COMP2208 Assignment: Search Methods

Nawab Mir
nm2g18
ID: 29981867

December 2019

Contents

1 Approach **3**

2 Evidence **4**
2.1 Executing the program 4
2.2 DFS 4
2.3 BFS 4
2.4 IDS 5
2.5 A* 5

3 Scalability **6**

4 Extras and Limitations **7**
4.1 Extras 7
4.2 Limitations 7

5 Code **9**
5.1 Node.py 9
5.2 State.py 10
5.3 Search.py 12

1 Approach

For this assignment, I opted to use the language **Python** as I have developed many projects using it and would be able to take advantage of the fact that it operates at a very high level. The resulting code may be slightly slower in execution time, but is extremely readable, with Python standard libraries already containing implementations of Stacks, Queues, and other data types. Since we have no care for the time complexity in terms of milliseconds, but rather in terms of nodes generated, Python serves as an excellent choice. In terms of the actual task, I opted to keep the problem as abstracted as possible so only small changes would need to be made to the standard implementations of the required searching algorithms. Planning is essential with this type of problem, so looking ahead I devised a form of structure. This involved three main classes: **Node**, **State** and **Search**.

The general idea was to have a **Node** containing a **State** with the capability of comparing itself to the **State** of another **Node** in order to evaluate if the solution had been reached. Other features of a **Node** include being able to generate it's own children, store the moves taken to reach this **Node**, the current depth, and the costs of traversing this node (for use with the A* algorithm). In order to combat memory issues that Python runs into, rather than storing the parent of this node as a reference, I opted to store the moves as an array of strings that denote the direction the agent moved in that could simply be passed to the children alongside the most recent move taken to reach that **Node**. This also made outputting to the terminal trivial when compared to having to re-traverse the path taken and print in reverse.

The **State** class was utilised to store the current *state* of the game as a 2D grid. This 2D grid is populated with A, $i \in [1, 2, 3]$ and X strings. A denoting the position of the agent, i denoting each unique movable tile, and X denoting a vacant tile. Alongside the grid, this class also stores the previous move taken to reach this state, and the current position of the agent. The current position of the agent could have been implemented as a function that searches for it, but since certain searches take an extremely long amount of time, shaving off a couple of milliseconds from each node generation can save minutes or even hours of time. Alongside these variables, the **State** also has functionality to initialise a grid with tiles in specific positions, return a list of possible states that could proceed this, and the ability to swap the positions of tiles contained within the grid.

The **Search** class contains all of tree searching algorithms: **DFS**, **BFS**, **DLS** $\rightarrow$ **IDS** and **A*** searches.

The Depth-First Search algorithm is very simple. A stack is initialised with a **Node** containing the starting state. The algorithm loops until the stack is empty. The **Node** at the top of the stack is then checked to see if it is the goal state, if not, all the children of the current node are generated and pushed onto the stack. This process repeats. If the stack is empty and the algorithm escapes the loop, then a solution is not found and an exception is raised. It should be noted that originally the DFS kept getting 'stuck' (since the next **Node** was deterministic defined from any state). The solution to this was to *shuffle* all the generated nodes in the **Node** class before it is returned. The issue that is apparent to this algorithm is that it only ever traverses one infinite path down the tree. It is clear that there is almost no logic, and the solution is almost always sub-optimal.

The Breadth-First Search algorithm operates similarly to the DFS algorithm, except the Stack syntax is swapped out for the Queue equivalent. What this achieves is allowing every node at the current depth of the tree to be expanded before expanding the next level, resulting in shallow solutions being found much earlier. BFS will always find the optimal solution to the problem, though the space required scales exponentially.

The Iterative-Deepening Search performs a true DFS up to a certain depth (meaning we don't just stumble down an infinite path) using a DLS and keeps incrementing the depth until a solution is found, or the depth limit has been reached. It should be noted that this re-generates previously generated nodes.

The A* algorithm calculates the cost of traversing from the current node to unopened nodes using the *open* and *close* lists. Using g as steps and h as the Manhattan distance of all the tiles from their goal location, we can generate a reasonable cost function f. By selecting the lowest f, we can reach the goal state faster.

2 Evidence

2.1 Executing the program

In order to execute the program, the following code serves as an example:

```
123     s = Search()
124     start_state = State([3, 3])
125     start_state.initialise_grid([[3, 0], [3, 1], [3, 2]], [3, 3])
126
127     end_state = State([3, 3])
128     end_state.initialise_grid([[1, 1], [2, 1], [3, 1]], [3, 3])
129
130     dfs = s.dfs(start_state, end_state)
```

Simply replacing s.dfs with s.bfs, for example, will result in a BFS being executed (as long as any extra arguments are provided like max depth)

2.2 DFS

Running the DFS search results in many different outputs. As the DFS technically traverses one infinite path, it could go on forever, until it finds a solution or runs out of memory. Listed below is an example output of running this algorithm (once the randomizing had been implemented).

```
Running DFS
Solution found:
Length of solution is: 9632
['RIGHT', 'UP', 'LEFT', 'UP', 'UP', 'DOWN', 'LEFT', 'LEFT', 'DOWN', 'UP', 'RIGHT',
 'RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'DOWN', 'UP', 'RIGHT', 'DOWN', 'LEFT', 'LEFT',
 'UP', 'UP', 'RIGHT', 'RIGHT', 'LEFT', 'UP', 'LEFT', 'RIGHT', 'RIGHT', 'RIGHT',
 'DOWN', 'DOWN', 'LEFT', 'RIGHT', 'LEFT', 'RIGHT', 'RIGHT', 'LEFT', 'RIGHT',
 'RIGHT', 'LEFT', 'UP', 'UP', 'RIGHT', 'DOWN', 'RIGHT', 'UP', 'UP', 'UP', 'LEFT',
 'RIGHT', 'LEFT', 'DOWN', 'LEFT', 'UP', 'DOWN', 'UP', 'RIGHT', 'DOWN', 'LEFT',
 'RIGHT', 'DOWN', 'LEFT', 'RIGHT', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'RIGHT', 'UP',
 'UP', 'UP', 'RIGHT', 'UP', 'UP', 'RIGHT', 'UP', 'RIGHT', 'RIGHT', 'LEFT', 'RIGHT',
 'UP', 'UP', 'RIGHT', 'DOWN', 'LEFT', 'LEFT', 'LEFT', 'LEFT', 'DOWN', 'RIGHT',
 'LEFT', 'RIGHT', 'LEFT', 'UP', 'DOWN', 'LEFT', 'RIGHT', 'DOWN', 'UP', 'LEFT',
 'LEFT', 'DOWN', 'RIGHT', 'RIGHT', 'LEFT', 'RIGHT', 'RIGHT', 'DOWN', 'UP', 'DOWN',
 'DOWN', 'RIGHT', 'UP', 'RIGHT', 'UP', 'UP', 'UP', 'DOWN', 'LEFT', 'LEFT', 'RIGHT',
>>>
```

We can conclude from this result that DFS does not produce an optimal solution to the problem. The problem solution lies at depth 14, but here we have a solution of 9632. DFS technically could find an optimal path, but it is effectively random in how it does this with a probability tending towards 0.

2.3 BFS

Running the BFS algorithm results in an optimal solution. By expanding every single node until depth d and expanding a random amount of nodes at depth d-1, the algorithm has an upperbound of nodes it will generate before it will find a solution for any given problem, if d is known. Listed below is an example output of running this algorithm.

```
Running BFS
Solution found at depth 14:
['UP', 'LEFT', 'LEFT', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'RIGHT', 'UP', 'UP',
 'LEFT', 'DOWN', 'LEFT']
Time complexity: 1,090,229

>>>
```

We can conclude from this that BFS does eventually arrive at an optimal solution. The main back being that the memory usage is extremely high. All the nodes generated (1,090,229) are stored in memory. From the fact that the tree grows exponentially and every node being expanded, it is not hard to see that this algorithm fails on problems with a greater depth than 14.

2.4 IDS

Running the IDS algorithm results in an optimal solution. By performing DFS up to a certain limit, and incrementing the depth, the algorithm eventually arrives to a solution, though it *wastes* time generating nodes for all the trees of depth limit that do not contain the solution. Although it traverses all the nodes that a BFS algorithm would, it does so with significantly less operating memory since we do not hold failed paths.

```
Running IDS
Solution found at depth 14:
['UP', 'LEFT', 'LEFT', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'RIGHT', 'UP', 'UP',
 'LEFT', 'DOWN', 'LEFT']
Time complexity: 24,492,372

>>>
```

We can conclude from this that IDS does eventually arrive at an optimal solution. The actual time complexity in milliseconds is drastically higher than any other searching algorithm due to destroying and reconstructing the same tree (24,492,372) nodes - around 23 million of these are *wasted*. The main plus side is that we can definitely compute solutions to problems which BFS runs out of memory for, and we have some form of logic unlike DFS.

2.5 A*

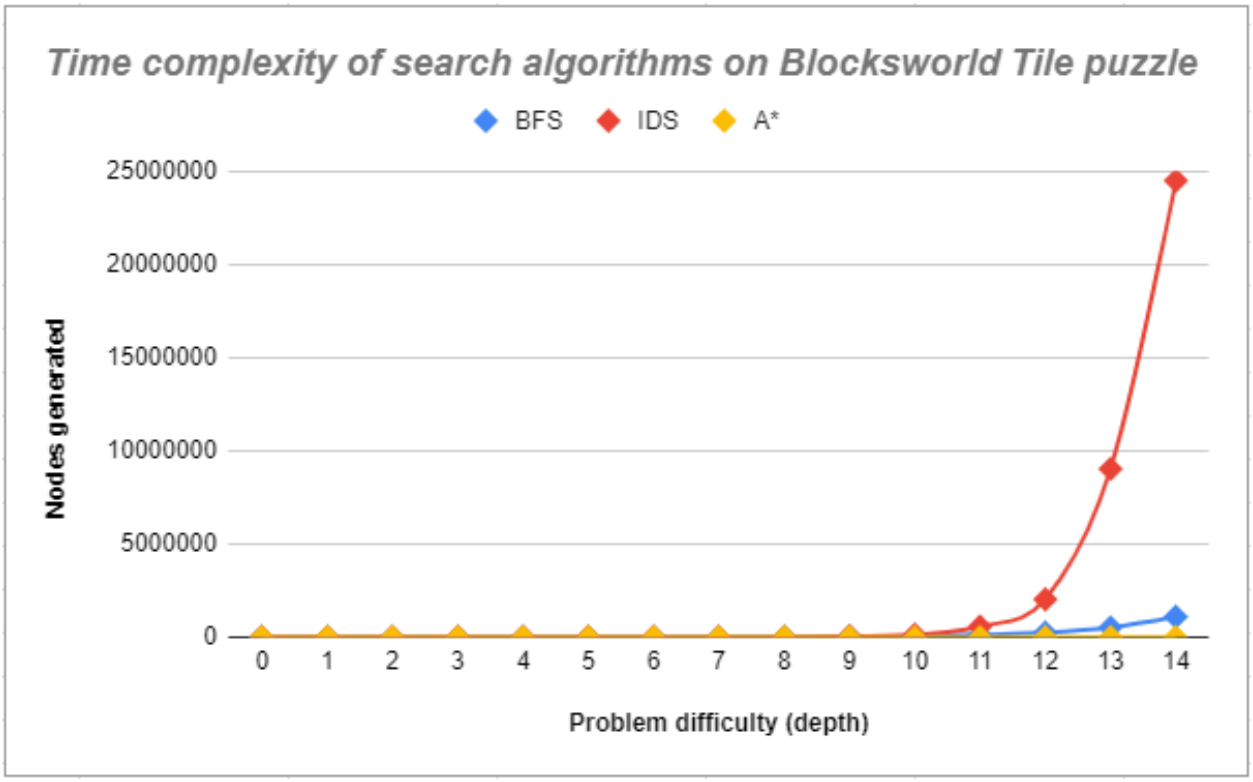
Running the A* algorithm results in an optimal solution. The utilisation of a heuristic cost from state to state definitely makes this the smartest algorithm out of all of the tested search methods. Now the actual nodes generated work towards a general goal, to get closer to the optimal solution without generating useless nodes that serve no purpose like the other *dumb* search algorithms.

```
Running A*
Solution found at depth 14:
['UP', 'LEFT', 'LEFT', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'RIGHT', 'UP', 'UP',
 'LEFT', 'DOWN', 'LEFT']
Time complexity: 1,016

>>>
```

We can conclude from this that A* very quickly arrives at an optimal solution, to the point that there is no comparison to the other algorithms. By the time another algorithm has reached depth 5, A* has already arrived at a solution (only 1,016 nodes generated).

3 Scalability



The above graph shows a plot of problems with solution depth $d \leq 14$ with DFS not included. The graph has been stopped at $d = 14$ as BFS begins to fail past this, due to lack of RAM (and possibly the sluggish nature of Python). This is because it is purely random and would not show any trend unless it was sampled thousands of times and averaged, and even that would only show a general relation between problem difficulty and nodes generated. We can see that the most scalable algorithm here is the A* algorithm which effectively uses a linear amount of time in comparison to the exponential nature of BFS and IDS. We can conclude that A* > BFS > IDS in terms of time complexity and scalability. DFS technically arrives at a solution faster than BFS and IDS, the solution is not optimal and thus has been discarded.

It is important to note that IDS generates a huge amount of nodes as displayed by the graph, but realistically when solving the problem, we only care about the final iteration which contains the solution (the rest are garbage collected). Not only this, but BFS actually generates an extra layer of nodes below the depth of the solution unlike IDS, which in reality makes it more efficient $\rightarrow T_{IDS}(n) : O(b^d), T_{BFS}(n) : O(b^{d+1})$.

If you were required to generate an *intelligent system* to solve this problem optimally in terms of space and time, A* would be the better choice by a huge margin.

4 Extras and Limitations

4.1 Extras

Several adaptations were made to the Node close in order to conserve memory and also allow certain algorithms to function. The first issue was the DFS becoming 'stuck'. This was because the code logic calculates the possible moves in the following order: [UP, DOWN, LEFT, RIGHT]. When iterating through these and pushing these onto the stack, it's clear to see that the first node to be popped off this is RIGHT (unless RIGHT is not possible, in which LEFT is at the top of the stack). This carries on infinitely and causes the agent to simply slide left until it hits a boundary, slide to the right and repeat. Shuffling the children once they are generated proved to be a fine solution, though slightly problematic for BFS and IDS.

Opting to store the moves taken to reach a certain node within itself as opposed to storing a reference to the parent node was an excellent idea. Although this *technically* should not affect the results, it is significantly more memory efficient and for such a problem, the difficulty can exponentially increase with only a single extra layer to the tree. This is particularly problematic for algorithms such BFS which store a huge amount of nodes in memory. Making this change made it reasonable to compute problems of a higher difficulty level without running out of heap space.

One adaptation I made to the problem was allowing for a resizable grid and the ability to have more or less than 3 movable tiles. Increasing either of these variables, increases the problem difficulty as there are more nodes and states to explore. This can become particularly problematic for all of the algorithms bar A* as they are generally unguided and focus on expanding as many nodes as possible; A* uses a heuristic to combat huge increases in problem difficulty. Interestingly, it was found that increasing problem difficulty in this way resulted in BFS running out of memory due to the necessity of storing a huge amount of nodes in memory (even after making the change to store moves taken rather than references to parent nodes). On the contrary, IDS held up with increases to problem difficulty as a result of its memory-efficient DFS nature.

4.2 Limitations

There have been several limitations and aspects that could've been altered to provide better, more reliable results. The first being that the shuffling of the child nodes mentioned in **4.2** directly affects the BFS and IDS algorithms. Although these algorithms will reach the goal state and produce the same optimal path, the order of expansion becomes probabilistic. This can be the difference between generating 2 extra nodes, or 200,000 at later problem difficulties. This was clearly a mistake, and has also affected the scalability results requiring a mean value. It would have been more appropriate to add a flag to trigger shuffling of children, only if the search method was DFS. This is slightly problematic though, as I tried to define these classes as generically as possible to improve re-usability. You can see from the **Search** class that the search methods are almost textbook defined except for the state checking branches to see if we have arrived at the goal state. This can easily be translated to other searching problems. However, for this problem, it was not optimal.

The other major issue was with the A* algorithm. Even though it had the lowest time complexity, it should have executed faster, which is down to the fact that I used 2 lists to track nodes. It's often advised to use priority queues to prevent slowdown from parsing these lists for the required node. A reduction from O(n) to O(1) can be huge when in situations where the lists are very, very large - if the grid-size is huge.

My method for generating problems of depth d was also very primitive. I opted to start with some random nodes and by eye, took d steps to somewhat randomise the input. I then marked the state I started with as the end state and the state I ended with as the start state. Although this works, a better solution would be to have some form of **ProblemGenerator** class that randomly creates grids and returns their start and end states using the same method I did. The **ProblemGenerator** would need to ensure that the start state $\neq$ end state for any generated problem. I could then pump random states into these algorithms to examine how well they perform, and have extra data / problem sets to display in the scalability study.

I actually could have optimised memory use even more for DFS. Seeing as it only ever traverses one infinite path, I technically could have deleted all child nodes apart from the one that was going to be selected. This would result in a reduction of 50-75 percent of nodes generated (since there is no reason to generate effectively dead nodes, in the same way we do not generate dead nodes that occur if we are against a boundary).

5 Code

5.1 Node.py

```
from random import shuffle

class Node:
    """
    This class generically defines a Node for the tree traversal algorithms
    Stores the current state of the game, the moves taken, and the depth of the node
    """
    def __init__(self, state, moves_taken=[], depth=0):
        self.state = state
        self.moves_taken = moves_taken
        self.depth = depth
        self.g = 0
        self.h = 0
        self.f = 0

    def generate_children(self):
        children = []
        moves = self.state.possible_moves()
        for move in moves:
            children.append(Node(move, moves_taken=self.moves_taken + [move.move_taken],
            depth=self.depth + 1))
        shuffle(children)

        return children

    def is_state(self, target_state):
        grid = self.state.grid
        grid[self.state.agent_pos[0]][self.state.agent_pos[1]] = 'X'
        target_state.grid[target_state.agent_pos[0]][target_state.agent_pos[1]] = 'X'
        if self.state.grid == target_state.grid:
            return True
        else:
            return False
```

5.2 State.py

```
class State:
    def __init__(self, agent_pos, grid=None, move_taken=None):
        self.grid = grid
        self.move_taken = move_taken
        self.agent_pos = agent_pos

    def initialise_grid(self, blk_pos, agent_pos, n=4):
        """
        :param blk_pos: a list (x,y) tuples of blocks
        :param agent_pos: (x,y) position of agent
        :param n: size of grid n*n
        """
        self.agent_pos = agent_pos
        self.move_taken = None
        self.grid = [['X']*n for _ in range(n)]
        self.grid[agent_pos[0]][agent_pos[1]] = 'A'
        for i in range(0, len(blk_pos)):
            self.grid[blk_pos[i][0]][blk_pos[i][1]] = str(i+1)

    def possible_moves(self):
        """
        Produces all possible moves from the current State

        :returns a list of States
        """
        moves = []
        ap = self.agent_pos

        # Move up
        try:
            new_pos = [ap[0]-1, ap[1]]
            temp = self.grid[new_pos[0]][new_pos[1]]
            new_state = State(new_pos, grid=self.swap_tiles(ap, new_pos), move_taken="UP")
            moves.append(new_state)
        except:
            pass

        # Move doum
        try:
            new_pos = [ap[0]+1, ap[1]]
            temp = self.grid[new_pos[0]][new_pos[1]]
            new_state = State(new_pos, grid=self.swap_tiles(ap, new_pos), move_taken="DOWN")
            moves.append(new_state)
        except:
            pass

        # Move left
        try:
            new_pos = [ap[0], ap[1]-1]
            temp = self.grid[new_pos[0]][new_pos[1]]
            new_state = State(new_pos, grid=self.swap_tiles(ap, new_pos), move_taken="RIGHT")
            moves.append(new_state)
```

```
        except:
            pass

        # Move right
        try:
            new_pos = [ap[0], ap[1]+1]
            temp = self.grid[new_pos[0]][new_pos[1]]
            new_state = State(new_pos, grid=self.swap_tiles(ap, new_pos), move_taken="UP")
            moves.append(new_state)
        except:
            pass

        return moves

    def swap_tiles(self, agent_pos, tile_pos):
        """
        :param agent_pos: position of the agent
        :param tile_pos: the tile the agent wants to move

        :returns new grid with swapped positions
        """
        temp_grid = self.grid
        if temp_grid[tile_pos[0]][tile_pos[1]] == "X":
            temp_grid[tile_pos[0]][tile_pos[1]] = 'A'
            temp_grid[agent_pos[0]][agent_pos[1]] = 'X'
        elif temp_grid[tile_pos[0]][tile_pos[1]].isdigit():
            temp_grid[agent_pos[0]][agent_pos[1]] = temp_grid[tile_pos[0]][tile_pos[1]]
            temp_grid[tile_pos[0]][tile_pos[1]] = 'A'
        else:
            raise Exception("Invalid swap")

        return temp_grid
```

5.3 Search.py

```
from collections import deque
from operator import attrgetter
from queue import Queue
from Node import Node
from State import State

class Search:
    """
    The class containing all search algorithms that the agent utilises
    """

    def dfs(self, start_state, end_state):
        t_complexity = 1
        initial_node = Node(start_state)
        stack = deque()
        stack.append(initial_node)
        while stack:
            node = stack.pop()
            if node.is_state(end_state):
                return [node, t_complexity]

            for n in node.generate_children():
                stack.append(n)
                t_complexity += 1

        raise Exception("No solution found!")

    def bfs(self, start_state, end_state):
        t_complexity = 1
        initial_node = Node(start_state)
        queue = Queue()
        queue.put(initial_node)
        while queue:
            node = queue.get()
            if node.is_state(end_state):
                return [node, t_complexity]

            for n in node.generate_children():
                queue.put(n)
                t_complexity += 1

        raise Exception("No solution found!")

    def dls(self, start_state, end_state, max_depth):
        t_complexity = 1
        initial_node = Node(start_state)
        stack = deque()
        stack.append(initial_node)
        while stack:
            node = stack.pop()
            if node.is_state(end_state):
                return [node, t_complexity]
```

```
            if node.depth < max_depth:
                for n in node.generate_children():
                    stack.append(n)
                    t_complexity += 1

        return [False, t_complexity]

    def ids(self, start_state, end_state, depth_limit):
        t_complexity = 0
        for i in range(depth_limit + 1):
            node, t = self.dls(start_state, end_state, i)
            t_complexity += t
            if node:
                return [node, t_complexity]

        raise Exception(f"No solution found for depth limit {depth_limit}!")

    def a_star(self, start_state, end_state):
        t_complexity = 1
        initial_node = Node(start_state)
        opened = [initial_node]
        closed = []
        while opened:
            node = min(opened, key=attrgetter('f'))
            if node.is_state(end_state):
                return [node, t_complexity]
            else:
                opened.remove(node)
                closed.append(node)
                for n in node.generate_children():
                    t_complexity += 1
                    is_closed = False
                    is_improvement = True

                    for c in closed:
                        if n.is_state(c.state):
                            is_closed = True
                            break

                    if is_closed:
                        continue

                    n.g = node.g + 1
                    n.h = self.heuristic_cost(n.state.grid, end_state.grid)
                    n.f = n.g + n.h

                    for o in opened:
                        if n.is_state(o.state) and n.g >= o.g:
                            is_improvement = False
                            break

                    if is_improvement:
                        opened.append(n)
```

```
        raise Exception(f"No solution found!")

    def heuristic_cost(self, grid, end_grid):
        cost = 0
        for i in range(len(grid)):
            for j in range(len(grid[i])):
                if grid[i][j] not in ['A', 'X']:
                    for k in range(len(end_grid)):
                        for l in range(len(end_grid[k])):
                            if grid[i][j] == end_grid[k][l]:
                                cost += abs(i-k) + abs(j-l)

        return cost
```