Final Year Project Report

**Full Unit – Interim Report**

Single-Agent and Multi-Agent In Maze Game

King San CHU

A report submitted in part fulfilment of the degree of

**BSc (Hons) in Computer Science**

**Supervisor:** Eduard Eiben



Department of Computer Science

Royal Holloway, University of London

December 06, 2023

**Declaration**

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 5219

Student Name: King San CHU

Date of Submission: 06/12/2023

Signature:



Table of Contents

[Abstract 3](#_Toc367967467)

[Chapter 1: Introduction 5](#_Toc367967469)

[1.1 The Problem 5](#_Toc367967470)

[1.2 Aims And Goals Of The Project 5](#_Toc367967470)

[1.3 Objectives – Milestones Summary 5](#_Toc367967470)

[Chapter 2: Algorithms 6](#_Toc367967471)

[2.1 Breadth First Search 5](#_Toc367967470)

[2.2 Depth First Search 5](#_Toc367967470)

[2.3 Dijstkra Algorithm 5](#_Toc367967470)

[1.1 A Star Search 5](#_Toc367967470)

[Chapter 3: Software Engineering 8](#_Toc367967472)

[3.1 Methodology 5](#_Toc367967470)

[3.2 Documentation 5](#_Toc367967470)

[Chapter 4: Self-evaluation And Expectation 9](#_Toc367967473)

[4.1 Self-evaluation 9](#_Toc367967474)

[4.2 Future expectation and implementation 9](#_Toc367967475)

[Bibliography 13](#_Toc367967481)

Abstract

The realm of this project presents exciting challenges for artificial intelligence **(AI)** and pathfinding algorithms. In the single-agent and multi-agent world, this project can help discover the fundamental problems of single and multi-agent pathfinding algorithms **(MAPF)[3]**, with adversarial decision-making through human-robot interaction **[5]** with the maze environment.

There is a lot of extensively conducted research in this area, displaying the efficacy of different search algorithms and game-playing strategies. A\* and Dijkstra's algorithms **[1]** have proven effective in single-agent scenarios, while Minimax and Alpha-beta pruning **[3]** are successful examples in Multi-Agent Systems **(MAS)[3]**. However, little attention is paid to integrating these algorithms into game development to improve users' gameplay experiences.

This project aims to fill the gap by incorporating advanced AI features into the maze game, from influencing popular algorithms and strategies seeking engagement and challenges for the players to enhance their gaming experience.

To commence the project, I need to imply theoretical solid foundations in basic pathfinding algorithms**[8]**, heuristic functions and adversarial search techniques **[6][7]**. This implementation will start with more straightforward pathfinding methods and fundamental understandings. I will also integrate more sophisticated algorithms while building up the game to optimise efficiency to the game environment as it advances. These goals could be achieved with the help of Distributed Artificial Intelligence **(DAI)[3]**and Distributed Problem Solving **(DPS)**methods.

This game will employ 'Tkinter' to create the Graphical User Interface (GUI), enabling graphical visualisation and user interaction. Programming in Python and utilising Tkinter will be essential skills to develop during this project, allowing for the seamless integration of pathfinding algorithms and AI techniques.

The outcomes and insights generated by this project will significantly contribute to the Artificial Intelligence sector by enhancing maze-based games with advanced AI capabilities. This advancement will unlock new dimensions and stimulate possibilities for enhancing the gaming experience across maze-based application platforms.

# Introduction

The pursuit of solving mazes has intrigued both enthusiasts and computer scientists for years. In game development and algorithmic exploration, creating a maze-solving game encapsulates a fascinating blend of challenge, strategy, and computational exploration. This project delves into the realms of maze-solving algorithms, centering on exploring single-agent and multi-agent approaches within a dynamic maze environment.

## The Problem

The tech industry continually grapples with various challenges when it comes to single-agent and multi-agent systems. These problems arise from the complexities of creating, managing, and optimising these systems, often impacting performance, scalability, and ethical considerations [8].

### Scalability And Efficiency

While single-agent systems excel in simplicity and efficiency, they need help scaling when confronted with complex tasks or vast datasets. A single agent might need help efficiently handling large data volumes or intricate decision-making processes. The limitations of single-agent systems in handling big data are often discussed. For instance, in "Challenges of Handling Big Data," authors emphasise the computational bottleneck faced by single-agent systems when processing large datasets. They stress the inefficiency of managing vast volumes of data due to limited processing power and memory [7].

Scalability remains a challenge in multi-agent systems, particularly in ensuring coordinated actions among numerous agents. Coherence and synchronisation become intricate as the number of agents grows, and the issues of communication overhead and decision-making complexity as the number of agents grows, affecting system scalability and efficiency, often leading to bottlenecks or inefficiencies in decision-making and communication.

### Collaboration And Communication

In scenarios demanding collaborative decision-making or actions, single-agent systems fall short due to their inability to coordinate with other entities. Such systems lack the capacity to leverage collective intelligence or share resources efficiently. Single agents have the advantage of simplicity but suffer from limited collaboration. The agent operates in isolation and does not benefit from information shared by other agents. It solely relies on its pathfinding algorithm and heuristic knowledge, which might need to be more efficient for more complex mazes [7].

While designed for collaboration, coordinating multiple agents poses challenges. Issues like communication overhead, conflicting objectives, or lack of a centralised control mechanism can hinder practical cooperation, impacting overall system performance. Multi-agent systems require efficient communication channels for agents to share information about their exploration, discovered paths, or obstacles. Conflicts arise when multiple agents attempt to traverse the same path or simultaneously aim to reach the same destination. Implementing conflict resolution mechanisms without compromising the efficiency of individual agents' paths is a complex task. Establishing effective communication while avoiding information overload or bottlenecks is crucial [8].

### Ethical And Regulatory Consideration

Ethical concerns encompass issues related to bias, fairness, and accountability in the decision-making processes of single-agent systems. Lack of transparency and interpretability in algorithms might lead to unintended biases or unfair outcomes. Players should understand why specific paths are chosen or actions taken by the agents. Lack of transparency can lead to mistrust or confusion among users [9].

Algorithms may exhibit biases, affecting gameplay or agent behaviours. Developers must ensure fairness by minimising biases in pathfinding algorithms. Discriminatory paths or unfair advantages could negatively impact the gaming experience. Coordinating ethical behaviour among multiple agents raises intricate challenges. The decentralised nature of these systems can complicate the establishment of ethical guidelines, accountability mechanisms, and governance frameworks, amplifying concerns related to fairness and trust [10].

## Aims And Goals Of The Project

The central aim of this project is to simulate and run a maze-based game akin to Pac-Man, integrating both single-agent and multi-agent movement within the maze environment while introducing various constraints to enhance its complexity and intrigue. The main aim is to capture Pac-Man's essence while expanding its gameplay by using various pathfinding methods for individual agents and teamwork among multiple agents.

Creating a challenging and engaging gameplay experience is at the heart of this endeavour. By leveraging the principles of single-agent pathfinding algorithms, with Breadth-first search, Depth-first search, Dijkstra and A Star Search algorithm, the project aims to empower a solo navigating entity, similar to the iconic Pac-Man character, in efficiently navigating through the maze. This involves enabling the agent to find the shortest path to collect rewards or accomplish objectives while avoiding obstacles or adversaries within the confined maze space.

In collaboration, the project seeks to introduce multi-agent systems, wherein several agents operate simultaneously within the maze. These agents, each with distinct goals and constraints, contribute to an intricately woven fabric of maze exploration. The objective here is not just to navigate the maze but to foster interaction, coordination, and perhaps competition among agents, imitating a more realistic and dynamic environment akin to multiplayer games[10].

By adding layers of complexity and challenge, the project will implement various constraints within the maze environment. These constraints could range from different speeds or movement capabilities of agents, the presence of dynamic obstacles, varying rewards or objectives, to time-sensitive challenges, all designed to diversify and enrich the gaming experience [10]. For instance, some agents can move faster. In contrast, others might have constraints limiting their movements, thus fostering a diverse ecosystem of agents coexisting and manoeuvring within the maze's confines.

The ultimate goal of this project is not only to emulate the Pac-Man gameplay dynamics but also to push the boundaries by infusing it with the advancements and complexities of modern AI-driven pathfinding algorithms. By doing so, it aims to offer users an immersive and intellectually stimulating gaming experience, providing insights into the intricacies of both single-agent and multi-agent systems operating in maze-like environments. Through this, the project endeavours to contribute to AI-driven game development, setting new benchmarks in maze-based applications and artificial intelligence [12].

## Objectives – Milestone Summary

### Implementation of Search Algorithm

In this project, the primary focus was on implementing various search algorithms for maze-solving scenarios. The A Search Algorithm\* was a central highlight, allowing the program to find the shortest path from a starting point to a goal while considering heuristic estimations, enhancing the efficiency of pathfinding. Alongside A\*, the implementation of Dijkstra's Algorithm offered an alternative approach for discovering the shortest path, devoid of heuristic estimations, ensuring optimal paths without considering the target's location.

### GUI Integration

The integration of a Graphical User Interface (GUI) was pivotal for user interaction and visual feedback in the maze-solving process. Utilising Tkinter, the GUI provided an interactive platform to display maze structures, track pathfinding progress, and visualise agent movements. Including visual elements significantly enhanced the user's understanding and visualisation of the maze-solving algorithms, offering a seamless and informative experience.

### Achievements and Milestones

The project celebrated several milestones:

* Successful Algorithm Integration: The incorporation of four diverse search algorithms, namely A\*, Dijkstra's, Breadth-First Search (BFS), Dijkstra's Algorithm, and Depth-First Search (DFS), into the maze-solving system, allowed users to witness the varied strategies employed by these algorithms in solving maze complexities.
* Pathfinding Visualisation: Implementing an interactive GUI interface provided real-time visual feedback on the maze-solving process. It dynamically displayed the paths discovered by the algorithms, aiding users in comprehending and comparing the effectiveness of each algorithm in solving maze puzzles.
* Algorithmic Diversity and Comparison: Users could observe and compare each algorithm's efficiency and characteristics when navigating through varying complexities mazes. This allowed for an insightful analysis of their strengths and limitations.
* Error Handling and Robustness: Implementing comprehensive error checks and handling mechanisms ensured the program's robustness in managing diverse maze configurations and user inputs.
* User-Friendly Interface: The design of a user-friendly GUI using Tkinter enabled users to interact effortlessly with different algorithms and observe their outcomes on the maze grid.

The milestones represent a successful integration of diverse search algorithms and an interactive GUI for maze-solving scenarios. These accomplishments serve as a foundation for exploring more complex pathfinding strategies, refining the GUI interface, and introducing advanced features in subsequent iterations.

# Algorithms

## Breadth First Search

*Breadth-First Search****(BFS)*** is a fundamental graph traversal algorithm that explores nodes in a graph or tree data structure. The technique explores all neighbours of a particular node before moving on to its neighbouring nodes, effectively discovering the shortest path between nodes in an unweighted graph or the shallowest path to a target node in a tree.

The algorithm operates by traversing the graph level by level, starting from the root or a specified starting node and exploring all of its adjacent nodes before moving to the next level. It uses a queue data structure to keep track of nodes yet to be visited [11]. The basic idea is to explore the graph systematically, expanding the search from the root node to its neighbours, then to their neighbours, and so on, until it reaches the target node.

In BFS, nodes are visited in the order they were discovered. The algorithm begins by enqueueing the starting node into the queue and marking it as visited to avoid revisiting it. Then, it repeatedly dequeues nodes from the front of the queue, explores their neighbours, and enqueues those neighbours into the queue if they have yet to be visited.

def breadthFirstSearch(maze, start=None):

"""

Finds the shortest path in a maze using Breadth-First Search (BFS) algorithm.

Args:

- maze: The maze instance.

- start: The starting point for the BFS search. Defaults to the maze's dimensions if not provided.

Returns:

- A tuple containing three dictionaries representing the search path, BFS path, and forward path.

"""

if start is None:

start = (maze.rows, maze.cols)

frontier = deque()

frontier.append(start)

bfsPath = {}

explored = [start]

searchPath = []

while len(frontier) > 0:

currentCell = frontier.popleft()

if currentCell == maze.\_goal:

break

for direction in 'ESNW':

if maze.maze\_map[currentCell][direction] == True:

if direction == 'E':

childCell = (currentCell[0], currentCell[1]+1)

elif direction == 'W':

childCell = (currentCell[0], currentCell[1]-1)

elif direction == 'S':

childCell = (currentCell[0]+1, currentCell[1])

elif direction == 'N':

childCell = (currentCell[0]-1, currentCell[1])

if childCell in explored:

continue

frontier.append(childCell)

explored.append(childCell)

bfsPath[childCell] = currentCell

searchPath.append(childCell)

forwardPath = {}

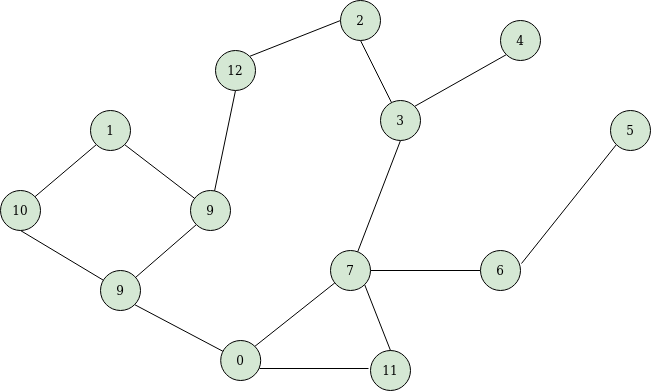
cell = maze.\_goal

while cell != (maze.rows, maze.cols):

forwardPath[bfsPath[cell]] = cell

cell = bfsPath[cell]

return searchPath, bfsPath, forwardPath



1. BFS Graph with queue to keep track of the next node to visit. [20]

In the function breadthFirstSearch, the code initialises a frontier using a deque data structure containing the starting point or a specified node. It explores neighbouring cells while systematically traversing the maze. The function employs a while loop to process the frontier by dequeuing cells, exploring their valid neighbouring cells, and appending them to the frontier for further exploration if they have not been visited.

The code utilises a dictionary (bfsPath) to record the path from the starting cell to each visited cell, thereby reconstructing the path taken. Additionally, it maintains explored and searchPath lists to keep track of cells that have been explored and the search path, respectively.

The while loop iterates until the frontier is empty or the goal cell is found, breaking the loop upon reaching the goal. This demonstrates the critical characteristic of BFS: exploring nodes level by level, ensuring the shortest path is found when applied to maze-solving scenarios.

The final section of the code constructs the forwardPath dictionary, which allows the reconstruction of the shortest path from the goal cell back to the starting cell. This corresponds to the process of tracing back the path taken from the goal to the starting point in the maze.

Overall, the code aligns with the BFS algorithm's principles, systematically exploring the maze and recording the path taken to find the shortest route from the starting point to the goal, as described in the previous explanation.

## Depth First Search

*Depth-first search (DFS)* is a fundamental graph traversal algorithm used to explore a graph or maze structure systematically [4]. It operates by starting at a chosen node and then exploring as far as possible along each branch before backtracking. Initially marking a node as visited, DFS then moves to adjacent unvisited nodes and explores these paths until it reaches a dead-end. It employs a stack or recursion mechanism to dive deeper into the graph, prioritising depth over breadth in its exploration strategy. DFS continues this process of exploration and backtracking until it has visited all reachable nodes or satisfied the goal condition [12].

This algorithm's characteristic deep exploration allows it to unravel intricate structures by following a path to the furthest possible extent before revisiting unexplored branches, contributing to its utility in a variety of applications, including maze solving, graph analysis, and network traversal[13].

def depthFirstSearch(self, start=None):

"""

Executes a Depth-First Search (DFS) algorithm to find a path in a maze.

Args:

- self: The maze instance.

- start: The starting point for DFS search. Defaults to the maze's dimensions if not provided.

Returns:

- A tuple containing three dictionaries representing the DFS search, DFS path, and forward path.

"""

if start is None:

start = (self.rows, self.cols)

explored\_cells = [start]

frontier = [start]

dfs\_path = {}

dfs\_search = []

while len(frontier) > 0:

current\_cell = frontier.pop()

dfs\_search.append(current\_cell)

if current\_cell == self.\_goal:

break

possible\_directions = 0

for direction in 'ESNW':

if self.maze\_map[current\_cell][direction] == True:

if direction == 'E':

child\_cell = (current\_cell[0], current\_cell[1]+1)

elif direction == 'W':

child\_cell = (current\_cell[0], current\_cell[1]-1)

elif direction == 'N':

child\_cell = (current\_cell[0]-1, current\_cell[1])

elif direction == 'S':

child\_cell = (current\_cell[0]+1, current\_cell[1])

if child\_cell in explored\_cells:

continue

possible\_directions += 1

explored\_cells.append(child\_cell)

frontier.append(child\_cell)

dfs\_path[child\_cell] = current\_cell

if possible\_directions > 1:

self.markCells.append(current\_cell)

forward\_path = {}

cell = self.\_goal

while cell != start:

forward\_path[dfs\_path[cell]] = cell

cell = dfs\_path[cell]

return dfs\_search, dfs\_path, forward\_path



1. How DFS function [21].

*The Depth-First Search (DFS)* algorithm begins by initialising a stack to keep track of visited nodes and paths. It starts the search from a specified starting cell within a maze or graph. If no start point is provided, it defaults to the maze's dimensions. The algorithm maintains a list of explored cells, a stack to traverse the maze, and dictionaries to store the DFS search, path, and forward path information.

During the search, DFS explores the neighbouring cells of the current node in a specific order ('ESNW' - East, South, North, West) to discover unvisited paths. If a valid path exists for each direction, the algorithm moves to the adjacent cell and marks it as explored while recording the traversal path.

Upon reaching a dead-end or when the goal cell is found, the algorithm backtracks by popping nodes off the stack, revisiting junctions, and exploring alternate paths. The process continues until all reachable nodes have been visited or the goal condition has been met.

The DFS function returns three dictionaries: dfs\_search contains a record of the explored cells during the search, dfs\_path stores the explored paths taken by the algorithm, and forward\_path denotes the sequence of cells from the goal cell back to the starting point. These data structures offer valuable information about the traversal, enabling path reconstruction and analysis.

## Dijkstra’s Algorithm

Dijkstra's algorithm is a fundamental pathfinding algorithm used to determine the shortest path between nodes in a graph, typically in scenarios without negative edge weights. It ensures that the shortest path from a source node to all other nodes in the graph is found. The algorithm employs a greedy approach by iteratively exploring the nodes based on their tentative distances from the source node [14].

Dijkstra's algorithm begins in a maze or graph by assigning initial distances to all nodes from the source node. It maintains a priority queue or a min-heap to select the next node to explore. The algorithm selects the node with the smallest tentative distance at each iteration and explores its neighbouring nodes.

def dijkstraPathFinding(self):

"""

Performs Dijkstra's algorithm to find the shortest path in a maze.

Returns:

- Dictionary representing the path found by Dijkstra's algorithm.

"""

start = (self.rows, self.cols)

g\_scores = {cell: float('inf') for cell in self.grid}

g\_scores[start] = 0

open\_cells = PriorityQueue()

open\_cells.put((0, start))

d\_path = {}

while not open\_cells.empty():

current\_cost, current\_cell = open\_cells.get()

if current\_cell == (1, 1):

break

for direction in 'ESNW':

if self.maze\_map[current\_cell][direction] == True:

if direction == 'E':

child\_cell = (current\_cell[0], current\_cell[1] + 1)

if direction == 'W':

child\_cell = (current\_cell[0], current\_cell[1] - 1)

if direction == 'N':

child\_cell = (current\_cell[0] - 1, current\_cell[1])

if direction == 'S':

child\_cell = (current\_cell[0] + 1, current\_cell[1])

temp\_g\_score = g\_scores[current\_cell] + 1

if temp\_g\_score < g\_scores[child\_cell]:

g\_scores[child\_cell] = temp\_g\_score

open\_cells.put((temp\_g\_score, child\_cell))

d\_path[child\_cell] = current\_cell

forward\_path = {}

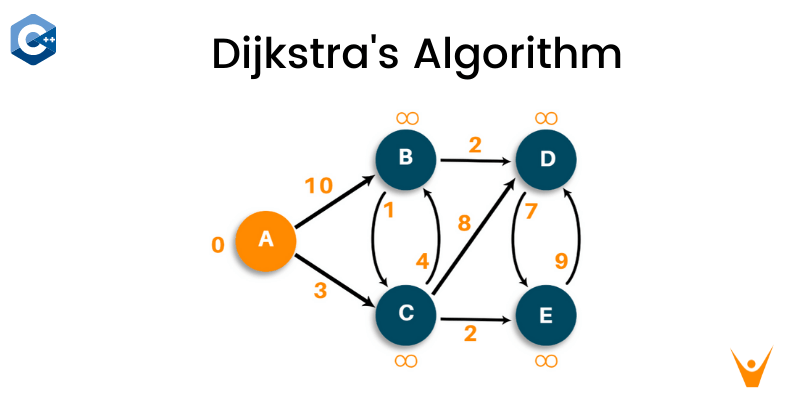
cell = (1, 1)

while cell != start:

forward\_path[d\_path[cell]] = cell

cell = d\_path[cell]

return forward\_path



1. Dijkstra’s Algorithm function. [22]

The dijkstraPathFinding function uses Dijkstra's algorithm to find the shortest path within a maze. The heuristic function calculates the heuristic (Manhattan distance) between two cells in the maze, aiding in estimating the distance between nodes.

The dijkstraPathFinding function initialises the starting cell and assigns initial values for the distances between cells. It employs a priority queue to traverse cells based on their tentative distances from the starting cell. The algorithm then evaluates neighbouring cells, updating their distances if a shorter path is discovered. The process continues until the algorithm reaches the goal cell, eventually constructing and returning the shortest path found.

This implementation efficiently explores the maze using Dijkstra's algorithm, continually updating the distances to determine the shortest path to the goal cell based on the provided maze structure and its constraints.

## A Star Search

The A\* search algorithm is a widely used heuristic-based pathfinding algorithm known for its efficiency in finding the shortest path between two nodes in a graph. It combines elements of Dijkstra's algorithm and greedy best-first search by using both the actual cost to reach a node from the start and an estimated heuristic cost to the goal [15]. A\* evaluates nodes based on the sum of the cost to reach that node from the starting node (known as the "g-score") and an estimated cost to the goal node (known as the "h-score"). This combination enables A\* to efficiently navigate through the search space while ensuring it reaches the goal optimally.

def aStarPathFinding(self):

"""

Performs A\* pathfinding algorithm to find the shortest path in a maze.

Returns:

- Dictionary representing the path found by A\* algorithm.

"""

start = (self.rows, self.cols)

g\_scores = {cell: float('inf') for cell in self.grid}

g\_scores[start] = 0

f\_scores = {cell: float('inf') for cell in self.grid}

f\_scores[start] = heuristic(start, (1, 1))

open\_cells = PriorityQueue()

open\_cells.put((heuristic(start, (1, 1)), heuristic(start, (1, 1)), start))

a\_path = {}

while not open\_cells.empty():

current\_cell = open\_cells.get()[2]

if current\_cell == (1, 1):

break

for direction in 'ESNW':

if self.maze\_map[current\_cell][direction] == True:

if direction == 'E':

child\_cell = (current\_cell[0], current\_cell[1] + 1)

if direction == 'W':

child\_cell = (current\_cell[0], current\_cell[1] - 1)

if direction == 'N':

child\_cell = (current\_cell[0] - 1, current\_cell[1])

if direction == 'S':

child\_cell = (current\_cell[0] + 1, current\_cell[1])

temp\_g\_score = g\_scores[current\_cell] + 1

temp\_f\_score = temp\_g\_score + heuristic(child\_cell, (1, 1))

if temp\_f\_score < f\_scores[child\_cell]:

g\_scores[child\_cell] = temp\_g\_score

f\_scores[child\_cell] = temp\_f\_score

open\_cells.put((temp\_f\_score, heuristic(child\_cell, (1, 1)), child\_cell))

a\_path[child\_cell] = current\_cell

forward\_path = {}

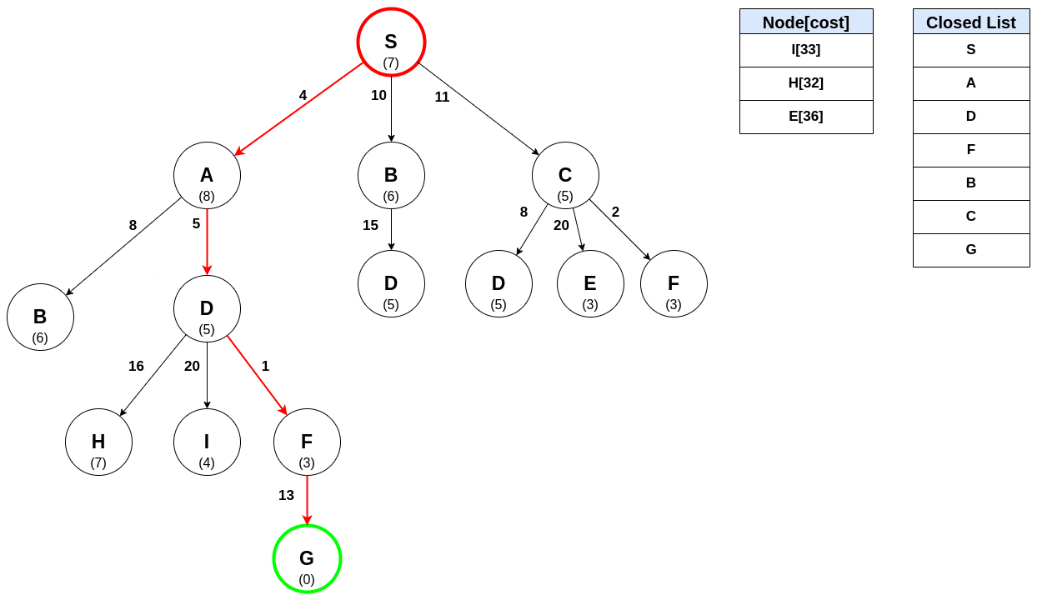
cell = (1, 1)

while cell != start:

forward\_path[a\_path[cell]] = cell

cell = a\_path[cell]

return forward\_path



1. An example of how A\* Algorithm chooses the path [23]

This algorithm commences by setting up the initial conditions: defining the starting point, 'start', and establishing two dictionaries, 'g\_scores' and 'f\_scores'. These dictionaries store the cumulative cost ('g\_scores') and the anticipated overall cost ('f\_scores') from the starting cell to any other location within the maze.

Subsequently, a priority queue named 'open\_cells' is initiated to hold cells based on their estimated total cost. The algorithm populates this queue by initially adding the starting cell along with its estimated cost, as per the heuristic function employed.

During the exploration phase, the algorithm continually extracts the cell from 'open\_cells' with the lowest estimated total cost. For each of the adjacent cells—computed for four directions: East, West, North, and South—it evaluates potential movement possibilities.

Calculating the tentative 'g\_score' for each adjacent cell involves adding 1 to the cost of the current cell. The algorithm then computes the 'f\_score' by summing the 'g\_score' and the heuristic value (Manhattan distance) from the adjacent cell to the goal.

The algorithm then updates the 'g\_score' and 'f\_score' and enqueues the adjacent cell into the 'open\_cells' queue if the computed 'f\_score' is lower than the previously recorded 'f\_score'. Additionally, it tracks the parent cell for each adjacent cell within the 'a\_path' dictionary.

Once the goal cell is reached, the algorithm retraces its steps from the goal back to the starting point using the 'a\_path' dictionary. This backtracking facilitates the reconstruction of the shortest path, which is stored in the 'forward\_path' dictionary.

Finally, the algorithm returns the shortest path discovered, encapsulated within a dictionary ('forwardPath') containing the sequential cells from the start to the goal. The A\* search algorithm elegantly balances the cost to reach the current cell with an estimated cost from that cell to the goal, ensuring an optimal path through the maze is identified.

# Software Engineering

The software engineering aspect of the "Single Agent and Multi-Agent Search in Maze Game" project involves a robust implementation of various search algorithms and game design paradigms. This endeavour encompasses creating and integrating algorithms like Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra's algorithm, and A\* Search into the maze environment. Additionally, it involves the orchestration of both single-agent and multi-agent interactions within the maze. Overall, the software engineering facet of this project accentuates the integration of diverse algorithms and design strategies, emphasising optimal performance and interactive user experiences within the maze environment.

## Methodology

The "Single-Agent and Multi-Agent Search in Maze Game" project encompasses various methodologies to tackle the complex challenge of developing a maze-based gaming environment. At its core, the project aims to simulate a maze-based game reminiscent of Pac-Man, integrating both single-agent and multi-agent movement. The overarching goal is to create an immersive experience, leveraging a blend of pathfinding algorithms and strategies designed for individual agents and coordinated interactions among multiple agents [7].

A pivotal aspect of this endeavour revolves around creating a challenging and engaging gameplay experience. This involves harnessing the principles of single-agent pathfinding algorithms like Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra's algorithm, and A\* Search to enable a solitary navigating entity, similar to the iconic Pac-Man character, to traverse the maze efficiently. The goal is to enable the agent to identify the shortest path to collect rewards or achieve objectives while evading obstacles or adversaries within the maze[5].

Simultaneously, the project delves into multi-agent systems, where several agents operate concurrently within the maze environment. Each agent possesses distinct goals and constraints, contributing to a complex and intricate maze exploration. This segment aims to foster interaction, coordination, and even competition among the agents [2], creating an environment reminiscent of multiplayer games that reflect realistic and dynamic scenarios.

To augment the complexity and challenge of the game, the project implements diverse constraints within the maze environment. These constraints introduce layers of difficulty, varying from differing speeds or movement capabilities of agents, the presence of dynamic obstacles, diverse objectives or rewards, to time-sensitive challenges [13]. This diverse range of constraints enriches the gaming experience, establishing a diverse ecosystem of agents coexisting and navigating within the maze's boundaries.

This project's ultimate objective is to encapsulate the gameplay dynamics of Pac-Man and extend its complexity by incorporating modern advancements in AI-driven pathfinding algorithms. The project aims to offer users an intellectually stimulating gaming experience through this fusion, providing insights into the intricacies of single-agent and multi-agent systems operating within maze-like environments. This effort aspires to contribute significantly to AI-driven game development, setting new benchmarks in maze-based applications and artificial intelligence.

## Documentation

### Implemented Algorithms

1. Breadth-First Search (BFS)

BFS is a systematic search algorithm that explores all neighbour nodes before moving to the next level. Its primary purpose is finding the shortest path from a starting point to a goal within a maze. It is commonly employed for navigation, ensuring an agent reaches its goal while avoiding obstacles efficiently.

1. Depth-First Search (DFS)

DFS explores as far as possible along a branch before backtracking, exhaustively examining paths. While it may not necessarily find the shortest path, it offers an alternative pathfinding method with different characteristics compared to BFS.

1. Dijkstra's Algorithm

This algorithm finds the shortest path in a weighted graph from a source node to all other nodes, specifically effective for pathfinding with non-negative edge weights, offering various traversal possibilities based on weight-based constraints.

1. A\* Algorithm

A\* is a best-first search algorithm that uses heuristics to guide the search process, efficiently finding the shortest path by considering both cost and heuristic information.

### Files and Modules

1. Maze Module (maze.py)

This module generates maze environments, facilitating maze creation, representation, and navigation. It offers the flexibility to create diverse mazes of varying complexities, sizes, and themes for maze-based games.

1. Agent Module (agent.py)

Designed to handle agent behaviours and functionalities within the maze, this module creates, controls, and sets goals for agents. It allows for diverse agent types with unique abilities and behaviours, supporting single-agent and multi-agent interactions.

1. Text Label Module (textLabel.py)

This module manages textual information within the maze environment. It enhances visualisation by displaying essential information, and its future use can improve user interfaces and offer informative feedback during gameplay.

1. COLOR Module (COLOR.py)

This module defines colour schemes used in the maze environment by storing colour codes for visualisation purposes. It can be further extended to support various themes and visual customisations.

### Tkinter

Tkinter serves as the primary GUI toolkit in this maze project, providing a user-friendly interface to interact with and visualise the maze environment and its agents.

The Tkinter component in this maze project embodies several essential components and functionalities, contributing significantly to the interactive visualisation of the maze environment. Tkinter facilitates the comprehensive display of the maze's structural elements, effectively presenting the grid layout, walls, open paths, and diverse maze themes [18]. It enables users to visually comprehend the maze's intricacies and distinct features, providing a holistic view of the environment.

Moreover, Tkinter is a vital tool for representing agents within the maze. It offers a graphical representation of agents, illustrating their movements, positions, and interactive behaviours within the maze structure. This graphical representation aids users in tracking and understanding the agents' behaviours, contributing to a more immersive and engaging experience.

The user interaction aspect of Tkinter plays a pivotal role in enhancing user engagement and control within the maze game. Through user-friendly controls and buttons, Tkinter allows users to initiate various algorithms, commence maze navigation, and visualise the pathfinding process seamlessly. Additionally, it displays crucial textual information, such as path lengths, algorithm names, and other game-related details, providing essential insights and feedback to the users[18].

Tkinter's integration capabilities are noteworthy, seamlessly embedding with the maze modules to retrieve data and visualise maze structures, agents, and paths obtained from diverse algorithms. Its event-driven programming facilitates efficient handling of user interactions, triggering specific functions to execute actions like running algorithms, controlling agents, or updating the maze display.

The integration of Tkinter in this project offers numerous advantages. Its simplicity and intuitiveness empower users to navigate and interact with the maze game effortlessly. Its flexibility in customising user interface elements, visuals, and controls significantly enhances the user experience. Tkinter's compatibility with Python and other modules allows seamless integration, making it suitable for incorporating maze-related functionalities and visuals.

Tkinter offers various avenues for future development and enhancement within the maze project. Advanced visualisation capabilities can be explored to elevate maze visuals and agent representations, further immersing users in the gaming experience. User interface refinement remains a potential focus area to improve usability and engagement, offering a more polished and appealing interface. Additionally, introducing more controls or features for user interaction and maze customisation can augment the game's complexity and engagement.

Tkinter plays a crucial role in the maze project, providing an efficient and user-friendly platform for visualising maze environments, managing agent movements, and enabling user interactions. Its flexibility and integration capabilities pave the way for further enhancements and refinements in the maze game's GUI.

With its modular structure and foundational functionalities, the project offers ample scope for future enhancements. Potential areas include introducing new algorithms, dynamic maze elements, user interface improvements, and theme variations, contributing to a more engaging and immersive maze-based gaming experience.

# Self-Evaluation and Expectations

## Self-Evaluation

### ****Achievements****

This project has achieved substantial milestones, marking significant progress in maze-solving algorithms and GUI development. The successful implementation and integration of multiple search algorithms—Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra's Algorithm, and A\* Search—have been pivotal achievements. Additionally, developing a user-friendly GUI using Tkinter to visualise maze structures, pathfinding algorithms, and agent interactions represents a significant accomplishment. The seamless integration of maze generation, agent control, and visualisation modules has contributed to a robust and functional maze-solving application.

### ****Skills and Knowledge Acquired****

The project journey has been instrumental in acquiring a spectrum of technical skills and domain knowledge. A deepened understanding of search algorithms, heuristic evaluation, and their practical applications in maze navigation has been gained. Proficiency in Python programming language, particularly in implementing algorithms, creating graphical interfaces, and managing diverse modules, has been honed. Moreover, the project has provided insights into GUI development using Tkinter, offering valuable exposure to event-driven programming, user interface design, and interactive elements.

### ****Challenges Faced and Lessons Learned****

Encountering time constraints amidst the project's expansive scope posed a significant challenge, affecting optimal algorithm implementation and GUI feature enrichment. Additionally, code communication among modules and functionalities presented hurdles in maintaining code coherence and ensuring seamless integration between diverse components. These challenges provided crucial insights into the importance of meticulous planning, task prioritisation, and streamlined communication in project management. The need for clear code documentation, modular development practices, and effective collaboration methodologies was emphasised. Overcoming these challenges has imparted valuable lessons in time management, fostering more precise communication strategies and enhancing code readability for improved project cohesion.

### ****Personal Growth****

Navigating the challenges, particularly time constraints and code communication hurdles, led to substantial personal growth. The experience enhanced my adaptability, bolstered my time management skills, and deepened my understanding of effective code structuring and communication strategies. Embracing time constraints fostered resourcefulness and an ability to prioritise tasks effectively. Moreover, grappling with code communication challenges reinforced the importance of concise documentation, adherence to coding standards, and collaborative practices for cohesive project development.

### ****Areas for Improvement****

Building upon these challenges, improvements can be made in managing time constraints by adopting agile methodologies, setting realistic milestones, and employing practical project management tools. Code communication can be strengthened through standardised documentation practices, consistent naming conventions, and routine code reviews to ensure clarity, coherence, and maintainability. Additionally, fostering an environment of open communication among team members and emphasising knowledge sharing can enhance code comprehension and integration among diverse modules. These efforts will streamline project development and enhance the overall quality and sustainability of the maze-solving application.

## Expectations

### Implementing Multi-Agent Scenarios and Interactive Elements

The future development of the maze game involves introducing multi-agent scenarios and dynamic elements to enhance gameplay complexity and engagement. Including multiple agents operating within the maze environment offers intricate interactions and strategic gameplay opportunities. To achieve this, the integration of algorithms such as Minimax, Alpha-Beta pruning, and Expectimax becomes paramount.

### Utilizing Minimax Algorithm for Multi-Agent Interactions

Integrating the Minimax algorithm allows in-depth analysis of multi-agent interactions within the maze. The game can simulate strategic decision-making by evaluating possible moves and counter-moves among agents. Over time, new agents with distinct behaviours and goals can be introduced, contributing to an evolving and challenging gameplay experience.

### Applying Alpha-Beta Pruning for Efficient Exploration

Alpha-Beta pruning, coupled with the Minimax algorithm, optimises the exploration of decision trees, enhancing computational efficiency. This approach streamlines the search process in mazes of varying sizes, leading to faster and more effective decision-making. Visualising agents' actions based on the pruned tree adds depth to the gameplay, offering insights into the decision-making rationale.

### Employing Expectimax Agent for Probabilistic Behaviour

Introducing the Expectimax agent introduces an element of probabilistic behaviour into the maze game. It accounts for an agent's suboptimal choices, simulating scenarios where agents might only sometimes make the best decisions. This adds realism to the agents' behaviours, creating an environment that mirrors real-world uncertainties and imperfect decision-making processes.

### Considerations and Limitations

However, it is crucial to acknowledge the algorithm's reliance on heuristics and its potential limitations in dynamic environments. Environments with shifting walls, appearing obstacles, or other dynamic elements challenge the algorithms' decision-making. The adaptability of the algorithms to dynamically changing scenarios and their sensitivity to heuristic parameters should be considered for accurate and efficient gameplay.

### Advanced Visualization Features

There is significant potential for integrating more advanced visualisation tools and features within the GUI in future project iterations. These additional features offer users a more comprehensive and intuitive understanding of the maze-solving process. Visual aids such as real-time algorithm execution visualisation, step-by-step algorithm analysis, or interactive graphical representations of search spaces could be implemented. By introducing these advanced visualisation elements, users can gain deeper insights into the behaviour of various algorithms, enhancing their understanding of pathfinding strategies and maze navigation methods.

### Performance Optimization

Continued efforts toward optimising the project's performance are essential to future development. Specifically, focusing on enhancing algorithm efficiency and GUI responsiveness, especially when handling larger maze grids, is a priority. Exploring techniques to streamline algorithm execution, reduce computational complexity, or implement parallel processing can significantly improve the overall user experience. Optimising the GUI's responsiveness to accommodate larger-scale mazes ensures a smooth and seamless interaction for users navigating through various maze complexities. By addressing performance concerns, the project can offer users a more efficient and enjoyable experience while working with diverse sizes and complexities of mazes.

This expanded version elaborates on implementing multi-agent scenarios interactive elements and integrating specific algorithms to enhance the maze game's gameplay experience. It also highlights the considerations and potential limitations in dynamic environments, offering a comprehensive overview of the future development expectations.

Bibliography

[1] Barnouti, N. , Al-Dabbagh, S. and Sahib Naser, M. (2016) Pathfinding in Strategy Games and Maze Solving Using A\* Search Algorithm. Journal of Computer and Communications, 4, 15-25.

[2] Drew, D.S. Multi-Agent Systems for Search and Rescue Applications. Curr Robot Rep 2, 189–200(2021).https://doi.org/10.1007/s43154-021-00048-3

[3] Tjiharjadi, S., Razali, S. and Sulaiman, H.A., 2022. A systematic literature review of multi-agent pathfinding for maze research. Journal of Advances in Information Technology Vol, 13(4).

[4] GeeksforGeeks,(n.d.).Types of Environments in AI. Available at: https://www.geeksforgeeks.org/types-of-environments-in-ai/ (Accessed: 2/10/2023)

[5] Dresner, K. and Stone, P., 2008. A Multiagent Approach to Autonomous Intersection Management. Journal of Artificial Intelligence Research, [online] 31, pp.591-656. Available at: https://www.jair.org/index.php/jair/article/view/10909/26012

[6] Russell, S.J. and Norvig, P., 2016. Artificial Intelligence: A Modern Approach, 3rd ed. Pearson.

[7] Millington, I. and Funge, J., 2009. Artificial Intelligence for Games, 2nd ed. CRC Press

[8] Nilsson, N.J., 2014. Principles of Artificial Intelligence. Morgan Kaufmann.

[9] Human-AI Research, Dayton, OH | Kairos Research. <https://kairosresearch.com/our-work/>

[10] Tjiharjadi, S., Razali, S. and Sulaiman, H.A., 2022. A systematic literature review of multi-agent pathfinding for maze research. Journal of Advances in Information Technology Vol, 13(4).

[11] Ueno, K., Suzumura, T., Maruyama, N. *et al.* Efficient Breadth-First Search on Massively Parallel and Distributed-Memory Machines. *Data Sci. Eng.* 2, 22–35 (2017). <https://doi.org/10.1007/s41019-016-0024-y>

[12] Joshi, V. (2018) *Demystifying depth-first search*, *Medium*. Available at: https://medium.com/basecs/demystifying-depth-first-search-a7c14cccf056

[13] Kozen, D.C. (1992). Depth-First and Breadth-First Search. In: The Design and Analysis of Algorithms. Texts and Monographs in Computer Science. Springer, New York, NY. <https://doi.org/10.1007/978-1-4612-4400-4_4>

[14] Permana, Silvester Dian Handy et al. “Comparative Analysis of Pathfinding Algorithms A \*, Dijkstra, and BFS on Maze Runner Game.” (2018).

[15] P. O. N. Saian, Suyoto and Pranowo, "Optimized A-Star algorithm in hexagon-based environment using parallel bidirectional search," 2016 8th International Conference on Information Technology and Electrical Engineering (ICITEE), pp. 1-5, 2016, <https://doi.org/10.1109/ICITEED.2016.7863246>.

[16] Liu, Xiang, and Gong, Daoxiong. “A Comparative Study of A-Star Algorithms for Search and Rescue in Perfect Maze.” 2011 International Conference on Electric Information and Control Engineering,  <https://doi.org/10.1109/iceice.2011.5777723>

[17] Difference Between Bfs and Dfs. <https://www.engineerknow.com/2023/04/difference-between-bfs-and-dfs.html> (2.1)

[18] Moore, A. D. (2018). Python GUI Programming with Tkinter: Develop responsive and powerful GUI applications with Tkinter. Packt Publishing Ltd.

[19] Pornavalai, C., Shiratori, N., & Chakraborty, G. (1996). Neural network for optimal steiner tree computation. Neural Processing Letters, 3(3), 139-149.

[20] Jose K, “Graph Theory | Breadth First Search” Towards Data Science (2020) <https://towardsdatascience.com/graph-theory-breadth-first-search-d898e9f40fdd>

[21]Depth first search tutorials & notes: Algorithms. HackerEarth. (n.d.). <https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/>

[22] Bhadaniya, S. (2021, May 20). *Dijkstra’s algorithm in C++: Shortest path algorithm*. FavTutor. https://favtutor.com/blogs/dijkstras-algorithm-cpp

[23] TECHBREAUX. (2023, April 12). *AI: Search algorithms: A\* search*. Codecademy. https://www.codecademy.com/resources/docs/ai/search-algorithms/a-star-search