# Instructions

This program has both the command line interface and a graphical user interface that allow the user to create new maze and visualize the path finding process. In order to use the command line interface, the user use the command: search <filename> <method> where filename is the path to a .txt file that specifies the maze configuration (including start, goals, and wall points). Method is the search algorithm that will be used in the path finding process. There are 6 different methods, including Breadth-first search (BFS), Depth-first search (DFS), Greedy Best first search (GBFS), A Star Search (AS), Bidirectional Search (CUS1), Iterative Deepening A Star Search (CUS2) (Note: you only need to specify the abbreviation inside the brackets, not the full name). If the arguments are valid, the program will run the searching process and produce the following output:

filename method number\_of\_nodes  
path

where filename is the path to the .txt file, method is the search algorithm, number\_of\_nodes is the ????, and path is either a sequence of moves in the solution to reach one goal from the start location or the message “No solution found” if there is no solution.

On the other hand, the user can use the graphical interface to create a new maze.txt file and see the searching process visually. To create a new maze, the user have to speicified the row, column, start, goals, and walls points. To select a point in a maze, the user can simply click on a square in the maze and press the confirm button. Additionally, for the goals and walls, you can select multiple squares and remove selected squares by clicking on the already selected squares.

A picture containing crossword puzzle, text, appliance, fruit

Description automatically generated

After creating the maze, the user can save the file and export it to a location in the computer. Next, the user can go back to the homepage to import the maze.txt file that are just created and start the searching process. Finally, in the search page, the user can select one of 6 algorithms to see the searching process.

A picture containing website

Description automatically generated

Them legend

# Introduction

In the Robot Navigation problem, the environment is an N x M grid where N and M are both greater than 1. Moreover, there are a number of occupied wall cell that can not be reached. A robot is initally located in one of the cell and it is required to avoid the walls and find a path to reach one of the designated goals. To find a solution for this problem, we can use several search algorithm such as Breadth-first search and Depth-first search in order to find a correct path. Therefore, in this report, we will look at some common algorithms that has been used in the program and compare them based on their time complexity, memory.

# Search Algorithms

## Uninformed search algorithms

Uninformed search algorithms are basic algorithms that can find a solution in a search tree using a brute-force method without using any domain-specific knowledge to guide the search. Some common uninformed algorithms are Breadth-first search, Depth-first search, and Bidirectional Search:

* Breadth-first search (BFS): In BFS, the algorithm will start at the root node and keep expanding the shallowest unexpanded node until it finds a solution. Therefore, it will check all nodes at the same depth level before moving on to explore the nodes at the next depth. As a result, BFS is can find the shortest path from the root node to the solution if the actions are unweighted. Furthermore, BFS store the nodes on the queue in a first-in first-out principle to ensure the shallowest node is explored first. Its time complexity is *O*(*bd*+ 1),where b is the branching factor, and d is the distance from the start node to the goal node. Moreover, because the BFS need to store all the nodes at the same depth; therefore, the memory consumption will be huge if the maximum width in the graph is large. Them completeness
* Depth-first search (DFS): Instead of exploring the shallowest unexpanded node, the DFS will check the deepest unexpanded node. The algorithm will start at the root node and recursively explore all the children node along the same branch as far as possible before backtracking and check different branches. As a result, the found solution is not guaranteed to be the shortest path from the root node to the solution. To implement DFS, we can use a stack data structure in a last-in first-out faction to make sure the deepest node is explored first. For the time complexity, it is O(bd) sdf where d is the depth of the goal node ?. For the memory consumption, because depth first search only need to keep track of the nodes in the chain; therefore it would typically consume less memory than BFS Them completeness
* Bidirection search: Rather than starting at the first node and keep exploring the children node until we find a solution, bidirection start searching simultaneously at both the first node and goal node and keep exploring until it finds a common node. If we use BFS as the method to explore the start and goal node, our solution will be the shortest path if the actions are unweighted. For the time complexity, as each search only need to check half of the depth, the time complexity is O(bd/2 +bd/2)

## Informed search algorithms

On the other hand, informed algorithm use domain-specific knowledge to guide the search and find the solution; therefore, the search will be more efficient in terms of time and search space. Specificly, informed algorithms use heuristic functions to evaluate how “promising” a node is (i.e how likely that node will lead to the goal) and explore the most promosing node in the current search tree. In the Robot navigation problem, we can evaluate the nodes by using the Manhattan distance which is the sum of the distance along the x and y axes between 2 points: f(A,B) = |xA – xB| + |yA – yB| where A(xA,yB), B(xB,yB) are the coordinates of 2 points A and B. Notably, the manhttan distance is admissible as it never overestimates the actual distance to reach the goal. The 3 common informed search algorithm that is used in the program are Greedy Best First Search, A\* Search, and Iterative Deepening A\* (IDA):

* Greedy Best First Search (GBFS): In GBFS, we only use the heurisitc function to evaluate the nodes; therefore f(n) = h(n). In particular, GBFS estimates the cost of each possbile path to the goal and select the path with the lowest cost. To do this, GBFS stores the nodes in a queue and sort them based on their heurisitc value to choose the node with the lowest value to explore. However, the solution produced by GBFS is not always the optimal path with the lowest cost as it only choose the most “promising” node and not consider the cost it has taken so far to reach that node from the start. The time and space complexity for GBFS is O(bm) with b as the branching factor and m is the maximum depth of the tree.
* A\* Search (AS): In AS, we evalute a node by combining its heurisitc function h(n) and the cost that it has taken so far to reach that node g(n); therefore f(n) = h(n) + g(n). In paritular, AS store the nodes in a priority queue and sort them based on f(n) to select the path with the lowest cost. Because AS takes the travelled distance into account, it can avoid exploring the expensive path and search other paths with less cost. Especially, the solution produced by AS is guaranteed to be the shortest path if the heurisitic function h(n) is admissible.The time complexity of A\* is dependent on the structure of the graph and the heuristic function; in the worst case the number of expanded nodes is exponential with the path length of the solution. In addition, one drawback of the A\* algorithm is the memory consumption as it stores all the generated nodes.
* Iterative Deepening A\* (IDA): IDA is a variant of Iterative Deepening Depth First search that use heuristic function to evaluate the possible paths. In particular, IDA only explore the nodes with f(n) value smaller than the threshold number and put the nodes with higher f(n) value into a pruned list. If there is no more nodes to explore and the solution is not found, it will update the threshold value to the smallest f(n) value in the pruned list and restart the whole search again. Similar to A\*, IDA will find the optimal path to the goal if the heuristic function is admissible. However, IDA will use less memory than A\* as it is a depth-first search algorithm

# Implementation

In this section, we will discuss the implementation of the mentioned search algorithms in pseudocode and analyze the differences between the algorithms in more details. Firstly, each search algorithm is implemented in a class and derived from an abstract class called SearchAlgorithm that specifies the required attributes and methods. However, each algorithms will have different implementation and search for the solution in different ways.

## Breadth-first search

This is the pseudo code for the breadth-first search:

function BFS:

let frontier be a queue

let visited be a list

let start be the root node

frontier.queue(start)

visited.append(start)

while frontier is not empty:

node := frontier.dequeue()

if node == goal:

found solution, stop program

else:

for children of node:

if children not in visited:

frontier.queue(children)

visited.append(children)

no solution found, stop program

As the first step, breadth-first algorithm start the searching process by inserting the root node into the frontier queue, and mark it as visited. Next, it removes a node in the frontier queue to check if it is a goal. If the goal is found, the program will stop, otherwise, it will expand the current node by exploring its children. For each children of the node, if the children was not visited before, the program will add it to the frontier and marked it as visited. This proccess will be repeated until a solution is found or the program cannot find any solution. Notably, BFS use queue data structure to stores the node in a first in first out principle to ensure the shallowest node is expanded first.

## Depth-first search

function DFS:

let frontier be a stack

let visited be a list

let start be the root node

frontier.push(start)

while frontier is not empty:

node := frontier.pop()

visited.append(node)

if node == goal:

found solution, stop program

else:

for children of node:

if children not in visited:

frontier.push(children)

no solution found, stop program

The implementation of Depth-first search is very similar to Breadth-first search; however it use a stack to stores the nodes instead of a queue. In particular, the stack works in a last in first out fashion; therefore, the most recently added node will be explore first. This allows the Depth-first search to explore the deepest node in the frontier as far as possbile until it hits the maximum depth.

## Bidirectional Search

function BidirectionalSearch:

let sourceFrontier be a queue

let desFrontier be a queue

let sourceVisited be a list

let desVisited be a list

while sourceFrontier and desFrontier is not empty:

perform BFS for sourceFrontier

perform BFS for desFrontier

if sourceVisited and desVisited has a common node:

solution found, stop program

no solution found, stop program

In the bidirection search, we have 2 simultaneous search, one start from the root node while the other starts at the goal node. The program will keep searching until one of the frontiers is empty. If during the process, we find a common node in the sourceVisited and desVisited we stop the program and return the solution. If one of the frontiers is empty and there is no soution yet, we can conclude that there is no solution for the problem and stop the program. Notably, as we use the BFS algorithm for the searches, it is guaranteed that the solution will be the shortest path from the root node to goal node.

## Greedy best first search

function GBFS:

let h(n) be the Manhattan distance of n

let frontier be a priority queue

let visited be a list

let start be the root node

frontier.queue(start)

visited.append(start)

while frontier is not empty:

node := the node with the smallest h(n) in the frontier

if node == goal:

found solution, stop program

else:

for children of node:

if children not in visited:

frontier.queue(children)

visited.append(children)

no solution found, stop program

In informed search, we can use heuristic function which is domain-specific knowledge to guide the search and make the process more efficient. In particular, we can assess the nodes by using the h(n) function which is the manhattan distance from the current node to the goal. For every iteration, we evaluate every possible paths and explore the most “promising” node based on the heuristic function. To implement this, the algorithm store the nodes in a priority queue and sort the nodes in the frontier based on their h(n) value. As a result, the search process will be more efficient than the uninformed methods. However, the GBFS algorithm does not always produce the optimal path, as it does not takes the distance it takes to travel to the current node into account.

## A\* Search

function AS:

let f(n) = h(n) + g(n)

let frontier be a priority queue

let visited be a list

let start be the root node

frontier.queue(start)

while frontier is not empty:

newNode := the node with the smallest f(n) in the frontier

visited.append(newNode)

if newNode == goal:

found solution, stop program

else:

for children of newNode:

if node in visited with same location and f(node)<f(children):

skip children

elif node in frontier with same location and f(node)<f(children):

skip children

else:

frontier.queue(children)

no solution found, stop program

In A\* search, we evaluate a node by combining both the Manhattan distance and the total cost it has taken so far to reach that node: f(n) = h(n) + g(n). As a result, the solution produced by A\* is guarantted to be optimal as long as the heuristic function is admissible. Moreover, we also check the children node to see if it has been already visited before adding it to the frontier. If there is a node with the same location either in the visited or frontier list with the lower f(n) value than the children node, we can discard that children node and move on to the next node.

## Iterative Deepening A\*

while True:

let f(n) = h(n) + g(n)

let prunedList be a list

let frontier be a priority queue

let visited be a list

let start be the root node

frontier.queue(start)

let threshold := f(start)

while frontier is not empty:

newNode := the node with the smallest f(n) in the frontier

visited.append(newNode)

if newNode == goal:

found solution, stop program

if f(newNode) > threshold:

prunedList.append(newNode)

skip newNode

else:

for children of newNode:

if node in visited with same location and f(node)<f(children):

skip children

elif node in frontier with same location and f(node)<f(children):

skip children

else:

frontier.queue(children)

if prunedList is empty:

no solution found, stop program

else:

threshold := smallest f(n) value in prunedList

Iterative Deepening A\* search is a variant of Iterative Deepening Depth-first search and use heuristic function to evaluate the node. In particular, IDA only explore the node whose h(n) value is smaller than a threshold number and add all the nodes with the higher value into a pruned list. The algorithm will keep searching until the frontier is empty and if the solution is not found and the prunedList is not empty, it will update the threshold value to be the smallest f(n) value in the prunedList and restart the whole search process again until it finds a solution. If both the frontier and prunedList are empty, then we can conclude that there is no solution for the problem and stop the program.

# Features/Bugs/Missing

For the assignment, the program has implemented all the required features as well as a graphical interface to allow the user to create, import maze configuration and see the search process.

* Breadth-first search alogirhtm: the uninformed algorithm will produce a sequence of moves to reach the goal node from the start node
* Depth-first search algorithm: the algorithm will produce a solution for the maze if one exists.
* Greedy best-first search algorithm: an informed search algorithm that will evaluate the possible paths using heuristic function and return a solution if one exists
* A\* search algorithm: an informed search algorithm that is guaranteed to produce the shortest path.
* Uninformed custom search (Bidirectional search): This algorithm will return the shortest solution from the start node to the first goal node specified in the maze file.
* Informed custom search (Iterative Deepening A\* Search): the algorithm will produce the shortest path to reach one of the goals.
* Command line interface: given the maze file, and the search method, this program will return a series of actions to reach the goal node from the start node if one exists.
* Create a new maze configuration using a graphical user interface: the user can create a new maze configuration using a graphical interface
* Import maze configuration using a graphical user interface: the user can also import an existing maze configuration to the GUI program
* Visualize search process using a graphical user interface: Finally, the user can see the search process happens in real time using the graphical interface

# Resesarch

In addition to the command line interface, the program also includes a graphical interface that allows the user to create new maze configuration, import existing maze files and see the searching process happens in real time. In terms of implementation, the graphical interface program was written entirely in Python and heavily uses the Tkinter library which a cross-platform framework that provides a fast and easy method to create graphical applications. In this section, we will discuss the main functionalities of the GUI program and provide some screenshots to demonstrate them.

Graphical user interface, application

Description automatically generated

caption

In the homepage, the user can either create a new maze configuration or import an existing maze file to the program. If the user clicks on the “Create Maze” button, they will have to specifies the maze configuration including the number of row, column, start, goals, and wall squares:

A screenshot of a game

Description automatically generated with medium confidence

After specifying the required information, the user can click on the “save” button which will open a save dialog box that allows the user to specify where they want to save the maze configuration. Next, to solve the maze and start the searching process, the user need to go back to homepage and import the .txt file they just created:

A screenshot of a game

Description automatically generated with medium confidence

After importing the maze configuration, the user can finally see the searching process by pressing on the “Search” button. Now, the user can select one of the six algorithms including Breadth-first, Depth-first, Greedy best-first, A\*, Bidirectional, and the Iterative Deepening A\* algorithms to see the searching process happens in real time. If the algorithm finds a solution, the sequences of moves to reach the goal node will be printed out in the screen; otherwise, a message saying “No solution found” will be displayed:

Chart

Description automatically generated

## Conclusion

References:

<https://en.wikipedia.org/wiki/Breadth-first_search>

<https://www.geeksforgeeks.org/illustrate-the-difference-in-peak-memory-consumption-between-dfs-and-bfs/>

<https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/>

<https://en.wikipedia.org/wiki/Bidirectional_search>

<https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/>

[https://en.wikipedia.org/wiki/Iterative\_deepening\_A\*](https://en.wikipedia.org/wiki/Iterative_deepening_A*)

https://ai.stackexchange.com/questions/8821/how-is-iterative-deepening-a-better-than-a