VIETNAM NATIONAL UNIVERSITY

HO CHI MINH UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

LAB 01: N-PUZZLE PROBLEM

<u>Course</u>: Artificial Intelligence

<u>Students:</u>

Hoàng Bảo Khanh (22127183)

<u>Lecturers:</u>

PhD Nguyễn Ngọc Thảo

M.S Lê Ngọc Thành

M.S Nguyễn Hải Đăng

M.S Nguyễn Trần Duy Minh

Ho Chi Minh city, March 17th, 2024.

CHECKLIST:

	Features	**Complete**
Algorithm	UCS	100%
	A*	100%
Standard features	Inversion Distance	100%
	Check solvable puzzle	100%
	Move the blank square	100%
	Create the default puzzle (with 8, 15, 35 puzzle)	100%
	Measure the runing times	100%
	Measure the consumed memory	100%
Extra features	GUI for user	100%
	Create the new puzzle	100%
	Check the valid puzzle	100%

DESCRIPTION FUNCTIONS:

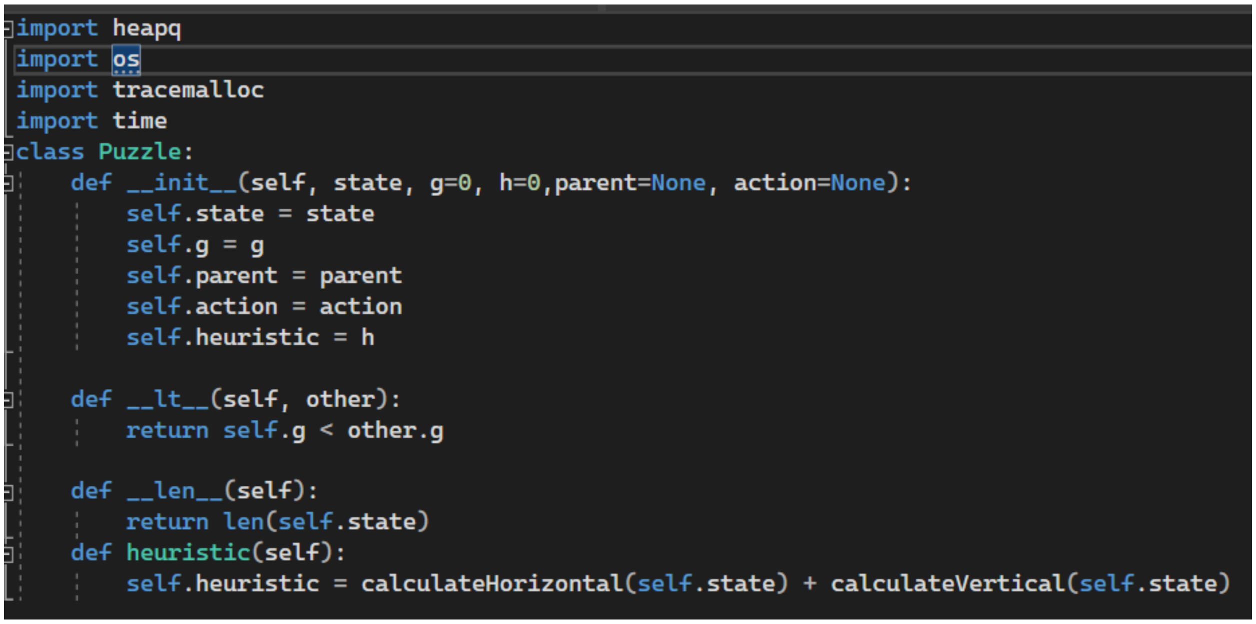
1. Library:

In order to solve the N-puzzle problem, I wrote the Python code and use many libraries:

- heapq: to create the Priority Queue.
- tracemalloc: to measure the consumed memory used in the program.
- time: to calculate the run time of the algorithm.

2. Initialize the Puzzle:

I used class to initialize the Puzzle. The puzzle included the state of the board, path cost (g variable in the class), heuristic value (h variable in the class), parent of state (parent) and the action of the blank box (moved left, right, up, or down). For calculating the heuristic value of the puzzle, I used the Inversion Distance.



```python
import heapq
import os
import tracemalloc
import time
class Puzzle:
    def __init__(self, state, g=0, h=0,parent=None, action=None):
        self.state = state
        self.g = g
        self.parent = parent
        self.action = action
        self.heuristic = h

    def __lt__(self, other):
        return self.g < other.g

    def __len__(self):
        return len(self.state)
    def heuristic(self):
        self.heuristic = calculateHorizontal(self.state) + calculateVertical(self.state)
```

Image 1: Puzzle structure in the program

3. Check the solvable puzzle:

There are many ways to create the initial state of the puzzle, and not all of this can be solved to the goal state. Therefore, I created the function to check if the puzzle is solvable or not.

At first, I wrote the function to calculate the number of inversions.

For example, if we have the square matrix below:

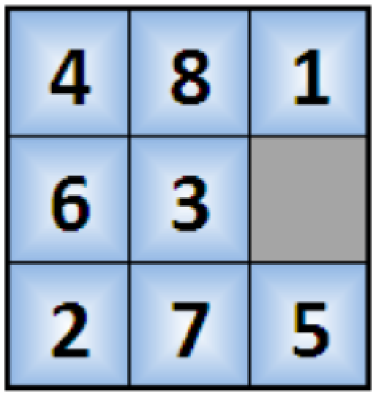


Image 2: Matrix 3x3 (source: aptech.vn)

To find the number of inversions, the program considers unravelling the square into a single row of tiles (from left to right, top to bottom).

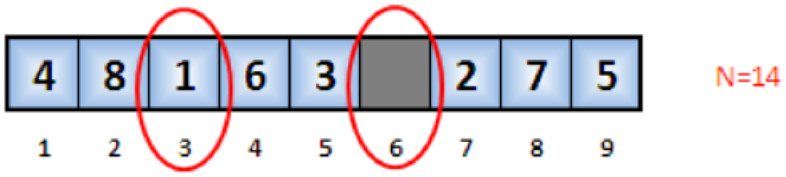


Image 3: Row of tiles

And then, with each square, the program counts the tiles which value is smaller than the tiles' value.

Supposed that we have the the number of inversions is N, the puzzle is solvable when:

- The size is odd, and N is even.
- The size is even:
    - N is even and the blank square is in the even row.
    - N is odd and the blank square is in the odd row.

4. Implement Uniformed Cost Search:

Uniformed Cost Search is the search algorithm by comparing the path cost. To implement this feature, I used two main lists: frontier list and visited list.

Moreover, I also installed many functions to support running this algorithm:

- Expand fucntion: to add the expanded node.
- Find blank space function: to find the blank square in the puzzle.
- Action function: to store the action from the initial node to the present node.
- Result function: to reach a new state with the appropriate action.

5. **Graph-search A* with Inversion Distance heuristic:**

Initially, I installed the function to calculate the Inversion Distance. Based on the Puzzle blog of Michael Kim, the Inversion Distance is installed by the value of vertical and horizontal.

To find the value of vertical, the program unravels the puzzle into single row of tiles (from left to right, top to bottom) and counts the number of inversions. The program runs the same with horizontal function, however the program will unravel into the row from top to bottom, left to right.

In the main function to run the A*, I used 3 main lists: frontier list, g_score list (which stores the path cost of the state) and the f_score list (which stores the total value of the path cost and the heuristic value). The main function is also supported by other functions.

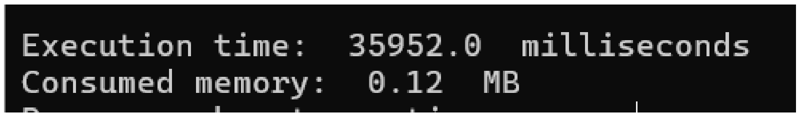
6. **Running times and Consumed memory**
a. **8 puzzles:**

For 8 puzzles, I chose this test case of Puzzle to measure the run times and consumed memory: [[2, 0, 5], [8, 3, 4], [1, 7, 6]], (0 is the blank square).

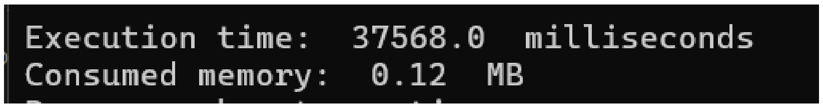
In each algorithm, the program takes 3 runs and the statistics will be collected in average of 3 times run.

- **Uninformed-cost search (UCS):**

The first test:

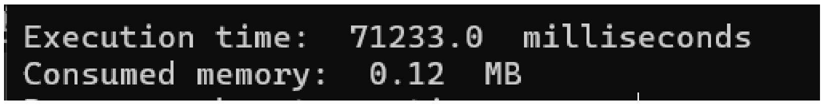

```
Execution time:  35952.0  milliseconds
Consumed memory:  0.12  MB
```

The second test:



```
Execution time:  37568.0  milliseconds
Consumed memory:  0.12  MB
```

The third test:



```
Execution time:  71233.0  milliseconds
Consumed memory:  0.12  MB
```

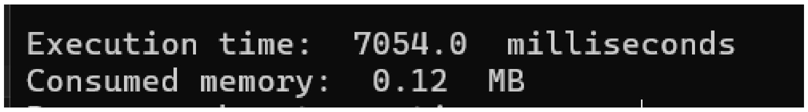
In average:

Execution time: 48251 milliseconds.

Consumed memory: 0.12 MB.

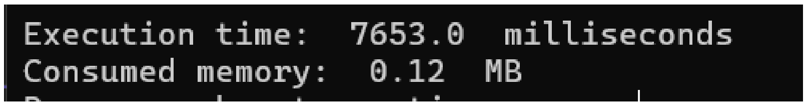
- **A* Algorithm:**

The first test:



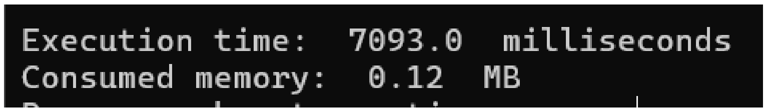
```
Execution time:  7054.0  milliseconds
Consumed memory:  0.12  MB
```

The second test:



```
Execution time:  7653.0  milliseconds
Consumed memory:  0.12  MB
```

The third test:



```
Execution time:  7093.0  milliseconds
Consumed memory:  0.12  MB
```

In average:

Execution time: 7266.67 milliseconds.

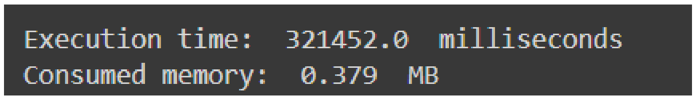
Consumed memory: 0.12 MB.

b. 15 puzzles:

For 15 puzzles, the program has the default Puzzle: [[1, 3, 4, 8], [6, 2, 10, 7], [5, 9, 12, 0], [13, 14, 11, 15]].

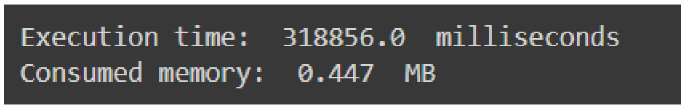
- **Uninformed-cost search (UCS):**

The first test:



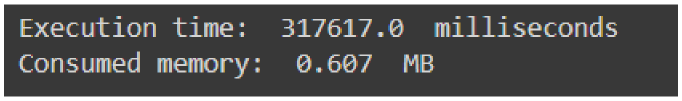
```
Execution time:  321452.0  milliseconds
Consumed memory:  0.379  MB
```

<u>The second test:</u>



```
Execution time:  318856.0  milliseconds
Consumed memory:  0.447  MB
```

<u>The third test:</u>



```
Execution time:  317617.0  milliseconds
Consumed memory:  0.607  MB
```

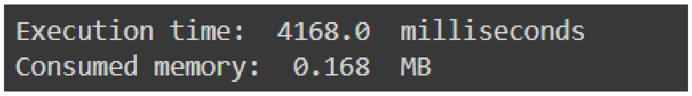
In average:

Execution time: 319308.33 milliseconds.

Consumed memory: 0.478 MB.

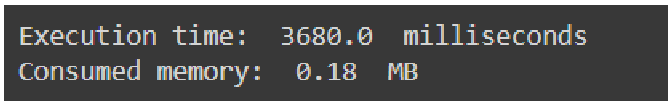
- **A* algorithm:**

<u>The first test:</u>



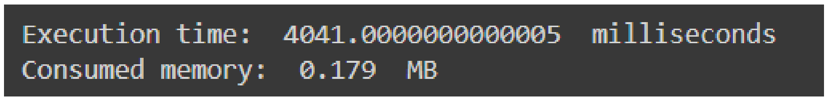
```
Execution time:  4168.0  milliseconds
Consumed memory:  0.168  MB
```

<u>The second test:</u>



```
Execution time:  3680.0  milliseconds
Consumed memory:  0.18  MB
```

<u>The third test:</u>



```
Execution time:  4041.0000000000005  milliseconds
Consumed memory:  0.179  MB
```

In average:

Execution time: 3963 milliseconds.

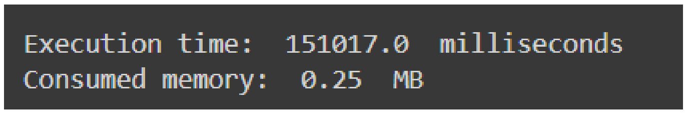
Consumed memory: 0.176 MB.

c. 35 puzzles:

The default 35-puzzle in the program is [[1, 2, 3, 4, 5, 6], [0, 8, 9, 10, 11, 12], [7, 20, 14, 15, 17, 18], [13, 19, 21, 16, 23, 24], [25, 26, 27, 22, 28, 30], [31, 32, 33, 34, 29, 35]]

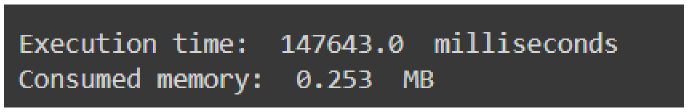
- **Uninformed-cost search (UCS):**

<u>The first test:</u>



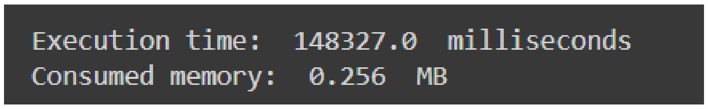
```
Execution time:  151017.0  milliseconds
Consumed memory:  0.25  MB
```

<u>The second test:</u>



```
Execution time:  147643.0  milliseconds
Consumed memory:  0.253  MB
```

<u>The third test:</u>



```
Execution time:  148327.0  milliseconds
Consumed memory:  0.256  MB
```

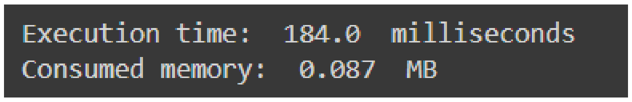
<u>*In average:*</u>

Execution time: 148995.67 milliseconds.

Consumed memory: 0.253 MB.

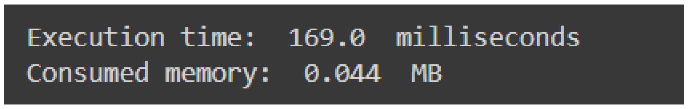
- **A* algorithm:**

<u>The first test:</u>



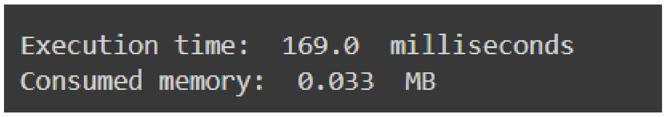
```
Execution time:  184.0  milliseconds
Consumed memory:  0.087  MB
```

<u>The second test:</u>



```
Execution time:  169.0  milliseconds
Consumed memory:  0.044  MB
```

<u>The third test:</u>



```
Execution time:  169.0  milliseconds
Consumed memory:  0.033  MB
```

<u>*In average:*</u>

Execution time: 174 milliseconds.
Consumed memory: 0.0547 MB.

<u>Statistical table of running times and consumed memory</u>:

	Running time (ms)			Memory (MB)		
Algorithms	N = 8	N = 15	N = 35	N = 8	N = 15	N = 35
UCS	48251	313308.33	148995.67	0.12	0.478	0.253
A*	7266.67	3963	174	0.12	0.176	0.0547

<u>Conclusions:</u>

- In general, the running times of A* algorithm are considerably smaller than UCS. Therefore, A* algorithm is a more optimal search algorithm.
- The long running time causes the increase of consumed memory.
- The UCS algorithm is implemented based on the path cost of the state. The A* algorithm depends on the path cost and heursitic value. Therefore, the A* algorithm has more perception and is more efficient when searching for the goal.
- The UCS and A* have the modest consumed memory, which is less than 1 MB due to those algorithms only store the neccesarry node.

REFERENCES

[1]. Blog: Solving the 15 puzzle, Michael Kim, January 27, 2019.
[2]. A* Search Algorithm, Geeksforgeeks.
[3]. Welcome to N-Puzzle, tristanpenman.
[4]. N-puzzle: Tìm hiểu về cách giải bài toán, aptech.vn