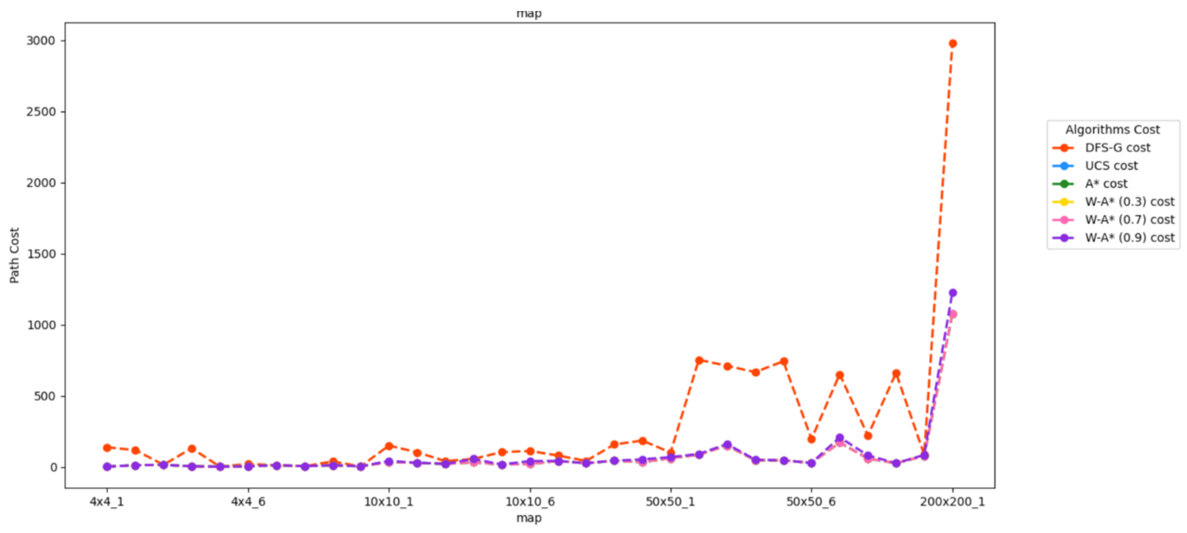
A* Search:

- Cost: Optimal and similar to UCS but with fewer expanded nodes.

- Expanded Nodes: Fewer than UCS due to heuristic guidance, making the search more efficient.

- Path and Cost: Guarantees the optimal path and cost efficiently.

Weighted A* Search:



- Cost and Expanded Nodes: Higher weights lead to fewer expanded nodes but slightly higher costs. The balance depends on the heuristic weight.

- Weights 0.3, 0.7, 0.9: Higher weights reduce the number of expanded nodes but may increase the path cost slightly.

Expectations:

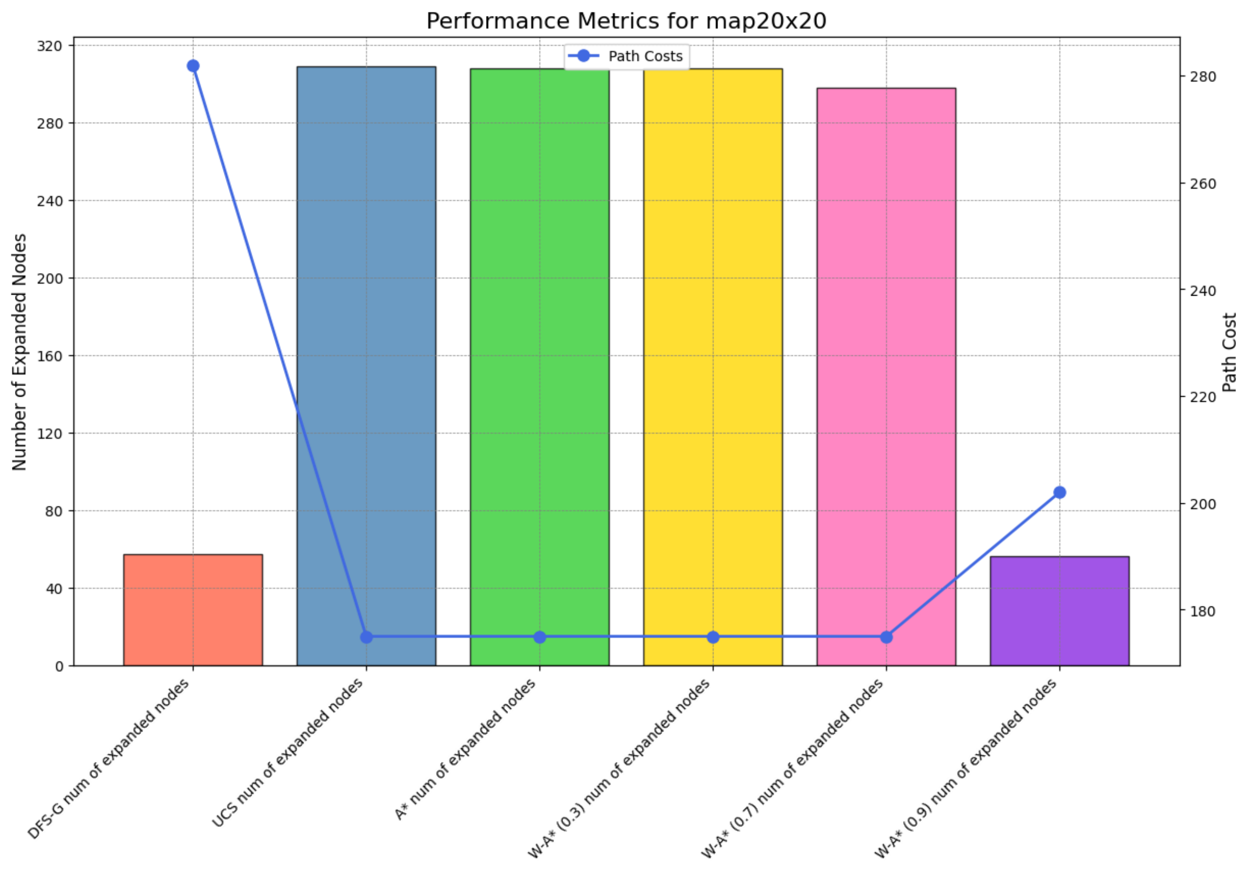
The results match expectations:

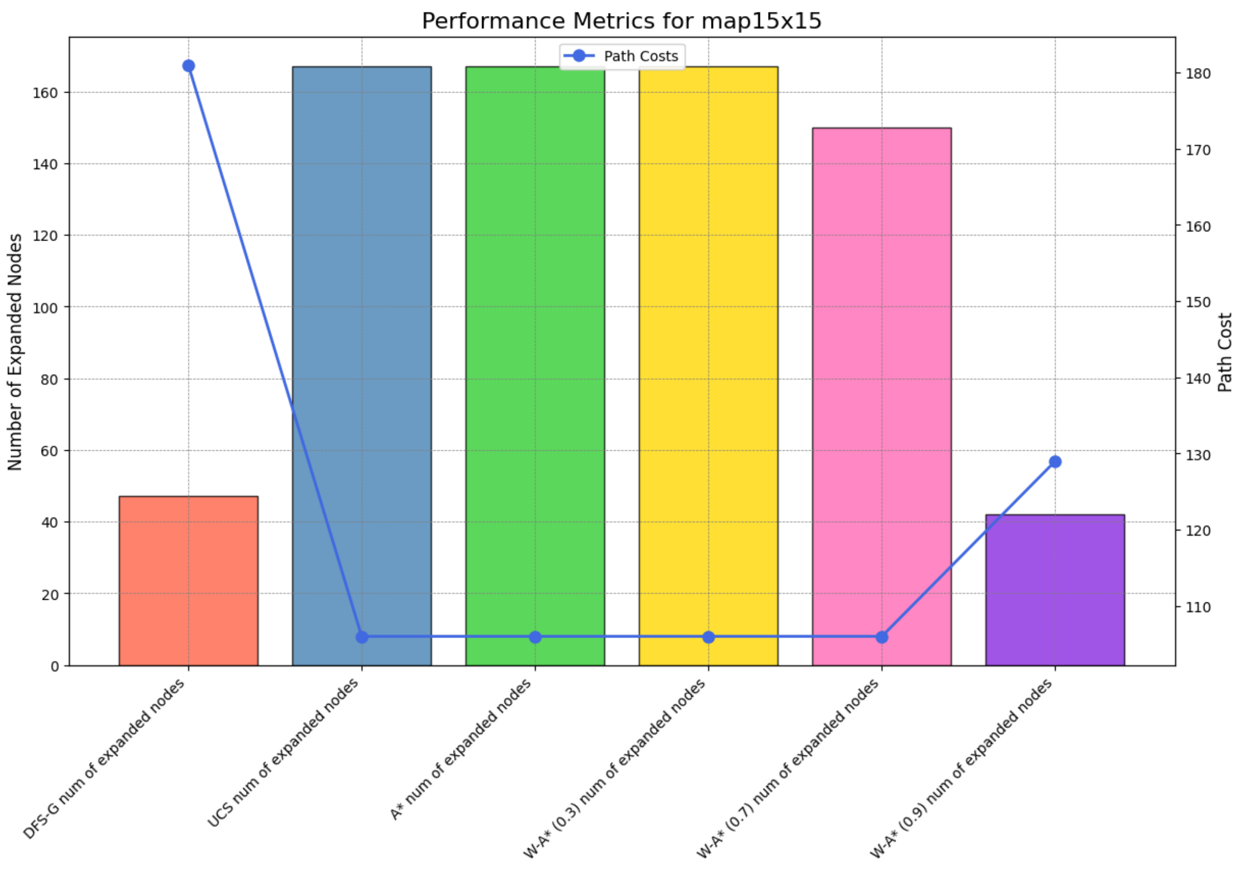
- DFS-G finds suboptimal paths with higher node expansions.

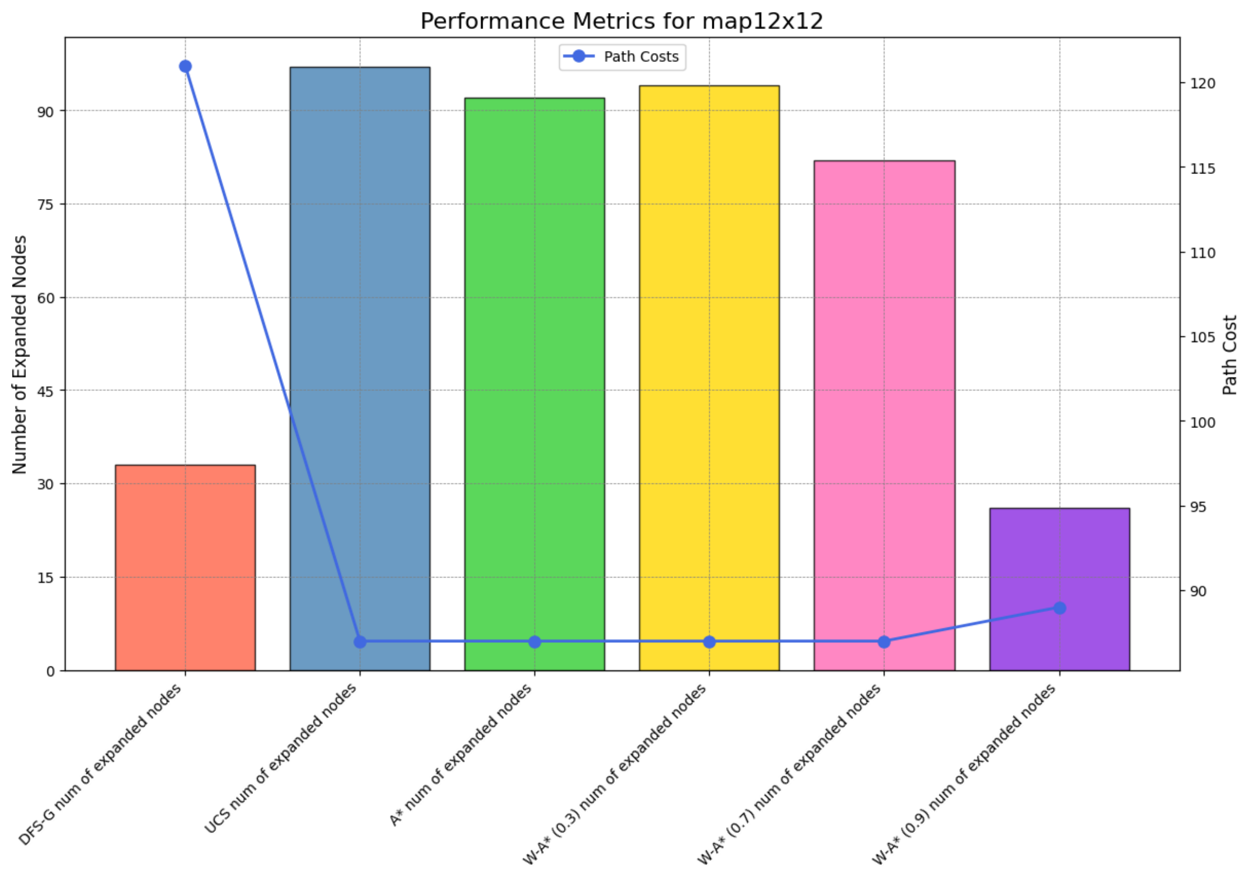
- UCS finds optimal paths but expands many nodes.

- A* finds optimal paths with fewer node expansions due to the heuristic.

- Weighted A* balances path cost and node expansions based on the weight.


Performance Metrics for map20x20
Path Costs
Number of Expanded Nodes
Path Cost
0
40
80
120
160
200
240
280
320
180
200
220
240
260
280
DFS-G num of expanded nodes
UCS num of expanded nodes
A* num of expanded nodes
W-A* (0.3) num of expanded nodes
W-A* (0.7) num of expanded nodes
W-A* (0.9) num of expanded nodes


Performance Metrics for map15x15
Path Costs
Number of Expanded Nodes
Path Cost
0
20
40
60
80
100
120
140
160
110
120
130
140
150
160
170
180
DFS-G num of expanded nodes
UCS num of expanded nodes
A* num of expanded nodes
W-A* (0.3) num of expanded nodes
W-A* (0.7) num of expanded nodes
W-A* (0.9) num of expanded nodes



Effect of a More Informed Heuristic:

With a more informed heuristic:

- A* and Weighted A* would expand even fewer nodes.

- Path costs might stay the same if the heuristic is admissible and consistent.

Conclusion:

- DFS-G: Inefficient with higher costs and more expansions.

- UCS: Reliable for optimal paths but expands many nodes.

- A*: Efficient with fewer node expansions and optimal paths.

- Weighted A*: Flexible balance between path cost and efficiency based on the weight.

A more informed heuristic would improve the efficiency of A* and Weighted A*, reducing the number of expanded nodes while maintaining optimal paths.

Q9

1) S= {(w_{i1} w_{i2} w_{i3} w_{i4} w_{i5} w_{i6} w_{i7} w_{i8} w_{i9}…w_{in} | w_{ij} є {chines words in the document} w_{ij} != w_{ik} for all j and k. i є {0….n!}}. In simple words S is the group of permutations of the words in the document where each word appears once.
2) nPn = n! where n is the number of words in the file.
3) Yes, because every word appears once, if a word is not in its place there will always be an option to switch it with the word that is in its place giving us a utility score of +1 compared to the last step resulting in every step leading to another until we solve the problem.
4) 1. Yes, sidestep adds the option of going from utility u to utility u in the next step but as we shown before there will be a u+1 so the algo will go in that direction.
   2. No, as they will act the same they will return the same results.
5) Yes because in each step we are guaranteed to improve our utility score and since our utility score is bounded by n we are assumed to reach it.