



Robot navigation

Assignment 1 - Tree Based Search

TRAN DUC ANH DANG
103995439

Table of Contents

Abstract	2
Instruction	2
Introduction	2
Uninformed Search	3
WHAT IS UNINFORMED SEARCH?	3
SEARCH AND A.I.....	3
SEARCH ALGORITHMS	3
Depth First Search (DFS):	3
Breadth First Search (BFS)	4
Depth Limited Search (DLS) cus_1 (Custom 1 Search)	5
Iterative Deepening Search (IDS) cus_1ext (Custom 1 Search Extended).....	6
Informed Search	6
WHAT IS INFORMED SEARCH?	6
SEARCH AND A.I.....	7
SEARCH ALGORITHMS	7
POSSIBLE HEURISTIC FUNCTIONS:	7
GREEDY BEST FIRST SEARCH (GBFS):	8
A STAR (A*):	8
ITERARTIVE DEEPENING A STAR (IDA*) cus_2:	9
Algorithms Experiment	10
Uninformed Search:.....	10
Informed Search: Using Manhattan	10
Informed Search: Using Chebyshev	10
Informed Search: Using Euclidean	11
Experiment Conclusion:.....	11
Research.....	11
Conclusion	13
References.....	14

Abstract

In this task, I will explore the performance of tree-based search algorithms in solving the provided Robot Navigation problem. The problem involves navigating a robot through a grid to reach a goal cell. I have been implemented both informed and uninformed search algorithms as required, including Depth First Search (DFS), Breadth First Search (BFS), Greedy Best First Search (GBFS) and A* search. Furthermore, I have extended to do a custom search that hasn't been taught in any lectures, for a custom uninformed search, I chose to do Depth Limited Search (DLS) and Iterative Depth Search, for a custom informed search, I decided to pick up Iterative Deepening A* search. I also adding a various heuristic functions such as Manhattan, Chebyshev, and Euclidean distance to guide the informed search algorithms. The performance of the algorithms in terms of efficiency, and completeness has suggested that A* search combine with Manhattan Heuristic function has outperform every other uninformed search algorithms in most cases. This works has provided a better understanding of search algorithms and its applications in real world scenarios.

Instruction

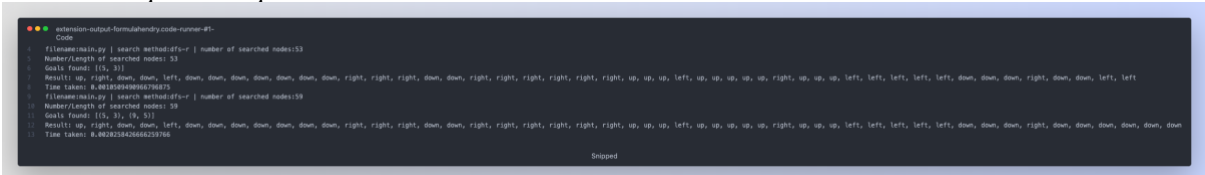
In this Option B program, the Agent (Robot) starts from a specific position and the aims is to reach one of the goal states while avoiding walls in the grid.

Instructions to run the program:

1. Make sure Python and PyGame installed on the system.
For Python please visit: <https://www.python.org/>
For PyGame please visit: <https://www.pygame.org/>
2. Provided code such as algorithm.py, grid.py and main.py must be in the same directory.
3. Create your grid text file, example is shown in RobotNav-test.txt and make sure it is name as **map.txt**:
 - First line: Grid dimensions
 - Second line: Initial state of the agent
 - Third line: Goal states for the agent
 - Remaining lines: Walls
4. Run main.py script in terminal by typing:
 - Python3: `python3 main.py`
 - Python: `python main.py`Or you can double click on run.bat file to run the program.
5. Choose the algorithms to start the search.
6. Choose the heuristic functions, and if you have to re-select each time you want to run (default is Manhattan Distance)
7. Result will be visualise and also print out to the CLI.

Note: The code used in this report is when I have completed the extended research.

Here is the expected output:



```
extension-output-formulahehdy.code-runner-47-
Code
File: main.py | search method:dfs | number of searched nodes:53
Number/Length of searched nodes: 53
Goal found: (5, 3)
Result: up, right, down, down, left, down, down, down, down, down, right, right, right, right, right, right, right, up, up, up, up, up, up, right, up, up, up, left, left, left, left, left, down, down, down, right, down, down, left
Time taken: 0.402825890879625
File: main.py | search method:dfs | number of searched nodes:59
Number/Length of searched nodes: 59
Goal found: (5, 3), (9, 3)
Result: up, right, down, down, left, down, down, down, down, down, right, right, right, down, down, right, right, right, right, right, up, up, up, up, up, up, right, up, up, up, left, left, left, left, left, down, down, right, down, down, down, down, down
Time taken: 0.402825890879625
Snipped
```

Introduction

This is an Option B program, that designed to discover the path to reach a goal using variety of uninformed and informed search algorithms. The purpose of this task is to guide an agent from the starting point to the destination (goal point).

This program uses uninformed search algorithms such as:

1. Depth First Search (DFS): An algorithm that investigates nodes as deeply as possible prior to backtracking. DFS may not always return the shortest path throughout the journey.
2. Breadth First Search (BFS): An effective algorithm that explores neighbor nodes evenly and guaranteeing the discovery of the shortest path if exist.
3. Depth Limited Search (DLS): A type of DFS that restricts the search depth, helping to prevent infinite loop in cases of cycles or unsolvable environments.

4. Iterative Deepening Search (IDS): An algorithm that combines both BFS and DFS search algorithm by iterative DLS with increasing depth limits until solution is found.

Additionally, not only uninformed search algorithms were applied, but this program also features informed search that uses heuristics to guide the search process. The algorithm such as:

1. Greedy Best First Search (GBFS): An algorithm that selects the next node to visit based on heuristic function $f(n) = h(n)$.
2. A Star (A^*): An algorithm that combines the strengths of BFS and GBFS by considering the cost of path travel so far and the heuristic cost to the goal $f(n) = g(n) + h(n)$.
3. Iterative Deepening A^* (IDA^*): An algorithm that iteratively increases the cost of threshold, it is combining the benefits of A^* , Iterative Deepening Depth First Search and it is particularly useful in many cases with limited memory resources.

Uninformed Search

WHAT IS UNINFORMED SEARCH?

Uninformed Search, also known as blind search, is a class of general-purpose algorithms that operates in a brute-force way. These algorithms lack any specific knowledge about the problem domain and rely solely on the problem's structure to find a solution. Some uninformed search techniques including in this report are Breadth-First Search (BFS), Depth-First Search (DFS), Depth Limited Search (DLS), Iterative Deepening Search (IDS). Due to their exhaustive nature, uninformed search strategies can be less efficient than informed or heuristic search methods. However, they can still be useful for simpler problems or when no domain-specific information is available to guide the search process.

SEARCH AND A.I

Uninformed search plays a significant role in the field of Artificial Intelligent (A.I) as it represents a fundamental approach to solving problems, particularly in the context of search and optimization task. It is known as a search algorithms provide a foundation for developing more advanced search techniques and serve as a basis for understanding problem solving in A.I

These algorithms can also be useful in teaching and learning AI concepts, as they illustrate the importance of search strategies in problem-solving and provide a basis for comparison with more advanced algorithms like informed search and heuristic search methods.

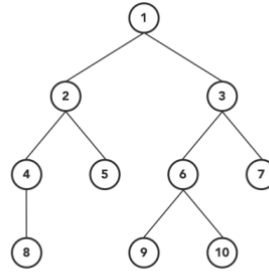
SEARCH ALGORITHMS

Tree based search is a common approach used in uninformed search algorithms, where the search space is represented as a tree structure. Each node in the tree represents a state in the problem domain. The root node represents the initial state, and the goal is to find a path in the tree that leads to a goal node, which represents a solution to the problem.

There are some common tree-based uninformed search algorithms that has been used in this assignments are:

Depth First Search (DFS):

This is a search algorithm commonly used in graph theory and AI for traversing or exploring graphs and trees. It explores the search space by visiting a node and then recursively exploring its children before backtracking (LIFO - Last In First Out). It is known as a search algorithm that always expands the deepest node in the frontier. It might not be optimal but can be made optimal using path checking.



Here is an example of DFS algorithm has been applying in this assignment using python to find the path to its goal:

```

algorithm.py

103 def dfs(self, position, path, stack=None):
104     if stack is None:
105         stack = []
106
107     if position in self.grid.goal_states:
108         self.path = path
109         # print("Here is DFS SELF PATH:", self.path)
110         return True
111
112     # print(position)
113     self.visited.add(position)
114     # print(self.visited)
115     self.search_movement.append(position)
116
117     for neighbor in self.get_neighbors(position):
118         direction = (neighbor[0] - position[0], neighbor[1] - position[1])
119         direction_map = {(-1, 0): 'up', (0, -1): 'left', (1, 0): 'down', (0, 1): 'right'}
120         move_direction = direction_map[direction]
121         if neighbor not in self.visited:
122             stack.append(neighbor)
123             if self.dfs(neighbor, path + [move_direction], stack):
124                 return True
125             stack.pop()
126     self.visited.remove(position)
127     return False
  
```

Snipped

How is that function above works? | Implementation

This is an implementation of DFS search algorithm in Python.

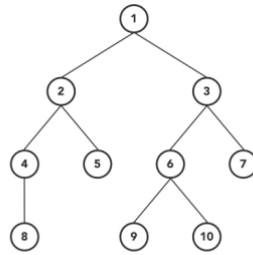
- If the current position is not a goal state, it is added to the visited set. The function iterates over the neighbors of the current position, calculating the move direction for each neighbor, and maps the direction to a string using a dictionary called direction_map. If the neighbor has not been visited, it is appended to the stack (frontier).

Then, the function calls itself recursively with the neighbor as the new position and updates the path accordingly. If the recursive call returns True, the function returns True, indicating a solution has been found. Otherwise, the neighbor is removed from the stack (backtracking).

After the loop, the current position is removed from the visited set, and the function returns False, indicating that no solution was found from the current position.

Breadth First Search (BFS)

This is a search algorithm commonly used in graph theory and AI for traversing or exploring graphs and trees. It explores the search space level by level starting from the root node. It is checking all the nodes at the current depth before moving on the next depth (FIFO - First In First Out). This approach ensures that BFS always finds the shortest path to the goal.



Here is an example of BFS algorithm has been applying in this assignment using python to find the path to its goal:

```

134 def bfs(self):
135     start = self.grid.starting_location
136     queue = [(start, [])] # Each element in the queue is a tuple containing a position and the path to that position
137
138     while queue:
139         position, path = queue.pop(0) # FIFO
140
141         # print("BFS Position:", position)
142
143         self.search_movement.append(position)
144
145         if position in self.grid.goal_states:
146             return '; '.join(path) + ';'
147
148         if position not in self.visited:
149             self.visited.add(position)
150
151         for neighbor in self.get_neighbors(position):
152             direction = (neighbor[0] - position[0], neighbor[1] - position[1])
153             direction_map = {(-1, 0): 'up', (0, -1): 'left', (1, 0): 'down', (0, 1): 'right'}
154             move_direction = direction_map[direction]
155             queue.append((neighbor, path + [move_direction])) # Enqueue the neighbor with its path
156             # print("BFS Path:", path)
157
158     return "No path found."
  
```

Snipped

How is that function above works? | Implementation

This is an implementation of BFS search algorithm in Python.

The main loop of the function iterates while the queue is not empty:

- Firstly, it dequeues the first element (FIFO) from the queue into position and path variables. Check if the current position is in goal states, if so, return the path. If position is not visited, add it to visited set. Then it is checking for each neighbor of the current position and enqueue the neighbor to the queue.

Depth Limited Search (DLS) | cus_1 (Custom 1 Search)

This is the search that combines the principles of DFS with a predefined limit. DLS is used a graph theory and A.I to traverse or explore graphs and trees while avoiding infinite loops and excessive memory consumption that can occur with standard DFS in some cases.

DLS can be implemented using recursion or an explicit stack data structure to maintain the search state. The algorithm is particularly useful when the search space has a large branching factor or depth, and memory constraints are critical. However, it may not find the optimal solution if the depth limit is not adequately set.

Here is an example of DLS algorithm has been applying in this assignment using python to find the path to its goal:

```

algorithm.py

244 def cus_1(self, position, goal, maxDepth, path):
245     if position in goal:
246         self.path = path
247         return True
248
249     self.visited.add(position)
250     self.search_movement.append(position)
251
252     if maxDepth <= len(path):
253         return False
254
255     for neighbor in self.get_neighbors(position):
256         direction = (neighbor[0] - position[0], neighbor[1] - position[1])
257         direction_map = {(-1, 0): 'up', (0, -1): 'left', (1, 0): 'down', (0, 1): 'right'}
258         move_direction = direction_map[direction]
259
260         if neighbor not in self.visited:
261             next_path = path.copy()
262             next_path.append(move_direction)
263             if self.cus_1(neighbor, goal, maxDepth, next_path):
264                 return True
265     self.visited.remove(position)
266
267     return False

```

Snipped

How is that function above works? | Implementation

- This is an implementation of Depth Limited Search algorithm in Python. This function works similarly with DFS but once its maximum depth is reached, and solution to its goal hasn't been found it will return False which indicating that no solution was found at this depth.

Iterative Deepening Search (IDS) | cus_1ext (Custom 1 Search Extended)

This search algorithm is also known as Iterative Deepening Depth First Search, that combines the advantages of DFS and BFS. It used in graph theory and A.I to traverse or explore graphs and trees while avoiding memory constraints of BFS and the infinite loop issues that occur with standard DFS.

This algorithm works by iteratively applying DLS with increasing depth limits. It starts with a depth limit of 0 and increments it after each iteration until the goal is found or maximum depth is reached. By doing so, it is also exploring the search space level by level which is very similar to BFS but with the memory efficiency of DFS.

Here is an example of IDS algorithm has been applying in this assignment using python to find the path to its goal:

```

algorithm.py

270 def cus_1ext(self, position, goal, maxDepth, path):
271     for i in range(maxDepth):
272         self.visited = set() #This always needs to be reset because it needs to perform multiple DLS search!
273         if self.cus_1(position, goal, i, path):
274             return True
275
276     return False

```

Snipped

Informed Search

WHAT IS INFORMED SEARCH?

Informed Search, also known as heuristic search, is a class of search algorithms that use additional knowledge or heuristics about the problem domain to guide the search process. These algorithm use heuristics to estimate the cost or distance from the current state to the goal state, which will enabling them to focus on more promising paths and increase its efficiency. These algorithms are very useful when the search space is large. Some informed search techniques including in the report are Greedy Best First Search (GBFS), A Star (A*), Iterative Deepening A* (IDA*). These algorithm are widely used in many applications and can significantly improve in its efficiency; however, the effectiveness are heavily depending on the quality of heuristic function that used to guide the search.

SEARCH AND A.I

Informed search plays a significant role in the field of Artificial Intelligent (A.I) as it allows for more efficient problem solving by incorporating the knowledge in the form of heuristic functions. As heuristic functions are used to guide the search therefore it can significantly improve the performance in terms of time and space complexity, especially with large search spaces.

The effectiveness of an informed search algorithm often depends on the quality of the heuristic function, which should be easy to compute and provide an accurate estimate of the cost or distance to the goal.


SEARCH ALGORITHMS

Tree based search in informed search refers to search algorithms that traverse a tree like structure, such as a state space tree, by employing a heuristic function to guide the search towards the goal node. As it is known for using the knowledge to prioritize nodes with a higher chance of leading to the goal which is much more efficient than uninformed search.

There are some common tree-based informed search algorithms that has been used in this assignments are:

POSSIBLE HEURISTIC FUNCTIONS:

There are a few possible heuristic formula that could be found, such as Manhattan distance, Chebyshev and Euclidean.



```
53 # h(n) heuristic
54 # Mahattan = |x1-x2| + |y1-y2|
55 def manhattan_distance(self, position, goal): # <----- Recommended
56     x = abs(position[0] - goal[0])
57     y = abs(position[1] - goal[1])
58     return x + y
59
60 # h(n) heuristic
61 # Chebyshev = max(|x1 - x2|, |y1 - y2|)
62 def chebyshev(self, position, goal):
63     x = abs(position[0] - goal[0])
64     y = abs(position[1] - goal[1])
65     return max(x, y)
66
67 # h(n) heuristic
68 # Euclidean sqrt((x2 - x1)^2 + (y2 - y1)^2)
69 def euclidean(self, position, goal):
70     x = (position[0] - goal[0]) ** 2
71     y = (position[1] - goal[1]) ** 2
72     return math.sqrt(x + y)
73
```

Snipped

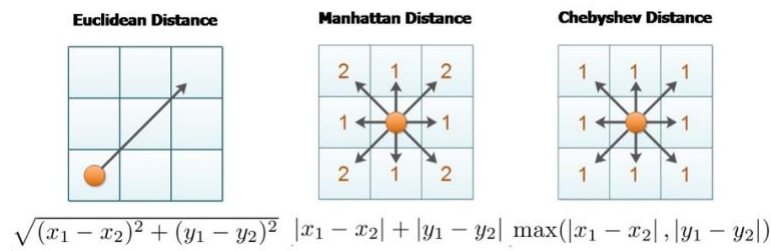
What are those?

The Manhattan, Chebyshev, and Euclidean distance heuristic are the distance based heuristic functions used in informed search algorithms using the estimate cost to reach the goal from its current state to guide the path to its goal.

1. **Manhattan Heuristic:** It is a standard heuristic for a square grid that is well known for Taxicab distance or the City Block distance, which calculates the distance between two real valued vectors or the sum of absolute differences of x and y coordinates between the current state and the goal state.
Manhattan Distance formula: $|x_1 - x_2| + |y_1 - y_2|$
2. **Chebyshev Heuristic:** This heuristic function considers the maximum of the absolute differences of x and y coordinates between the current state and the goal state.
Chebyshev Distance formula: $\max(|x_1 - x_2|, |y_1 - y_2|)$

- Euclidean Heuristic: This heuristic functions calculate the shortest distance between current state and the goal state. This formula is also well known when applying in calculating the distance between 2 points on the graph.

Euclidean Distance formula: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$



GREEDY BEST FIRST SEARCH (GBFS):

Greedy Best First Search (GBFS) is a search algorithm used in AI and graph theory to explore graph and trees by prioritizing nodes based on heuristic function. The fact that it is call "greedy" because it selects the most promising node to expand according to the heuristic function without considering the cost to reach the current node. The focus is to reach the goal as quickly as possible and may not always find the optimal solution.

Here is an example of GBFS algorithm has been applying in this assignment using python to find the path to its goal:

```

167 def greedy_best_first(self):
168     start = self.grid.starting_location
169     goal_states = self.grid.goal_states
170
171     open_list = [(start, 0, [])]
172     closed_list = set()
173
174     while open_list:
175         if not goal_states:
176             return "No goals left to find."
177
178         open_list.sort(key=lambda x: min(self.heuristic(x[0], goal) for goal in goal_states)) # Sort by heuristic (min cost to goal)
179         position, cost, path = open_list.pop(0)
180         self.search_movement.append(position)
181
182         if position in goal_states:
183             return '; '.join(path) + ';'
184
185         closed_list.add(position)
186         self.visited.add(position)
187
188         for neighbor in self.get_neighbors(position):
189             if neighbor not in closed_list:
190                 direction = (neighbor[0] - position[0], neighbor[1] - position[1])
191                 direction_map = {(-1, 0): 'up', (0, -1): 'left', (1, 0): 'down', (0, 1): 'right'}
192                 move_direction = direction_map[direction]
193                 neighbor_cost = cost + 1
194                 existing_neighbor = [entry for entry in open_list if entry[0] == neighbor]
195                 if not existing_neighbor:
196                     open_list.append((neighbor, neighbor_cost, path + [move_direction]))
197
198     return "No path found."

```

How is that function above works? | Implementation

This is an implementation of Greedy Best First Search algorithm in Python. This algorithm chooses the next position to visit based on the heuristic function, which estimates the cost to reach the goal state. The fact that it is called **greedy** because the position it chose seems closest to the goal, even if that choice might not lead to the optimal solution.

A STAR (A*):

A* or can be call as A Star, it is a search algorithm used in Artificial Intelligence (A.I) for pathfinding and graph traversal. This algorithm aims to find the shortest or least cost path from a starting node to a goal node in a graph a tree. It is using heuristic function to estimate the cost or distance from the current node to the goal that makes the search process more efficient. The efficiency of the A* algorithm depends on the quality of the heuristic function, which should be easy to compute, admissible, and consistent.

```
algorithm.py

202 def a_star(self):
203     start = self.grid.starting_location
204     goal = self.grid.goal_states[0]
205
206     open_list = [(start, 0, [])]
207     closed_list = set()
208
209     while open_list:
210         open_list.sort(key=lambda x: x[1] + self.heuristic(x[0], goal)) # Sort by total cost (cost so far + heuristic)
211         position, cost, path = open_list.pop(0)
212         self.search_movement.append(position)
213
214         if position == goal:
215             return ';'.join(path) + ';'
216
217         closed_list.add(position)
218         self.visited.add(position)
219
220         for neighbor in self.get_neighbors(position):
221             if neighbor not in closed_list:
222                 direction = (neighbor[0] - position[0], neighbor[1] - position[1])
223                 direction_map = {(-1, 0): 'up', (0, -1): 'left', (1, 0): 'down', (0, 1): 'right'}
224                 move_direction = direction_map[direction]
225                 neighbor_cost = cost + 1
226
227                 existing_neighbor = [entry for entry in open_list if entry[0] == neighbor]
228                 if not existing_neighbor or existing_neighbor[0][1] > neighbor_cost:
229                     open_list.append((neighbor, neighbor_cost, path + [move_direction]))
230
231     return "No path found."
```

Snipped

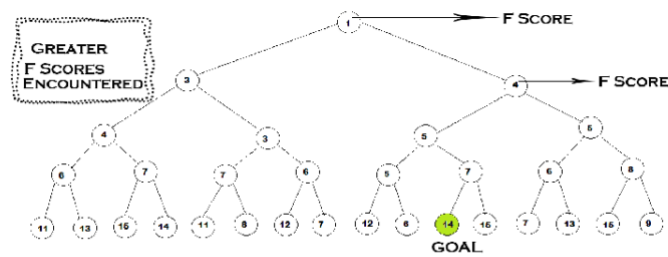
How is that function above works? | Implementation

- This is an implementation of the A* search algorithm in Python. The goal is to find the shortest path from the starting location to the goal location in a grid. The algorithm uses the Manhattan distance as a heuristic function to estimate the distance between the current position and the goal. This A* search implementation uses the Manhattan distance as a heuristic function, making it suitable for grid pathfinding problems where diagonal movements are not allowed. The algorithm will find the optimal path if such a path exists.

ITERATIVE DEEPENING A STAR (IDA*) | cus_2:

Iterative Deepening A* (IDA*) is a search the idea of combining of the other search algorithms such as of depth first search, iterative deepening as well as A* search algorithm. It is used for solving graph traversal and pathfinding problems, and it is very handy in many cases where memory is constraint.

In this algorithm, its initial threshold is calculated as the sum of the cost from the start state to the goal state using a heuristic function. Then it will perform a DFS algorithm, visiting nodes that have the total cost less than or equal to the threshold. If goal is not found, the threshold will increase, and it will performs the search again until the goal is found or no more nodes to visit.



```

202 def cus_2(self):
203     def search(position, cost, heuristic, path, threshold):
204         f = cost + heuristic
205         if f > threshold:
206             return f, None
207         if position in self.grid.goal_states:
208             return f, path
209         minimum = float("inf")
210         self.visited.add(position)
211         self.search_movement.append(position)
212         for neighbor in self.get_neighbors(position):
213             if neighbor not in self.visited:
214                 direction = (neighbor[0] - position[0], neighbor[1] - position[1])
215                 direction_map = {(-1, 0): 'up', (0, -1): 'left', (1, 0): 'down', (0, 1): 'right'}
216                 move_direction = direction_map[direction]
217                 self.visited.add(neighbor)
218                 new_cost = cost + 1
219                 new_heuristic = self.heuristic(neighbor, self.grid.goal_states[0])
220                 next_threshold, result = search(neighbor, new_cost, new_heuristic, path + [move_direction], threshold)
221                 self.visited.remove(neighbor)
222                 if result is not None:
223                     return next_threshold, result
224                 minimum = min(minimum, next_threshold)
225         return minimum, None
226
227 start = self.grid.starting_location
228 goal = self.grid.goal_states[0]
229 threshold = self.heuristic(start, goal)
230
231 while True:
232     threshold, result = search(start, 0, threshold, [], threshold)
233     if result is not None:
234         return ';'.join(result) + ';'
235     if threshold == float("inf"):
236         return "No path found."

```

Snipped

How is that function above works? | Implementation

- This is an implementation of Iterative Deepening A* search algorithm in Python. It is combining the benefits of A*, Iterative Deepening Depth First Search. The algorithm progressively increases the depth limit until it finds a solution.

Algorithms Experiment

Here are a few results that have been taken when performing the search in the provided RobotNav-test.txt

Uninformed Search:

Algorithms	Time Average	Optimal	Complete	Derivative
DFS	1.223 ms	No	Yes	
BFS	0.410 ms	Yes	Yes	
DLS dept > 15	0.214 ms	No	Yes	DFS
IDS	2.342 ms	Yes	Yes	DLS

Informed Search: | Using Manhattan

Algorithms	Time Average	Optimal	Complete	Derivative
GBFS	0.218 ms	No	Yes	BestFS
A*	0.228 ms	Yes	Yes	BestFS
IDA*	0.220 ms	Yes	Yes	A*

Informed Search: | Using Chebyshev

Algorithms	Time Average	Optimal	Complete	Derivative
GBFS	0.218 ms	No	Yes	BestFS

A*	0.268 ms	Yes	Yes	BestFS
IDA*	0.954 ms	Yes	Yes	A*

Informed Search: | Using Euclidean

Algorithms	Time Average	Optimal	Complete	Derivative
GBFS	0.308 ms	No	Yes	BestFS
A*	0.344 ms	Yes	Yes	BestFS
IDA*	1.98 ms	Yes	Yes	A*

Experiment Conclusion:

In this experiment for the grid map search experiment, Breadth First Search (BFS) seems to outperform the other uninformed search algorithms which being both **Optimal** and **Complete** the search with an average time of 0.410ms. While Depth Limited Search performing the worst comparing to its competitor as the result in the end is **Incomplete** if the dept limit is less than or equal to 15. On the other hand, informed search algorithms rely heavily on Heuristic function which proves that Manhattan Distance heuristic delivered the best performance, with A* and IDA* showing similar efficiency as A* only slightly faster with an average time of 0.228ms. Meanwhile, while applying Chebyshev and Euclidean Distance as its heuristic function, A* maintained both **Optimal** and **Complete** the search with its average time to complete increased but makes no difference from Manhattan. Overall, for this experiment, A* with Manhattan Distance as its Heuristic function proved to be the most efficient for this grid map search problem.

As this task only applying the algorithms on one simple maze, the performance of the algorithms may vary when applying in a more difficult task, but as theory, uninformed search algorithms tend to perform worse on much difficult search. On the other hand, informed search is expected to outperform uninformed search in a much difficult search when applying a heuristic function to guide them. However, performance of these algorithms still heavily depends on how complex the task is and the choice of heuristic. As in this experiment, A* that using Manhattan heuristic might still be a good choice for harder maze, but its relative performance against other algorithms and heuristic could change.

Therefore, to improving the accuracy of this experiment as well as the conclusion on the performance of each algorithms that has been use in this task, more additional experiment must be done with more complex maze for a better data analysing in the end.

Research

In this Research, I have chosen to implement the following additional features:

1. Modify the program to find the shortest path that visits all green cells (goals)
2. Create a GUI to display the environment and visualise the search algorithm in action.

Modifying the program to visits all green cells:

I modified the main loop to iterate through all goal states and update the goal states accordingly. While the algorithm is not None, the program executes a search for the next goal state and updates the starting location and goal states for each iteration. This process repeats until all goal have been visited. To be able to achieve this, I made use of variables such as goal_index, goals_found and search_nodes, which will help to keep tracking the progress and updating the grid's goal states.

```

main.py

231 while algorithm is not None:
232     for event in pygame.event.get():
233         if event.type == pygame.QUIT:
234             running = False
235             algorithm = None
236
237     screen.fill(black)
238
239     if not path_drawn:
240         if goal_index < len(goal_states):
241             grid.initial_state(screen)
242             search_nodes = list(search_nodes)
243             grid.draw_path(screen, path, search_nodes)
244             path_drawn = True
245             pygame.time.delay(5000)
246             search_nodes_length = len(search_nodes)
247             print(f"filename: main.py method:{algorithm} number of searched nodes:{search_nodes_length}")
248             print(f"Number/Length of searched nodes: {search_nodes_length}")
249             if path is not None and path_output != 'No path found.':
250                 current_position = grid.starting_location
251                 for direction in path:
252                     move = None
253                     if direction == 'up':
254                         move = (-1, 0)
255                     elif direction == 'down':
256                         move = (1, 0)
257                     elif direction == 'right':
258                         move = (0, 1)
259                     elif direction == 'left':
260                         move = (0, -1)
261                     if move:
262                         current_position = (current_position[0] + move[0], current_position[1] + move[1])
263
264                 # print("Here is current pos: ",current_position)
265                 found_goal = current_position
266                 if found_goal in goal_states:
267                     goals_found.append(found_goal)
268                     goal_states.remove(found_goal)
269
270                 # Reset for the next goal state
271                 starting_location = grid.starting_location
272                 grid.goal_states = goal_states[goal_index:] # Update the grid's goal states
273                 grid = Grid(rows, cols, cell_size, starting_location, grid.goal_states, walls)
274                 search = Search(grid)
275                 path_drawn = False
276                 search_nodes = []
277
278             print(f'Goals found: {goals_found}')
279             print(f'Result: {path_output}')
280             print(f'Time taken: {duration}')
281
282         try:
283             path, path_output, duration, search_nodes = execute_search(algorithm)
284         except IndexError as e:
285             algorithm = None
286         else:
287             if path_output == 'No path found.':
288                 goal_index += 1
289                 starting_location = grid.starting_location
290                 grid.goal_states = goal_states[goal_index:] # Update the grid's goal states
291                 grid = Grid(rows, cols, cell_size, starting_location, grid.goal_states, walls)
292                 search = Search(grid)
293                 path_drawn = False
294                 search_nodes = []
295             else:
296                 algorithm = None
297                 path = None
298                 neighbour_nodes = []
299                 search_nodes = []
300                 goals_found.clear()
301                 dimensions, starting_location, goal_states, walls = read_grid_setup(textFile)
302                 grid = Grid(rows, cols, cell_size, starting_location, goal_states, walls)
303                 search = Search(grid)
304
305     pygame.display.flip()
306     choose = True
307

```

Snipped

Building a GUI and Visualiser:

A GUI was built using pygame library to display the environment and visualise the search algorithm. The visual shows the changes happening to the search tree and the shortest path the agent (robot) takes to visit all goals (green cells). The draw_path function in Grid class is responsible for visualising the robot's path and search nodes.

Optimisation visualisation for large search spaces:

For some search algorithms, the searched_nodes dataset can be quite large that leading to longer visualisation times. I have tried to optimise the delay and nodes_per_iteration variables and draw_path function to balance visualisation time and clarity.

```

1  # grid.py
2
3  import pygame
4
5  class Grid:
6      def __init__(self, rows, cols, cell_size, starting_location, goal_states, walls):
7          self.rows = rows
8          self.cols = cols
9          self.cell_size = cell_size
10         self.starting_location = starting_location
11         self.goal_states = goal_states
12         self.walls = walls
13
14         # Drawing Initial States (MAP/Grid)
15         def initial_state(self, screen):
16             rects = []
17             for i in range(self.rows):
18                 for j in range(self.cols):
19                     color = (255, 255, 255) # White background
20                     if (i, j) == self.starting_location:
21                         color = (255, 0, 0) # Red (starting location)
22                     elif (i, j) in self.goal_states:
23                         color = (0, 255, 0) # Green (goal states)
24
25                     rect = pygame.Rect(i * self.cell_size, j * self.cell_size, self.cell_size, self.cell_size)
26                     pygame.draw.rect(screen, color, rect)
27                     rects.append(rect)
28
29             # Draw walls
30             for wall in self.walls:
31                 x, y, width, height = wall
32                 rect = pygame.Rect(x * self.cell_size, y * self.cell_size, width * self.cell_size, height * self.cell_size)
33                 pygame.draw.rect(screen, (192, 192, 192), rect)
34                 rects.append(rect)
35
36             # Draw grid lines on top of everything
37             for i in range(self.rows):
38                 for j in range(self.cols):
39                     rect = pygame.Rect(i * self.cell_size, j * self.cell_size, self.cell_size, self.cell_size)
40                     pygame.draw.rect(screen, (10, 10, 10), rect, 1)
41                     rects.append(rect)
42
43             return rects
44
45         # Draw path + Visualize
46         def draw_path(self, screen, path, search_nodes, delay=100, nodes_per_iteration=10):
47             rects = self.initial_state(screen)
48             pygame.display.update(rects)
49
50             # I don't know why but search nodes are extremely hard to optimize may be there is a large amount of searched nodes
51             # Draw Searched Nodes
52             visited_nodes = set()
53             search_node_surface = pygame.Surface((self.cols * self.cell_size, self.rows * self.cell_size), pygame.SRCALPHA)
54
55             while search_nodes:
56                 updated_rects = []
57
58                 chunk, search_nodes = search_nodes[nodes_per_iteration:], search_nodes[nodes_per_iteration:]
59
60                 for node in chunk:
61                     if node in visited_nodes:
62                         continue
63
64                     visited_nodes.add(node)
65
66                     vx, vy = node
67                     rect = pygame.Rect(vx * self.cell_size, vy * self.cell_size, self.cell_size, self.cell_size)
68                     pygame.draw.rect(search_node_surface, (0, 255, 255), rect)
69                     updated_rects.append(rect)
70
71                 screen.blit(search_node_surface, (0, 0))
72                 pygame.display.update(updated_rects)
73                 pygame.time.delay(delay)
74
75             # Reset
76             rects = self.initial_state(screen)
77             pygame.display.update(rects)
78
79             # Start Drawing Path
80             current_position = self.starting_location
81             for direction in path:
82                 move = None
83                 if direction == 'up':
84                     move = (-1, 0)
85                 elif direction == 'down':
86                     move = (1, 0)
87                 elif direction == 'right':
88                     move = (0, 1)
89                 elif direction == 'left':
90                     move = (0, -1)
91                 if move:
92                     current_position = (current_position[0] + move[0], current_position[1] + move[1])
93                     rect = pygame.Rect(current_position[0] * self.cell_size, current_position[1] * self.cell_size, self.cell_size, self.cell_size)
94                     pygame.draw.rect(screen, (255, 255, 0), rect)
95                     pygame.display.update(rects)
96                     pygame.time.delay(delay)
97
98             Snipped

```

I have also generated a test map that will be included in the submissions.

Conclusion

In this conclusion, it will be talking about the purpose of the task.

The purpose of this task is to develop problem solving skills for the people that are new into this field, in this case Artificial Intelligent (A.I). Not only that, but this task will also develop critical think, and programming some basic tree-based search algorithms that everyone into this field must know. This task will help to understanding the principles of search algorithms, performance characteristic and its application in real world scenarios. Furthermore, this task also encourages in self learning by exploring search methods outside of the lectures in this case (DLS, IDS, IDA*).

In Option B, it is required to implement both informed and uninformed search algorithms to find a solution for RobotNavigation problem. This task involves writing a program to model grid environment, the robot, and its movement towards the goal by applying search algorithm.

In Research part, the path finding program has been significantly enhanced with new features and optimises as best as it could, including the ability to visit all goals (green cells) with the shortest path to its goal and its a GUI for visualisation.

References

- Nandy, A. (n.d.). 4 types of tree traversal algorithms. Towards Data Science. Retrieved from <https://towardsdatascience.com/4-types-of-tree-traversal-algorithms-d56328450846>
- Warren, D. H. D. (1969). An improved program for tree search. Simon Fraser University. Retrieved from http://www.sfu.ca/~arashr/warren.pdf?fbclid=IwAR2D0y8Xgn1F0Sjk2NwrrErAG5tlgorQZXLHIN57C3ZkyTpua_BCRrjvCU
- Wikipedia. (n.d.). Iterative deepening A*. Retrieved from https://en.wikipedia.org/wiki/Iterative_deepening_A*
- Chauhan, S. (n.d.). Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS). GeeksforGeeks. Retrieved from <https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/>
- FavTutor. (n.d.). Breadth First Search in Python: Algorithm and Examples. Retrieved from <https://favtutor.com/blogs/breadth-first-search-python>
- Educative. (n.d.). How to implement Depth First Search in Python? Retrieved from <https://www.educative.io/answers/how-to-implement-depth-first-search-in-python>
- Ghosh, A. (n.d.). IDA* Algorithm in General. Algorithms Insight. Retrieved from <https://algorithmsinsight.wordpress.com/graph-theory-2/ida-star-algorithm-in-general/>
- OpenGenus IQ. (n.d.). Euclidean vs Manhattan vs Chebyshev Distance. Retrieved from <https://iq.opengenus.org/euclidean-vs-manhattan-vs-chebyshev-distance/>
- Irfan, M., & Basalamah, S. (2018). Comparative Analysis of Pathfinding Algorithms: A. Dijkstra and B. BFS on Maze Runner Game. ResearchGate. Retrieved from https://www.researchgate.net/publication/325368698_Comparative_Analysis_of_Pathfinding_Algorithms_A_Dijkstra_and_BFS_on_Maze_Runner_Game
- Kumar, A., & Sahoo, S. (2018). Purity results for applying Manhattan, Euclidean, Chebyshev, and Minkowski distance. ResearchGate. Retrieved from https://www.researchgate.net/figure/Purity-results-for-applying-Manhattan-Euclidean-Chebyshev-and-Minkowski-distance_tbl2_346502652
- Patel, A. (n.d.). Heuristics. Amit's Game Programming Information. Retrieved from <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>