Course Code : SOF106

Course Name : Principles of Artificial Intelligence

Lecturer : Shamini Raja Kumaran

Academic Session : 2022/09

Assessment Title : Final Assessment

Submission Due Date :

Prepared by :

Student ID	Student Name
AIT2204016	Gabriel Choong Ge Liang
MEC2109494	Cao Rui
AIT2204234	Chau Teng Cheng
AIT2202007	Benjamin Chin Wei Hao

Date Received :

Feedback from Lecturer:

Mark:

Own Work Declaration

I/We hereby understand my/our work would be checked for plagiarism or other misconduct, and the softcopy would be saved for future comparison(s).

I/We hereby confirm that all the references or sources of citations have been correctly listed or presented and I/we clearly understand the serious consequence caused by any intentional or unintentional misconduct.

This work is not made on any work of other students (past or present), and it has not been submitted to any other courses or institutions before.

Signature: 

Date: 17/12/2202

MARKING RUBRICS

		Score and Descriptors				
No.		**Poor**	**Average**	**Excellent**	**Weight (%)**	**Mark**
Component 1: Project Development						
		0 - 4	**5 - 7**	**8 - 10**	**10**	
1	Code quality	Codes are poorly structured. Not fully functional. Not documented.	Codes are sufficiently documented and mostly functional. Satisfactorily structured	Codes are fully functional and well structured and documented		
		0 - 4	**5 - 7**	**8 - 10**	**10**	
2	Implementation of algoritm related to AI	Incomplete development of the method.	Complete development of method.	Completed\ enhanced the developed methods.		
		0-2	**3**	**4-5**	**5**	
3	Completeness and complexity of your project	The functionalities that are implemented are non-functional	All functionalities are implemented and functional	All the functionalities are fully implemented and functional Implemented extra functionalities.		
				Subtotal	**25**	
Component 2: Project Report + Project Demonstration						
		0 - 4	**5 - 7**	**8 - 10**	**10**	
1	Project Report	Poor writing quality Poor or no formatting / presentation Lack of technical content	Satisfactory writing quality, grammar and flow. Substantial technical content with used method.	Good writing quality, grammar and flow Well formatted and good presentation Demonstrate excellent technical content based the method used		
		0 - 4	**5 - 7**	**8 - 10**	**10**	
2	Presentation	Poor quality slides Poor time management Speech that is unclear	Satisfactory quality slides Speech that is satisfactory and understandable	High quality slides Good time management Speech that is clear and impactful		
		0-2	**3**	**4-5**	**5**	
3	Demonstration	Poor demonstration that is unclear The developed application/research is not functioning/implemented fully	Satisfactory demonstration The developed application/research is partially functioning/implemented fully	Good demonstration The developed application/research is fully functioning and logical		
				Subtotal	**25**	
				Grand Total	**50**	

Note to students: Please attach this appendix together with the submission of coursework

Optimal Path Finder

Abstract
Modern civilisations tend to have buildings in a network formation to use space efficiently. This however leads to some routing issues when we're in a time crunch. The path that most people take tends not to be the fastest. By applying the A* search algorithm, we're interested to see if we could idealise the route usage and find out the most efficient path from one building to another.

The A* algorithm accounts for the path cost, which is the distance between two buildings (nodes) and the heuristic cost, which is some arbitrary value we chose to give a certain building. These two parameters could influence the routing process significantly. The outcome of this algorithm is the minimum cost of the combination of the path cost and heuristic cost.

Introduction
A* search algorithm calculates the minimum f (n) for every cumulative step, where f (n) = g (n) + h (n). G (n) is called the path cost and h (n) is called the heuristic cost. We mapped out the whole area of the campus by using an undirected graph. Each node and path was given their heuristic value and path cost respectively. This is then used as a map in the implementation of our program.

An A* algorithm was written with the Python programming language. The campus map was fed into the algorithm and the user input was taken, namely the current location of the user and the destination of the user. The algorithm will then search for the optimal path based on the information given.

Methodology

Planning

Buildings on the campus were represented using nodes and draws. The user interface is made. The algorithm for A* is implemented using python language. Nodes is imported into the algorithm by calling the header file.

The data was collected and code was written for the algorithm we chose for this project.

Data collection

At this stage for the data collection, data was collected based on the campus map layout. What we did was to map out all the buildings on the campus by assigning them each a heuristic value and a path cost between each other. To make it easier similar buildings are given the same heuristic values. For the required code we implemented some of the materials that are collected from journals and websites alongside our knowledge.

Implementing

We selected the A* search algorithm for this optimal pathfinder as it uses a simple and efficient search algorithm that is minimizing the path cost and heuristic value for every iterative step between two nodes in a graph.

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

g(n) is the path cost which is the distance between the current node and the start node. Whereas h(n) is the heuristic value from the current node to the end node.

Implementation

```
def astar(route, start, end):

    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    # Initialize both open and closed list
    open_list = []
    closed_list = []

    # Add the start node
    open_list.append(start_node)

    # Loop until you find the end
    while len(open_list) > 0:

        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index

        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)
```

To start we had to define some of the functions like start and end nodes, and open and closed lists. This way the code runs from the start node and ends in the end node. Similarly, open and closed list just indicates that the values given have a start and an end. The time complexity of A* relies on the heuristic value.

```
    # Found the goal
    if current_node == end_node:
        path = []
        current = current_node
        while current is not None:
            path.append(current.position)
            current = current.parent
        return path[::-1] # Return reversed path

    # Generate children
    children = []
    for new_position in campus: # Adjacent squares

        # Get node position
        node_position = new_position

        # Make sure within range
        if node_position[0] in location.keys():
            continue

        # Create a new node with parent: current_node and positi
        new_node = Node(current_node, node_position)

        # Append
        children.append(new_node)
```

This part reverses the paths after the goal has been found. The nodes get updated to a new position when a new node is found.

```
# Loop through children
for child in children:

    # Child is on the closed list
    for closed_child in closed_list:
        if child == closed_child:
            continue

    # Assign f, g, and h values
    child.g = child.g + 1

    child.h = heuristic_cost[child.parent.position]

    child.f = child.g + child.h

    # Child is already in the open list
    for open_node in open_list:
        if child == open_node and child.g > open_node.g:
            continue

    # Add the child to the open list
    open_list.append(child)
```

A closed list is a collection of all expanded nodes. It indicates the nodes that are already searched to prevent the search from visiting the same nodes again and again. Whereas the open list, is a collection of all generated nodes that are neighbours of the expanded nodes.

```
from sys import exit

# User input class
You, yesterday | 1 author (You)
class UserInput:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def set_start(self, start):
        self.start = start

    def set_end(self, end):
        self.end = end

    def get_start(self):
        return self.start

    def get_end(self):
        return self.end

def answer(list):
    for i in list:
        if i != list[-1]:
            print(i, end=" > ")
        else:
            print(i)

def exit():
    exit(0)
```

Here are just some common functions that are crucial as user inputs so that the code knows what to output.

```python
from model import astar
from functions import UserInput, answer
from route import location

def main():

    current_location = UserInput(" ", " ")
    current_location.set_start(input())
    current_location.set_end(input())

    start = current_location.get_start()
    end = current_location.get_end()
      You, 2 weeks ago • Created main.py
    map = location
    global path
    path = astar(map, start, end)
    answer(path)


if __name__ == '__main__':
    main()
```

This is the main function of the whole program. Users are required to input the start and end locations to obtain the path. Some of the programs are imported into the main.py program to run it.

Implementation

To start we had to define some of the functions like start and end nodes, and open and closed lists. This way the code runs from the start node and ends in the end node. Similarly, open and closed list just indicates that the values given have a start and an end. The time complexity of A* relies on the heuristic value.

This part reverses the paths after the goal has been found. The nodes get updated to a new position when a new node is found.

A closed list is a collection of all expanded nodes. It indicates the nodes that are already searched to prevent the search from visiting the same nodes again and again. Whereas the open list is a collection of all generated nodes that are neighbours of the expanded nodes.

Here are just some common functions that are crucial as user inputs so that the code knows what to output.

This is the main function of the whole program. Users are required to input the start and end locations to obtain the path. Some of the programs are imported into the main.py program to run it.

Validation/ Verification

After the code was done, it was tested by multiple input and output pairs. The results yielded were not 100% accurate all the time and there were shorter paths. The implementation was not perfect in this situation.

We tried various input-output pairs but the result wasn't what we expected. The routing has lots of issues since we didn't account for the difference in altitude. The project was carried out in an ideal situation where only the path cost and heuristic cost mattered. However, we're able to implement this search algorithm and find out how suitable it is in real-life situations.

Conclusion

After conducting our research and analysis on the A* search algorithm, we have learned that it is a powerful tool for finding optimal solutions in pathfinding and graph traversal problems. It combines the benefits of both breadth-first search and best-first search, allowing it to consider both the cost of reaching a particular node and an estimated cost to the goal.

Overall, our project has been successful in achieving its goals of understanding the A* algorithm and evaluating its performance. We have found that the A* algorithm performs well in a variety of contexts, although its efficiency can be affected by the quality of the heuristics used.

There are several ways that the project could be improved in the future. One possibility would be to explore more advanced heuristic functions, as these can have a significant impact on the efficiency of the algorithm. Additionally, it may be helpful to compare the performance of the A* algorithm to other pathfinding algorithms to better understand its strengths and weaknesses.

Overall, the A* algorithm has met our expectations and has proven a reliable and effective tool for finding optimal solutions.