**Artificial Intelligence**

**Project**



Best Path Finder

**Subject :** Artificial Intelligence

**Course Code :** INT-404

**Teacher’s Name:** Ankita Wadhawan

Github Link:

**Students :**

Akash Abhay Paliwal Aayush Thakur

11805342 11805354 11804907

Roll no: 48 49 47

Abstract

This python program help us to find the best path using three different algorithms namely BFS,DFS and A\*.In this project we have created some basic implementation of GUI to have a interaction with user for basic inputs like selecting a start point ,selecting an end point and also have some obstacles. In the first pop up menu user have to select the size of the grid and the type of algorithm that they need to preform then after that user have to select the start and end point of for the path.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

As per the given inputs given in this GUI interface further a grid will be created of the selects size and on the selected start and ends point algorithm will be performed and the best path will be shown .

Introduction

A python project for finding the best path via user inputs and interacting with user through GUI and finally show the best possible path from one point to another. In this project we have applied different concepts together by combing different searching modules and using the help of pygame we were able to visualize the best paths between the pint.

Different path finding algorithm available in this program are Breadth First Search (BFS), Depth first search (DFS), and A\* algorithm. User is free to choose any of the given algorithm for any path, as all three algorithm works on different concepts so the paths shown by them will also be different in different scenarios.

A screen shot of a building

Description automatically generated

A close up of a logo

Description automatically generated

**Literature Review**

**Graph traversals**

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

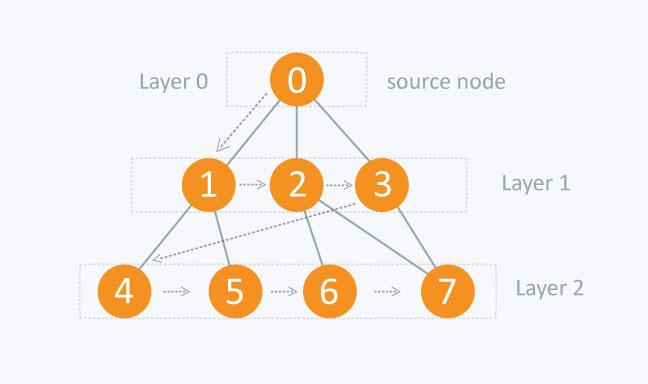
**Breadth First Search (BFS)**

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer



Consider the diagram.

The distance between the nodes in layer 1 is comparatively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

**Depth First Search (DFS)**

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:  
Pick a starting node and push all its adjacent nodes into a stack.  
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.  
Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.



**Time complexity** O(V+E), when implemented using an adjacency list.

**A\* Search Algorithm**

Informally speaking, A\* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.  
And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A\* Search Algorithm comes to the rescue.

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, water, etc.). There can be many ways to calculate this ‘h’ which are discussed in the later sections.

A\* expands paths that are already less expensive by using this function:

f(n)=g(n)+h(n),*f*(*n*)=*g*(*n*)+*h*(*n*),where

* f(n)*f*(*n*) = total estimated cost of path through node n*n*
* g(n)*g*(*n*) = cost so far to reach node n*n*
* h(n)*h*(*n*) = estimated cost from n*n* to goal. This is the heuristic part of the cost function, so it is like a guess.

A picture containing cage, clock

Description automatically generated

In the grid above, A\* algorithm begins at the start (red node), and considers all adjacent cells. Once the list of adjacent cells has been populated, it filters out those which are inaccessible (walls, obstacles, out of bounds). It then picks the cell with the lowest **cost**, which is the estimated f(n). This process is recursively repeated until the shortest path has been found to the target (blue node). The computation of f(n)*f*(*n*) is done via a heuristic that usually gives good results.

**Proposed Methodology**

Some notes on our approach and issues we came across.

**Tools used:**

* Tkinter: To create a window with different options for the user to select which search technique and grid size to use.
* Pygame: Generates the actual visualization of the search algorithm in progress

### **Choice for Visualization**

In hindsight we don't know if Pygame was the best choice to create the visualization. Maybe Matlab or something else would have been better but we didn't have experience with any GUI libraries so we figured we would try out PyGame and see how it went. All things considered, we are happy with how it turned out.

### **Threads**

One of the main issues We had was how to separate the visualization and the actual algorithm. we could have merged the algorithms inside the main game loop that created the visuals but that seemed a little unreliable and felt like a bad approach to the problem. And once the PyGame loop is running (i.e. making the grid show up)no other code will run.

We decided to try Python's Threading library and it worked well aside from a few hiccups with PyGame. We initially tried to have PyGame's game loop split off in a separate thread but we quickly discovered PyGame isn't thread safe and it's better to handle the game-loop in the main thread. We ended up making the search algorithm run as a separate thread and Pygame in the main.

### **Tkinter**

We wanted to have an interface to choose options about the grid and which algorithm to use. Since tkinter is included with Python it seemed like the logical choice.

One of the hurdles we had with this was setting up a way to limit the indices available to choose from for the start/goal cells. We wanted to have different grid sizes so these values had to change based on that grid size was chosen.

We thought the easiest way to do this would be a paging window system so the indices could update based on the selections made in the previous window. We also wanted to allow the option to return and change selections from previous windows. It turns out this was a little more complicated to achieve than we thought it would be, but luckily there are some excellent tutorials on StackOverflow for how to do this.

### **PyGame**

Creating the grid in PyGame was one the easier parts of this project. It was fairly simple to generate a grid window with the draw.line method in PyGame. The only slightly tricky thing was converting locations on the PyGame grid to indices in the matrix used in the search algorithm.

### Features to Implement

Some of the things that we would like to improve or implement:

* Implement another window to start the search algorithm after walls have been added instead of having to press spacebar
* Add the ability remove walls after they have been added
* Add a way to increase the weight of cells to see how A-star reacts to an increased weight in certain cells
* Implement Dijkstra's after above bullet point.

**RESULT AND DISCUSSION**

We had a lot of fun with this project and definitely learned a lot along the way. Although we was familiar with BFS and DFS we didn't have as much knowledge on A-Star implementation and this project definitely gave me a better understanding of how it works.

We Combined different implementation together like different searching strategy and GUI tkinter although we used pygame for the first time it was really very good experience using it we learnt a lot through stack over flow and our teacher Ankita Wadhawan .

What we got to learn during this process was :

* Using git hub .
* How to combine code.
* How to code together with others .
* How to use pygame.
* Implementation of different search strategy.
* Merging of code with GUI.
* Using threads for parallel programming .
* Combining different python files in one.
* And much more.