RUTGERS

THE STATE UNIVERSITY OF NEW JERSEY

Project Report for Intro to AI (CS 520)

Fast Trajectory Replanning

Submitted to Rutgers, the State University of New Jersey

Towards the partial fulfillment for the Award of the Degree of

MASTERS OF SCIENCES

In Computer and Information Sciences

2022-2023

By

Niyati Jain - 216008777 (nsj39)

Agrani Swarnkar - 219002416 (as4154)

Dhruv Patel - 219000059 (dp1224)

Under the Guidance of

Mr. Abdeslam Boularias

Department of Computer Science Engineering

School of Computing and Information Technology

Rutgers University – New Brunswick

New Jersey

Table of Contents

List of Figures:

Abstract

This project relates to video game technologies and is part of our effort to use computer games as a motivator in projects without having to use game engines. A fast trajectory replanning algorithm is essentially moving game characters, that we call ‘intelligent agent’, to a given target or goal in an initially unknown but finite gridworld. Heuristic search specially A*(A star) is among the most widely used single-agent search techniques and thus effective for this Artificial Intelligence project. This project describes the working of A*, and modifying it as a form of two techniques- Repeated Forward A* and Repeated Backward A* as well as extending it to a more effective variant- Adaptive A*. Adaptive A* is an modified version of Repeated Forward A* that updates the heuristic such that minimum number of states are explored. This is the first project as a part of our course CS520- ‘Introduction to Artificial Intelligence’ as an objective to better understand different search techniques.

Introduction

One of the fundamental single-agent search approaches is heuristic search, precisely, A*. Heuristic search is an informed search technique that typically expands the node that seems the closest to the target. In other words, it is an estimation of the shortest or the cheapest path from the current state to the goal state.

A* algorithm is an extension of the heuristic search technique in a way such that the cost applied in travelling from the start state to the current state is also taken into account for us to get a better understanding of the most optimal path of all the best options. Repeated Forward A* is again an algorithm in which an intelligent agent finds its goal using the A* technique but when it realises that the path is blocked or not exactly the shortest presumed path, it comes back into the previous state with the knowledge of that previous route and why its not supposed to go there. repeated Backward A* is the same technique which works the opposite way considering the goal to be the start state and vice versa but the tree traversal in the end happens in reverse resulting in the same output.

In this study, we are also implementing Adaptive A*, a generalization of Repeated Forward A*, that solves a sequence of related search problems more quickly than A*, as it uses data from earlier searches to update the h-values. Because it just converts consistent h-values into more knowledgeable consistent h-values, adaptive A* is straightforward to comprehend and simple to execute.

In order to respond to challenges that demand for computational thinking, we have programmed the given questions and then built a comprehensive code that implements all of these stated algorithms as per instructions.

Part 0: Setting Up the Environment

The primary part of the project was to set up an environment for our 'Agent', the virtual character that moves towards the goal. To do the same, we initially generated a grid-like structure resembling a maze as the environment. The grid is created using a 2D array of dimensions equal to the size of the grid and we input values in the corresponding cells by traversing through each one them using the Depth first search technique taught in class. We use the same since it has a time complexity of O(V) where V is the number of vertices in contrast to two nested for loops which has the time complexity of O(V^2). In the createfinalGrid() function of our project, we first set all the cells as unvisited and indicated them using the '#' symbol. After that, we iterated through all the cells of the grid one by one until each cell was visited[2]using stack. We push all the neighbours of the visited cells in the stack using the direction vectors and pop them one by one. We also use the isValid() function in our program to check whether a cell has been visited or not. Further, in the createFinalGrid() method, we used the random function to mark the cell as blocked or unblocked. If the probability of the number generated using the random function was greater than 30%, the cell was marked as unblocked denoted by the symbol '.'. Using this procedure, 30% of the total cells in the grid are blocked and 70% of the cells are unblocked.

The next step in the process of setting up the environment was to set the position of our Agent and the Target. This process was also completed with the help of the random function where we randomly set the position of the Agent and the Target as 'A' and 'T' respectively. We mark the coordinates of the agent in the pos_a_x and pos_a_y variable and the target in the pos_t_x and pos_t_y variable.

```
# Initialize direction vectors
dRow = [0, 1, 0, -1]
dCol = [-1, 0, 1, 0]

# Function to check if mat[row][col] is unvisited and lies within the
boundary of the given matrix
def isValid(row, col):
 global ROW
 global COL
 global vis

 # If cell is out of bounds
 if (row < 0 or col < 0 or row >= n or col >= n):
     return False

 # If the cell is already visited
 if (vis[row][col]):
     return False

 # Otherwise, it can be visited
 return True

def createfinalGrid(row, col, grid, n):
 global dRow
 global dCol
 global vis

 vis = [[False for i in range(n)] for j in range(n)]
 # Initialize a stack of pairs and push the starting cell into it
 st = []
 st.append([row, col])

 # Iterate until the stack is not empty
```

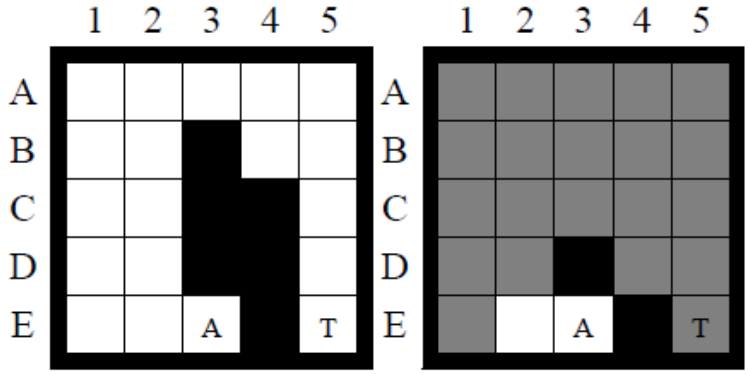
```
while (len(st) > 0):
    # Pop the top pair
    curr = st[len(st) - 1]
    st.remove(st[len(st) - 1])
    row = curr[0]
    col = curr[1]

    # Check if the current popped cell is a valid cell or not
    if (isValid(row, col) == False):
        continue

    # Mark the current cell as visited
    vis[row][col] = True

    num = random.random()
    if num >= 0.2:
        grid[row][col] = '.'

    # Push all the adjacent cells
    for i in range(4):
        adjx = row + dRow[i]
        adjy = col + dCol[i]
        st.append([adjx, adjy])
posa_x = int((random.random()) * n)
posa_y = int((random.random()) * n)
post_x = int((random.random()) * n)
post_y = int((random.random()) * n)
grid[posa_x][posa_y] = 'A'
grid[post_x][post_y] = 'T'
return grid
```

A snapshot of how our grid looks once we have set up the blockages and the position of the Agent and the Target is as shown below in Figure 0.1:

```
['.', '.', '.', '#', '#']
['.', '#', 'T', '#', '.']
['.', '.', '#', '#', '#']
['A', '#', '#', '.', '.']
['.', '#', '.', '.', '.']
```

Figure 0.1

In the above figure, we can see that the blocked cells are denoted by the symbol '#' and unblocked cells are denoted by '.'. The position of the Agent 'A' is (3,0) and the position of the Target 'T' is (1,2).



(a) (b)

Figure 0.2

In Figure 0.2, part (a) shows a better illustration of the initial state where the Agent is at (E,3) and Target is at (E,5). Part (b) shows that when Agent A starts from position (E,3), it can only see the highlighted path and only the blocked cells in its vicinity are visible to A. Rest all the cells of the grid are unblocked according to A.

As instructed, we create a 50 gridworld environment of n=101 that is 101 X 101 dimensions using the python library 'pickle'. This library has modules that help us serializing or converting a Python object into a byte stream and vice versa. A Grid folder is created where all the Grids which are named according to the chronology of which they are generated are stored using the *pickle.load()* method. The documents are marked the *'rb'* which gives us the read and write access. The coordinates of the agent and target are noted for every grid and the corresponding search technique runs in the loop in order to calculate the average time taken and compare them for various algorithms implemented.

```
if timecalculation==0:
 with open('/content/Grids/Grid_1.data', 'rb') as f:
   grid= pickle.load(f)

 for i in range(n):
   for j in range(n):
     if grid[i][j] == 'A':
       pos_a_x= i
       pos_a_y= j
     if grid[i][j] == 'T':
       pos_t_x= i
       pos_t_y= j
```

Once applying the desired filters for the algorithm and setting up the dimensions of the grid as 5, we can visualize the grid with the agent moving as shown below in Figure 0.3:

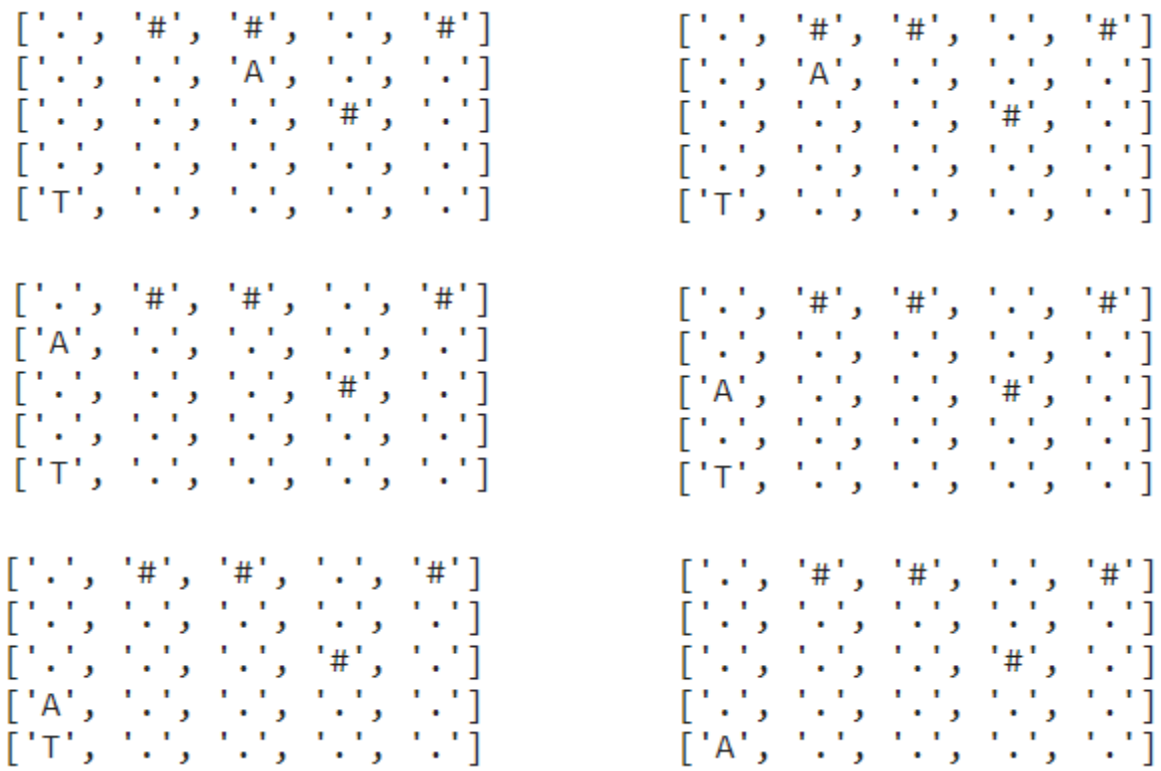


Figure 0.3: Time step 1 to 6

Part 1: Understanding the methods

Solution a) The procedure that the A* algorithm follows is that it allocates a particular cost to every cell of our generated maze and then selects the path which has the minimum cost. The cost of the cell is computed using the function f(n) and stored in the f array which further consists of the summation of two functions, that is, g(n) and h(n).

$$f(n) = g(n) + h(n)$$

g(n): The actual cost of the path to reach from the start cell, where the Agent is, to the goal cell, where the Target is calculated using the *setgval()* function in the code and stored in the *g* array. As g-value is the actual cost of the path, and we are assuming the cost of movement to adjacent or neighboring cell as 1, the g-value of the next cell is incremented by 1 unit from the current cell where the Agent is present.

```
def setgval(bgrid, pos_a_x, pos_a_y, pos_t_x, pos_t_y, gval):
   try:
       if bgrid[pos_a_x][pos_a_y] == '.' or bgrid[pos_a_x][pos_a_y] == 'A'
or bgrid[pos_a_x][pos_a_y] == 'T':
           gval[pos_a_x][pos_a_y] += 1
           #calculated g value by adding 1 as the cost taken to go to that
state from the current state
   except IndexError:
       pass
   return gval
```

h(n): h(n) is the heuristic function. It is the estimated cost of the path from the start cell to the target or goal cell calculated using the *sethval()* function in our program and stored in the *h* array. This function computes the Manhattan Distance between the current position of the agent and the goal state which is the sum of the absolute

difference of the x coordinates of the cell and the Target and the absolute difference of the y coordinates of the cell and the cell of the Target. Computing the h value using the Manhattan Distance is preferred because it is regular in the grid environment where our Agent can move in one of the four main directions, that is, up, down, right and left.

```
def sethval(bgrid, pos_t_x, pos_t_y, h):
   for i in range(0,n):
     for j in range(0,n):
       if bgrid[i][j] == '.' or bgrid[i][j] == 'A'or bgrid[i][j] == 'T':
         h[i][j] = abs(i - pos_t_x) + abs(j - pos_t_y)
         #calculated manhattan distance by subtracting the x and y
coordinates of the state from the goal
   return h
```

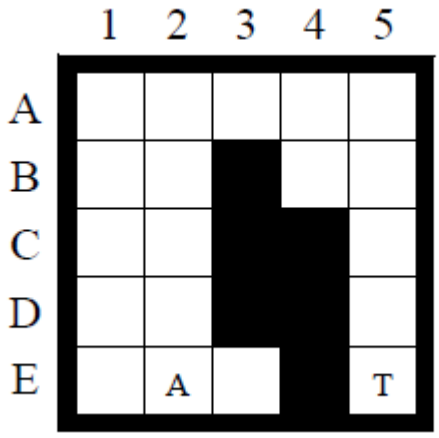


Figure 1.1

In the given figure, in the initial state when the agent does not know which path is blocked, the first move of the agent will be towards east instead of north. This is because of the usage of h values that calculate the estimated lowest cost path from the Agent to the Target. In the given figure example above, the Target is towards the east direction of the Agent. Hence, the Manhattan Distance from the Agent to

the Target will be lowest. But once we calculate and compare the f values of all the adjacent unblocked cells, the agent moves along the right direction.

Solution b) The Agent is in a known grid environment, so it will have a finite number of steps. The total number of moves made by our Agent would be equal to the summation of the total number of searches done by the Agent and the total number of moves made by the Agent in each search. During each search the Agent will find at least one unblocked neighboring cell. If there are no unblocked cell then the Agent instantly discovers that it is not possible for it to reach the Target. Hence, we can say that the Total number of searches is bounded by the number of unblocked cell. Similarly, the total number of moves that the Agent will make in each search is also bounded by the number of unblocked cells as the Agent can move only in the unblocked cells.

Since the Total number of moves made by the Agent is equal to the Total number of searches multiplied by the number of moves made in each search, and both the processes are bounded by the number of unblocked cells, we can say that the upperbound of the number of moves of the agent until it reaches the target or discovers that this is impossible is equal to the number of unblocked cells squared.

Let S be the set of all the searches that the Agent makes while finding its path to the Target or the goal cell. And let N be the set of number of moves in each search.

$$S = [s_{1,}\, s_{2,}\, s_{3,\ldots,}\, s_n]$$

$$N = [n_{s1,}\, n_{s2,}\, n_{s3,\ldots,}\, n_{sn}]$$

The Total number of steps(T) can be written as:

$$T = s_1 * n_{s1} + s_2 * n_{s2} + s_3 * n_{s3} + \ldots + s_n * n_{sn}$$

$$T = \Sigma\, s_i * n_{si}$$

where i = 0 to n.

Now $S \in O(n)$ and $N \in O(n)$, $T \in O(n*n)$, i.e., $T \in O(n^2)$

Part 2: The Effect of Ties

A problem that could arise when our Agent is following the ComputePath() function is that it is faced by a situation where more than one cell has the same minimum f-values. Now the Agent has to decide which cell it should expand next, that is, it has to break ties. It has the ability to break ties either in favor of cells with smaller g-values or larger g-values. For implementing this methodology to break the ties is to use the equation c * f(n) - g(n) where c is constant with a very large value. The value of c would be greater than any generated g-value.

For example, if we assume the value of c to be 10,000 then we can use the equation

10,000*f(n) - g(n).

As we discussed in the above statement, the value of g(n) can be of two types :- Larger g-values and Smaller g-values. We implemented this functionality in our code using the effectofties() function which is used to break ties when the Agent is faced by a situation where more than one neighboring cells have the same f-values.

```
def effectofties(f,g, priority, openlist):
 aaa=0
 if priority==0:
   hq.heapify(openlist)
   aaa = hq.heappop(openlist)
   # prioritising greater g values if f values are same
 elif priority==1:
   heap_neg = [-x[0] for x in openlist]
   hq.heapify(openlist)
   aaa = hq.heappop(openlist)
   #prioritising lesser g values if f values are same
 return aaa
```

When priority is set as 0, then the ties are broken using larger values of g. Whereas when the priority is set as 1, then the ties are broken using smaller values of g.

To check what the effect of both the methods of breaking the ties has on the runtime of the Repeated Forward A* algorithm, we stored the 50 grids in a variable using the python library ‘pickle’. Then we computed the Average time for the running of all the 50 grids first for Larger values of g using the priority 0, then for Smaller values of g using the priority 1. The Average runtime for both of the methodologies are shown in the table below:

S. No.	Priority(0 or 1)	Average Time taken for 50 gridworlds
1.	Larger(0)	0.0477400779724121
2.	Smaller(1)	0.044813060760498

Table 2.1

According to our observations the time taken for running the Forward A* algorithm is slightly greater when we take larger values of g instead of smaller values of g. The runtime values have a very negligible difference. This is because the methodology for the computation for both Priority 0, i.e., larger g-values and Priority 1, i.e., smaller g-values is similar, just that occasionally different paths will be selected for the agent based on the difference in calculating the best possible successor which much depends on the gridworld generated. On a general level, we can’t expect a significant difference in the time consumed for each of these conditions.

Part 3: Forward vs. Backward

We know that the Agent will definitely reach the Target if there are no blockages separating it from the Target. Initially our Agent only knows about the blockages in its neighboring cells and it is unaware of the blocked path that lies ahead. Hence, it expands the unblocked cell that has the minimum f-value. The final shortest presumed path on which the Agent moves is stored in the array *closedlist.*

Whenever the Agent is met with a blocked path while moving towards the Target, it retraces its path by popping the elements of the *closedlist* from which no further path is admissible and starts searching for a new shortest path from the cell again. This methodology is called the Repeated A* algorithm. A* algorithm can search either from the agent's current cell in the target's direction,that is, forward A* or from the target's current cell in the opposite direction, that is, backward A*, producing Repeated Forward A* or Repeated Backward A*, respectively.

In the traditional A* algorithm when the Agent comes across a blocked cell, it doesn't yet know whether it would have been beneficial to take a detour to find the right path first, which would necessitate restarting and finding the path from scratch, or whether continuing the search will still produce a better outcome. In contrast, if our agent starts to find a path using backward A*, that is, from the Target and then it encounters the blocked path, it can treat it as open and proceed expanding through that cell.

In our code we have declared a variable called *'direction',* and a function called *RepeatedAstar*() to implement Forward and Backward A*. When the value of the variable *direction* is set as 0, the function *RepeatedAstar*() performs Forward A*,

whereas, when the value of the variable *direction* is set as 1, then the function *RepeatedAstar*() performs Backward A*.

```
def RepeatedAstar(grid, pos_a_x, pos_a_y, pos_t_x, pos_t_y, direction, priority, timecalculation):
 h = [[0]*n for i in range(n)]
 g = [[0]*n for i in range(n)]
 f = [[0]*n for i in range(n)]
 openlist=[]
 # stores the c*f(n)-g(n) of the neighbours of the state along with its coordinates so that we can pick the most optimal neighbour
 closedlist=[]
 # stores the final list of explored states in the shortest presumed path
 if timecalculation==0:
   for gr in grid:
     print(gr)
   print()
 if direction==0:
   curr_x, curr_y = pos_a_x, pos_a_y
   goal_x, goal_y = pos_t_x, pos_t_y
   #For Repeated Forward A star
 elif direction==1:
   curr_x, curr_y= pos_t_x, pos_t_y
   goal_x, goal_y= pos_a_x, pos_a_y
   # For Repeated Backward A star
 h = sethval(grid, goal_x, goal_y, h)
 ret = ComputePath(grid, curr_x, curr_y, goal_x, goal_y, h, g, f, openlist, closedlist, n, priority)
 openlist.clear()
 while ret==0 and len(closedlist)>0:
   dis_x = ([item[0] for item  in closedlist])
   dis_y = ([item[1] for item  in closedlist])
   f[dis_x[0]][dis_y[0]]+=1000
   closedlist.pop(0)
```

```
   if len(closedlist)>0:
     rr_x = ([item[0] for item  in closedlist])
     rr_y = ([item[1] for item  in closedlist])
     curr_x= rr_x[0]
     curr_y=rr_y[0]
     ret = ComputePath(grid, curr_x, curr_y, goal_x, goal_y, h, g, f,
openlist, closedlist, n, priority)
     openlist.clear()
     #if ret=0, i.e. it is blocked from all ends or that is not the
feasible path, it should go back one state from the closed list,increase
the g value of the state which cant be explored again and repeat the
process again
 #Printing
 if timecalculation==0:
   if direction==0:
     answer_x = ([item[0] for item  in closedlist])
     answer_y = ([item[1] for item  in closedlist])
     l= len(closedlist)
     for no in range(0,l-1):
       t1= answer_x[no]
       t2 = answer_y[no]
       grid[t1][t2]= '.'

       i = answer_x[no+1]
       j = answer_y[no+1]
       grid[i][j]= 'A'
       for grids in grid:
         print(grids)
       print()
   if direction==1:
     answer_x = ([item[0] for item  in closedlist])
     answer_y = ([item[1] for item  in closedlist])
     l= len(closedlist)
     for no in range(l-1,0, -1):
```

```
        t1= answer_x[no]
        t2 = answer_y[no]
        grid[t1][t2]= '.'

        i = answer_x[no-1]
        j = answer_y[no-1]
        grid[i][j]= 'A'
        for grids in grid:
          print(grids)
        print()
    print ("Step value:", l)
```

The *RepeatedAstar()* , if direction variable is 0, we are implementing Forward Repeated A* and if direction variable is 1, then we are implementing Backward A*. The coordinates of the Agent are stored in the current cell(curr_x, curr_y) and the coordinates of the Target are stored in the goal cell(goal_x, goal_y). Similarly for Backward A*, the coordinates of the Agent are stored in the goal cell(goal_x, goal_y) and the coordinates of the Target are stored in the current cell(curr_x, curr_y), because here we have to search from the Target cell.

Next, we set the h-values for every cell of the grid using the *sethval()* function. Then we run the *ComputePath()* function, that moves the Agent in the direction of the target and returns the value of the *ret* variable. If the Agent reaches the target, that is, the coordinates of the Agent is the same as that of the Target, then the value of ret is equal to 1. Whereas, if the Agent is met with any blocked cell and is unable to move forward, the value of the ret variable is returned as 0. When the ret is 0, the Repeated A* algorithm is implemented where the current state of the closed list is discarded and it's g value is increased by 3 so that the agent doesn't fluctuate between these two states repeatedly and the previous explored state is set

to be the current state. Now, the agent knows that the discarded state has an increased g value since it was not the feasible choice and hence it avoids going there unless it has no option. Backward A* works essentially in the same manner but while printing, closedlist is traversed in the reverse direction in order to move the agent towards to goal following the backtracking path of the goal towards the agent.

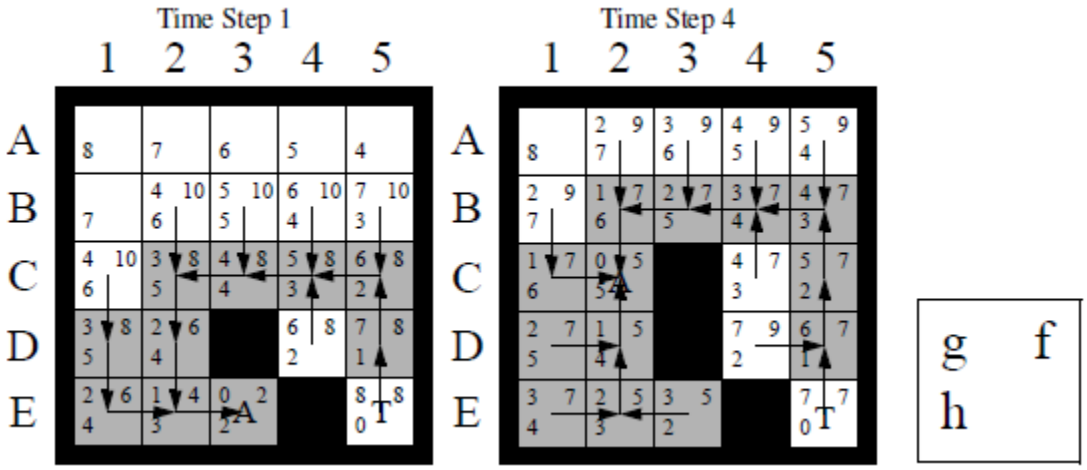


Figure 3.1: Repeated Forward A*

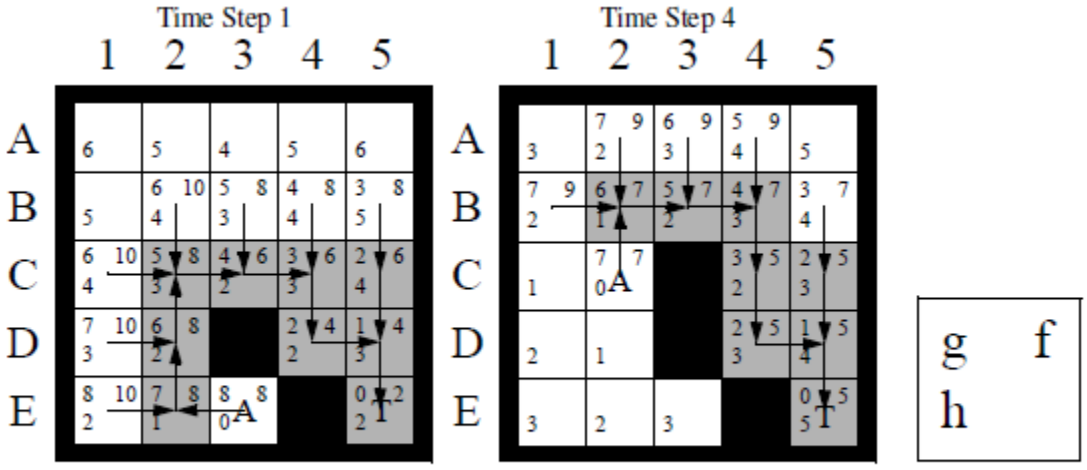


Figure 3.2 : Repeated Backward A*

To compare the differences between the running time of both Forward and Backward A*, we stored the 50 grids in a variable using the python library ‘pickle’. Then we computed the Average time for the running of all the 50 grids first keeping the direction variable as 0, i.e., Repeated Forward A*, then keeping the direction variable as 1, i.e., Repeated Backward A*. To prevent a situation where the Agent encounters equal f-values for more than one neighboring cell, we have broken the ties with the help of larger g-values.

The Average runtime for 50 gridworlds for Forward and Backward A* is shown in the table given below:

S. No.	Type of A*	Average Time taken for 50 gridworlds
1.	Repeated Forward A*	0.0477400779724121
2.	Repeated Backward A*	0.0431150913238525

Table 3.1

As it is evident from the above table, the Average time taken for a 50 gridworld environment for Repeated Backward A* is slightly less than Repeated Forward A*. Since the pathways provided by the tree-pointers move from the destination states to the start states for forward searches and must thus be reversed, forward searches have a runtime overhead compared to backward searches. The difference is the runtime is almost negligible because the methodology of implementation of both the functions is similar. The only difference is that in Repeated Forward A* we are searching from the Start cell to the Target and in Backward A* we are searching from the Target to the Start cell. Hence, the runtime mainly depends on the grid environment.

Part 4: Heuristics in the Adaptive A*

Our project states the fact that “Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions”.

The Manhattan distance is calculated in the initial state and the h-value of the maze is consistent and unchanged throughout the functioning of the algorithm. It gives the exact minimum cost of moving the Agent in the direction of the target, given there were no blockages present, so it is said to be admissible.

If a heuristic never overestimates the actual cost of reaching a nearby target, it is admissible. A heuristic is consistent if it never overestimates the real step cost when moving between neighboring nodes a and b.

If a heuristic function h consistently calculates the cost to achieve the goal and provides the lowest cost, it is considered to be admissible.

$$\forall n : h(n) \leq h*(n),$$

where h(n) is the actual cost of taking the quickest route from n to the goal state.

If h is admissible then the tree search A* is optimal.

We can prove this statement with the help of a lemma or an assumption.

We take the assumption as : *h is admissible*.

Now, let n be a node such that n*State = goal cell

where n does not lie on the optimal path of the Agent to the Target.

Next, we will take the first node on the optimal path that has not yet been expanded as n′. A* will expand the node n′ before n because n′ lies on the optimal path and

A* algorithm expands any node present on the optimal path before the node that lies on any non-optimal path even if it contains the goal.

Hence, we can say that A* is optimal[2].

Now we have to prove the assumption. For that let C* denote the cost of the optimal path. We know that the node n is non-optimal and it has n*state = goal. The value of f(n) is calculated when n is added to the openlist. Since n*state = goal, the value of f(n) = g(n), that is, the real cost of the path from the start to n.

For n′, i.e., the first node of the optimal path that has not been expanded,

$$f(n') = g(n') + h(n') \leq C^* \leq g(n) = f(n)$$

First inequality is derived from the fact that h is admissible and the second one is derived from the definition of minimum cost. Hence, we can say that optimal node n′ should be expanded before the non-optimal node n.

A heuristic function h is said to be consistent if

$$\forall (n, a, n') : h(n) \leq C(n, a, n') + h(n')$$

where C(n, a, n′) is the step cost for going from n to n′ using action a. This is called the Triangle Inequality.

The heuristic values that we use throughout the program are calculated only once in the start of the program, i.e, the Manhattan values. Since these values are consistent, then the h-values are consistent with respect to the Goal state.

The main function of the program is the ComputePath() function. As the name suggests, it computes the path of our Agent moving towards the Target.

```
def ComputePath(grid, curr_x, curr_y, goal_x, goal_y, h,g, f, openlist,
closedlist, n, priority):
  count = 0
  foundPath=0
  c= 2*n+1

  while count< 3*n:
    count+=1
    closedlist.append([curr_x, curr_y])
    hq.heapify(openlist)

    if([goal_x, goal_y])==([curr_x, curr_y]):
      foundPath = 1
      return foundPath
      break

    temp = []

    if (curr_x + 1) <n and (curr_y) < n and (curr_x + 1) > -1 and (curr_y)
> -1:
          if grid[curr_x + 1][curr_y] != '#':
            g = setgval(grid, curr_x + 1, curr_y, goal_x, goal_y, g)
            f[curr_x+1][curr_y] = h[curr_x+1][curr_y] +
g[curr_x+1][curr_y]
            eot= c*f[curr_x+1][curr_y] - g[curr_x+1][curr_y]
            temp = (eot, ((curr_x + 1),(curr_y)))
            hq.heappush(openlist, temp)
            # checking down neighbour if unblocked and within the range
            #then calculating its f,g and h values as well as pushing into
the openlist
```

```
    if (curr_x - 1) <n and (curr_y) < n and (curr_x - 1) > -1 and (curr_y)
> -1:
        if grid[curr_x - 1][curr_y] != '#' and curr_x - 1>=0:
          g = setgval(grid, curr_x - 1, curr_y, goal_x, goal_y, g)
          f[curr_x-1][curr_y] = h[curr_x-1][curr_y] + g[curr_x-1][curr_y]
          eot= c*f[curr_x-1][curr_y] - g[curr_x-1][curr_y]
          temp= (eot, ((curr_x - 1),(curr_y)))
          hq.heappush(openlist, temp)
          # checking up neighbour if unblocked and within the range
          #then calculating its f,g and h values as well as pushing into
the openlist


    if (curr_x) <n and (curr_y + 1) < n and (curr_x) > -1 and (curr_y+ 1)
> -1:
        if grid[curr_x][curr_y + 1] != '#':
          g = setgval(grid, curr_x, curr_y + 1, goal_x, goal_y, g)
          f[curr_x][curr_y+1] = h[curr_x][curr_y+1] + g[curr_x][curr_y+1]
          eot= c*f[curr_x][curr_y+1] - g[curr_x][curr_y+1]
          temp= (eot, ((curr_x),(curr_y + 1)))
          hq.heappush(openlist, temp)
          # checking right neighbour if unblocked and within the range
          #then calculating its f,g and h values as well as pushing into
the openlist


    if (curr_x) <n and (curr_y- 1) < n and (curr_x) > -1 and (curr_y -1) >
-1:
        if grid[curr_x][curr_y - 1] != '#' and curr_y - 1>=0:
          g = setgval(grid, curr_x, curr_y - 1, goal_x, goal_y, g)
          f[curr_x][curr_y-1] = h[curr_x][curr_y-1] + g[curr_x][curr_y-1]
          eot= c*f[curr_x][curr_y-1] - g[curr_x][curr_y-1]
          temp= (eot, ((curr_x),(curr_y - 1)))
          hq.heappush(openlist, temp)
```

```
          # checking left neighbour if unblocked and within the range
          #then calculating its f,g and h values as well as pushing into
the openlist


    if len(openlist)==0:
        return foundPath;


    aa= effectofties(f, g, priority, openlist)
    curr_x= aa[1][0]
    curr_y= aa[1][1]
    if ([(curr_x), (curr_y)] in closedlist):
      tr = closedlist.pop()
      tr_x = tr[0]
      tr_y = tr[1]
      g[tr_x][tr_y]+= 3
      #if the selected state was already explored then
      #increasing the g value of the current state so that it does not
keep moving between these two states
    openlist.clear()
```

In the ComputePath() function, the current coordinates of the agent are stored in the variable closedlist. The uper limit of the number of steps that the Agent can move is set as 3*n, where n is the number of rows of the grid. If the uper limit is exceeded, then that implies that the path cannot be found. If the coordinates of the current position of the agent are the same as the coordinates of the goal cell, the variable foundpath is set as 1 and we return the value of foundpath. If that is not the case, we check the neighboring cells of the Agent. We check whether the neighboring cells are unblocked. If so, we calculate th f, g, h and eot values of each

unblocked cell. The eot value refers to the function c*f(n) - g(n) which is used to break ties in case of same values of f as explained in Part 2 of the report.

Next, we push the values eot and the coordinates of the neighbors in the openlist using heappush(). After that, we sort the open list with respect to the eot values using the heapify() function.

We call the effectofties() function to find out the cell with the least eot value. The cell with the least eot value is returned and set as the current cell. If that cell is present in the closed list, that is, it has already been explored, then we pop that value and increase the g value of that cell so that it does not keep moving between these two states.

Part 5: Heuristics in the Adaptive A*

Adaptive A* is the most recent incremental heuristic search algorithm which adjusts its h-values based on data from past searches, which allows it to solve a series of related search problems more quickly than A*. In essence, it converts consistent h-values into more knowledgeable consistent h-values. Since consistent h-values continue to be consistent even after action cost increases, this enables it to determine shortest pathways in state spaces where the action costs can increase over time.

Adaptive A* updates (= overwrites) the consistent h-values with respect to the goal state of all expanded state s after an A* search by executing $h(s) := g(s_{goal}) - g(s)$. This principle was first used in and later resulted in the independent development of Adaptive A*. The updated h-values are again consistent with respect to the goal state. They also dominate the immediately preceding h-values at least weakly. Thus, they are no less informed than the immediately preceding h-values and an A* search with the updated h-values thus cannot expand more states than an A* search with the immediately preceding h-values (up to tie breaking).

```
def AdaptiveAstar(grid, pos_a_x, pos_a_y, pos_t_x, pos_t_y, priority,
timecalculation):
   h = [[0]*n for i in range(n)]
   g = [[0]*n for i in range(n)]
   f = [[0]*n for i in range(n)]
   openlist=[]
   # stores the c*f(n)-g(n) of the neighbours of the state along with its
coordinates
   # so that we can pick the most optimal neighbour
   closedlist=[]
```

```
   # stores the final list of explored states in the shortest presumed
path
   if timecalculation==0:
     for gr in grid:
       print(gr)
     print()
   curr_x, curr_y = pos_a_x, pos_a_y
   goal_x, goal_y = pos_t_x, pos_t_y
   h = sethval(grid, goal_x, goal_y, h)
   ret = ComputePath(grid, curr_x, curr_y, goal_x, goal_y, h, g, f,
openlist, closedlist, n, priority)
   openlist.clear()
   while ret==0 and len(closedlist)>0:
     dis_x = ([item[0] for item  in closedlist])
     dis_y = ([item[1] for item  in closedlist])
     f[dis_x[0]][dis_y[0]]+=1000
     closedlist.pop(0)
     if len(closedlist)>0:
       rr_x = ([item[0] for item  in closedlist])
       rr_y = ([item[1] for item  in closedlist])
       curr_x= rr_x[0]
       curr_y=rr_y[0]
       for abc in closedlist:
         abc_x = abc[0]
         abc_y = abc[1]
         h[abc_x][abc_y]= g[goal_x][goal_y]- g[abc_x][abc_y]
         ret = ComputePath(grid, curr_x, curr_y, goal_x, goal_y, h, g, f,
openlist, closedlist, n, priority)
         openlist.clear()
         #if ret=0, i.e. it is blocked from all ends or that is not the
feasible path, it should go back one state from the closed list,
         #increase the g value of the state which cant be explored again
and repeat the process again
```

```
#Printing
if timecalculation==0:
  answer_x = ([item[0] for item  in closedlist])
  answer_y = ([item[1] for item  in closedlist])
  l= len(closedlist)
  for no in range(0,l-1):
    t1= answer_x[no]
    t2 = answer_y[no]
    grid[t1][t2]= '.'

    i = answer_x[no+1]
    j = answer_y[no+1]
    grid[i][j]= 'A'
    for grids in grid:
      print(grids)
    print()
  print ("Step value:", l)
```

To compare the differences between the running time of both Repeated Forward A* and Adaptive A*, we stored the 50 grids in a variable using the python library ‘pickle’. Then we computed the Average time for the running of all the 50 grids first keeping the direction variable as 0, i.e., Repeated Forward A*, then keeping the direction variable as 1, i.e., Repeated Backward A*. To prevent a situationwhere the Agent encounters equal f-values for more than one neighboring cell, we have broken the ties with the help of larger g-values. The Average runtime for 50 gridworlds for Forward and Backward A* is shown in the table given below:

S. No.	Type of A*	Average Time taken for 50 gridworlds
1.	Repeated Forward A*	0.0477400779724121
2.	Adaptive A*	0.0466012477874755

Table 5.1

As it is evident from the above table, the Average time taken for a 50 gridworld environment for Repeated Forward A* is slightly greater than Adaptive A*.

The reason for this observation is that Adaptive A* uses its experience with earlier searches in the sequence to speed up the current A* search. That is why it runs faster than Repeated Forward A*.

Part 6: Statistical Significance

Differences in performance between two search algorithms may be systematic in nature or simply the result of sampling noise (=bias displayed by the chosen test cases since the number of test instances is always constrained). In order to determine whether the performance differences are systematics in nature, we can utilize Statistical Hypothesis Tests. Statistical Hypothesis Testing is one of the most crucial processes to perform when we are utilizing a random sample to draw inferences about a population[3].

In the majority of cases, while we are testing our project, it is simply impossible to observe the entire population to understand its properties. For instance, in our project for fast trajectory replanning, we are working on a gridworld of 50 grids. It is difficult to compute the running time of 50 grids of 101*101 dimensions for each of Repeated Forward A*, Repeated Backward A* and Adaptive A*, for Larger as well as Smaller values of g. We can simplify this procedure by collecting a randomly generated sample and then use statistics to analyze that sample.

For instance, it might be possible that the mean of the sample that we generated is equal to the mean of the actual population. The difference between the sample statistic and the population value is called Sample Error. Hence, To help distinguish between a random error and a genuine effect, we employ a hypothesis test to compare the means.

In Hypothesis testing when we perform statistical analysis on a randomly generated sample data, we assess two mutually exclusive theories about the

properties of a population. These theories are referred to by statisticians as the null hypothesis and the alternative hypothesis. To establish which hypothesis the data support, a hypothesis test evaluates your sample statistic and takes into account an estimate of the sample error.

Another important term while performing Statistical hypothesis testing is ‘Effect’. Effect can be defined as the difference between the population value and the value computed by the null hypothesis.

Null hypothesis

The null hypothesis basically states that the size of effect tends to zero, i.e., there is no effect. The null hypothesis is generally denoted by $H_{0.}$

In any kind of hypothesis testing, the researchers are testing an effect of some sort. This is because they want to identify some kind of advantage or difference.

In some cases, it might be possible that they are unable to find any kind of effect. This scenario, where the effect is equal to 0 is called Null Hypothesis. Hence, if we can dismiss the null hypothesis, we can go forward with the alternative hypothesis.

Alternative Hypothesis

The next theory that is mutually exclusive to null hypothesis is the Alternative Hypothesis theory. In contrast to the null hypothesis, Alternative Hypothesis is the type of hypothesis testing where effect is not equal to zero, that is,population parameter does not equal the null hypothesis value. Although we would require a solid proof to debunk the null hypothesis and apply Alternative hypothesis. The Alternative Hypothesis is denoted H_1

References

[1]. https://www.geeksforgeeks.org/depth-first-traversal-dfs-on-a-2d-array/

[2].http://rl.cs.rutgers.edu/fall2019/lecture3.pdf

[3].https://statisticsbyjim.com/hypothesis-testing/statistical-hypothesis-testing-overview/

[4].http://www.eecs.harvard.edu/cs286r/courses/spring08/reading6/CohenTutorial.pdf