KATHMANDU UNIVERSITY

Department of Mathematics

Dhulikhel, Kavre

A Project Report on

"Path Finding in Graph Data Structure"

[Code No.: COMP 202]

(For partial fulfillment of 2nd Year/ 1st Semester in Computational Mathematics)

Submitted by:

Aayush Shrestha (21)

Prashant Timalsina (29)

Submitted to:

Dr. Prakash Poudyal

Department of Computer Science and Engineering

Submission Date: _____

Acknowledgement

We would like to thank our course instructor as well as our project supervisor, **Dr Prakash Poudyal,** for his consistent support and feedback throughout this project. His feedback and suggestions helped direct the project to its completion. We would also like to extend our appreciation to Group A members, **Aayush Shrestha** and **Prashant Timalsina** for their time and dedication which led the project to fruition. Lastly, we would like to thank Kathmandu University, the Department of Mathematics and the Department of Computer Science & Engineering and everyone else for their direct and indirect role to support us in this project.

Abstract

A graph in the data structure is a non-linear data structure that comprises a finite number of nodes/ data and edges/links between nodes. It is used in real life to represent social networks, road connections etc.

Pathfinding or pathing is the plotting, by a computer application, of the shortest route between two points. It is a more practical variant of solving mazes.

This field of research is based heavily on Dijkstra's algorithm for finding the shortest path in a weighted graph and A * on an unweighted graph. Along with the two mentioned algorithms, the program includes an Ant Colony Optimization algorithm to create a Hamiltonian circuit in the given complete graph

The program is created using Python programming language and mainly uses the data structure of graphs via adjacency matrix, queues, and arrays. It is expected that we successfully create a program that can create an interactive grid, have a manual selection of the nodes and algorithm to use and finally display a path for the given selection.

Keywords: Graph, Nodes, Weighted Graph, Unweighted Graph, Complete Graph, Hamiltonian Circuit, Adjacency Matrix

TABLE OF CONTENTS

Contents

List of Figures

Figures **Page No.**

List of Tables

Tables **Page No.**

List of Acronyms / Abbreviations

1. OS: Operating System
2. UI: User Interface
3. ACO: Ant Colony Optimization
4. A*: A star or A search

Chapter 1: Introduction

1.1. Background

A graph in the data structure is the collection of nodes and edges between nodes that describe their link or relation. In simple words, a graph consists of nodes where each node contains a data field and a reference(link) to the other node to describe a relation. The relation can be directed, undirected, weighted or unweighted or no relation at all(disjoint).

Pathfinding or pathing is the plotting, by a computer application, of the shortest route between two points. Pathfinding is closely related to the shortest path problem, within graph theory, which examines how to identify the path that best meets some criteria (shortest, cheapest, fastest, etc.) between two points in a large network.

The Dijkstra algorithm finds the shortest path quickly in scenarios where efficiency isn't the main priority. A* is used pretty commonly in pathfinding as it is a good mix of efficiency and quickness. Lastly, Ant Colony Optimization (ACO) algorithm is a relatively recent algorithm used mainly as a solution to the Travelling Salesman Problem. Unlike the other two algorithms, ACO doesn't guarantee the shortest path but rather on average it provides a relatively short path. As it is a probabilistic approach.

1.2. Objectives

- To create an Interactive grid, capable of selecting different kinds of nodes.
- To be able to offer a choice of algorithm to be used
- To create a visualization for working of the algorithms
- To add the functionality of clearing the grid, going back to the interface or reperforming the algorithm.

1.3. Motivation and Significance

The algorithms that we implemented have a real application in the field of map traversal. It is a fairly common approach that is used by companies like Google in their google map feature. The code we have implemented as proof of the concept of finding the optimal path for different scenarios can be used in maze solving, pathfinding, file search etc.

Chapter 2: Design and Implementation

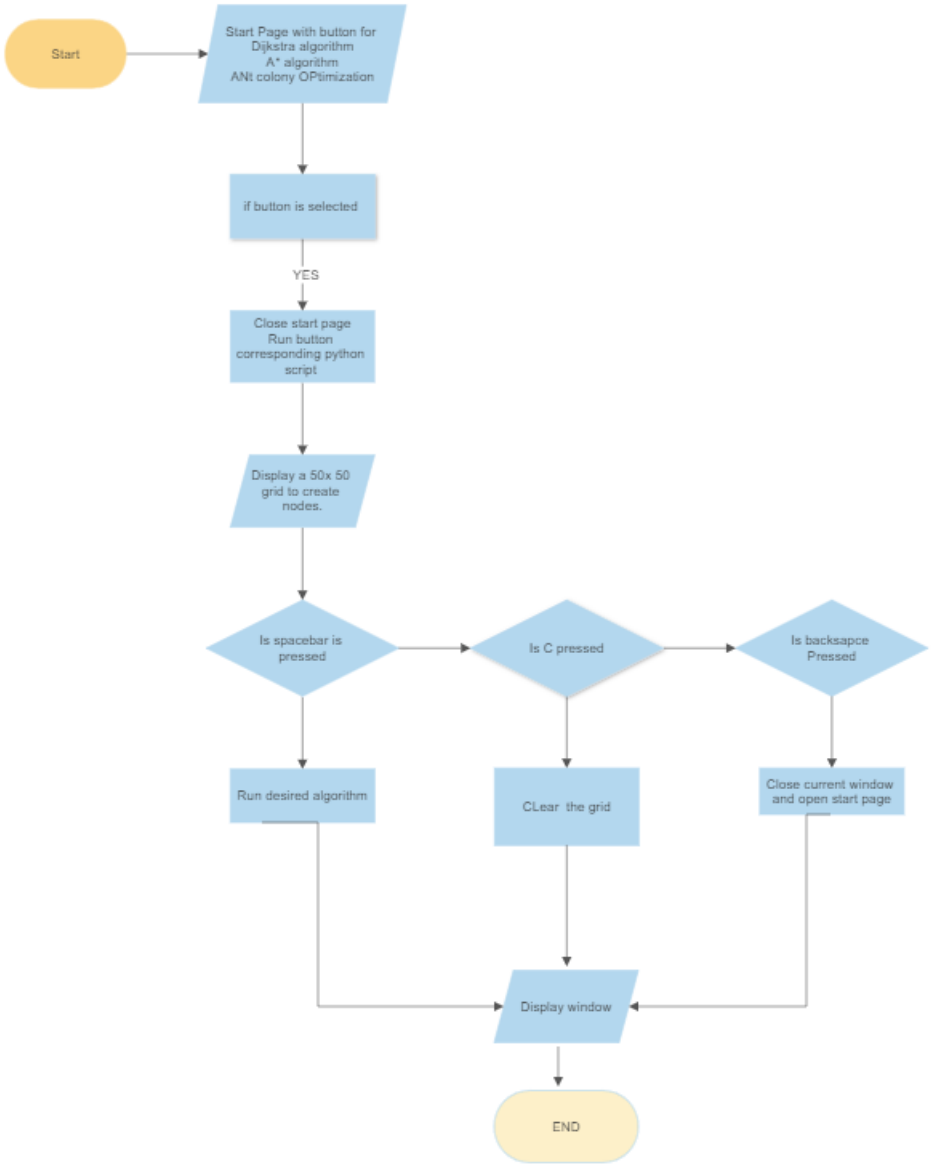

Fig 1: Flowchart of the Program

Upon starting the program, on the starting page, the user is presented with buttons corresponding to the 3 algorithms implemented in this project, that is, Ant Colony Optimization, A* algorithm and Dijkstra Algorithm. Once the user presses a button, the starting page will terminate and the corresponding grid appears. Based on the Algorithm used, nodes can be selected as For Dijkstra; right mouse button and mouse movement for endpoint, holding left mouse button and mouse movement for walls. For A*; the first left mouse button click gives the start node, the second left mouse button gives the end node, and the mouse movement while holding the left mouse button gives walls. Finally, for Ant colony optimization, the left mouse button selects the nodes. All grids are 50x50 box grids in a 700x700 pixel window and have key-related features such as; Spacebar to run the algorithm, C to clear the grid and backspace to return to the starting page. In the ant colony algorithm, the number of nodes, ants and generation is set at 7 for each but can be changed in the program script file using the variable point, ants and gener respectively.

Fig 2: Starting Page

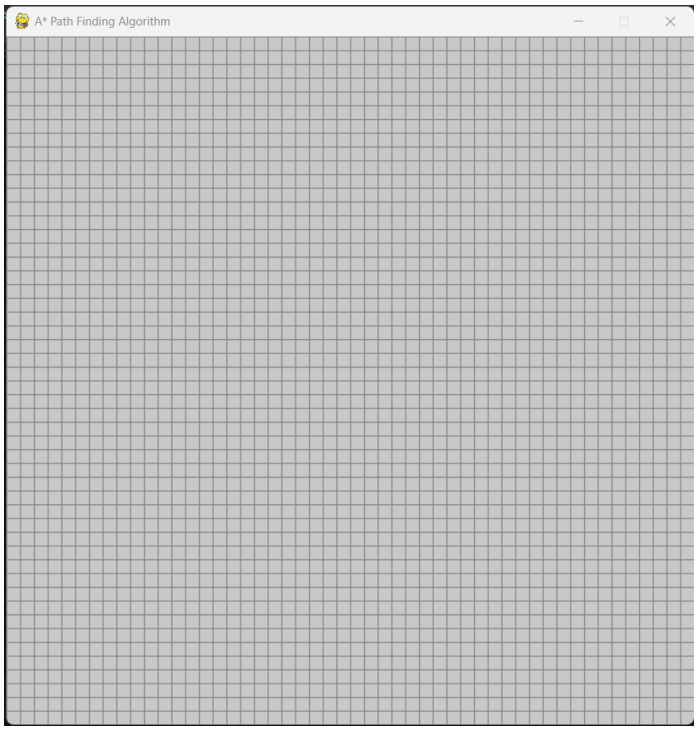

Fig 3: Starting Grid For A*

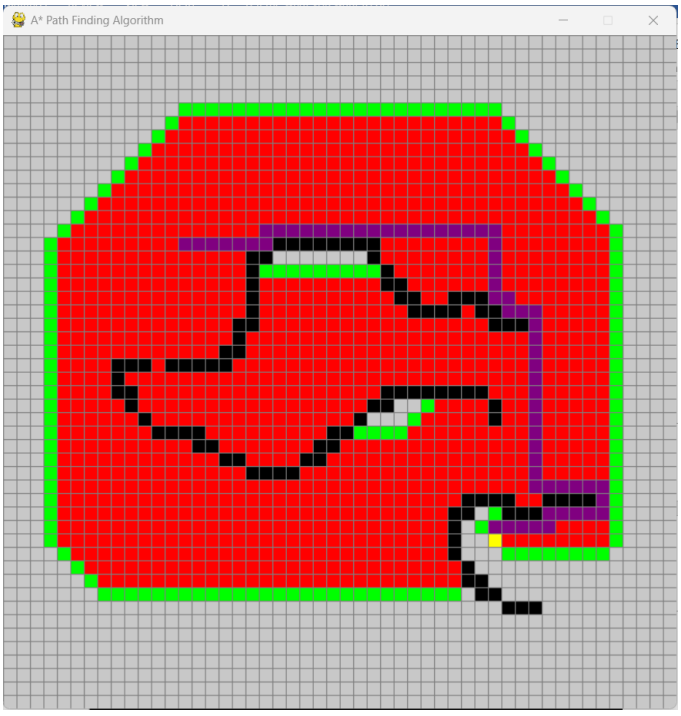

Fig4: Path Visualization in A *

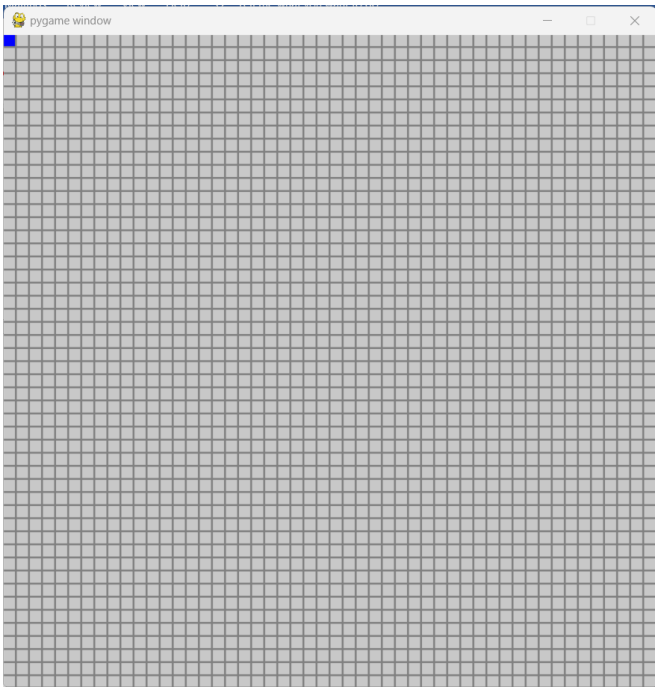

Fig 5: Starting Grid for Dijkstra

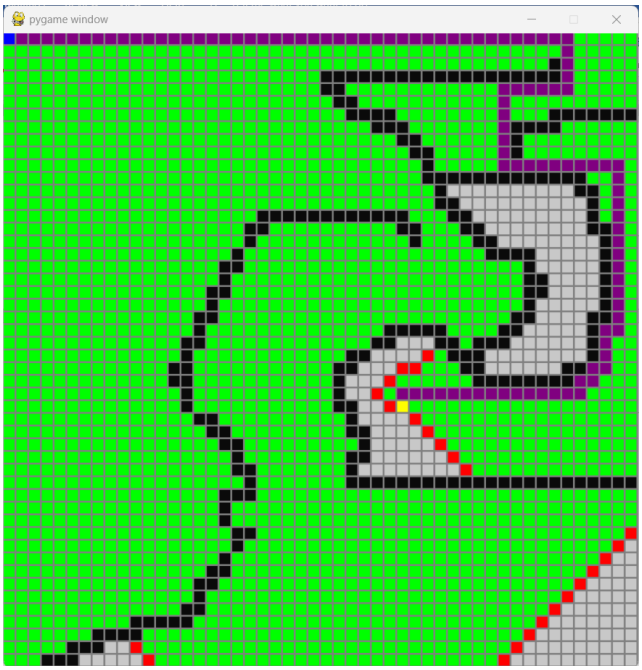

Fig 6: Path Visualization in Dijkstra

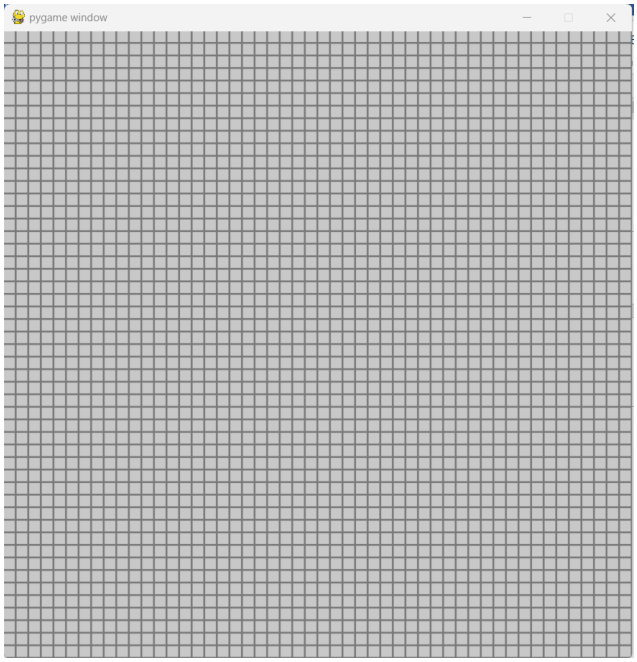

Fig 7: Starting Grid for ACO

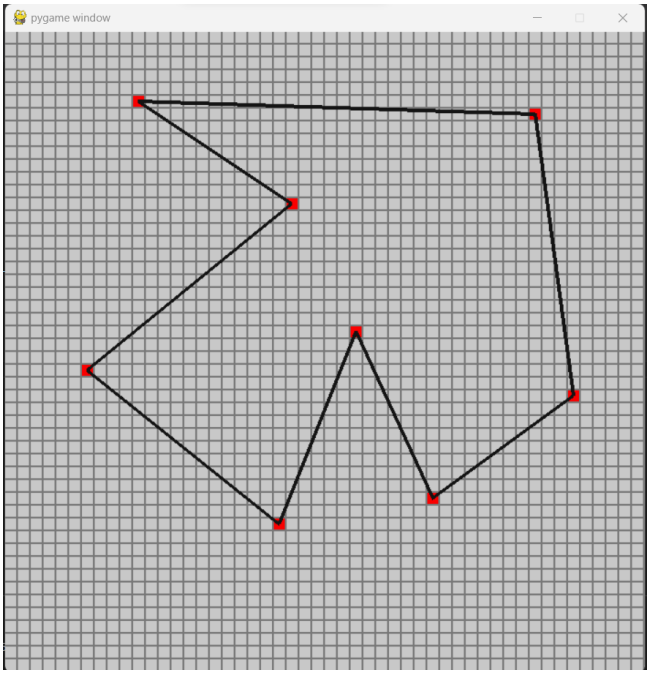

Fig 8: Path Visualization in ACO

The flowchart of the program is shown above and the general flow of program is described. The screenshots of the running program are also attached above. And the complete version of the project program is hosted in our GitHub which can be visited via GitHub.

2.1. System Requirement Specification

2.1.1. Software Specification

2.1.1.1. **Interpreter:** Python 3 (version 3.7 recommended)

2.1.1.2. **OS:** Windows XP and above or similar Linux

2.1.1.3. **Python Package:** pygame, NumPy, random, os, queue, math

2.1.2. Hardware Specification:

The minimum requirements of hardware needed for this program are:

Processor: Any Intel Core processor or similar
Storage: 30 KB (.exe excluded) of available free space
RAM: 500 MB or above

Chapter 3: Code and Explanation

For the initial start page, we used the pygame package and its pygame widgets feature to create a interactive window with 3 buttons. When the button is pressed the start window is closed and the corresponding python script is sent to the system for execution.

```
# Creates the button with optional parameters
button1 = Button(
    # Mandatory Parameters
    win,  # Surface to place button on
    200,  # X-coordinate of top left corner
    100,  # Y-coordinate of top left corner
    300,  # Width
    100,  # Height

    # Optional Parameters
    text='Ant Colony',  # Text to display
    fontSize=50,  # Size of font
    margin=20,  # Minimum distance between text/image and edge of button
    inactiveColour=(200, 50, 0),  # Colour of button when not being interacted with
    hoverColour=(150, 0, 0),  # Colour of button when being hovered over
    #pressedColour=(0, 200, 20),  # Colour of button when being clicked
    radius=20,  # Radius of border corners (leave empty for not curved)
    onClick=lambda: now(1)  # Function to call when clicked on
)
```

Fig 9: Python code for Pygame button.

```
def now(x):
    pygame.quit()
    if x == 1 :
        os.system('python ACO2.py')
    elif x == 2 :
        os.system('python A_star.py')
    elif x == 3 :
        os.system('python Dijkstra.py')
# Set up Pygame
pygame.init()
win = pygame.display.set_mode((700, 700))
```

Fig 10: Function to run corresponding to the button

3.1. Dijkstra Algorithm

In Dijkstra Algorithm, the node at the top (0,0) is preset as the start node. And if the Right mouse button is pressed when the mouse is in motion the first node it hovers over is now the target node. Lastly, if the Left mouse button is pressed when the mouse is in motion all the node it hovers over is now the wall. A class named box is created and customized for this algorithm to store the required information of the node.

```
class Box:
    def __init__(self, i, j):
        self.x = i
        self.y = j
        self.start = False
        self.wall = False
        self.target = False
        self.queued = False
        self.visited = False
        self.neighbours = []
        self.prior = None

    def draw(self, win, color):
        pygame.draw.rect(win, color, (self.x * box_width, self.y * box_height, box_width-2, box_height-2))

    def set_neighbours(self):
        if self.x > 0:
            self.neighbours.append(grid[self.x - 1][self.y])
        if self.x < columns - 1:
            self.neighbours.append(grid[self.x + 1][self.y])
        if self.y > 0:
            self.neighbours.append(grid[self.x][self.y - 1])
        if self.y < rows - 1:
            self.neighbours.append(grid[self.x][self.y + 1])
```

Fig 11: Python Code For class Box

Once the necessary nodes are selected, algorithm can be run by pressing the Spacebar key. When Dijkstra's algorithm picks which path to expand next, it looks at all of the paths it has seen so far, and picks the path with the shortest total length.

```
if begin_search:
    if len(queue) > 0 and searching:
        current_box = queue.pop(0)
        current_box.visited = True
        if current_box == target_box:
            searching = False
            while current_box.prior != start_box:
                path.append(current_box.prior)
                current_box = current_box.prior
        else:
            for neighbour in current_box.neighbours:
                if not neighbour.queued and not neighbour.wall:
                    neighbour.queued = True
                    neighbour.prior = current_box
                    queue.append(neighbour)
```

Fig 12: Dijkstra algorithm implementation

3.2. A* Algorithm

Dijkstra's algorithm has one big downside: it expands the shortest path, regardless of how close that path gets to the destination (LibGDX Pathfinding, 2018). To solve this issue, we implement the A * algorithm which also knows the end point, unlike the Dijkstra algorithm. The point to be noted here is that we can't know for certain that the distance we get is same as how far we are. So instead of using actual distance, we use heuristics, simply an educated guess. And depending upon the Heuristics used, A* may not always give the shortest path but the path that seems relatively short and feasible while being effective

```python
def algorithm(draw, grid, start, end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}
    g_score = {spot: float("inf") for row in grid for spot in row}
    g_score[start] = 0
    f_score = {spot: float("inf") for row in grid for spot in row}
    f_score[start] = h(start.get_pos(), end.get_pos())

    open_set_hash = {start}

    while not open_set.empty():
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()

        current = open_set.get()[2]
        open_set_hash.remove(current)

        if current == end:
            reconstruct_path(came_from, end, draw)
            end.make_end()
            return True

        for neighbor in current.neighbors:
            temp_g_score = g_score[current] + 1

            if temp_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = temp_g_score
                f_score[neighbor] = temp_g_score + h(neighbor.get_pos(), end.get_pos())
                if neighbor not in open_set_hash:
                    count += 1
                    open_set.put((f_score[neighbor], count, neighbor))
                    open_set_hash.add(neighbor)
                    neighbor.make_open()
```

Fig 13: Python code for A * algorithm

3.2.1 Heuristics

What A* Search Algorithm does is that at each step it picks the node according to a value-'f' which is a parameter equal to the sum of two other parameters – 'g' and 'h'. At each step it picks the node/cell having the lowest 'f', and process that node/cell.
We define 'g' and 'h' as simply as possible below
g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls). (*A* Search Algorithm - GeeksforGeeks*, 2016).

For our program, we used the Manhattan distance as an approximate heuristic which takes the absolute value in the difference of x and the absolute value of the difference in y.

```
def h(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return abs(x1 - x2) + abs(y1 - y2)
```

Fig 14: Function for returning Manhattan distance.

3.3. Ant Colony Optimization

Ant Colony Optimization is a graph traversal algorithm that is made in accordance with how ants create a path from their base to a source of food or danger and back to their nest. Here, in our program, we implemented our version of ACO. First, the user selects some nodes. The number of nodes, ant and generation is predefined in the script code.

Once all the nodes are selected, an adjacency matrix is created to store the value of the distance between the two nodes corresponding to the row and column index. two copy of this matrix is made which will act as our preference matrix and temporary matrix for each ant.

All the ants start at the same node which is the node that was first selected. For every current node, the order of nodes to then be travelled is then shuffled randomly. A random number is generated between 1 and X. and the first unvisited node in the shuffled order whose distance value is less than the random value and greater than 0 is then selected as the new current node. The index of the previous node is stored then in an array. And all the distance to and from the previous current node to any other node is then turned 0.

Here X is the visibility variable which starts at 10 and then is incremented by 10 for every loop where a new current node is not found. Its max possible value for our program is set at 100.

Once a path is found by every ant in a generation, the shortest path among them is found using a linear search. And the edges in that path have their preference value reduced by 10% which will in turn increase their probability of being selected.

```
if event.type == pygame.KEYDOWN :
    if event.key == pygame.K_SPACE and len(NOD)>=point:
        random.shuffle(NOD)
        for i in range(0,point-1):
            for j in range(i+1,point):
                dist[i][j] = dist[j][i]=(((NOD[i].x-NOD[j].x)**2) +  (NOD[i].y-NOD[j].y)**2)**.5
                pref[i][j] = pref[j][i] = dist[i][j] //1
       # print(pref)
        for gen in range(0, gener ):
            Astart(gen)
            print(parr[:,ant])
        begin_search=True
```

Fig 15: Python code to create weighted adjacency matrix.

```
def Astart(gen):
    i,j,n,n1,x=0,0,0,0,0
    a= list(range(1,point))
    global parr
    parr = np.zeros((point+2,ant+1))
    for v in range(0,ant):
        p_temp=pref.copy()
        n=0
        while n!=point-1:
            random.shuffle(a)
            if n1==n and x<100:
                x=x+10
            n1=n
            for j in a:
                rand =  np.random.randint(low=5, high= x)
                if p_temp[i][j]<=rand and  p_temp[i][j] !=0:
                    parr[n+1][v]=j
                    if n!=0:
                        p_temp[:,i]=p_temp[i,:]=0
                    n=n+1
                    w=dist[i][j]
                    parr[point+1][v] +=w
                    i=j
                    break
        parr[n+1][v]=0
        p_temp[i,:]=0
        p_temp[:,i]=0
        parr[point+1][v] +=dist[i][j]
    l= parr[point+1][0]
    m=0
    for k in range(1,ant):
        if l>parr[point+1][k]:
            m=k
    parr[:,ant]= parr[:,m]
```

Fig 16: Python Code for ACO algorithm

3.4. UI

The UI is made as similar as possible for all 3 algorithms. Starting with the Color theme as is visible in the figures above. Also, we have added the following functionality.

Pressing the spacebar will run/re-run the algorithm. Pressing the c key will clear the grid and pressing the backspace key will close the current algorithm window and open the start page.

```
if event.type == pygame.KEYDOWN :
     if event.key == pygame.K_SPACE and len(NOD)>=point:
          #begin algorithm
          begin_search=True
     elif event.key == pygame.K_c:
          # restart the grid
          main()
     elif event.key == pygame.K_BACKSPACE:
          # go to the start page
          pygame.quit()
          os.system('python all.py')
```

3.5. Time Complexity

In computer science, time complexity is a very essential computational complexity that describes the tentative amount of computer time it takes to run an algorithm. Since an algorithm's running time may vary due to differences in OS or processor, inputs of the same size, commonly the worst-case time complexity is considered. In cases, where the worst case happens rarely or only with a specific condition, we use the average time complexity The time complexity of algorithms is most commonly expressed using the big O notation. It's an asymptotic notation to represent the time complexity. The Big O notations for the different algorithms can be seen below:

Here V is the number of nodes or vertices, E is the number of edges, b is the average number of edges from each node, d is the number of nodes in the resulting nodes, A is the number of Ants, G is the generations.

Algorithm	Time Complexity	
	Average Case	Worst Case
Dijkstra	$O(V^2)$	$O((\|V\|+\|E\|) * \log\|V\|)$
A*	O(E)	$O(b^d)$
ACO	$O(AGV^2)$	infinity

Table 1: Time Complexity of the used algorithms

The worst time complexity of ACO being infinite is a rare case when the random number is always less than any weightage of the respective edge.

Chapter 4: Discussion on the Achievements

We had a great time working together on this project and experienced a lot of challenges. We tried to research and implement the algorithms that were outside the bounds of our course syllabus and we were successful in doing that. We learned and understood the basics of UI using the pygame package. The running program which works as plan is finally implemented.

Features:

1. **Starting page**: A page to select which algorithm to use
2. **Interactive grid:** Nodes can be manually selected with different attributes
3. **Visualization:** Different color for different type of nodes and display of path.
4. **Key based command:** Different functionality based on the key pressed can be observed.

Chapter 5: Conclusion and Recommendation

The code shown above was used to create a program with application of different path finding algorithm in graph, with flow of the program as shown and the output of the program as shown in the screenshots. This project helps to understand the concept of graph and the issues related to its traversal and its representation. The concept of data structures and algorithms successfully implemented to develop this program.

5.1. Limitations

1. All the files (4 .py files) have to be in the same folder.
2. Lack of a .exe file.
3. There is no feature to pause/ replay the algorithm.
4. In the ACO, the same node can be selected twice which causes an error.
5. Pressing Spacebar to replay algorithm in Dijkstra doesn’t work.

References:

1. *LibGDX Pathfinding*. (2018, November 3). Happy Coding. https://happycoding.io/tutorials/libgdx/pathfinding
2. *A* Search Algorithm - GeeksforGeeks*. (2016, June 16). GeeksforGeeks. https://www.geeksforgeeks.org/a-search-algorithm

Bibliography:

- Cox, G. (2020, August 13). *A* Pathfinding Algorithm | Baeldung on Computer Science*. Baeldung on Computer Science. https://www.baeldung.com/cs/a-star-algorithm
- Prasanna. (2022). *A* Search Algorithm*. Enjoyalgorithms.com. https://www.enjoyalgorithms.com/blog/a-star-search-algorithm
- *A* Search Algorithm - GeeksforGeeks*. (2016, June 16). GeeksforGeeks. https://www.geeksforgeeks.org/a-search-algorithm/
- Tech With Tim. (2020). A* Pathfinding Visualization Tutorial - Python A* Path Finding Tutorial [YouTube Video]. In *YouTube*.
- Lague, S. (2021). Coding Adventure: Ant and Slime Simulations [YouTube Video]. In *YouTube*.
- Learn by Example. (2021). Solving the Travelling Salesman Problem using Ant Colony Optimization [YouTube Video]. In *YouTube*.
- jamesmcguigan. (2022, August 5). *Ant Colony Optimization Algorithm*. Kaggle.com; Kaggle. https://www.kaggle.com/code/jamesmcguigan/ant-colony-optimization-algorithm
- Shekhawat, A., Poddar, P., & Boswal, D. (2009). *Ant colony Optimization Algorithms : Introduction and Beyond*. https://mat.uab.cat/~alseda/MasterOpt/ACO_Intro.pdf
- Max teaches Tech. (2022). Python Pathfinding Vizualisation [YouTube Video]. In *YouTube*