# CHECKERS GAME USING MINMAX ALGORITHM WITH ALPHA-BETA PRUNING

## NAMES/MATRIC NOS

1. FAFOWORA OLASUNKANMI OLAYINKA - 160805029
2. OLAGESIN SAMUEL FEYINTOLUWA - 160805025
3. MUDASHIRU TOHEEB - 160805048
4. OLALEKAN RILIWAN ABIODUN - 160805051

## COURSE CODE:528

## Session: 2022, 2nd Semester

# ABSTRACT

The problem given was to develop an AI GUI visualization module that of six Uniformed Search methods (Depth-First Search [DFS], Depth-Limited Search [DLS], Iterative Deepening Search [IDS], Breath-First Search [BFS]. Bidirectional Search [BIDI], and Uniform-Cost Search [UCS]). Alongside this module, we are to develop a Checkers game-playing program that can play a human player using the Minimax algorithm with alpha-beta pruning. The Agile methodology was used and the results obtained were a two-player checkers game with computer capable of predicting moves using mini-max alpha-beta pruning and a visualization of uninformed search algorithms.

# INTRODUCTION

## 1.1 BACKGROUND OF STUDY

Checkers is a 2 player board game played on a square board with 64 cells of alternating colors (usually black and red). Each player starts with his 12 pieces, commonly called “queen” or “men”, placed on dark squares of the board. The aim of the game is to capture all of the opponent’s pieces or block them so they cannot move any further.

On a player’s turn, the player may move one of his piece diagonally forward or backward into an empty space. If a player’s piece is adjacent to an opponent’s piece and the cell directly behind it is free, the player can jump over the opponents’s piece to remove it from the board. When a player’s piece reachers the opposite end of the game board, it is crowned and becomes a “King” that can move back and forth. The player with pieces left on the board wins the game.

Checkers games have been developed over time that allow users play against other players, or against bots that run on a specific algorithm, some of these algorithms perform better than others.

## 1.2 PROBLEM STATEMENT

In the development of an AI checkers game using minimax and alpha-beta pruning, we might

be faced with a computationally expensive implementation due to checkers having a large

branching factor. The challenge is to develop an efficient algorithm that can quickly

evaluate the best move while minimizing the number of nodes explored. The second challenge is to visualize six uninformed search algorithms (Depth-First Search [DFS], Depth-Limited Search [DLS], Iterative Deepening Search [IDS], Breath-First Search [BFS]. Bidirectional Search [BIDI], and Uniform-Cost Search [UCS]).

## 1.3 AIM AND OBJECTIVES

The aim of this project is to develop an efficient checkers game with Minimax and Alpha-Beta pruning algorithm with the goal of creating a high-performing AI that can play competitively against human players. Objectives of the game are:

1. Implement the Minimax algorithm with alpha beta pruning to create a basic checkers game AI that can play against human players.
2. Optimize the Minimax algorithm with alpha beta pruning by reducing the search space and improving the evaluation function to reduce the computational cost of the algorithm.

The aim and objective of the visualization task are to develop a visualization module for the six uninformed algorithms given.

1.4 METHODOLOGY

The methodology used for this project is the Agile methodology.

# LITERATURE REVIEW AND REAL-LIFE APPLICATIONS

## 2.1 MINIMAX ALGORITHM

The Minimax algorithm is a decision making-making algorithm commonly used in

two-player zero-sum games, where each player is trying to maximize their own score or

minimize their opponents score. The minimax algorithm works by exploring all possible

moves and their outcomes to determine the best move to make. The Minimax algorithm uses

any of the two strategies where the entire game tree is traversed to the leaf node or the tree is

traversed only to a set depth and is then evaluated. In many two-player games, such as chess

or go, there are an enormous number of possible moves that a player can make at any given

time. For example, in chess, there are approximately 20 legal moves that can be made for

each piece on the board. This means that after just two moves, there are already 400 possible

positions to evaluate. This number grows exponentially with each additional move, making

it infeasible to evaluate all possible moves to determine the best one. As the Minimax

algorithm explores the entire game tree to determine the best move to make, it is soon

realised that in some cases, most of the branches of the game tree are not worth exploring

further because they do not lead to desirable outcomes. Therefor reduing the search tree by

eliminating branches that are not worth exploring further can significantly improve the

efficiency of the algorithm.

### 2.1.1 MINIMAX ALGORITHM WITH ALPHA-BETA PRUNING

Alpha-Beta pruning is a commonly used technique for reducing the search tree in the

minimax algorithm. It works by maintaining upper and lover bounds on the score that can be

achieved for each node in the game tree. If the score for a particular node falls outside of

these bounds, then the algorithm can eliminate that branch of the tree as it cannot lead to a

desirable outcome. This allows the algorithm to prune large portions of the search tree and

focus on the most promising branches, resulting in significant savings in computational

resources. There have been many research done on Minimax Algorithm using Alpha-Beta pruning, for example, Rijul, Rishabh, et al, 2018 developed a Connect-4 game using Minimax algorithm and Alpha-Beta pruning and the Minimax algorithm alone. Their findings revealed that when both methods are presented with the same level of difficulty, the two algorithms behave differently in terms of iteration count and time taken with minimax and alpha-beta taking less time and performing less iterations to generate the game state. Sravya, Tejashwini and Sanjan, 2020 implemented the two types of alpha beta pruning methods on Tic-Tac-Toe and Checkers game, Parallel and Sequential methods. The results of their research showed that parallel Alpha Beta Pruning was more efficient and consumes lesser time than that of sequential alpha beta pruning. Vinay, Manjari and Nagalakshmi, 2020 developed an arbitrary decision two-player board game called Plot-4 and implemented a self learning player into it. The self learning player improves the game by predicting the opponents move ahead of time. Trained using three algorithms, Random, Minimax and Alpha-Beta Pruning, the results, based on time complexity, space complexity, latency and payoff all favoured Alpha Beta pruning as it was able to make ideal moves by evaluating a lesser number of nodes in the game tree, proving to be an efficient algorithm.

### 2.1.2 STRENGTHS OF MINIMAX ALGORITHM WITH ALPHA-BETA PRUNING

1. Optimal decision making: the minmax algorithm with alpha-beta pruning is guaranteed to find the optimal decision in two-player zero-sum games, where both players play optimally and there is complete knowledge of the game.
2. Efficient pruning of the search space: Alpha-Beta pruning allows the algorithm to efficiently prune search trees, eliminating branches that are not worth exploring further, and reducing the time and computational power required to evaluate all possible moves.
3. Ease of implementation: the minimax algorithm with alpa-beta pruning is relatively easy to implement and be applied to many problems with two-player zero-sum structure.

### 2.1.3 WEAKNESSES OF MINIMAX ALGORITHM WITH ALPHA-BETA PRUNING

1. Limited to two-player zero-sum games: The minmax algorithm with alpha-beta pruning is limited to two-player zero-sum games, where both players play optimally and there is complete knowledge of the game. It cannot be used for more complex problems that do not have a two-player zero-sum structure.
2. Can be computationally expensive: While alpha-beta pruning can help reduce the computational power required to evaluate all possible moves, the minimax algorithm can still be computationally expensive, especially for large search spaces.
3. Order of Evaluation: the efficiency of alpha-beta pruning depends on the order in which moves are evaluated. If a good move is evaluated later in the search, the algorithm may nor prune as many nodes, leading to longer computation times.
4. Win not guaranteed: while the minimax algorithm with alpha-beta pruning is optimal, it does not guarantee a win in games where there is an element of chance, such as card games. It can only make the best decision given the available information.

## 2.2 UNINFORMED SEARCH ALGORITHMS

### 2.2.1 DEPTH-FIRST SEARCH

The Depth-First Search (DFS) is a graph search algorithm that begins its search at the root node and continues its search to the deepest node before backtracking to unexplored nodes. A *visited* node array prevents processing a node more than once. DFS traversals may occur more than once in a graph.

Features:

* Time Complexity: O(bm)
* Space Complexity: O(bm)
* Completeness: No
* Optimal: No
* Data Structure: Stack

Where

b - branching factor of the graph

m - tree depth of graph

| STRENGTHS | WEAKNESSES |
| --- | --- |
| 1. Without doing much additional research, the answer can be found. | 1. As the cut-off depth decreases, time complexity increases. |
| 2. Since only the nodes on the current path are kept, depth-first search uses less memory than other methods | 2. It's possible that some states repeat more than once and finding the desired node cannot be guaranteed. |

Real World Applications

1. Can be used for solving puzzles with only one solution, such as a maze or a sudoku puzzle
2. It is used in web crawlers
3. It can also be used in maze generation
4. Backtracking
5. Topological Sorting

### 

### 2.2.2 DEPTH LIMITED SEARCH

The Depth-Limited Search (DLS) is a search algorithm that works the same as the depth-first search algorithm, except that in depth-limited search, a level limit, a start, and a goal node are added here. The reason a level limit is introduced here is to prevent exhaustive searches on graphs with infinite levels.

Features:

* Time Complexity: O(bl)
* Space Complexity: O(bl)
* Completeness: No
* Optimal: No
* Data Structure: Stack

Where

b - branching factor of the graph

l - search depth limit of graph

| STRENGTHS | WEAKNESSES |
| --- | --- |
| 1. Compared to DFS, depth-limited search is more effective and uses less time and memory. | 1. Incompleteness is a drawback of depth-limited search. |
| 2. DLS ensures that a solution, if one exists, will be discovered in a limited amount of time. | 2. When a problem has multiple possible solutions, it might not produce the best answer. |

Real World Applications

1. Can be used for solving puzzles with only one solution, such as a maze or a sudoku puzzle
2. It is used in web crawlers
3. It can also be used in maze generation
4. Backtracking
5. Topological Sorting

### 2.2.3 ITERATIVE DEEPENING SEARCH

Iterative Deepening Search (IDS or IDDFS) is a graph search algorithm and a variant of BFS in which a depth-limited version of depth-first search is periodically run with increasing depth limitations until the target is discovered.

Iterative deepening search is a search algorithm that combines the fast search feature of DFS and the good space complexity of BFS.

Features:

* Time Complexity: O(bd)
* Space Complexity: O(bd)
* Completeness: No
* Optimal: Yes
* Data Structure: Queue

Where

b - branching factor of the graph

d - tree depth of solution from graph

| STRENGTHS | WEAKNESSES |
| --- | --- |
| 1. A benefit of IDDFS can be seen in in-game tree searching, where the IDDFS search operation aims to enhance heuristics, scores of searching nodes, and the depth definition, to enable efficiency in the search algorithm | 1. The primary issue with IDDFS is the wasted amount of time and wasteful calculations that occur at each depth. |
| 2. In IDDFS, a solution can be found and there’s hope to locate it if it exists | 2. IDDFS also has the drawback of making repeated trips to some nodes, which might slow down the search |

Real World Applications

1. IDS can be used to solve games with trees, such as chess, checkers, and more.

### 2.2.4 BREADTH-FIRST SEARCH

Breadth-First Search is a recursive graph algorithm that begins by exploring all of the nearby nodes before moving down each level of the graph from the root node. It chooses the closest node and investigates each node that hasn't been explored yet.

Features:

* Time Complexity: O(bd)
* Space Complexity: O(bd)
* Completeness: Yes
* Optimal: Yes
* Data Structure: Queue

Where

b - branching factor of the graph

d - tree depth of solution from graph

| STRENGTHS | WEAKNESSES |
| --- | --- |
| 1. It will discover a solution with the fewest number of steps if there are multiple solutions. | 1. It uses a lot of memory since it needs to save every node from the current level before moving on to the next one. |
| 2. It is a complete algorithm. BFS is sure to find a solution if a solution exists. | 2. Finding a solution that is far deep in the level of the graph will take time. |

Real World Applications

1. BFS is used in GPS navigation systems to locate nearby locations.
2. The BFS algorithm is used in networking when we wish to broadcast a few packets.

### 2.2.5 BIDIRECTIONAL SEARCH

Bidirectional search is a graph search algorithm that seeks out the shortest route between a source and a goal vertex. It conducts two searches simultaneously, one forward and one backward search. The forward search begins from the source vertex toward the goal vertex and the backward search begins from the goal vertex toward the source vertex.

Features:

* Time Complexity: O(bd/2)
* Space Complexity: O(bd/2)
* Completeness: Yes
* Optimal: Yes
* Data Structure: Queue

Where

b - branching factor of the graph

d - tree depth of solution from graph

| STRENGTHS | WEAKNESSES |
| --- | --- |
| 1. A great advantage of the bidirectional search algorithm over BFS is the speed at which it locates the goal node. | 1. Without the goal node specified, the algorithm cannot work. |

Real World Applications

1. For finding the shortest path in maps
2. Can be used to generate synthetically generating bidirectional texture functions (BTFs)

### 2.2.6 UNIFORM COST SEARCH

Uniform Cost Search (UCS) is a graph algorithm that finds a path from the source to the destination is found using the least expensive cumulative cost. Starting at the root, nodes are expanded in accordance with the minimum cumulative cost.

Features:

* Time Complexity: O(bd)
* Space Complexity: O(bd)
* Completeness: Yes
* Optimal: Yes
* Data Structure: Priority Queue

Where

b - branching factor of the graph

d - tree depth of solution from graph

| STRENGTHS | WEAKNESSES |
| --- | --- |
| 1. Since the least path is assumed to be taken at each state, it is regarded as an optimal solution. | 1. The amount of storage required to run this algorithm is enormous. |
| 2. It helps in finding the route with the lowest total cost from the root node to the destination node in a weighted graph that has a different cost for each of its edges. | 2. Due to the algorithm's consideration of all potential routes from the root node to the destination node, it may become stuck in an infinite loop. |

Real World Applications

1. Can be used in maps to find the most optimal path from street A - C
2. Can be used in games like chess to find the least punishing move to make

# ALGORITHM DESIGN

## 3.1 MINMAX ALGORITHM

### 3.1.1 MINIMAX ALGORITHM DESIGN

The Minimax algorithm works by searching throught the game tree to find the best move for

the current player, assuming the opponent will also make their best move. The algorithm

design is defined as follows:

1. The game state is defined.
2. Player’s turn is determined.
3. A list of possible valid moves are generated.
4. For each generated move:
5. Apply the current move to the game state to create a new game state.
6. Using recursion, evaluate the new game state using minimax algorithm, alternating between maximizing and minimizing the value of each possible move.
7. Determine the value of the evaluated move by comparing it to the previously evaluated moves based of whether the current player is minimizing or maximizing.
8. Return the best move found in step 4.

### 3.1.2 MINIMAX ALGORITHM DESIGN WITH ALPHA BETA PRUNING

The algorithm design for Minimax with Alpha Beta pruning is similar to that of Minimax

without alpha-beta pruning, with the only difference being the parameters used to prune parts

of the tree that are guaranteed to be irrelevant.

1. The game state is defined.
2. Player’s turn is determined.
3. A list of possible valid moves are generated.
4. For each generated move:
   1. Apply the current move to the game state to create a new game state.
   2. Recursively evaluate the new game state using the minimax algorithm, alternating between maximizing and minimizing the value of each possible move.
   3. Update the alpha beta value based on the value of the evaluated move. Alpha is the highest value found so far and, beta is the lowest value found so far.
   4. If beta is less than or equal to alpha, prune the rest of the subtree and make the current move the best move.
5. Return the best move found in step 4.

## 3.2 UNIFORMED SEARCH ALGORITHMS

### 3.2.1 DEPTH-FIRST SEARCH

This algorithm works by first marking the current node as visited. Then, it iterates over all of the neighbors of the current node. If a neighbor has not been visited yet, the algorithm recursively calls itself on the neighbor. This process continues until all of the nodes in the graph have been visited.

Pseudocode:

def dfs(visited, graph, node):

"""

Performs a depth-first search on the given graph, starting at the given node.

Args:

visited: A set of nodes that have already been visited.

graph: The graph to search.

node: The node to start the search at.

Returns:

A list of the nodes that were visited in the order they were visited.

"""

# Mark the current node as visited.

visited.add(node)

# For each neighbor of the current node:

for neighbor in graph[node]:

# If the neighbor has not been visited yet:

if neighbor not in visited:

# Recursively call DFS on the neighbor.

dfs(visited, graph, neighbor)

# Return the list of visited nodes.

return visited

### 3.2.2 DEPTH LIMITED SEARCH

This algorithm works by first marking the current node as visited. Then, it iterates over all of the neighbors of the current node. If a neighbor has not been visited yet and its depth is less than or equal to the limit, the algorithm recursively calls itself on the neighbor. This process continues until all of the nodes in the graph have been visited or the limit has been reached.

Pseudocode:

def dls(visited, graph, node, limit):

"""

Performs a depth-limited search on the given graph, starting at the given node, with a limit.

Args:

visited: A set of nodes that have already been visited.

graph: The graph to search.

node: The node to start the search at.

limit: The maximum depth to search.

Returns:

A list of the nodes that were visited in the order they were visited.

"""

# Mark the current node as visited.

visited.add(node)

# For each neighbor of the current node:

for neighbor in graph[node]:

# If the neighbor has not been visited yet and its depth is less than or equal to the limit:

if neighbor not in visited and int(node\_with\_levels[neighbor]) <= limit:

# Recursively call DFS on the neighbor.

dls(visited, graph, neighbor, limit)

# Return the list of visited nodes.

return visited

### 3.2.3 ITERATIVE DEEPENING SEARCH

This algorithm works by first initializing the visited set to be empty. Then, it iterates over the depths from 0 to *maxDepth*. For each depth, it clears the visited set and performs a depth-limited search with the current depth.

If the target node is found during the depth-limited search, the algorithm returns *Tru*e. Otherwise, the algorithm continues iterating over the depths. If the target node is not found after iterating over all of the depths, the algorithm returns False.

Pseudocode:

def iterDeepSearch(src, target, maxDepth):

"""

Performs an iterative deepening search on the given graph, starting at the given node, with a maximum depth.

Args:

src: The source node.

target: The target node.

maxDepth: The maximum depth to search.

Returns:

True if the target node was found, False otherwise.

"""

# Initialize the visited list.

visited = []

# Iterate over the depths from 0 to maxDepth.

for depth in range(maxDepth + 1):

# If the target node has been found, return True.

if target in visited:

return True

# Clear the visited set.

visited.clear()

# Perform a depth-limited search with the current depth.

dls(visited, graph, src, target, False, depth)

# Return False if the target node was not found.

return False

def dls(visited, graph, node, goal\_node, foundGoalNode, limit):

"""

Performs a depth-limited search on the given graph, starting at the given node, with a limit.

Args:

visited: A set of nodes that have already been visited.

graph: The graph to search.

node: The node to start the search at.

goal\_node: The goal node.

foundGoalNode: A boolean flag that indicates if the goal node has been found.

limit: The maximum depth to search.

Returns:

A list of the nodes that were visited in the order they were visited.

"""

# Mark the current node as visited.

visited.add(node)

# If the current node is the goal node, return True.

if node == goal\_node:

foundGoalNode = True

return True

# For each neighbor of the current node:

for neighbor in graph[node]:

# If the neighbor has not been visited yet and its depth is less than or equal to the limit:

if neighbor not in visited and int(node\_with\_levels[neighbor]) <= limit:

# Recursively call DFS on the neighbor.

dls(visited, graph, neighbor, goal\_node, foundGoalNode, limit)

# If the goal node was not found, return False.

if not foundGoalNode:

return False

# Return the list of visited nodes.

return visited

### 3.2.4 BREADTH-FIRST SEARCH

This algorithm works by first marking the current node as visited. Then, it creates a queue to store the nodes that have not yet been visited. The algorithm repeatedly pops the first node off of the queue and marks it as visited. For each neighbor of the current node, the algorithm checks if the neighbor has already been visited. If the neighbor has not been visited, the algorithm adds the neighbor to the queue. This process continues until the queue is empty, which means that all of the nodes in the graph have been visited.

Pseudocode:

def bfs(visited, graph, node):

"""

Performs a breadth-first search on the given graph, starting at the given node.

Args:

visited: A set of nodes that have already been visited.

graph: The graph to search.

node: The node to start the search at.

Returns:

A list of the nodes that were visited in the order they were visited.

"""

# Mark the current node as visited.

visited.add(node)

# Create a queue to store the nodes that have not yet been visited.

queue = [node]

# While the queue is not empty:

while len(queue) > 0:

# Remove the first node from the queue.

s = queue.pop(0)

# For each neighbor of the current node:

for next in graph[s]:

# If the neighbor has not been visited yet:

if next not in visited:

# Mark the neighbor as visited.

visited.add(next)

# Add the neighbor to the queue.

queue.append(next)

# Return the list of visited nodes.

return visited

### 3.2.5 BIDIRECTIONAL SEARCH

This algorithm works by first initializing the visited set to be empty. Then, it creates two queues, one for the forward search and one for the backward search. The algorithm repeatedly pops the first node off of the forward queue and marks it as visited. For each neighbor of the node, the algorithm checks if the neighbor has already been visited. If the neighbor has not been visited, the algorithm adds the neighbor to the forward queue. This process continues until the forward queue is empty.

The algorithm then switches to the backward search. It repeatedly pops the first node off of the backward queue and marks it as visited. For each neighbor of the node, the algorithm checks if the neighbor has already been visited. If the neighbor has not been visited, the algorithm adds the neighbor to the backward queue. This process continues until the backward queue is empty.

If either queue is empty before the other, the goal node was not found. Otherwise, the algorithm returns the list of visited nodes.

def bidirectional\_search(graph, start, goal):

"""

Performs a bidirectional search on the given graph, starting at the given start node and ending at the given goal node.

Args:

graph: The graph to search.

start: The start node.

goal: The goal node.

Returns:

A list of the nodes that were visited in the order they were visited.

"""

# Initialize the visited set.

visited = set()

# Create two queues, one for the forward search and one for the backward search.

forward\_queue = [start]

backward\_queue = [goal]

# While both queues are not empty:

while forward\_queue and backward\_queue:

# Pop the first node from the forward queue.

node = forward\_queue.pop(0)

# If the node is the goal node, return the list of visited nodes.

if node == goal:

return visited

# Mark the node as visited.

visited.add(node)

# For each neighbor of the node, add it to the forward queue.

for neighbor in graph[node]:

if neighbor not in visited:

forward\_queue.append(neighbor)

# Pop the first node from the backward queue.

node = backward\_queue.pop(0)

# If the node is the start node, return the list of visited nodes.

if node == start:

return visited

# Mark the node as visited.

visited.add(node)

# For each neighbor of the node, add it to the backward queue.

for neighbor in graph[node]:

if neighbor not in visited:

backward\_queue.append(neighbor)

# If both queues are empty, the goal node was not found.

return None

# SOFTWARE DESIGN

## 4.1 CHECKERS GAME

The Checkers game developed offers two game modes - player vs player and, player vs

computer. The player vs computer mode has the computer running the Minimax algorithm

with alpha-beta pruning. Following the rules of checkers, we design an 8x8 interface to

represent the checkered board and 12 each pieces to represent the players. The two piece

colors are Red and White with White representing the AI in Human vs Computer mode and

the Red piece making the first move. The software also takes into account, King pieces, this

is when a piece reaches the last row of the opponents side of the board.

## 4.1.1 GAME STATE REPRESENTATION

The game state is represented by a data structure that stores the current position of all pieces

on the board, the player whose turn it is and, any other relevant game information such as the

number of captured pieces.

4.1.2 USER INTERFACE

The user interface allows players view the current game state and make moves by selecting a

piece and a valid move destination.

## 4.1.3 GAME LOGIC

The game logic handles the movement of pieces on the board and the determination of the

winner. This is implemented using functions that perform move validation, game state update

and checking for win condition or draw condition.

## 4.1.4 MINIMAX WITH ALPHA BETA PRUNING

The minimax algorithm with alpha-beta pruning is used to determine the best move for the

current player. This is implemented using a recursive function that evaluates each possible

move by simulating the game to a certain depth and using a heuristic function to assign a

score to each position. The alpha-beta pruning technique is implemented to optimize the

algorithm by cutting of search paths that are guaranteed to be irrelevant.

## 4.1.5 HEURISTIC EVALUATION

The Heuristic Evaluation should evaluate the value of a game state based on the position of

pieces on the board, the number of pieces, and other relevant factors such as the potential for

future moves. This is implemented as a function that takes the current game state and returns

a numerical value.

## 4.1.6 GAME MODES

The software program has two modes that can be played by the user. These modes represent

how the game is run on start up and ensures the entire state of the game for that instance runs

the appropriate way.

## 4.1.7 DIFFICULTY LEVEL

The software program must have a difficulty level for the game mode Human vs Computer.

This is determined by the level of depth the computer is meant to search through.

## 4.1.8 INTEGRATION

All of these components are integrated into a working program that runs the game and allows

the player make moves and view the current game state.

# SOFTWARE IMPLEMENTATION

## 5.1 CHECKERS GAME

### 5.1.1 CHECKERS GAME IMPLEMENTATION OVERVIEW

The programming language adopted for this project is the Python programming language. The Integrated Development Environmant used for this project is Pycharm Community Edition.

#### 5.1.1.1 INTERFACE IMPLEMENTATION

The first step is to build the user interface for playing the game. This is achieved using pygame, a python graphical library for developing multimedia applications like games. With pygame, we are able to design a 8x8 display that serves as our checkers board. A width and height of 800 is set to create the board with equal number of rows and columns of values 8. The total square size is calculated as the division of the width by the column values, which gives a value of 60.

The first point of contact is the initialization of the *GameLogic* class, this class creates a new instance of the *Board* class which creates the board pieces in the board list. The board list is a multidimensional array of 8x8 size and has instances of Piece at points that denote where the pieces Red and White would be on the checkers board. For cells that do not contain any pieces, the value 0 is used in position of that cell. At the end of this process, the multidimensional list board now holds the structure of the checkers board, we can now proceed to build and fit the elements on the user interface.

The *setBoard(window)* method in the *Board* class is invoked, this method is responsible for setting the cells on the checkers board. This works by iterating through the elements of the *board* list and invoking the *drawpiece(window)* on it. The *drawpiece(window)* method invokes the *circle* method in the *pygame* class and colors the cell with the color of the piece being iterated on. This method is also used to update King pieces so, it also checks if the piece is a king piece and places a crown image in the center of it.

This method first fills the entire screen with the color black. It iterates through the number of rows available and for each, it iterates again from the result of the modulus of that row number by 2 to the total number of rows, skipping two steps. The next point of action is to draw a rectangle by creating a cell and coloring it red on top of the black background.

#### 5.1.1.2 MOVING PIECES

Moving pieces on the checkers board works on a turn based system with the RED piece making the first move. Once a player initiates a move by clicking the mouse button, a *MOUSEBUTTONDOWN* event is recorded and the position of the mouse click is stored. A *GetRowAndColFromMouseClick(pos)* is then invoked and the approximate position of the x and y values are calculated based on the integer division by the Square Size. The next step is to check for the valid moves that can be made from the position of that piece, the *select* method is invoked and it fetches the piece that belongs to that row and column in the board list. If the piece is not 0 and the piece corresponds to its turn, the piece is assigned to the *selected* property and the *GetValidMoves(piece)* method is invoked on that piece. The valid moves are then returned and the board is updated by the *draw* method using a blue circle on the possible cells. The *select* method then tries to check recursively if there is an avenue for a double jump by passing the current position of the valid moves. Upon selecting any of the valid cells to move to, the *select* method is invoked again but this time, the *MovePiece(self.selected, row, col)* method is called instead. The *MovePiece* method takes the current piece, the row and col to be moved to. This sets the new value of the board list with the corresponding row and col and sets the current state of the board to the value of the piece, usually 0. This method also checks if the new position if the edge of the opponent’s and sets the piece to be a King piece. The list of valid moves are then removed from the board list to imitate a jump over an opponents piece. This same process is performed by the AI but using Minimax and Alpha Beta Pruning.

#### 5.1.1.3 MOVING PIECES WITH MINIMAX AND ALPHA BETA PRUNING

On the AI’s turn, the *minimax\_alpha\_beta* method is invoked and the current board state, the

depth(also termed difficulty level), the current player method(maximizing or minimizing), -infinity, +infiinty and the *GameLogic*

instance are passed into the method. The first step is to check there is already a winner, if no,

the Heuristic Function is evaluated and the result alongside the piece are returned as a tuple.

If not, check if the current player is trying to maximize, if yes, set the current *maxEvaluation*

to -infinity and the *best\_move* to None. Iterate through all the moves for that piece by

invoking the *get\_all\_moves* method, for each move, recursively call the *minimax\_alpha\_beta*

method and pass in the current move, the current depth - 1, set the Maximizing variable to

False, current alpha, current beta and the current game state and store the result in a variable

*evaluationResult*. We then check if the *evaluationResult* is greater than the *maxEvaluation*, if

it is, we set the current move to our *best\_move*. We then compare the greater value between

*alpha* and *beta*. If *alpha* is greater, we set our *best\_move* to the current move and exit the

Iteration and return a tuple of the *maxEvaluation* and the *best\_move*. The same logic is

applied if the player is trying to minimize except that, we work with the minimum values

between *minEvaluation* (which is ) and *evaluationResult*. Once a move has been gotten, we

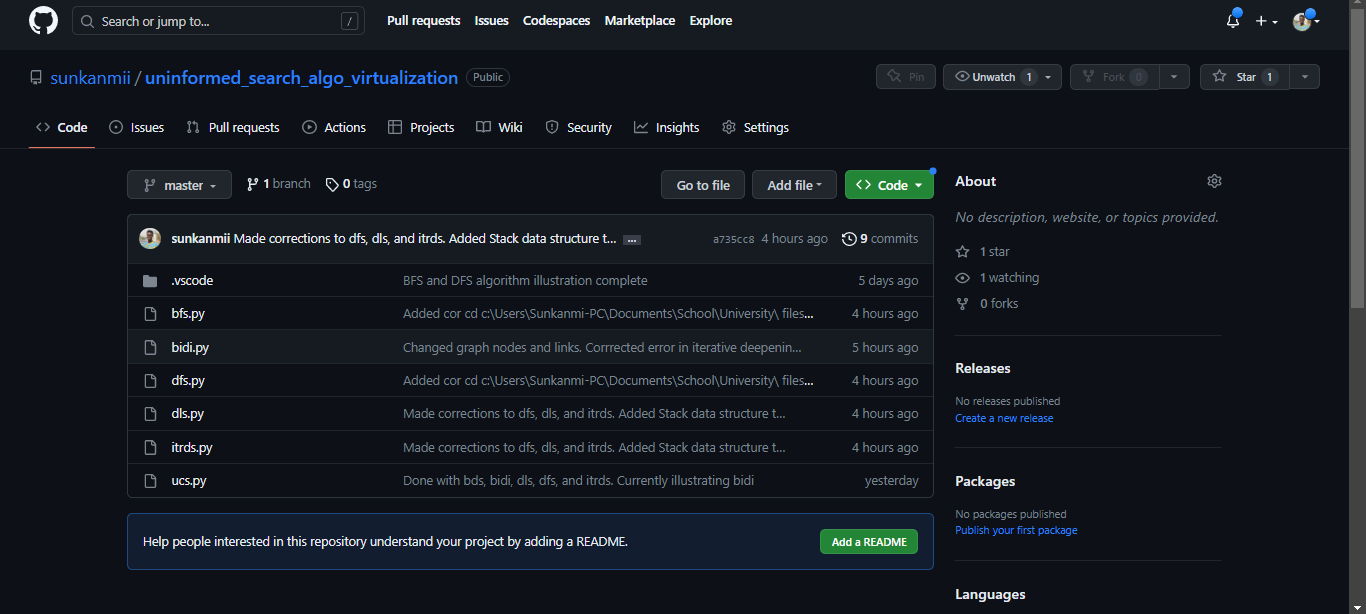
then simulate the move by updating the current state of the board with the state of the board

at the result of the *minimax\_alpha\_beta* method.

## 5.2 UNINFORMED SEARCH ALGORITHMS

* **Code Editor**: Visual Studio
* **Language:** Python
* **Visualization:** Matplotlib

The six uninformed search algorithm was implemented using the Python programming language and visualized using the Matplotlib library. The six uninformed search implementation code is uploaded on [Github](https://github.com/sunkanmii/uninformed_search_algo_virtualization).



***Link****:*[*https://github.com/sunkanmii/uninformed\_search\_algo\_virtualization*](https://github.com/sunkanmii/uninformed_search_algo_virtualization)

An example of the graph input some of the algorithms took in and visualized is:

# Graph

graph = {

'1' : ['2','3'],

'2' : ['4', '5'],

'3' : ['6'],

'4' : ['5'],

'5' : ['6'],

'6' : ['7'],

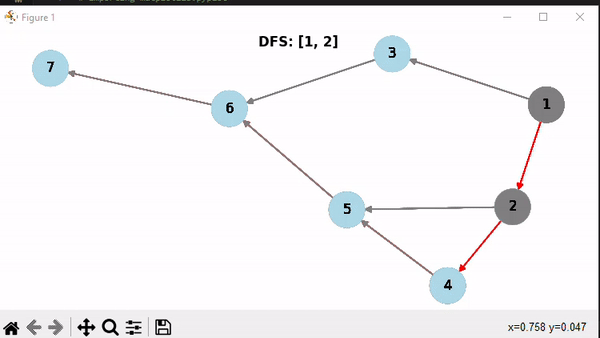
'7' : []

}

The graph input is a dictionary that maps nodes to their neighbors. Each node is a string, and each neighbor is a list of strings. For example, the node 1 has two neighbors, 2 and 3. Node 2 has two neighbors, 4 and 5. Node 3 has one neighbor, 6. And so on.

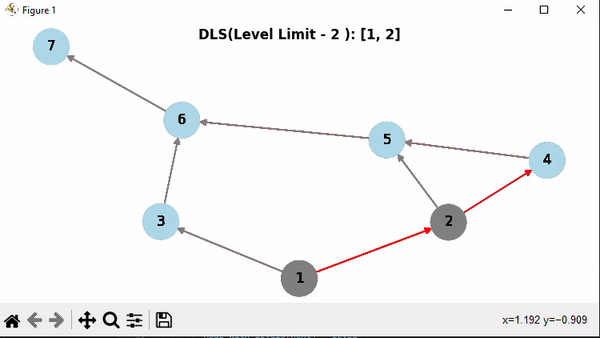
### 5.2.1 DEPTH-FIRST SEARCH

**Output**:

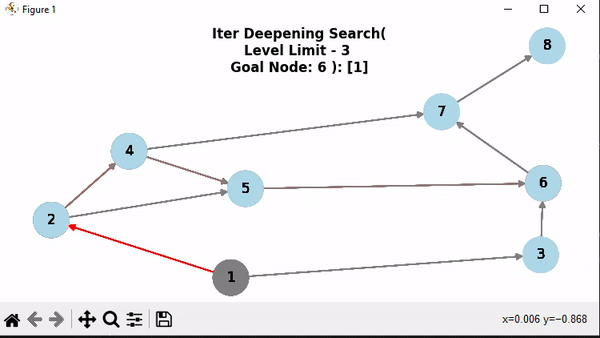


### 5.2.2 DEPTH-LIMITED SEARCH

**Output**:

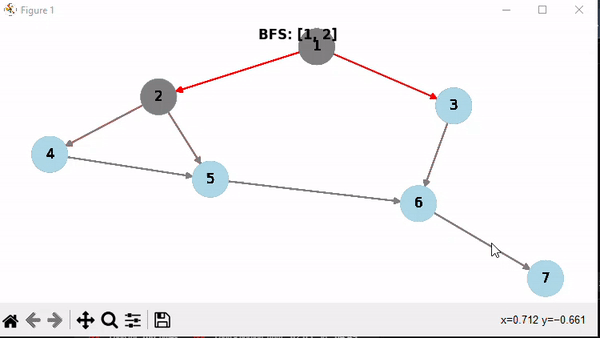


### 5.2.3 ITERATIVE-DEEPENING SEARCH

**Output**:  


### 5.2.4 BREADTH FIRST SEARCH

**Output**:



### 5.2.5 BIDIRECTIONAL SEARCH

Any graph can be used in any of these algorithms and to illustrate this, this graph was used in this bidirectional algorithm:

# Graph

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : ['4'],

'4' : ['8'],

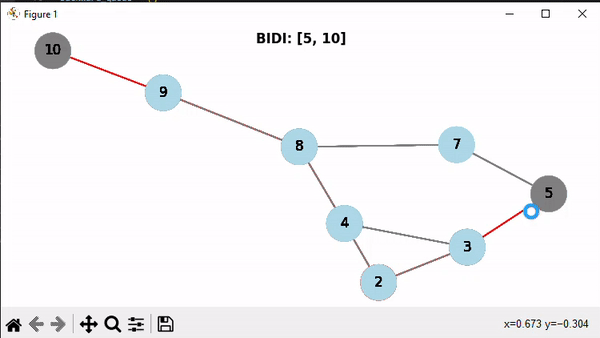
'8' : ['9'],

'9' : ['10'],

'10': []

}

**Output**:



# 

# RESULTS

## 6.1 CHECKERS GAME

During the Human vs Computer trial of the Checkers game, the human player was easily able

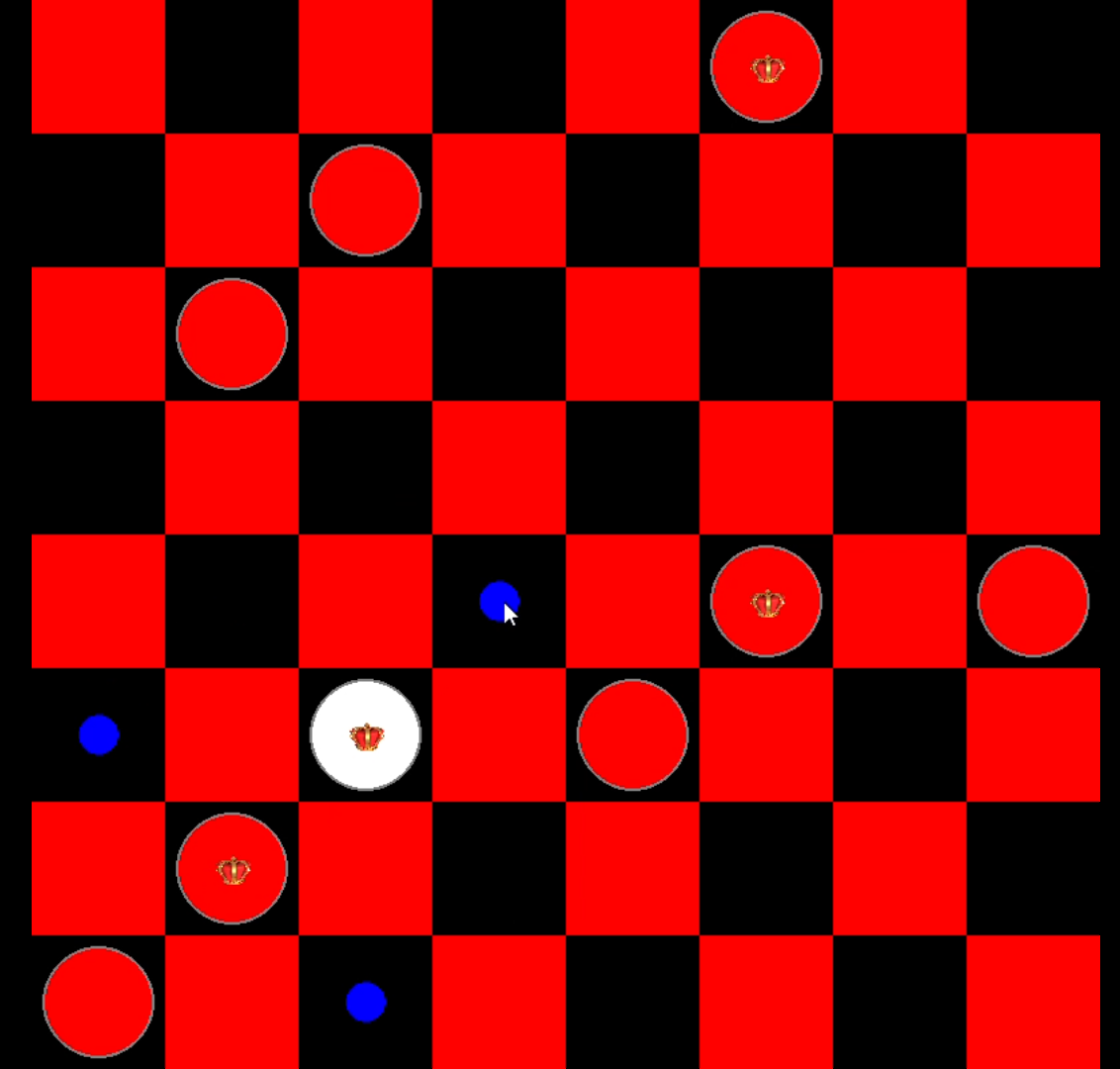
to beat the computer with an average of 6 red pieces left whenever the depth level set to 1.

When the depth value increases, it becomes more difficult to beat the AI in the game. It was

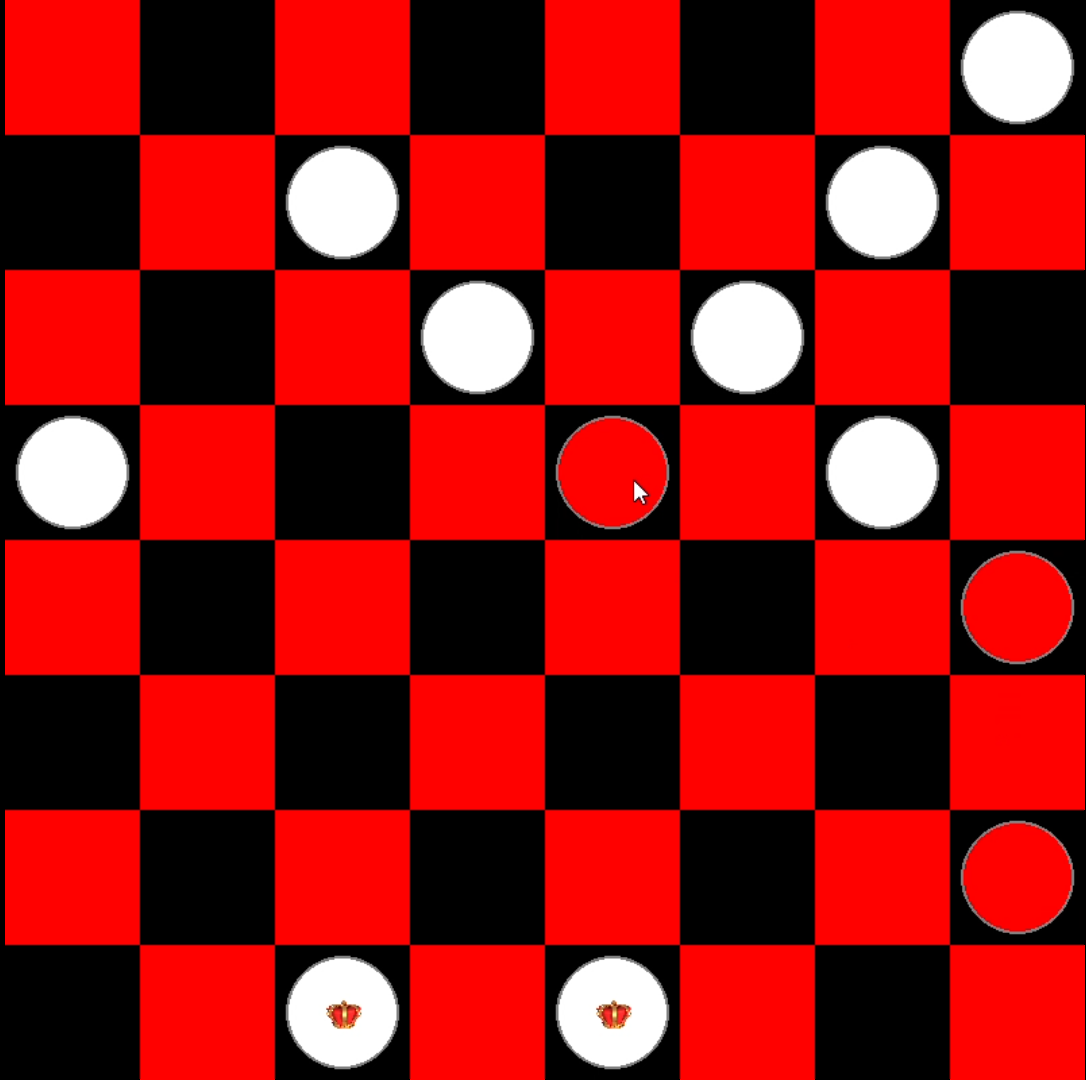
also noted that the time taken for the AI to make a move increased based on the

depth/difficulty level, this was as a result of the number of possible options being explored

by the algorithm.



[Checkers Player vs AI with depth 1](https://drive.google.com/file/d/1hxSGNPfXuaqsJGGFlHFg5P3-og_hB7AL/view?usp=sharing)



[Checkers Player vs AI with depth 4](https://drive.google.com/file/d/17SZ5J1pOTlyqsMboFsk8pMbWg8pWPNEA/view?usp=sharing)

# 

## 6.2 UNINFORMED SEARCH ALGORITHMS

The uniformed search algorithms were visualized using Python and other various tools listed. Link to code was provided for further review and outputs were produced in gif image format for easy view.

# CONCLUSION

## 7.1 CHECKERS GAME

The Checkers game developed using Minimax Algorithm with Alpha-Beta pruning proved to

be an effective algorithm in developing an AI to compete against human players. With the

shortcomings of longer computing time as the depth of the search space increases, the AI

developed performed well. This logic of using the depth level as a difficulty level can be used

to develop a mobile app game where the user can set the difficulty to play with against the

AI.

## 7.2 UNINFORMED SEARCH ALGORITHMS

The uninformed search algorithms improved our ability to critically think and display a visualization of how nodes are traversed in these algorithms It was also deduced that uninformed search algorithms can be used to solve a wide variety of problems and can be used numerous real-life applications.

# APPENDICES

## 8.1 CHECKERS GAME

*select()* method - selects a piece and attempts to move it to the new col and row

def select(self, row, col):

if self.selected:

result = self.\_move(row, col)

if not result:

self.selected = None

self.select(row, col)

piece = self.board.GetPiece(row, col)

if piece != 0 and piece.color == self.turn:

self.selected = piece

self.validMoves = self.board.GetValidMoves(piece)

return True

return False

*Evaluate =* computes the heuristic score

def evaluate(self):

return self.whiteLeft - self.redLeft + (self.whiteKings \* 0.5 - self.redKings \* 0.5)

def MovePiece(self, piece, row, col):

self.board[piece.row][piece.col], self.board[row][col] = self.board[row][col], self.board[piece.row][piece.col]

piece.Move(row, col)

if row == ROWS - 1 or row == 0:

if piece.color == RED and not piece.isKingPiece:

self.redKings += 1

piece.MakePieceAKing()

if piece.color == WHITE and not piece.isKingPiece:

self.whiteKings += 1

piece.MakePieceAKing()

*setBoard -* for setting the background of the board

def setBoard(self, window):

window.fill(BLACK)

for row in range(ROWS):

for col in range(row % 2, ROWS, 2):

pygame.draw.rect(window, RED, (row \* SQUARE\_SIZE, col \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))

*Minimax\_alpha\_beta*

def minimax\_alpha\_beta(position, depth, maximizingPlayer, alpha, beta, game):

if depth == 0 or position.markWinner() is not None:

return position.evaluate(), position

if maximizingPlayer:

maxEvaluation = float('-inf')

best\_move = None

for move in get\_all\_moves(position, WHITE, game):

evaluationResult = minimax\_alpha\_beta(move, depth - 1, False, alpha, beta, game)[0]

if evaluationResult > maxEvaluation:

maxEvaluation = evaluationResult

best\_move = move

alpha = max(alpha, evaluationResult)

if beta <= alpha:

best\_move = move

break

return maxEvaluation, best\_move

else:

minEvaluation = float('inf')

best\_move = None

for move in get\_all\_moves(position, RED, game):

evaluationResult = minimax\_alpha\_beta(move, depth - 1, True, alpha, beta, game)[0]

if evaluationResult < minEvaluation:

minEvaluation = evaluationResult

best\_move = move

beta = min(beta, evaluationResult)

if beta <= alpha:

best\_move = move

break

return minEvaluation, best\_move

Complete code can be found [here](https://github.com/olagesin/Checkers)

## 8.2 UNINFORMED SEARCH ALGORITHMS

Pseudocodes:  
1. DEPTH-FIRST SEARCH

def dfs(visited, graph, node):

"""

Performs a depth-first search on the given graph, starting at the given node.

Args:

visited: A set of nodes that have already been visited.

graph: The graph to search.

node: The node to start the search at.

Returns:

A list of the nodes that were visited in the order they were visited.

"""

# Mark the current node as visited.

visited.add(node)

# For each neighbor of the current node:

for neighbor in graph[node]:

# If the neighbor has not been visited yet:

if neighbor not in visited:

# Recursively call DFS on the neighbor.

dfs(visited, graph, neighbor)

# Return the list of visited nodes.

return visited

2. DEPTH LIMITED SEARCH

def dls(visited, graph, node, limit):

"""

Performs a depth-limited search on the given graph, starting at the given node, with a limit.

Args:

visited: A set of nodes that have already been visited.

graph: The graph to search.

node: The node to start the search at.

limit: The maximum depth to search.

Returns:

A list of the nodes that were visited in the order they were visited.

"""

# Mark the current node as visited.

visited.add(node)

# For each neighbor of the current node:

for neighbor in graph[node]:

# If the neighbor has not been visited yet and its depth is less than or equal to the limit:

if neighbor not in visited and int(node\_with\_levels[neighbor]) <= limit:

# Recursively call DFS on the neighbor.

dls(visited, graph, neighbor, limit)

# Return the list of visited nodes.

return visited

3. ITERATIVE DEEPENING SEARCH

def iterDeepSearch(src, target, maxDepth):

"""

Performs an iterative deepening search on the given graph, starting at the given node, with a maximum depth.

Args:

src: The source node.

target: The target node.

maxDepth: The maximum depth to search.

Returns:

True if the target node was found, False otherwise.

"""

# Initialize the visited list.

visited = []

# Iterate over the depths from 0 to maxDepth.

for depth in range(maxDepth + 1):

# If the target node has been found, return True.

if target in visited:

return True

# Clear the visited set.

visited.clear()

# Perform a depth-limited search with the current depth.

dls(visited, graph, src, target, False, depth)

# Return False if the target node was not found.

return False

def dls(visited, graph, node, goal\_node, foundGoalNode, limit):

"""

Performs a depth-limited search on the given graph, starting at the given node, with a limit.

Args:

visited: A set of nodes that have already been visited.

graph: The graph to search.

node: The node to start the search at.

goal\_node: The goal node.

foundGoalNode: A boolean flag that indicates if the goal node has been found.

limit: The maximum depth to search.

Returns:

A list of the nodes that were visited in the order they were visited.

"""

# Mark the current node as visited.

visited.add(node)

# If the current node is the goal node, return True.

if node == goal\_node:

foundGoalNode = True

return True

# For each neighbor of the current node:

for neighbor in graph[node]:

# If the neighbor has not been visited yet and its depth is less than or equal to the limit:

if neighbor not in visited and int(node\_with\_levels[neighbor]) <= limit:

# Recursively call DFS on the neighbor.

dls(visited, graph, neighbor, goal\_node, foundGoalNode, limit)

# If the goal node was not found, return False.

if not foundGoalNode:

return False

# Return the list of visited nodes.

return visited

4. BREADTH FIRST SEARCH

def bfs(visited, graph, node):

"""

Performs a breadth-first search on the given graph, starting at the given node.

Args:

visited: A set of nodes that have already been visited.

graph: The graph to search.

node: The node to start the search at.

Returns:

A list of the nodes that were visited in the order they were visited.

"""

# Mark the current node as visited.

visited.add(node)

# Create a queue to store the nodes that have not yet been visited.

queue = [node]

# While the queue is not empty:

while len(queue) > 0:

# Remove the first node from the queue.

s = queue.pop(0)

# For each neighbor of the current node:

for next in graph[s]:

# If the neighbor has not been visited yet:

if next not in visited:

# Mark the neighbor as visited.

visited.add(next)

# Add the neighbor to the queue.

queue.append(next)

# Return the list of visited nodes.

return visited

5. BIDIRECTIONAL SEARCH

def bidirectional\_search(graph, start, goal):

"""

Performs a bidirectional search on the given graph, starting at the given start node and ending at the given goal node.

Args:

graph: The graph to search.

start: The start node.

goal: The goal node.

Returns:

A list of the nodes that were visited in the order they were visited.

"""

# Initialize the visited set.

visited = set()

# Create two queues, one for the forward search and one for the backward search.

forward\_queue = [start]

backward\_queue = [goal]

# While both queues are not empty:

while forward\_queue and backward\_queue:

# Pop the first node from the forward queue.

node = forward\_queue.pop(0)

# If the node is the goal node, return the list of visited nodes.

if node == goal:

return visited

# Mark the node as visited.

visited.add(node)

# For each neighbor of the node, add it to the forward queue.

for neighbor in graph[node]:

if neighbor not in visited:

forward\_queue.append(neighbor)

# Pop the first node from the backward queue.

node = backward\_queue.pop(0)

# If the node is the start node, return the list of visited nodes.

if node == start:

return visited

# Mark the node as visited.

visited.add(node)

# For each neighbor of the node, add it to the backward queue.

for neighbor in graph[node]:

if neighbor not in visited:

backward\_queue.append(neighbor)

# If both queues are empty, the goal node was not found.

return None