VIVEK BURUGUDDA RA1911003010316

HARSHIL AGARWAL RA1911003010325

RITESH RAI RA1911003010333

PUNEET SHARMA RA1911003010331

Prakhar Vijay RA1911003010337

**18CSC305J - ARTIFICIAL INTELLIGENCE**

**Week 5: Developing Best first search and A\*Algorithm for real world problems**

Aim: Developing Best first search and A\*Algorithm for real world problems.

**Best First Search:**

**Algorithm:**

Step 1 : Create a priorityQueue pqueue.

Step 2 : insert ‘start’ in pqueue : pqueue.insert(start)

Step 3 : delete all elements of pqueue one by one.

Step 3.1 : if, the element is goal . Exit.

Step 3.2 : else, traverse neighbours and mark the node examined.

Step 4 : End.

**Code:** C++ code for best first search is as follows:

#include <bits/stdc++.h>

using namespace std;

typedef pair<int, int> pi;

vector<vector<pi> > graph;

// Function for adding edges to graph

void addedge(int x, int y, int cost)

{

graph[x].push\_back(make\_pair(cost, y));

graph[y].push\_back(make\_pair(cost, x));

}

// Function For Implementing Best First Search

// Gives output path having lowest cost

void best\_first\_search(int source, int target, int n)

{

vector<bool> visited(n, false);

// MIN HEAP priority queue

priority\_queue<pi, vector<pi>, greater<pi> > pq;

// sorting in pq gets done by first value of pair

pq.push(make\_pair(0, source));

int s = source;

visited[s] = true;

while (!pq.empty()) {

int x = pq.top().second;

// Displaying the path having lowest cost

cout << x << " ";

pq.pop();

if (x == target)

break;

for (int i = 0; i < graph[x].size(); i++) {

if (!visited[graph[x][i].second]) {

visited[graph[x][i].second] = true;

pq.push(make\_pair(graph[x][i].first,graph[x][i].second));

}

}

}

}

// Driver code to test above methods

int main()

{

// No. of Nodes

int v = 14;

graph.resize(v);

// The nodes shown in above example(by alphabets) are

// implemented using integers addedge(x,y,cost);

addedge(0, 1, 3);

addedge(0, 2, 6);

addedge(0, 3, 5);

addedge(1, 4, 9);

addedge(1, 5, 8);

addedge(2, 6, 12);

addedge(2, 7, 14);

addedge(3, 8, 7);

addedge(8, 9, 5);

addedge(8, 10, 6);

addedge(9, 11, 1);

addedge(9, 12, 10);

addedge(9, 13, 2);

int source = 0;

int target = 9;

// Function call

best\_first\_search(source, target, v);

return 0;

}

**Output:**



**A\* Algorithm:**

**Algorithm:**

* Add start node to list
* For all the neighbouring nodes, find the least cost F node
* Switch to the closed list
  + For 8 nodes adjacent to the current node
  + If the node is not reachable, ignore it. Else
    - If the node is not on the open list, move it to the open list and calculate f, g, h.
    - If the node is on the open list, check if the path it offers is less than the current path and change to it if it does so.
* Stop working when
  + You find the destination
  + You cannot find the destination going through all possible points

**Code:** Python code for A\* algorithm is as follows:

class box():

"""A box class for A\* Pathfinding"""

def \_\_init\_\_(self, parent=None, position=None):

self.parent = parent

self.position = position

self.g = 0

self.h = 0

self.f = 0

def \_\_eq\_\_(self, other):

return self.position == other.position

def astar(maze, start, end):

"""Returns a list of tuples as a path from the

given start to the given end in the given board"""

# Create start and end node

start\_node = box(None, start)

start\_node.g = start\_node.h = start\_node.f = 0

end\_node = box(None, end)

end\_node.g = end\_node.h = end\_node.f = 0

# Initialize both open and closed list

open\_list = []

closed\_list = []

# Add the start node

open\_list.append(start\_node)

# Loop until you find the end

while len(open\_list) > 0:

# Get the current node

current\_node = open\_list[0]

current\_index = 0

for index, item in enumerate(open\_list):

if item.f < current\_node.f:

current\_node = item

current\_index = index

# Pop current off open list, add to closed list

open\_list.pop(current\_index)

closed\_list.append(current\_node)

# Found the goal

if current\_node == end\_node:

path = []

current = current\_node

while current is not None:

path.append(current.position)

current = current.parent

return path[::-1] # Return reversed path

# Generate children

children = []

for new\_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1),\

(-1, 1), (1, -1), (1, 1)]: # Adjacent squares

# Get node position

node\_position = (current\_node.position[0] + new\_position[0],\

current\_node.position[1] + new\_position[1])

# Make sure within range

if node\_position[0] > (len(maze) - 1) or node\_position[0] < 0 or \

node\_position[1] > (len(maze[len(maze)-1]) -1) or node\_position[1] < 0:

continue

# Make sure walkable terrain

if maze[node\_position[0]][node\_position[1]] != 0:

continue

# Create new node

new\_node = box(current\_node, node\_position)

# Append

children.append(new\_node)

# Loop through children

for child in children:

# Child is on the closed list

for closed\_child in closed\_list:

if child == closed\_child:

continue

# Create the f, g, and h values

child.g = current\_node.g + 1

child.h = ((child.position[0] - end\_node.position[0]) \*\* 2) + \

((child.position[1] - end\_node.position[1]) \*\* 2)

child.f = child.g + child.h

# Child is already in the open list

for open\_node in open\_list:

if child == open\_node and child.g > open\_node.g:

continue

# Add the child to the open list

open\_list.append(child)

if \_\_name\_\_ == '\_\_main\_\_':

board = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]]

start = (0, 0)

end = (6, 6)

path = astar(board, start, end)

print(path)

**Output:**

****