Assignment 1

Introduction to Artificial Intelligence

1. **Environments and Algorithms**
   1. **Generating a Maze: We generate the maze as follows,**
      1. Create a maze object with parameters dimension and probability
      2. Generate a 2-D array with dimension X dimension and initialize each entry as 1, depicting there are no obstacles.
      3. For the maze with dimension **d** and probability **p**, generate the number of obstacles **n\_o** using the following relation,

n\_o = (d X d) X p

* + 1. Generate random points (y , x) in the 2D array and ensure that none of the points are repeated. Thus, we ensure that each maze with dimension, d and probability, p has exactly n\_o obstacles.

For instance, for dimension = 100 & probability = 0.1, I generate 1000 obstacles. Then we randomly generate 1000 points and assign them a value of -1. We ensure that if the number of obstacles are 1000, we generate 1000 different points randomly using rand().

For ease of understanding and depiction, the following figure shows the maze for dimension = 10 and probability 0.2, which leads to 20 obstacles depicted as X.

A close up of a logo

Description automatically generated  
**Fig1. Maze depiction for dimension = 10 and probability = 0.2**

1. **Analysis and Comparison**

**Ques 2.1.** Find a map size (dim) that is large enough to produce maps that require some work to solve, but small enough that you can run each algorithm multiple times for a range of possible p values. How did you pick a dim?

**Solution.**

A\*using manhattan distance tends to give us the shortest path by travelling along the edges of the maze whereas using the Euclidean distance as the heuristic the algorithm tends to find the shortest path by moving diagonally from start towards the goal.

**For DFS, can you improve the performance of the algorithm by choosing what order to load the neighboring rooms into the fringe? What neighbors are `worth' looking at before others? Be thorough and justify yourself.**

|  |  |  |
| --- | --- | --- |
|  | 4 |  |
| 3 | Current state | 2 |
|  | 1 |  |

Note:The above figure considers the start state to be at the top left and the goal state to be at the bottom right.

Consider the case when we are at the current state and we have to decide what neighbors to load on the stack (fringe). We should prefer putting state 1 at the top of the fringe because it brings us closer to our goal. Then next in preference (if state 1 is not free) should be state 2 because it brings us closer to the goal in x direction. If state 2 is not free then the next state in preference should be state 3 as it is on the same level as the current state. Lastly, if no option is available then we should put state 4 at the top of the stack.

Thus in summary the neighbors worth looking (that should be put at the top of the stack) could be those which are closer to the goal from the start state in terms of either their x-coordinate or y-coordinate.

1. **Generating Hard Mazes**

**What local search algorithm did you pick, and why? How are you representing the maze/environment to be able to utilize this search algorithm? What design choices did you have to make to make to apply this search algorithm to this problem?**

We have implemented and analyzed the following local search algorithms:-

1. Hill Climbing Algorithm
2. Simulated Annealing Algorithm
3. Genetic Algorithm

The maze environment is converted into an 2D vector matrix (dim\*dim) filled with 0’s (blocked cell) and 1’s (open cell). The 2D vector matrix is part of a maze class which has the maze, its associated functions as well as variables for storing the values associated to the metrics (length of shortest path, total number of nodes expanded and maximum fringe size). We also have a point class for storing the co-ordinate points of the maze. The number of blocked cells is given by (dim\*dim\*p). While generating the maze we have taken care that the start (0,0) and goal (dim-1,dim-1) states are never blocked also, we ensure that once a cell is blocked, it cannot be blocked again.

**Hill Climbing Algorithm**

The input to this algorithm is a solvable maze with dimension (dim) and probability of obstacle (p) selected by user.

From this initial maze we construct the neighbor mazes by adding additional one and two obstacles. For adding one obstacle there are (dim \* dim – number of obstacles – 2{start and goal states}) locations and for adding two obstacles there are half as many as those available for adding one obstacle.

After generation of each neighbor maze we check whether the neighbor maze is solvable or not. If it is solvable we calculate the length of the shortest path for that maze and using this value as a priority we store the maze in a priority queue.

Once all the neighbor mazes are generated and stored in the priority queue then we select the best neighbor (top of the priority queue) to be the initial state if its priority value is greater than our current initial state. This process is repeated until a termination condition is encountered.

**Simulated Annealing Algorithm**

The input to this algorithm is a solvable maze with dimension (dim) and probability of obstacle (p) selected by user.

From this initial maze we construct the neighbor mazes by adding as well as removing additional one and two obstacles. For adding one obstacle there are (dim \* dim – number of obstacles – 2{start and goal states}) locations and for adding two obstacles there are half as many as those available for adding one obstacle.

After generation of each neighbor maze we check whether the neighbor maze is solvable or not. If it is solvable we calculate the length of the shortest path for that maze and using this value as a priority we store the maze in a priority queue.

Once all the neighbor mazes are generated and stored in the priority queue then we select the best neighbor (top of the priority queue) to be the initial state if its priority value is greater than our current initial state. Otherwise we compare the output of our acceptance probability function with a random number generated between 0 to 1 and accept a neighbor with lower probability than the initial state (to become the initial state for the next iteration).

This process is repeated until a termination condition is encountered.

**Genetic Algorithm**

The input to this algorithm is a solvable maze with dimension (dim) and probability of obstacle (p) selected by user.

From this initial maze we construct the (N=10) children by adding additional one obstacle to the initial state at a random open cell.

After generation of each child maze we check whether the child maze is solvable or not. If it is solvable we calculate the length of the shortest path for that maze and using this value as a priority we store the maze in a priority queue of children maze as well as a priority queue of children and parents maze.

Then we create the next generation (two children) by taking two child matrices at a time from the children maze priority queue. We take 60% of the first matrix ( (0.6\*dim) columns) and add it to 40% of the second matrix and thus get two children. After this we introduce mutation into the new children matrices by adding an extra obstacle in a random open cell. After generation of the mutated children maze we check whether they are solvable or not. If they are solvable we add then to the children and parents maze. Then, we select the best children from the 2N children and parents maze and set them as the children maze for the next iteration.

Genetic algorithm has the best chance for finding the hardest maze amongst the algorithms that we have implemented.

**Unlike the problem of solving the maze, for which the `goal' is well-defined, it is difficult to know if you have constructed the `hardest' maze. What kind of termination conditions can you apply here to generate hard if not the hardest maze? What kind of shortcomings or advantages do you anticipate from your approach?**

Termination conditions for Hill Climbing algorithm:-

1. When the maximum length of the shortest path has repeated itself for 20 times successively.

Termination conditions for Simulated Annealing algorithm:-

1. When the maximum length of the shortest path has repeated itself for 20 times successively.
2. After certain predefined number of iterations which represents after certain time.

Termination conditions for Genetic algorithm:-

1. When the maximum length of the shortest path has repeated itself for 20 times successively.
2. After certain predefined number of iterations which is analogous to number of generations.
3. No solvable children are generated.

One of our termination conditions is being able to control the number of iterations, this helps us in saving time and computing resources while running our algorithm for a large maze.

Shortcomings that I anticipate from my approach is that for example in hill climbing we might wrongly considers plateau (repeated value for a long time) as a local maxima. After a certain number of iterations we started seeing the same value being repeated for hill climbing as we were only making neighbors by adding obstacles. Whereas this was not the case in simulated annealing where we added (+1 and +2) and also removed (-1 and -2) obstacles which resulted in larger variations of the output results.

Another major shortcoming of our approach is that we may not be able to find the hardest maze because combining parents in genetic algorithm does not guarantee the children to be harder. Due to this the output from different generations does not vary much. Also there is no significant variations in the results from the children as there are lesser number of obstacles present initially in their parents as we increase only one obstacle at the time of mutation.

Hill climbing and simulated annealing algorithm take a longer time to run for larger mazes as compared to genetic because they generate all the possible neighbors with increasing number of obstacles (+1 and +2).

**Try to find the hardest mazes for the following algorithms using the paired metric:**

**DFS with Maximal Fringe Size**

**A\_-Manhattan with Maximal Nodes Expanded**

DFS with maximal fringe size

Maximal fringe size computed at return from the function(DFS) call.

Taking dim=50 and p=0.2 with number of generations=20 and Number of children(N)= 10

We get the following results:-

|  |  |
| --- | --- |
| Generation | Maximal Fringe Size |
| 1 | 602 |
| 2 | 609 |
| 3 | 609 |
| 4 | 604 |
| 5 | 590 |
| 6 | 588 |
| 7 | 604 |
| 8 | 600 |
| 9 | 599 |
| 10 | 594 |
| 11 | 537 |
| 12 | 597 |
| 13 | 572 |
| 14 | 599 |
| 15 | 598 |
| 16 | 557 |
| 17 | 512 |
| 18 | 511 |
| 19 | 525 |
| 20 | 547 |

The maximum value amongst the results is- 609

A\* Manhattan with maximal nodes expanded

Maximal nodes expanded computed at return from the function(A\_star) call.

Taking dim=50 and p=0.2 with number of generations=20 and Number of children(N)= 10

We get the following results:-

|  |  |
| --- | --- |
| Generation | Maximal Fringe Size |
| 1 | 1990 |
| 2 | 1982 |
| 3 | 1985 |
| 4 | 1981 |
| 5 | 1986 |
| 6 | 1982 |
| 7 | 1982 |
| 8 | 1982 |
| 9 | 1980 |
| 10 | 1978 |
| 11 | 1978 |
| 12 | 1978 |
| 13 | 1980 |
| 14 | 1979 |
| 15 | 1975 |
| 16 | 1981 |
| 17 | 1971 |
| 18 | 1976 |
| 19 | 1972 |
| 20 | 1969 |

The maximum value amongst the results is- 1990

**Do your results agree with your intuition?**

As the maze gets harder the solution path will tend to become more wiggly(passing through the maze)

For DFS with Maximal Fringe Size the fringe size at the end of the DFS operation should decrease as with more harder maze there will be more obstacles which leads to more points being popped from the fringe.

For A\*Manhattan with Maximal Nodes Expanded at the end of A\* operation should increase because A\*Manhattan travels along the edge of the matrix and as the maze becomes harder and harder, the algorithm has to expand more as it won’t be able to find a path along the side of the mazes.

1. **What if the Maze Were on Fire**