CSE5AIF

Artificial Intelligence Fundamentals

Lab 3: A* Search

Lab Objectives

The aim of this lab it to apply the A* search strategy to solve multiple search puzzles.

By the end of this lab, you will have a practical understanding of the A* search algorithm. You will utilize the code created in the previous lab to find the optimal solution to the 8-Puzzle with A*; then you will use what you have learnt to implement and solve the Shortest Path Puzzle.

Background Information

Heuristic Search

As noted previously in the subject lecture notes, there are two type of searching strategies, **Blind (Uninformed) Search** and **Heuristic (Informed) Search**.

In the previous Lab, we employed the Breadth-First Search algorithm to solve the 8-Puzzle problem; this is an example of Blind Search. This type of search algorithm has no information regarding what it is searching for or where it should search for it. As such, it tends to take much longer to find a solution, as it doesn't know which move will increase the probability of finding the goal.

On the other hand, we have Heuristic Search algorithms; these algorithms are aware of which moves will most increase the probability of finding the goal, using a heuristic value. This value is used to guide the algorithm by telling it which path will provide the solution as early on in the search as possible; providing the optimal route. The function that provides this value is dependent on the search problem, and varies based on application.

A* Search Algorithm

The A* search algorithm is utilized in numerous pathfinding and graph traversal applications and is regarded as having both high performance and accuracy.

Consider the 8-Puzzle problem, our objective is to sort the 3x3 grid in ascending order. How the A* search algorithm works, is that at each step of the search, it selects the next node according to a value denoted as **f**; selecting the node with the lowest **f** value. This **f** parameter is equal to the sum of two other parameters **g** and **h**; in the case of our 8-puzzle problem:

- **f** is the total cost of the node (**g** + **h**).
- **g** is the number of nodes traversed from the start node to get to the current node (the depth of the node).
- **h** is the heuristic - the number of tiles which are not in the correct position, when comparing the current node to the goal node (In the case of the 8-Puzzle).

Function Definitions

The following section provides some detail about new functions that we will use in the subsequent code.

Lambda Function

A Lambda function is a small unnamed function with the syntax:

lambda *arguments* : *expression*

e.g. A lambda function that adds 10 to the number passed in as an argument, and prints the result:

```
x = lambda a : a + 10
print(x(5))
```

this code will print a result of 15. More information explaining the lambda function can be found at: https://www.w3schools.com/python/python_lambda.asp

Sort Function

In the default use case, the sort function will sort a list, by value, in ascending order

e.g.:

```
list = {5, 8, 1, 2}
list.sort()
print(list) will now print {1, 2, 5, 8}.
```

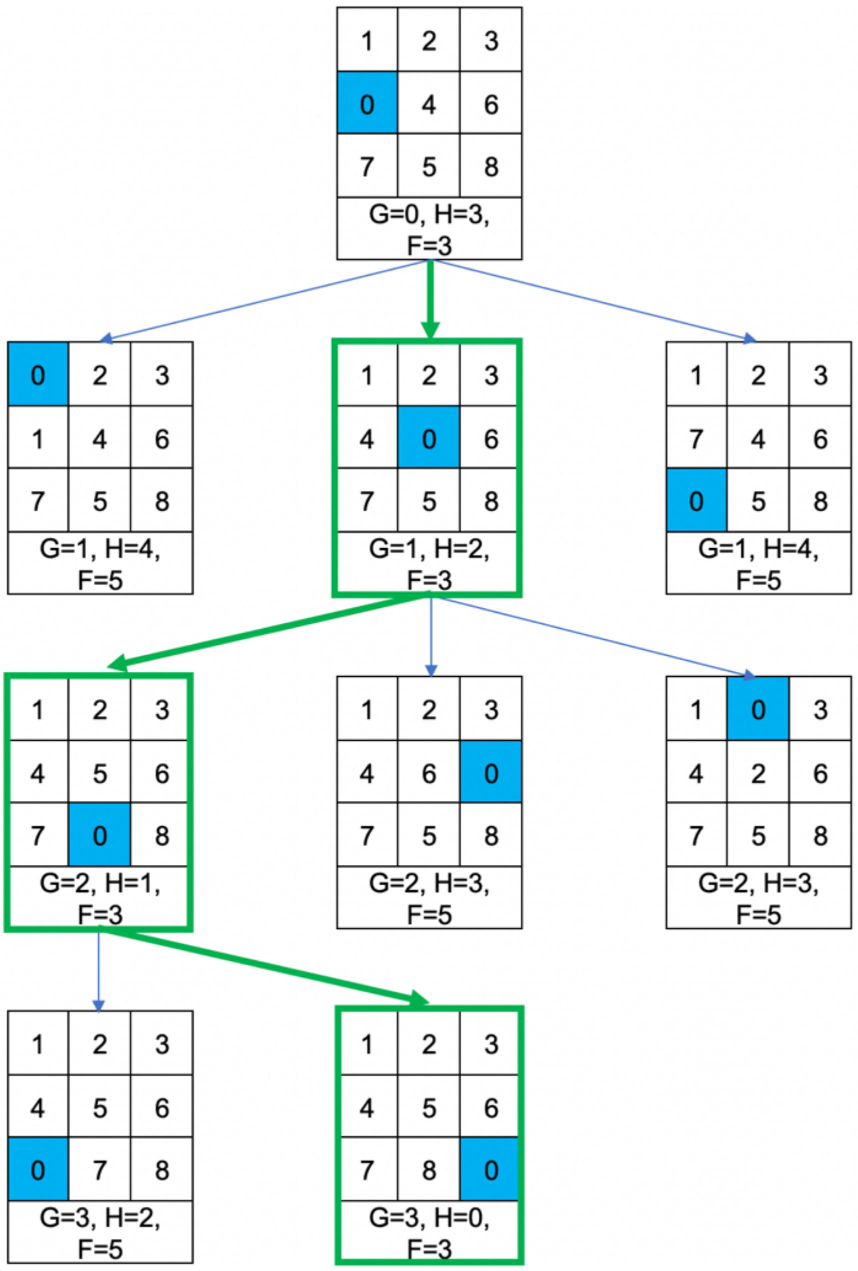
Using the “key” parameter allows us to provide **sort** with a function that we would like to sort our list by, with respect to the functions result. In the **a_star_search** function, we use this to sort **open_list** with respect to the **f** value of each node in the list. So, the resultant **open_list** will be sorted in ascending order by **f** value. More information about use can be found at: https://docs.python.org/3/howto/sorting.html

Lab Instructions - 8-Puzzle with A* Search

Similar to our BFS algorithm, the A* algorithm uses two lists named **open** and **closed.** The **open** list contains all the nodes that are in line to be checked as the goal or expanded to their neighbouring nodes.

The A* search algorithm solves the 8-puzzle problem by first expanding the zero-tile in all possible directions, calculating the **f** value at each node. Next, the **current_node** is moved into the list of **closed** nodes, and the newly created child nodes are put into a list of **open** nodes. The node with the lowest **f** score is selected and expanded again; continuing this pattern until we find the goal node.

In the following example, the path taken by the A* algorithm is highlighted in green. The **g**, **h** and **f** values are displayed for each node.



A more detailed method, outlining 8-Puzzle using A* search is as follows:

1. Take the node with the lowest **f_value** at the front of the **open** list as our **current_node** (initially this is our root node).

2. We check if the **current_node** is our goal node,
    a. if it is, we return the path taken up to this point as the solution;
    b. if it isn't, we continue

3. We now must expand the **current_node** to reveal its neighbouring nodes; this is done by moving the zero tile in each direction, appending each of these resultant children nodes to the **open** list.

4. Next, we loop through each of the newly created child nodes.
    a. If the node has previously been visited, we disregard it;
    b. Else, we append it to the **open** list

5. Once all child nodes have been checked, we remove the **current_node** from the **open** list and add it to the **closed** list.

6. We now sort the nodes in the **open** list by their **f_value**, in ascending order.

7. Finally, we return to step 1 and repeat this loop until we find the solution.

Implementation

In order to implement the A* search algorithm we must modify the code created in Lab 2. Create a new file named **lab_3_1.py** and open it in your chosen IDE. Copy all of your Lab 2 code into this file.

The code structure of **lab_3_1.py** is indicated below, all functions which will be modified/added are highlighted in red.

- class Node
    - __init__
    - create_child
    - move_right
    - move_left
    - move_up
    - move_down
    - goal_test
    - expand_node
    - print_puzzle
    - is_unsolvable
    - get_f_value
- class Search
    - breadth_first_search
    - a_star_search
    - path_trace
- Main code

To begin implementing the A* search algorithm, modify the **Node** class **__init__** function by adding two new class variables, **g** and **f**.

```
def __init__(self, puzzle):
    # List to store child nodes
    self.children = []
    # Variable, to store parent node (note: the root nodes parent is "None")
    self.parent = None
    # Current nodes puzzle state, set from the input parameter
    self.puzzle = puzzle
    # Index of zero tile in current puzzle (set in a future function)
    self.zero = 0
    # Depth of current Node with respect to root node
    self.g = 0
    # Variable to store the Nodes’ f value
    self.f = self.get_f_value()
```

Next, add a line to the **create_child** function; this will increment the **g,** depth variable for each child with respect to its parents’ depth.

```
def create_child(self, puzzle):
    # Create a child Node object using the input puzzle
    child = Node(puzzle)
    # Store the child node in the children list of the current node
    self.children.append(child)
    # Store the current node as the parent of the child node
    child.parent = self
    # Add one to the depth of the child
    child.g = self.g + 1
```

Within the **Node** class, define a function named **get_f_value**. This will be used to calculate each nodes’ **f** value, allowing us to sort our list of open nodes as required.

```
def get_f_value(self):
    h = 0
    # In this loop “i” iterates from 0 to 8 {0,1,2...,8},
    # as this matches the goal state for our puzzle,
    # we can use this to check if our puzzle is correct.
    for i in range(len(self.puzzle)):
        # Check to see if the nodes' puzzle values match the required goal state.
        if self.puzzle[i] != i:
            # If we find that a tile in our puzzle is in the incorrect position,
            # we increment the nodes' h value.
            h += 1
    # Finally, return the calculated value of f, where f = h + g
    return h + self.g
```

To add the A* search algorithm into our 8-Puzzle code, we must add the following **a_star_search** function to our **Search** class:

```
def a_star_search(self, root):
    # List to contain open nodes
    open_list = []
    # Set to contain visited nodes
    visited = set()
    # Add root node as open
    open_list.append(root)
    # Add root node as a visited state
    visited.add(tuple(root.puzzle))

    # Continuously loop over the code until we find a solution.
    while(True):
        # Take the next node with the lowest f value as the current node.
        # note: open_list is sorted by f value at the end of this loop.
        current_Node = open_list.pop(0)

        # Use the current_node's goal_test function to check if the puzzle is correct.
        if current_Node.goal_test() == True:
            # If we have found the goal state
            # Call the path_trace function and store the path to the current state.
            path_to_solution = self.path_trace(current_Node)
            # Print out total number of moves to reach goal state.
            print(len(visited))
            # Exit the a_star_search function and return the solution path.
            return path_to_solution

        # If current node is not the goal state, then expand to its neighbouring nodes
        current_Node.expand_node()

        # Loop through all nodes neighbouring the current node
        for current_child in current_Node.children:
            # if the current node hasnt been visited before,
            if tuple(current_child.puzzle) not in visited:
                # Add the child to the list of open nodes
                open_list.append(current_child)
                # Then add the current_child to set of visited nodes
                visited.add(tuple(current_child.puzzle))

        # Sort the list of open nodes by their f value, in ascending order
        open_list.sort(key=lambda x: x.f)
```

Finally, in the main block, comment out the original call to **breadth_first_search** and add a new one to use **a_star_search**.

```
if __name__ == "__main__":
    # The puzzle to be solved, you can modify this for any other configuration.
    puzzle = [8, 6, 7, 2, 5, 4, 3, 0, 1]
    # Create the root node of the puzzle
    root_puzzle = Node(puzzle)
    # Check if the puzzle is solvable
    if root_puzzle.is_unsolvable():
        print("Puzzle has no solution!")

    else:
        # Create the Search object
        search = Search()
        print("Finding solution...")
        # Get the time at the start of the search
        start = time()
        # Search for and get the solution using BFS
        # solution_path = search.breadth_first_search(root_puzzle)
        solution_path = search.a_star_search(root_puzzle)
        # Get the time at the end of the search
        end = time()
        # Reverse the solution path so that we can print inital_node to goal_node
        solution_path.reverse()
        # Loop throguh solution path nodes
        for i in range(len(solution_path)):
            # Print out node puzzle
            solution_path[i].print_puzzle()
        print("Number of steps taken:", len(solution_path)-1)
        print("Elapsed time:", end-start)
```

Running this code will now solve the 8-Puzzle problem with the A* search algorithm. Take note of the greatly improved execution times when switching from BFS to A*. (You can switch between the two by commenting/uncommenting the call to each respective function.)

Lab Instructions - Shortest Path with A* Search

The objective of the Shortest Path puzzle is to find the shortest path between two nodes. Solving this puzzle with the A* algorithm is almost identical to what was previously discussed for the 8-Puzzle; with the main difference being the calculation of the f value.

For example, take the following illustration of a shortest path puzzle:

7	6	5	6	7	8	9	10	11	▨	19	20	21	22
6	5	4	5	6	7	8	9	10	▨	18	19	20	21
5	4	3	4	5	6	7	8	9	▨	17	18	19	20
4	3	2	3	4	5	6	7	8	▨	16	17	18	19
3	2	1	2	3	4	5	6	7	▨	15	16	17	18
2	1	0	1	2	3	4	5	6	▨	14	15	16	17
3	2	1	2	3	4	5	6	7	▨	13	14	15	16
4	3	2	3	4	5	6	7	8	▨	12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Say that the green node 0 is our starting node; the blue node 19 is our end node; and the orange node 4 is our current node.

g: is the distance between the current node and the start node; counting back we can see our current node is 4 spaces away from our start node.

h: is the estimated distance from the current node to the end node. Counting the distance between the current node and the end node you can see that we must go right 6 spaces and up 3 spaces to reach our goal node.

Applying Pythagoras' Theorem ($a^2 + b^2 = c^2$**)** we get:

$$h = 6^2 + 3^2 = 45$$

(applying the square root to 45 is not necessary as using the same calculation on every node will get the same output).

f: is again, simply the total cost of the node (**g** + **h**). Hence, **f** = 45 + 4 = 49.

Method

The method for completing the shortest path puzzle is as follows:

1. Add the starting node to the **open_list**
2. Repeat the following:
    a. Look for the lowest **f** cost node on the **open_list**; This will be the **current_node**.
    b. For each of the 8 squares adjacent to the **current_node**
        i. If the adjacent node is not walkable (e.g. is a wall tile) or if it is on the **closed** list, ignore it,
        ii. Else, if it isn't on the **open_list**, add it to the **open_list**. Make the **current_node the** parent of this **child_node**. Record the **f**, **g**, and **h** costs of the square.
        iii. Else, if it is on the **open_list** already, check to see if this path to that node is better, using **g** cost as the measure. A lower **g** cost means that this is a better path. If so, change the parent of the node to the **current_node**, and recalculate the **g** and **f** scores of the node.
        iv. Re-sort the **open_list** by the **f** score.
    c. Stop repeating when you
        i. Find the target node, or
        ii. Fail to find the target node and the **open_list** is empty. In this case there is no possible path.
3. Save the path by working backwards from the end node, going from each node to its parent node until you reach the root node.

Implementation

We will be solving a 10x10 shortest path puzzle with our implementation of the A* Search algorithm.

Create a new file named **lab_3_2.py** in your working directory and open this with your chosen IDE.

At the end of this section, your code structure should be:

- class Node
    - __init__
    - Expand
- a_star_search
- Main code

Node Class

Similar to the Node class from Lab 2, this Node class will store all information regarding the node; including its child nodes, parent node, current position, and g, h, and f values.

Unlike the 8-Puzzle, the expand function will now be required to expand into all 8 directions (as we now have the ability to move diagonally too). We loop over a list of all possible movements, using their values to create child nodes with the new positions.

```
class Node():

    def __init__(self, parent=None, position=None):
        # Initialize all attributes that we want a node to have
        # List to store child nodes
        self.children = []
        # Store parent node for later path tracing
        self.parent = parent
        # Store position of node, with respect to puzzle
        self.position = position

        # Store the g,h, and f values
        self.g = 0
        self.h = 0
        self.f = 0

    def expand(self, puzzle):
        # Loop through all possible movements
        for movement in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]:

            # Get node position after moving
            node_position = (self.position[0] + movement[0], self.position[1] + movement[1])

            # Check that the new node is within the boundaries of the puzzle
            if node_position[0] <= (len(puzzle) - 1) and node_position[0] >= 0 and
node_position[1] <= (len(puzzle[len(puzzle)-1]) - 1) and node_position[1] >= 0:

                # Check if its trying to move into a position occupied by a wall
                if puzzle[node_position[0]][node_position[1]] == 0:

                    # Create new child
                    child = Node(self, node_position)
                    self.children.append(child)
```

A* Search Function

This search function pragmatically operates identically to the previously defined function. The modifications made are to account for the differing context of the puzzle, where some functionality is written in-line rather than in separate function blocks

```
def a_star_search(puzzle, start, end):
    # initialize node with no parent and "start" coords
    start_node = Node(None, start)
    # initialize node with unknown parent and "end" coords
    end_node = Node(None, end)

    # 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.pop(0)
        closed_list.append(current_node)

        # Check if it is the goal node
        if current_node.position == end_node.position:
            path_to_solution = []
            current = current_node
            while current is not None:
                path_to_solution.append(current.position)
                current = current.parent
            return list(reversed(path_to_solution))

        # # Generate children
        current_node.expand(puzzle)

        # Loop through children
        for child in current_node.children:
            flag = False
            # if the child with the same position isnt in the closed list:
            for closed_child in closed_list:
                if child.position == closed_child.position:
```

```
                flag = True

        # Create the f, g, and h values
        child.g = current_node.g + 1
        child.h = ((child.position[0] - end_node.position[0]) **
                2) + ((child.position[1] - end_node.position[1]) ** 2)
        child.f = child.g + child.h

        # Child is already in the open list
        for open_node in open_list:
            if child.position == open_node.position and child.g > open_node.g:
                # Dont add child to the open_list, move on to next one
                flag = True

        # Add the child to the open list
        if flag == False:
            open_list.append(child)

    open_list.sort(key=lambda x: x.f)

  print("No Solution Found!")
```

Main Code

The main code holds both the start and end points as well as the puzzle to solve. We can modify each of these parameters to solve countless configurations.

```
if __name__ == '__main__':
  puzzle = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]]

  start_point = (0, 0)
  end_point = (7, 8)

  path = a_star_search(puzzle, start_point, end_point)
  print(path)
```

Running this code will print the path to the solution (if it exists), from the start to the end node.

Practice Exercises

If you'd like to practice more and test your abilities, you can try to:

1. Solve the N-Queen Puzzle using A*.

2. Implement BFS or DFS to solve the Shortest Path Puzzle (if you previously have not attempted).

All of these search methods and puzzles have been previously explained in notes for Lecture 2 here.

References

https://docs.python.org/3/

https://medium.com/@sairamankumar2/8-puzzle-problem-aa578104ae15

https://sandipanweb.wordpress.com/2017/03/16/using-uninformed-informed-search-algorithms-to-solve-8-puzzle-n-puzzle/

http://benalexkeen.com/implementing-djikstras-shortest-path-algorithm-with-python/

https://blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288