Project 1: Search

1 Program Design

As a team, we choose **Python** as the programming language of our project to solve this sliding-tile problem.

1.1 Representations

In this project, we use several data types in Python to represent states and actions in the sliding-tile puzzle.

1.1.1 State representation

Firstly, we use *list* data type in Python to represent each state of the puzzle. As illustrated in Figure 1.1, this puzzle is the goal state with 0 in the top left, followed by a series of numbers increased by one in each tile to the bottom right. The top left is the number zero, which represents the blank space in the puzzle. The whole state can be represented by *list[0, 1, 2, 3, 4, 5, 6, 7, 8]*. The initial state is also represented in the *tuple* data type and it is converted from the plain text input.

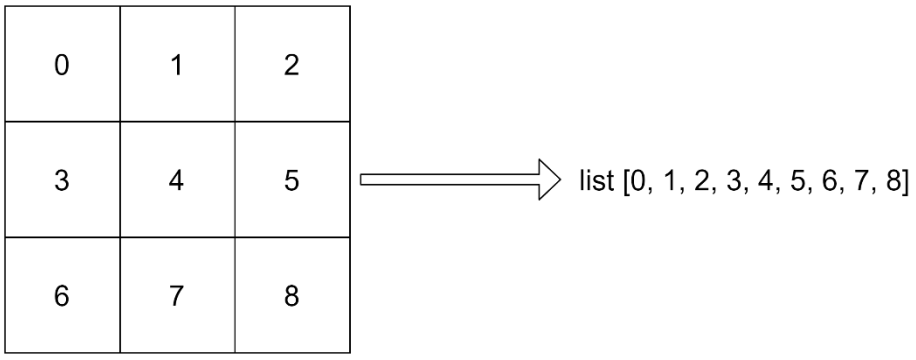

Figure1.1 State representation

1.1.2 Action representation

Secondly, we use *list* data type in Python to represent each action of moving a tile in the puzzle. As illustrated in Figure 1.2, the complete action list is [‘UP’, ‘DOWN’, ‘LEFT’, ‘RIGHT’] only when the blank space is in the center of the puzzle. To obtain action lists when blank space is in other locations, we create a *dictionary* of action fence and use indices to refer to different locations in the puzzle (Figure 1.3). When the blank space is in the top edge, we cannot move it up. Therefore, ‘UP’ action should be removed from the action list. When the blank space is in the bottom edge, we cannot move it down. In this case, ‘DOWN’ action should be removed from the action list. Similarly, when the blank space is in the left edge, we remove ‘LEFT’ action from the action list. Also, we delete ‘RIGHT’ action when the blank space is in the right edge. In our algorithm, we create a *dictionary* of action fences to represent this action limitation.

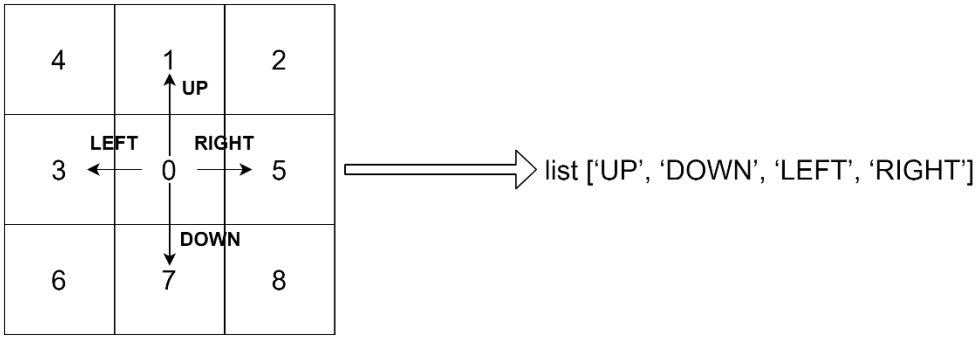

Figure1.2 Action representation

Figure1.3 Action fence based on the indices

Each time we move a tile in the puzzle, the blank space changes and we get to a new state. To represent this transition, we create a *dictionary* of action offset. When the blank space does 'UP' action, its index decreases by 3. When the blank space does 'DOWN' action, its index increases by 3. Also, we add 1 to the index of the blank space while doing the 'RIGHT' action and subtract its index by 1 while doing the 'LEFT' action. Then, we swap values at the index of the current blank space and the index that the blank space will be located at.

1.2 Classes and Functions

Our program contains two classes and seven functions. These two classes are Puzzle class and Node class.

Puzzle class has two object properties: the initial state of the puzzle and its action fence. This class has several methods to get the action fences, the goal state and the next state, to find the blank space, to filter actions and to compute the heuristic function.

Node class has four object properties: the current state, the parent node, the valid action of the parent node and the current depth. This class has methods of getting the next node, and the action list of a complete search.

The three main algorithms are Breadth-first Search (BFS), AStar Search(A*) and Iterative Deepening AStar Search (IDA*). The heuristic function used in A* and IDA* search is described in the section 1.2.1. As for IDA*, the depth bound is measured in terms of the f value, which is the sum of the search depth (g value) and the value estimated by the heuristic function (h value). If solution is not found at a given depth, the depth bound is increased to the minimum of the f-values that exceed the previous

bound.
In addition to these three algorithms, we create methods to read input files and get the initial state, to calculate the running time of each algorithm, and to write the results to output files.

1.2.1 Heuristic Function

In our A star search algorithm, we create three heuristic functions which can be chosen by the parameter 'h_flag'. The default value of 'h_flag' is 0.
The first heuristic function is to sum the steps for each title to reach their goal place
Calculate how many steps required for each title to achieve its goal place. For example, for a 3*3 grid, it will take a title with value 4 but placed in the bottom right at the index of 8 two steps (left and up) to reach its place.
The second heuristic function is inspired by Bubble Sort. For each title, we count the number of tiles in front of that tile that have bigger values. For example, for title with value 4, if tiles in front of it have values 1, 6 and 7, then we add 2 to the h value.
The third heuristic function is to count the number of tiles out of place except the blank. For example, for a 3*3 grid with value [1, 0, 2, 3, 4, 5, 6, 7, 8], as only the 1 is out of place, the h value is 1.

1.3 Run the Program

Before running our program, remember to update the input file path for three folders: easy, moderate and difficult. Then run the main function and you will get the results. On each difficulty level, you will get three folders for each of three main algorithms containing output files. In each folder, there are two kinds of output files that this program creates. One kind of files is the output of every single initial state given by each input file. This kind of files shows the initial state, the action list, the number of steps and the running time. Another kind of files ending with 'total_result' is the summary of all the results on the same difficulty level.

2 Collaboration

Contributions			Working time	
Coding	Classes	Puzzle		0.5h
		Node		0.5h
	Functions	BFS	20min	1h
		A Star	30min	2h
		IDA Star		4h
		Read input files	30min	
		Write output files	10min	
		Calculate the running time	20min	
		Main	1h	
Writeup		Program Design	3h	0.5h
		Collaboration	10min	5min
		Results	40min	

	Discussion	20min	1h

3 Results

We run our program on Jupitar.CIRC with 1 CPU and 6GB Memory. Table 3.1, 3.2, 3.3 show the running times in milliseconds of three search algorithms on three difficulty levels: easy, moderate and difficult. In this project, we call a puzzle "effectively solved" if our program can identify the shortest path in less than about a minute. As shown in tables below, all the running times are less than one minute, so we can say that all the puzzles are effectively solved by our program.

Table3.1 Running time (in milliseconds) of breadth-first search algorithm

Filename	Easy	Moderate	Difficult
3x3_1	0.192	27.932	42.575
3x3_2	0.408	22.079	1566.707
3x3_3	0.796	83.077	728.735
3x3_4	0.11	0.877	1920.019
3x3_5	0.345	1.172	260.946
3x3_6	0.885	25.202	392.783
3x3_7	0.627	0.455	331.04
3x3_8	0.656	0.815	281.57
3x3_9	0.168	6.627	146.559
3x3_10	1.375	3.052	1406.887
Mean	0.5562	17.1288	707.7821

Table3.2 Running time (in milliseconds) of AStar Search Algorithm

Filename	Easy	Moderate	Difficult
3x3_1	0.254	2.47	26.757
3x3_2	0.369	1.819	536.884
3x3_3	0.52	7.608	46.584
3x3_4	0.147	0.656	171.117
3x3_5	0.415	0.649	15.593
3x3_6	0.52	6.444	39.737
3x3_7	0.407	0.55	38.116
3x3_8	0.47	0.501	18.415
3x3_9	0.3	1.717	6.971
3x3_10	0.613	0.734	27.304
Mean	0.4015	2.3148	92.7478

Table3.3 Running time (in milliseconds) of Iterative Deepening AStar Search Algorithm

Filename	Easy	Moderate	Difficult
3x3_1	0.268	4.857	27.897
3x3_2	0.698	4.246	4324.574
3x3_3	0.473	18.463	55.326
3x3_4	0.109	0.667	249.187
3x3_5	0.368	0.552	54.841
3x3_6	0.667	18.604	108.024
3x3_7	0.492	0.693	126.501

3x3_8	0.514	0.532	26.223
3x3_9	0.283	3.019	7.558
3x3_10	0.441	1.191	63.207
Mean	0.4313	5.2824	504.3338

4 Discussion

4.1 Coding procedure

There are many packages including data structures make it convenient for coding, such as deque and priority queue. Moreover, for created classes, functions like comparison methods can be overridden for hash sort or other purposes.

4.2 Algorithm Effectiveness

Though the program should be compatible with **4*4** puzzles, it does not really work in any algorithms. For limited easy puzzles, like [1, 0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16], it takes only several steps to reach the goal state. However, for most puzzles, the program fails and always results in out of memory after a long time of operating in the Jupyter. Improvement should be made for a more robust program.

4.3 Time Efficiency

During coding, the first version of BFS and A* was effective but time-consuming. After experiments and analysis, we found that most time was spent in checking if the state is visited. At first, we used 'in' function and the type of visited states is *list*. According to the search, in terms of efficiency for 'in' function, *set* is the best, *dictionary(dict)* takes second place, and *list* is the worst one (set > dict > list). This is caused by their essence, where the time complexity for *list* is O(n), *set* is de-duplicated and works like a Red–black tree, the time complexity is O(logn), and for the *dict*, it is actually turned into a hash then do the same job as *set*. As a result, *dict* is slower than *set*. Therefore, for frequent 'in' function, *set* works most efficiently.

Furthermore, many other experiments are taken to improve the time efficiency, including replacing numpy.nrange with range and so on.

What is equally worth discussing is that differences between different algorithms. As shown in the results, A* Search Algorithm works most efficiently compared to BFS and IDA*. And BFS is the least time-efficient. For some puzzles, IDA* works better than A* while IDA* spent the longest time in other test cases. It can be speculated through its theory which combined deep first search and A*. A lot of efforts were taken to improve the time efficiency for IDA* but not huge progress was achieved.

Finally, we think that completing an AI project is not only a work of coding, the writeup is also important. Writing a report based on what we coded helps us organize our program better and think about what could be improved. Also, a writeup is necessary if others want to have a look at our project. They may struggle in our codes without explanation of a writeup.