CSE5AIF
Artificial Intelligence Fundamentals
Lab 2: 8-Puzzle with BFS

Lab Objectives

The aim of this lab is to apply what you have learnt from lab 1 and the subjects lectures, to create a state-space search algorithm.

Upon completing this lab, you should have a fully functioning breadth first search (BFS) algorithm that is able to solve the 8-Puzzle problem.

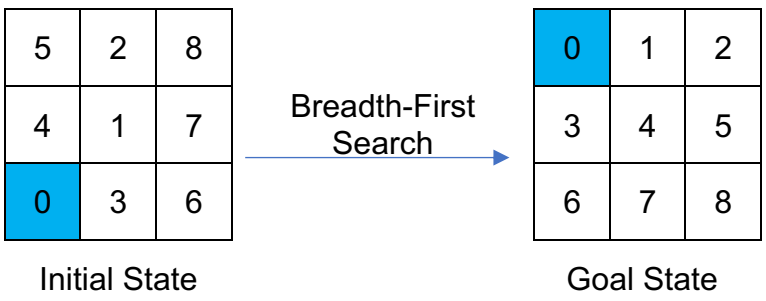
Background Information

Breadth-First Search

Breadth-First Search (BFS) is a method of state-space search which explores the state space in a level-by-level fashion. Only when there are no more states to be explored at a given level does the algorithm move on to the next level. We will not go over the details of BFS in this Lab as it has previously been discussed in Lecture 2 on problem spaces and search, this can be found on the subjects LMS.

N-Puzzle Objective

The N-Puzzle is a sliding puzzle that consists of a NxN frame of numbered tiles in a random tile with one tile missing. The object of the game is to arrange the tiles in order by making sliding moves, utilizing the empty space. For this lab we will be solving an 8-Puzzle, consisting of a 3x3 frame. The following shows a representation of our initial, and goal states:



8-Puzzle Solvability

The 8-Puzzle problem has 9! (362880) possible combinations; of these only half (181440) are actually solvable. The method of calculating if a particular permutation is solvable is as follows, for a given puzzle:

5	2	8
4	1	7
0	3	6

We must calculate the number of inversions by counting tiles that precede another tile with a lower number. So, for the puzzle in this case:

Tile	Precedes (Inversions)	Inversion Count
5	1,2,3,4	4
2	1	1
8	1,3,4,6,7	5
4	1,3	2
1	None	0
7	3,4	2
3	None	0
6	None	0
	Total:	14

As we have an even number of inversions the puzzle is solvable. If this inversion count produced an odd result, it would not be solvable.

Further explanation on this topic can be found here.

Lab Instructions

Create a new file

Making use of the information and notes from Lab 1, open Anaconda and launch Jupyter notebook.

Make sure you have your previously created **aif_env** selected.

Create a new file named **lab_2** in your working directory.

Note: Feel free to utilize any other IDE you are comfortable with in place of Jupyter Notebook.

Create the Node class

We start by creating a **Node** class. We want a **Node** instance to contain all of the information describing the puzzles state at a given point of the problem, as well as our various movement and auxiliary functions. We will represent each node's corresponding puzzle as follows:

1_0	4_1	3_2
7_3	0_4	6_5
5_6	8_7	2_8

Stored as a list, **puzzle** = [1, 4, 3, 7, 0, 6, 5, 8, 2].
(The index of each tile is shown in subscript)

Node Initialization

At the creation of a **Node** object, the **__init__** function will take the current puzzle state as an input parameter. Within this function, we will also initialize:

- a list, **children** to store future child nodes
- a variable, **parent** to store the parent node of our current node
- a variable **zero,** to store the index of our zero square

These variables will be set within future functions.

```
from time import time

class Node:
    # Initialization function, run at class creation
    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
```

Movement Functions

We have 4 options when attempting to move a tile, left, right, up, and down; each using the same base method:

1. Check if the move is possible;

(e.g. if the Zero square is in the first column, a move left is not possible; or if the Zero square is in the bottom row, a move down is not possible etc.)

2. Create a copy of the current nodes puzzle stored as **puzzle_copy**; our move operator will modify this copy as to keep the current nodes information intact.

3. Perform the movement.

4. Create the child node with the newly modified puzzle, setting parent/child variables to maintain the path.

Define a function named **create_child;** taking a **puzzle** as an input parameter. This will be called from within our movement functions, creating a child node using its modified puzzle after a move. It will also handle the **children** and **parent** variables for each node.

```
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
```

Define the first movement function, **move_right.** As our 3x3 puzzle is represented as a list of indices 0 to 8, we check that moving the zero-tile right is possible using the modulus function. First, we add 1 to our zero-tile index so that we have a value between 1 and 9, then we check the modulus of the index and the row width is not 0, showing that the zero-tile is not in the right-most column and the move is possible.

For example, in the initial puzzle:

1_0	4_1	3_2
7_3	0_4	6_5
5_6	8_7	2_8

Using the zero-index of 4 we calculate (4 + 1) % 3 = 2, so we know the zero tile is in the second column and a move right is possible. To perform the actual move, it is as simple as swapping the number stored at the zero tiles index, with the number stored one index over.

```
def move_right(self):
    # Check that the zero-tile is not in the right column
    if (self.zero + 1) % 3 != 0:
        # Create a copy of the current nodes puzzle to store the child's modified version
```

```
        puzzle_copy = self.puzzle[:]
        # Swap the position of the zero tile and the tile to its right
        puzzle_copy[self.zero], puzzle_copy[self.zero +
                                    1] = puzzle_copy[self.zero + 1], puzzle_copy[self.zero]
        # Create a child node using the newly modified puzzle
        self.create_child(puzzle_copy)
```

We can now define the remaining moves **move_left, move_up,** and **move_down**, modifying each function to check their respective limits and perform the desired move.

```
def move_left(self):
    # Check that the zero-tile is not in the left column
    if self.zero % 3 != 0:
        # Create a copy of the current nodes puzzle to store the child's modified version
        puzzle_copy = self.puzzle[:]
        # Swap the position of the zero tile and the tile to its left
        puzzle_copy[self.zero], puzzle_copy[self.zero -
                                    1] = puzzle_copy[self.zero - 1], puzzle_copy[self.zero]
        # Create a child node using the newly modified puzzle
        self.create_child(puzzle_copy)

def move_up(self):
    # Check that the zero-tile is not in the top row
    if self.zero > 2:
        # Create a copy of the current nodes puzzle to store the child's modified version
        puzzle_copy = self.puzzle[:]
        # Swap the position of the zero tile and the tile above it
        puzzle_copy[self.zero], puzzle_copy[self.zero -
                                    3] = puzzle_copy[self.zero - 3], puzzle_copy[self.zero]
        # Create a child node using the newly modified puzzle
        self.create_child(puzzle_copy)

def move_down(self):
    # Check that the zero-tile is not in the bottom row
    if self.zero < 6:
        # Create a copy of the current nodes puzzle to store the child's modified version
        puzzle_copy = self.puzzle[:]
        # Swap the position of the zero tile and the tile below it
        puzzle_copy[self.zero], puzzle_copy[self.zero +
                                    3] = puzzle_copy[self.zero + 3], puzzle_copy[self.zero]
        # Create a child node using the newly modified puzzle
        self.create_child(puzzle_copy)
```

Checking the Goal Node

Define a function named **goal_test,** this will be used to loop over the current nodes puzzle and check if it is the goal configuration or not.

```
    def goal_test(self):
        # Loop over length of puzzle
        for i in range(len(self.puzzle)):
            if i != self.puzzle[i]:
                # If Every tile of the puzzle is not correct, return false
                return False
        # If Every tile of the puzzle is correct, return true
        return True
```

Expanding Nodes

Define a function named **expand_node**, this will expand our current node, checking all neighbouring nodes. This will lead to the creation of child nodes by utilizing the newly created movement functions.

```
def expand_node(self):
    # Loop over the current puzzle and find the index of the zero-tile
    for i in range(len(self.puzzle)):
        if self.puzzle[i] == 0:
            self.zero = i
    self.move_right()
    self.move_left()
    self.move_up()
    self.move_down()
```

Printing Results

Define a function **print_puzzle**, this will later be used to iterate over the puzzle and print out the puzzle configurations at each node in the goal path.

```
    def print_puzzle(self):
        print()
        m = 0
        for i in range(3):
            for j in range(3):
                print(self.puzzle[m], end=" ")
                m += 1
            print()
```

Checking Puzzle Viability

One final function must be defined to complete our Node class, **is_solvable**. As discussed previously, this function is used to check the input puzzle configuration and tell the user if it is possible to achieve the goal configuration.

```
    def is_unsolvable(self):
        print(self.puzzle)
        count = 0
        for i in range(8):
            for j in range(i, 9):
                if self.puzzle[i] > self.puzzle[j] and self.puzzle[j] != 0:
                    count += 1
        if count % 2 == 1:
            return True
        else:
            return False
```

Create the Search class

Define a new class named **Search**, this class will be used to contain various search methods. For this lab we will use the Breadth First Search algorithm, however, you may want to expand this to include additional methods. In the **Search** class, define a new function named **breadth_first_search**. This function will execute the previously discussed BFS algorithm to find a path to the goal state.

```
class Search:
    def breadth_first_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))

        while(True):
            # Get next node to search from the top of the list of open nodes
            current_Node = open_list.pop(0)
            # Check if the current node is the goal state
            if current_Node.goal_test():
                # If we have found the goal state, store the path to the current state
                path_to_solution = self.path_trace(
```

```
                current_Node)
            # and, print out total number of moves to reach goal state
            print(len(visited))
            return path_to_solution

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

        # Loop through all nodes neighbouring the current node
        for current_child in current_Node.children:
            # If neighbouring child hasn't previously been visited
            if (not (tuple(current_child.puzzle) in visited)):
                # Add neighbouring child to list of open nodes
                open_list.append(current_child)
                # Add current child to set of visited nodes
                visited.add(tuple(current_child.puzzle))
```

Trace the Path

Define one last function within the **Search** class named **path_trace**; this will be used to trace the path back to the initial node once we have found the goal node. An input parameter **node** is used to pass our goal node into this function, we call this function from within **breadth_first_search** once our goal state has been located.

```
def path_trace(self, node):
    # Store the input node
    current = node
    # Create a list named path, this will store all nodes in the path
    path = []
    # Append the initial node to the path list
    path.append(current)
    # Loop while our current node isn't the root node (as our root node's parent is "None")
    while current.parent != None:
        # Set current node to the parent of the previous node
        current = current.parent
        # Append the current node to the path list
        path.append(current)
    # Return the final path from root node to goal node
    return path
```

Run your code

At the bottom of our file, we must create some code to utilize everything we have produced up until this point.

Note: the statement at the beginning of this block “if __name__ == "__main__"’ is used to check if the module (the source file) is being run as the main program, or if the module is called as a result of an import in another file.

```
if __name__ == "__main__":
    # The puzzle to be solved, you can modify this for any other configuration.
    puzzle = [6, 7, 5, 4, 3, 0, 2, 1, 8]
    # 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)
        # 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)
```

Now that the code is completed running it will complete a BFS of the **puzzle,** printing out the puzzle at each node, the number of steps, and time take to achieve the goal state.

Exercises

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

1. Implement Depth First Search to solve the 8-Puzzle.
2. Solve the N-Queen Puzzle using BFS or DFS.
3. Shortest Path using BFS or DFS

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/