A* Algorithm for Path Planning

AI: Foundations and Applications Assignment
AI61005, Autumn, 2021

Assignment Final Report

Group Members

Keerthi Sree Marrapu	20MF10015
Sudeshna Bose	21MM61R04
Pendem Ganesh	20ME10061

Interim Report

AI: Foundations and Applications Assignment
AI61005, Autumn, 2021

Problem Statement

Problem Description

To build an algorithm, which can be possibly used in autonomous vehicles, to find the optimum path considering static and real-time obstacles assuming that the map and all possible paths are known beforehand. Taking that real-time obstacle can be detected only within a specific range from the vehicle.

Formal Problem Statement

We have a Beginning Position(Starting Position) an End Position(Goal). In today's world, we widely use GPS (Global Positioning System). Hence, we are aware of the obstacles that are there in the path from the initial to the final position. However, there is a possibility that obstacles might come in the path, which we did not know previously. Examples of such barriers can be roadblocks, construction work, any accident that has taken place etc.

Inputs:

initial position
goal position
obstacles (both predefined and defined at a later stage)

Output:

We have to find the best way from the initial position to the final position or Goal State.

Background

For centuries self-driving cars have captivated human minds. The origin of the Autonomous Vehicles (AVs) journey can be traced back to the late 1400s when Leonardo da Vinci prescribed a path for a self-propelled cart. Sperry Gyroscope Co. developed a prototype autopilot for Post’s flight around the world in 1933. Henceforth several inventions such as Cruise Control (Ralph Teetor (Dana, Inc.), 1945-1958), Vehicle-Mounted Camera (James Adams and Les Earnest (Stanford Univ.) – Stanford Cart, 1961-71), Dynamic Vision (Ernst Dickmanns – VaMoR, 1987) and the currently being widely used technology LIDAR (1960s-present) were made. With these advancements in technology, global positioning (GPS), computing power, digital mapping, AI and sensor systems, we can now bring AVs into a reality. This digitalisation of vehicles brings in the need for newer technology and skills.

Importance

Self-driving vehicles, a quintessentially 'smart' innovation, are not conceived intelligent. The calculations that control their developments are learning as the innovation arises. Self-driving vehicles address a high-stakes trial of the forces of AI, just as an experiment for social learning in innovation administration. Society is finding out with regards to the innovation while the innovation finds out with regards to society. Understanding and overseeing the governmental issues of this innovation implies asking, 'Who is realizing, what are they realizing, and how are they picking up?' Focusing on the victories and disappointments of social learning around the much-pitched accident of a Tesla Model S in 2016, it can be contended that directions and talking points of AI in transport represent a significant administration challenge. Self-driving or 'independent' vehicles are incorrectly named. Likewise, they are formed by suppositions about friendly necessities, practical issues, and monetary freedoms with different advancements. Overseeing these advancements in the public premium means working on social learning by valuably captivating the possibilities of AI.

AI Mapping

Since the problem has to address issues posed by real-time obstacles that cannot be determined beforehand until reaching a specific range while some domain knowledge, such as goal state, the vehicles' correct location, is known, we have decided to take up A* to solve this problem.

Programming Language to be used: **Python 3**
Platform being used for visualising the code: **pygame**

A* Algorithm

Each Node n in the algorithm has a cost $g(n)$ and a heuristic estimate $h(n)=g(n)+h(n)$

Assume all $c(n,m)>0$

1. [Initialize] Initially the OPEN List contains the Start Nodes. $g(s)=0$, $f(s)=h(s)$.CLOSED list is Empty.
2. [Select] Select the Node n on the OPEN List with minimum $f(n)$. If OPEN is empty. Terminate with Failure.
3. [Goal Test, Terminate] If n is Goal, then Terminate with Success and path from s to n.
4. [Expand]
    a. Generate the successors n_1,n_2,.........n_k, of node n, based on the State Transformation Rules.
    b. Put n in LIST CLOSED
    c. For each n_i, not already in OPEN or CLOSED List,compute

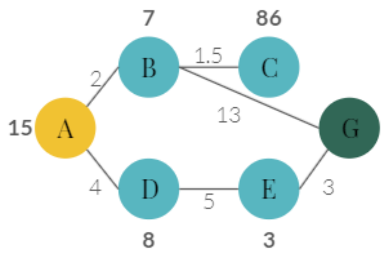
       i) $g(n\_i)=g(n)+c(n,n\_i)$,$f(n\_i)=g(n\_i)+h(n\_i)$, Put n_i in th OPEN list

    d. For each n_i, not already in OPEN, if $g(n\_i)>g(n)+c(n,n\_i)$, then revise cost as:

5. [Continue] Go to Step 2.

A* Algorithm Example

f(n)=g(n)+h(n) where,
n = next node on the path
g(n) = **cost of the path** from beginning node to n
h(n) = **heuristic function** estimating cost of cheapest path from n to the goal



1. A = 0 + 15 = 15
2. A → B = 2 + 7 = 9
   A → D = 4 + 8 = 12
3. A → B → C = (2 + 1.5) + 86= 89.5
   A → B → G = (2 + 13) + 0 = 15
4. A → D → E = (4 + 5) + 3 = 12
5. A → D → E → G = (4 + 5 + 3) + 0 =12

Hence, the most cost efficient path is:

A → D → E → G = (4 + 5 + 3) + 0 =12

Why A* Algorithm is good?

A* is the most well-known decision for pathfinding because it is genuinely adaptable and can be utilized in a broad scope of contexts. Pathfinder calculations like A* assist with arranging things as opposed to delaying until we find the problem. They act proactively as opposed to responding to a circumstance. A* can have as an answer a hub that it has chosen for development; it is ideal. Moreover, A* is usually more preferable over Dijkstra (that Google Maps uses), as it performs informed and not uninformed searches. A* expands more promising vertices.

Solution Approach

Code that we have worked on is as follows: (Comments are added to explain each part)

```
#A* Algorithm for Path Planning

#Team Members = [(Keerthi Sree Marrapu, 20MF10015),(Sudeshna Bose, 21MM61R04), (Pendem
Ganesh, 20ME10061)]

import math
import pygame       #Using pygame environment to visualise A star Algorithm
import time         #To delay the program for better understanding of the working of the
code
from queue import PriorityQueue  #Efficient way to find the minimum element
```

```
#Global Variables
SPAN = 800
ROWS = 60
SCREEN = pygame.display.set_mode((SPAN, SPAN))
pygame.display.set_caption("A* Algorithm for Path Planning")
```

Here we are specifying the colors:
Yellow = Initial position
Green = Goal position
Black= Obstacles in the Path

```
#Specifying Color codes
BEGIN_C = (227, 180, 72)
END_C = (42, 161, 15)

FREET_C = (255, 255, 255)
GRID_C = (238, 237, 231)
OBSTACLE_C = (0, 0, 0)

CLOSEDT_C = (210, 43, 43)
OPENT_C = (144, 238, 144)

FINALP_C = (0, 48, 96)
```

This class job is to keep track of the colors and what location they are in, is it a start node or obstacle etc.

```
class Tile:
      def __init__(self, row, col, span, total_rows):
            #Defining variables corresponding to each tile
            self.row = row
            self.col = col
            self.x = row * span
            self.y = col * span
            self.color = FREET_C
            self.span = span
            self.total_rows = total_rows

      #Define the colors
      def mark_begin(self):
            self.color = BEGIN_C

      def mark_end(self):
            self.color = END_C
```

```
    def mark_obstacle(self):
        self.color = OBSTACLE_C

    #Find position of the vehicle
    def get_pos(self):
        return self.row, self.col

    #Tiles indicating their staus - whether they are in open list / closed list
    def mark_open(self):
        self.color = OPENT_C

    def mark_closed(self):
        self.color = CLOSEDT_C

    #Final path color
    def create_path(self):
        self.color = FINALP_C

    #Checking if the given tile is an obstacle. If obstacle, function returns TRUE
    def check_obstacle(self):
        return self.color == OBSTACLE_C

    #Reset grid
    def reset(self):
        self.color = FREET_C

    #Drawing the grid
    def draw(self, screen):
        pygame.draw.rect(screen, self.color, (self.x, self.y, self.span, self.span))
```

Adjacent tiles will only have those tiles which can be taken, i.e. , it removes tiles that are obstacles.

```
#adj_tiles will only have those tiles which can be taken, i.e., it removes tiles that are 
obstacles
    def update_adj_tiles(self, grid):
        self.adj_tiles = []

        #Moving Forwards
        if self.col < self.total_rows - 1 and not grid[self.row][self.col + 
1].check_obstacle():
            self.adj_tiles.append(grid[self.row][self.col + 1])
        #Moving Backwards
        if self.col > 0 and not grid[self.row][self.col - 1].check_obstacle():
            self.adj_tiles.append(grid[self.row][self.col - 1])
        #Moving Upwards
        if self.row > 0 and not grid[self.row - 1][self.col].check_obstacle():
```

```
                    self.adj_tiles.append(grid[self.row - 1][self.col])
            #Moving Downwards
            if self.row < self.total_rows - 1 and not grid[self.row +
1][self.col].check_obstacle(): # DOWN
                    self.adj_tiles.append(grid[self.row + 1][self.col])
```

We are going to define a Heuristic Function for our Algorithm

```
#Gives the heuristic estimate of a node
def dist(t1, t2):
        x1,y1 = t1
        x2,y2 = t2
        #Manhattan Distance is being returned (since we are considering our vehicle cannot
traverse diagonally)
        return abs(x1-x2)+abs(y1-y2)
```

We need a Data structure to hold all of these tiles, so that we can use them.

```
def algorithm(draw, grid, begin, end):
        count = 0
        open_list = PriorityQueue()
        open_list.put((0, count, begin)) #First parameter here is the f(x)=0

        #To keep track of which tile we came from,
        from_tile = {}

        #We initialise all unexplored tiles with g and h values to be infinity, assuming
that it takes infinity to reach there

        gValue = {tile: float("inf") for row in grid for tile in row}
        gValue[begin] = 0  #At starting node, g(x) is zero

        fValue = {tile: float("inf") for row in grid for tile in row}
        #f(x) = h(x)+g(x)
        #h(x)=heuristic function=distance between beginning tile and end tile
        fValue[begin] = dist(begin.get_pos(), end.get_pos()) + gValue[begin]

        #To keep track of what items are there in open_list, making a set
        open_list_set = {begin}

        while not open_list.empty():

                #To quit the loop,
                for event in pygame.event.get():
                        if event.type == pygame.QUIT:
                                pygame.quit()
```

```
            present_tile = open_list.get()[2] #To get the tile parameter from open_list
            open_list_set.remove(present_tile) #To remove duplicates

            if present_tile == end:
                  rebuild_path(from_tile, end, draw)
                  end.mark_end()
                  #time.sleep(0.8)
                  return True

            for adj_tile in present_tile.adj_tiles:
                  #Adding 1 because we are assuming the distance between a tile and its 
adjacent tile is 1
                  tent_gValue = gValue[present_tile] + 1

                  #Finding the minimum g value and keeping track of that path
                  if tent_gValue < gValue[adj_tile]:
                        from_tile[adj_tile] = present_tile
                        gValue[adj_tile] = tent_gValue

                        #f(n)=g(n)+h(n)
                        fValue[adj_tile] = tent_gValue + dist(adj_tile.get_pos(), 
end.get_pos())

                        #If the adj_tile has lesser f value, and is not in the 
open_list, then we increase count and we add it to the list
                        if adj_tile not in open_list_set:
                              count += 1
                              open_list.put((fValue[adj_tile], count, adj_tile))
                              open_list_set.add(adj_tile)
                              adj_tile.mark_open()
                              #time.sleep(0.8)

            draw()

            if present_tile != begin:
                  present_tile.mark_closed()

      return False
```

```
def rebuild_path(from_tile, present_tile, draw):
      while present_tile in from_tile:
            present_tile = from_tile[present_tile]
            present_tile.create_path()
            #time.sleep(0.5)
            draw()
```

Building the required grid map

```
#Building the required grid map
def build_grid(rows, span):
      grid = []
      width = span // rows
      for i in range(rows):
            grid.append([])
            for j in range(rows):
                  tile = Tile(i, j, width, rows)
                  grid[i].append(tile)

      return grid

def draw_grid(screen, rows, span):
      width = span // rows
      for i in range(rows):
            pygame.draw.line(screen, GRID_C, (0, i*width), (span, i*width))

            for j in range(rows):
                  pygame.draw.line(screen, GRID_C, (j*width, 0), (j*width, span))

def draw(screen, grid, rows, span):
      screen.fill(FREET_C)

      for row in grid:
            for tile in row:
                  tile.draw(screen)

      draw_grid(screen, rows, span)
      pygame.display.update()
```

Returning the position of tiles

```
#returns the position of the tiles
def get_clicked_pos(loc, rows, span):
      width = span // rows
      j, i = loc

      row = j // width
      col = i // width

      return row, col
```

main() of the program

```
def main(screen, span):

      grid = build_grid(ROWS, span)
```

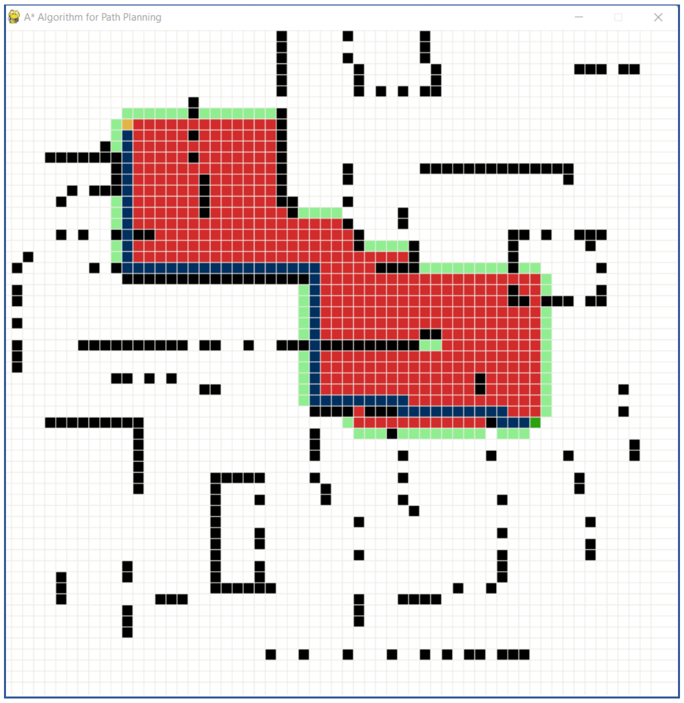
```
    begin = None
    end = None

    run = True
    while run:

        #draws the grid
        draw(screen, grid, ROWS, span)

        #Ends running when we click the 'x' button
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False

            #Using left mouse button to define the beginning, ending tiles and
obstacles
            if pygame.mouse.get_pressed()[0]: # Left mouse button
                loc = pygame.mouse.get_pos()
                row, col = get_clicked_pos(loc, ROWS, span)
                tile = grid[row][col]

                #Colors the tiles to mark the beginning and the ending tiles
                if not begin and tile != end:
                    begin = tile
                    begin.mark_begin()
                elif not end and tile != begin:
                    end = tile
                    end.mark_end()

                #Colors the obstacles
                elif tile != end and tile != begin:
                    tile.mark_obstacle()

            #Using right click to reset the definitions
            elif pygame.mouse.get_pressed()[2]: # Right mouse button
                loc = pygame.mouse.get_pos()
                row, col = get_clicked_pos(loc, ROWS, span)
                tile = grid[row][col]
                tile.reset()
                if tile == begin:
                    begin = None
                elif tile == end:
                    end = None

            if event.type == pygame.KEYDOWN:

                #Click enter to start the algorithm
```

```
                                if event.key == pygame.K_RETURN and begin and end:
                                        for row in grid:
                                                for tile in row:
                                                        tile.update_adj_tiles(grid)

                                        algorithm(lambda: draw(screen, grid, ROWS, span), grid,
begin, end)

                                #To reset screen
                                if event.key == pygame.K_r:
                                        begin = None
                                        end = None
                                        grid = build_grid(ROWS, span)

        pygame.quit()

main(SCREEN, SPAN)
```

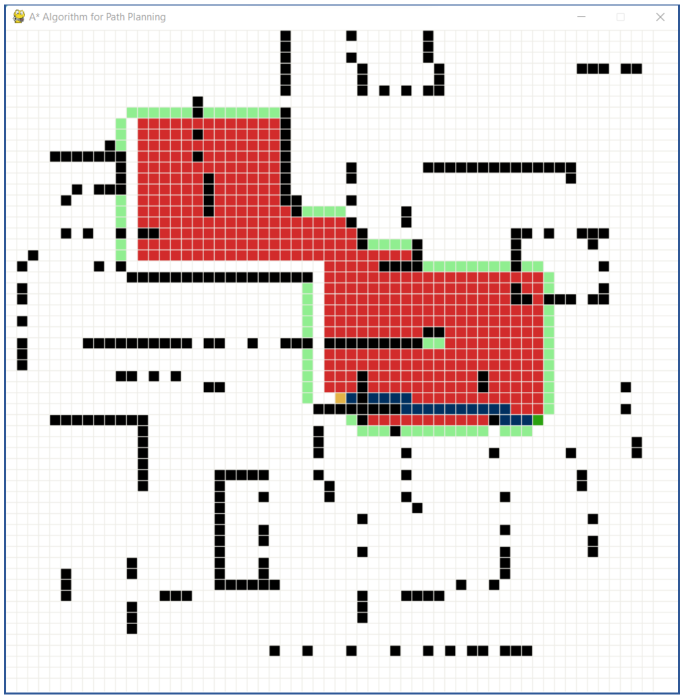
Github Link:
https://github.com/MKSree066/AIFA_Assignment_21/commit/15d15cf49c54e575a7ce342a32d9c9e3c0eb5d61

Youtube Link:
AIFA Group Project | A* Algorithm for Path Planning | pygame platform |

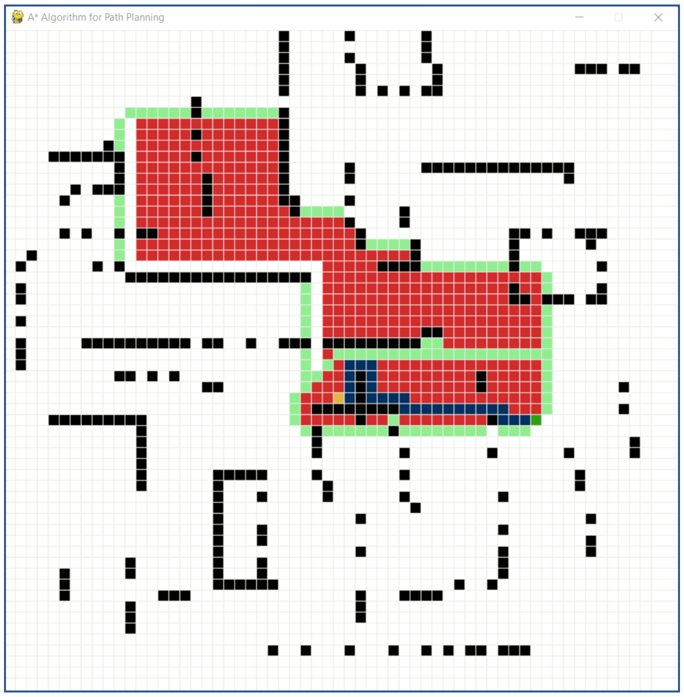
Working of Code (Visualisation snaps)



Picture 1: Given the initial position, final position and obstacles, the best path has been made.



Picture 2: In case the vehicle encounters an obstacle after traversing some of the path…



Picture 3: The vehicle has to change its initial state and run the algorithm once more to find the optimum path

Future Prospects

The Algorithm can be modified into Hybrid A*, thereby considering vehicle dynamics as well.

Object localisation and motion prediction can be made using Machine Learning Algorithms.

Further Scopes of Research

- Deep learning for collision detection/traffic prediction in IoV
- New deep learning approaches to steering angle control in autonomous vehicles.
- Convolutional neural networks for vision and environment perception machine Learning/graph learning for autonomous vehicles
- Meta-heuristic algorithms for Deep Learning in vehicle to sensor communication for IoV
- Multi-Criteria for reinforcement learning for vehicle to vehicle communication

References

1. What are the most important technologies that led to autonomous transportation?
2. https://www.mygreatlearning.com/blog/a-search-algorithm-in-artificial-intelligence/
3. python.org
4. pygame.org