Problem 1: Best First (using f=h), Uniform Cost (using f=g), A* (f = g+h)

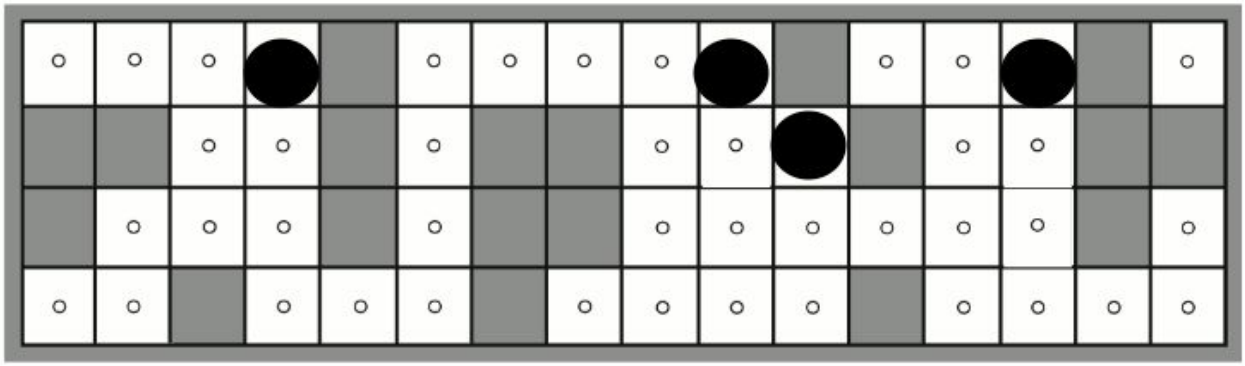
- Best first search
  - **Goal State Reached:** G1
  - **States Popped off open:** S, B, F, G1
  - This is because the path will follow the heuristic cost of the nodes. It reaches the starting node and pops it off the open set. S has two child nodes B and C with a heuristic cost of 3 and 5 respectively. The search will go to B as it has the lower h cost adding it to the closed set. Adding C to the open set. B then has two child nodes E and F with cost of 3 and 1 respectively. The search will go to F since it has the lower cost adding E to the open set and F to the closed set. From there F has one child with a cost of 0 which is the G1 goal state.
- Uniform Cost Search
  - **Goal State Reached:** G2
  - **States Popped off open:** S, B, E, D, C, F, A, G2
  - This is because the path will follow the edge cost of the nodes. It reaches the starting node and pops it off the open set. S has two children B and C with edge costs of 1 and 5 respectively. The search will then go to B as it has the lowest edge cost and pop it off from the open set. From there B has two child nodes F and E which have a total cost of 5 and 3. The search will then go to E as it has the lowest total cost. It will then pop E off the open set. E has two child nodes D and G2 to a total cost of 4 and 7. It then pops D off open and D has two children A and G2 with cost of 6 and 6. Then looking at our open list we then have C being the cheapest cost so then we pop that off and it has one child node of G1 with a cost of 13. The next lower cost in open would then be F with a cost of 5. It is then popped off and has one child node of g1 with a cost of 7, updating the cost of g1. Then the next lowest cost is A with a cost of 6 is popped off which only has the child node of the starting node. This leaves G1 and G2 left in the open set and the cheapest cost is G2 with a total cost of 6.
- A* search
  - **Goal State Reached:** G2
  - **States Popped off open:** S, B, E, D,G2
  - This is because the path will follow the edge costs and the heuristics together. The search will start off with S and pop it off of open. S has two children nodes, B and C with a cost of 4 and 10. It will then pop off B as it has the lower cost. B is then popped off the open set. B then has two nodes E and F with both of them having a cost of 5. The search will then choose E as it comes before F as defined in the question. E is then popped of the open set and it has two child nodes G2 and D with a cost of 4 and 3. D is then popped off the open set as it has the lower cost of 3. D has two child nodes A and G2 with a cost of 11 and 2. G2 then has the lowest cost in the set and is then popped off.

Problem 2: A* Termination and Goal States

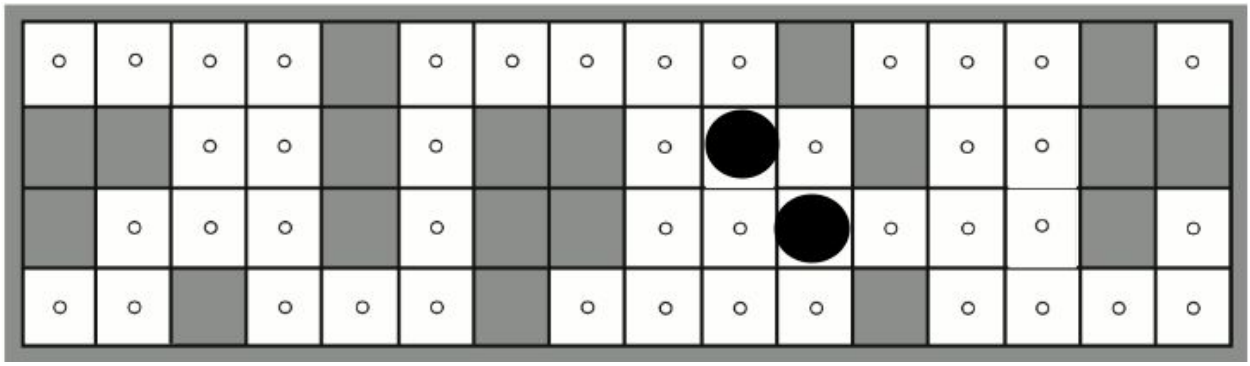
- **Why not Terminate A* as soon as a goal node is expanded to the frontier list?**
    - The reason why you wouldn't terminate A* as soon as the goal node or a goal node is expanded and added to the frontier list is because A* would lose its optimality. This would be caused by a node that has less total cost to the goal state to be ignored removing the optimal solution that A* tries to achieve.
- **Show that doing so would violate an Important property of A* and explain your example.**
    - In this example we can see that when we expand S there are two nodes A and G A has a cost of 1 and a heuristic cost of 1. G is the goal and has a cost of 5 and a heuristic cost of 0. If we just popped G off the frontier the cost to reach the goal would be 5. Yet if we followed the normal rules of A* we would see that the A has a child node of of the goal with a total cost of 3. As you can see by not popping off the goal state as soon as it is expanded to the frontier A* will not keep its optimality, which is a key property of A*.

Problem 3: Consider the world and an agent that can sense the dark grey boundaries

- **Given an initial unknown initial state and precept = {N, E} mark all possible states**



- **After taking the action to move South into a new state**



Problem 4: True and False Questions

- **Depth first search is guaranteed to find the goal**
  - False, DFS is not guaranteed to find the goal as it is not complete
- **All nodes with the same cost as the goal node are expanded before A* terminates**
  - False, Nodes with the same cost are sometimes expanded before A* terminates
- **An admissible heuristic always over estimates the cost to goal**
  - False, an admissible heuristic is one that never over estimates the cost to the goal
- **Uniform Cost search is equivalent to A* with a heuristic = 0**
  - True, This is true because if the heuristic cost of A* is zero then A* will search based only on the edge costs. Uniform cost only searches based on the edge costs thus they would behave the same way if the heuristic cost is equal to zero.
- **If best-first search finds some goal node in finite time, then depth first search will also find a goal node in finite time.**
  - True, this is true because both best-first and and depth first are incomplete in finite space. Since both best-first and depth-first have the same completeness and they share the same complexity then if best-first finds a goal node then depth-first will also find a goal node in finite time
- **If all transactions (edge costs in graph) have the same cost, then uniform cost and breadth first search will give the exact same solution**
  - True, this is because uniform cost search relies on the edge cost of the nodes to decide where to go to next. If the node cost are the same then the search will pop off the first thing in the queue. This is the exact same method that breadth-first search use and thus will act the same way.

Problem 5 Short answer

- **Which of the following is simular to simulated annealing**
  - Iterative deepening, this is because iterative deepening keeps trying and trying different limits until it reaches the last limit. This is similar to the gradient decent method used by simulated annealing as it keeps trying to move randomly if it gets caught in a local minimum.
- **Assume someone gives you two heuristics h1(node) and h2(node), both of which are admissible heuristics. Explain your answer.**
  - The heuristic that would be better would be h3(node) = max(h1(node) , h2(node)). This one is chosen because both heuristics are admissible, meaning that they are never overestimated. This means that h1 and h2 are going to be less than or equal to h* (h* being the most accurate heuristic). Meaning that if we choose the maximum out of those two nodes then it will make h3 as close to h* as possible.
- **The primary purpose of the explored list in search algorithm shown in Figure 3.7 is to produce the path from the goal node found back to the initial state.**
  - False, this is because the explored list holds all the nodes and the corresponding costs to see if their is a cheaper cost of expansion. Take for example the

explored list for the Uniform cost search in question 1. The explored list is not the path but it is rather all of the nodes that are expanded during the search.

Problem 6: State based three letter word search problem

- **What would be a state**
    - A possible way to define a state for this problem would be any combination of a 3 letter word defined in the dictionary available to the program. We have a starting word CAT and from there we have a some three letter words that can be generated by moving one letter at a time making the child nodes. As an example we could change CAT into BAT and CAB each being child nodes of CAT and being valid states.
- **What would be an edge (transition)**
    - A edge transition would be how many times the word was manipulated by one letter each to reach that valid state. For example the edge cost to get from CAT to BET would be two as two letter were changed to get to that state while also still being three letter words. The edge costs would not indicate the optimal path to reach the goal state but rather how many letters have been changed to get to the current state of that node.
- **What would be a good heuristic function? It is fine to assume your h function knows the words that are in the inital state and the goal state.**
    - A good heuristic function would be one that is able to calculate how many words can be generated that have the goal letters in the correct position and how many letters are also in the correct position at the same time. For example CAT has no letters in the correct position and (to my limited knowledge) cannot generate any words with the correct letters in them so the cost of the starting node would be 6. Then for example BAT can generate a word with one letter in the correct position (BAG) so it's h cost would 4 as it can generate a word with a letter in the correct position. DAB would have a heuristic cost of 5 because it has one letter in the correct position but is not able to generate a child node with another letter in the correct position.
- **Would your answer to part 3 be an admissible heuristic? Explain**
    - No as the heuristic is able to overestimate how far away the node is from being solved. Take for example DAB, dab would have a heuristic cost of 5 but from DAB we can get to DAT. From DAT we can transition to DOT and from there we can transition to DOG. So even though DAB cannot make a child with another letter in place one of its children is able to get to the correct solution in 2 steps meaning that the step it takes to get to the goal from DAB would only be 3. This would indicate that our heuristic is not admissible.

Programming Assignment 1:

To make world continuous:

- Change def Actions(): so that whenever the x or y coordinate reach a boundary on the map but not an object it will shift to the corresponding x or y coordinate. It would do this by switching the min and the max value in the x or y coordinate provided there is no object blocking it there. I.E in the python code the starting coordinate is zero for x and y and the max distance is 255 so if an y coordinate gets to 255 and tries to go to 256 it would then make the y coordinate 0 as it would then loop back to the minimum. The same would happen with the x coordinate as well.

Effect of different heuristics:

- Manhattan vs. Euclidian 2 heuristics
    - Manhattan is calculated by the distance the x and y values are away from the goal. Euclidian 2 is calculated by the the distance x and y values are away from the goal giving us dx and dy. From dx and dy we get the euclidian 2 distance which is calculated by 2 * sqrt (dx^2 + dy^2). The main difference between these two heuristics will be that the manhattan distance will more admissible as it will have less of a chance to over estimate. Euclidian 2 heuristic has a higher chance chance to over estimate since it takes the square root of the distances and multiplies it by two. On average this will result in the manhattan distance having more expansions and less nodes in the queue. The Euclidian distance will have less expansions but more nodes in the queue.

Effects of Different Grid Sizes:

- ½ size of the original grid
    - This will cause the search function to run slower as it will need to expand more nodes. This is caused by the grid size determining the amount of nodes generated in the state space. The smoothness of this line will be very smooth as it is able to expand more nodes it is able to make a path that does not jump around and makes it more smooth. It will also allow for the search to be able to reach smaller paths around the obstacles that if the grid size the search would not be able to reach.
- 2 times the original grid size
    - This will cause the search function to run faster as it will need to expands significantly less nodes. This is caused by the grid size determining the amount of nodes generated in the state space. This will make the path much more jagged as the nodes are bigger making it jump further when drawing the path. It will also make it significantly harder to get in narrow path or even impossible in some cases as the nodes might be so big that they would not be able to be generated as they would encompass some of the obstacles. If the goal state in this robot

arm example is to get into a very narrow passage then it will fail to ever get into that narrow passage.

How might you bias A* away from the obstacles

You can bias A* away from the obstacles by modifying the heuristic cost of the nodes next to the obstacles. When the nodes are generated and the obstacles are placed into the c-space then any nodes next to the c-space obstacles will have their heuristic cost inflated to bias A* away from expanding to them. This can be done by when A* is generating the nodes heuristic cost it will see it a obstacle is in the path distance around that x and y coordinate. If the obstacle is close to the node and a known obstacle is in the path then the heuristic can be inflated to make A* bias away from nodes close to the obstacle.

Generating solutions with RRT implementations

- **Defining a good Sampling function for the space**
  - A good sampling function for the RRT implementation would be a random sampling function. This would make the c-space for our problem dense with probability 1. This will mean that the algorithms used are probabilistically complete which means that the probability given enough nodes of finding an existing solution converges to one.
- **A single RRT using extend**
  - a single RRT using extend would branch out based on the sampling function. It would rapidly explore the c-space and given enough time would find the goal state given enough time. This would take a significant amount of time as the nodes would grow but in random directions.
- **A bidirectional RRT using extend and connect**
  - There would be two RRT's one at the starting node and one at the goal node. One of the best implementations would be to have the extend RRT in an open area and the connect RRT in a more enclosed area surrounded by obstacles. This would allow the extend to rapidly explore the c-space open and the connect to fill in the area surrounded by obstacles. This will result in the connect trying to connect to the extend and eventually connect to the extend RRT.
- **A bidirectional RRT using two connects**
  - using two RRTs one at the starting node and one at the goal node. They would rapidly expand outwards and then try to connect to each other. This out of the three given approaches would result in the quickest results as the trees would expand towards each other at a converging rate. meaning that the chance of the branches escaping a narrow passageway is greatly increased.

Programming Assignment 2:

- **Hill Climbing Solution:**
    - Hill climbing solution imposes its own problems but so far is the quickest method out of all three implementations. Most of the time it will be stuck in a local minimum and will need to be taken out of that minimum for the program to find a solution. If we did not have the check for the local minimum and a way to escape said local minimum the program would fail almost constantly.
- **Simulated Annealing**
    - Simulated annealing proved to be a good solution but at the same time would still run into constraints based on the number of expansions it took and sometimes based on how many expansions allowed would not find the correct answer. For the testing conditions we have the given temperature is 4000 and the cooling factor used was .99 a general good expansion limit seems to be around 30,000 which stops any calculations that would last too long but would also allow for smaller faster expansions. Playing around with the temperature and cooling factor some of the lower results that still returned seemed to be around 4000 and .99 for the temperature and cooling factor.
- **Genetic Algorithm**
    - The genetic algorithm seemed to take the most expansions but also at the same time always returned a correct answer that didn't need help getting out of a local minimum or running out of expansions. The population size that we used was 7 and the mutation probability that seemed to work and give faster times was 0.2

Expansion Cost			
	All on first row	Diagonal across	mixed up
Hill Climbing	18 (Needing to jump out of a local min)	17 (Needs to jump out of local min)	2
Simulated Annealing	2339	5205	2015
Genetic Algorithm	10408	23612	4504

Below are the outputs of the various programs

Hillclimbing Output:

All in the same row
running hill climbing with array: (0,0),(1,0),(2,0),(3,0),(4,0),(5,0),(6,0)
local max reached. retrying
(0,1),(1,5),(2,4),(3,5),(4,6),(5,1),(6,4)
solution: (0,3),(1,0),(2,2),(3,5),(4,1),(5,6),(6,4)
count: 18

Diagonal
running hill climbing with array: (0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6)
local max reached. retrying
(0,4),(1,5),(2,6),(3,0),(4,5),(5,1),(6,0)
local max reached. retrying
(0,4),(1,1),(2,2),(3,5),(4,1),(5,5),(6,0)
solution: (0,3),(1,6),(2,2),(3,5),(4,1),(5,4),(6,0)
count: 17

Mixed
running hill climbing with array: (0,0),(1,6),(2,1),(3,5),(4,2),(5,4),(6,3)
solution: (0,4),(1,6),(2,1),(3,5),(4,2),(5,0),(6,3)
count: 2

Simulated Annealing Output:
All in the same row
Starting with array: (0,0),(1,0),(2,0),(3,0),(4,0),(5,0),(6,0)
solution: (0,3),(1,5),(2,0),(3,2),(4,4),(5,6),(6,1)
2339

All in the diagonal
(0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6)
solution: (0,4),(1,6),(2,1),(3,3),(4,5),(5,0),(6,2)
5205

Mixed Up
(0,0),(1,6),(2,1),(3,5),(4,2),(5,4),(6,3)
solution: (0,2),(1,0),(2,5),(3,3),(4,1),(5,6),(6,4)
2015

Genetic Algorithim:

All in the same row
Starting with array: (0,0),(1,0),(2,0),(3,0),(4,0),(5,0),(6,0)
solution: (0,4),(1,6),(2,1),(3,5),(4,2),(5,0),(6,3)
10408

All in the diagonal
Starting with array: (0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6)
solution: (0,5),(1,0),(2,2),(3,4),(4,6),(5,1),(6,3)
23612

Mixed Up
Starting with array: (0,0),(1,6),(2,1),(3,5),(4,2),(5,4),(6,3)
solution: (0,4),(1,6),(2,1),(3,5),(4,2),(5,0),(6,3)
4504

These outputs are not surprising based on the randomness of how these algorithms function. The algorithms also behave similarly based on what we have learned in class. Hill climbing always returns good results as long as it doesn't get to a local minimum or maximum. If there is not a catch with Hillclimbing it will fail to find a correct answer. Simulating annealing will randomise the layout of the board and slow down the pieces it moves based on the temperature and the cooling ratio. It will continue to run until it finds the solution or the temperature hits zero or too many nodes are expanded. Finally the genetic algorithm behaves like we expected it to. It uses the principles of simulated annealing to generate a population by reproducing and having crossover and if the mutation probability is hit then a genetic mutation will occur within the population shaking things up further. Hillclimbing expands the least amount of nodes but at the same time fails constantly, Simulated Annealing expands more nodes but is sometimes able to find a solution based on the temperature and cooling ratio, finally Genetic Algorithm uses some of the principles found in Simulated Annealing (mainly handling local minimums and maximums) and then creates children nodes based on the nodes in the population with a chance of mutation occurring, causing it to expand more nodes making it cover more of the map of the solution space.