personal-code-assignment-report

March 13, 2025

1 Pseudocode

1.1 Input Process

[7]:
```
vector<vector<int>> get_board():
    read input_string
    parse input into 2x3 board
    return board
```

In this function I use getline to handle the input string, and store it into a 2x3 vector.

1.2 Compute Manhattan Distance

[8]:
```
int manhattan(board):
    compute sum of Manhattan distances for all tiles
    return distance
```

This function calculating the manhattan distances as heuristic function.

1.3 Define State Structure

[9]:
```
struct State:
    board, g, h, path

    char get_priority(move): return priority order

    operator<(other_state):
        return compare (g + h), or path priority
```

This struct record the state of the board at a certain time. Including actual cost g, heuristic function h, and current path. And a function used to define the priority of the path.

1.4 Generate Next States

[10]:
```
vector<State> get_next_state(current_state):
    find empty tile position
    generate new states by moving empty tile in valid directions
    return list of new states
```

This function return a vector that store states, it will look for the neighbors of given state. And automatically update g by plus 1, calculate the manhattan and updating the path to the state.

1.5 Convert Board to String

[11]:
```
string board_to_str(board):
    return string representation of board
```

trans board vector to string, later it will be used to check for state has been visited or not.

1.6 Solve Puzzle Using A* Search

[ ]:
```
string solve_puzzle(board):
    initialize priority queue pq, visited map
    create start_state, INSERT into pq and visited

    while pq is not empty:
        pop state with lowest (g + h)
        if goal reached, return path

        for each neighbor in get_next_state(current_state):
            if neighbor not in visited or has lower g:
                update visited, push into pq

    return "None"
```

This function using A* search. It used a priority queue to determine which path to explore next, the one that has lowest g + h will be considered, if they have same cost, the function will compare their path, following by l -> u -> d -> r priority to determine. if it find a solution, return the path, otherwise it return None.

1.7 Main Function

[7]:
```
int main():
    board = get_board()
    solution_path = solve_puzzle(board)
    print solution length and path
```

Implement the above function and output results.

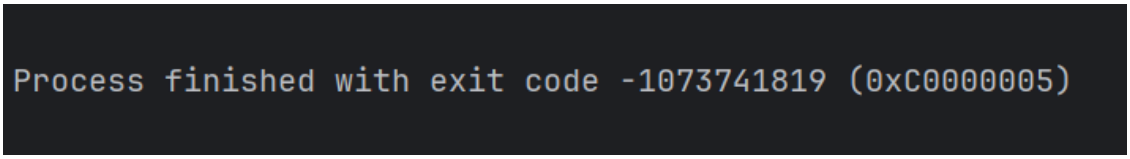
2 Problems Encountered During Implementation

2.1 Input Format

During coding, the first problem I met is the input format, I thought the input should be a continuous string without blank, and it caused problem in testing. I've simply used a for loop to iterate through the string, but this makes it impossible to read the contents of trailing spaces because I'm using the cin command.

[1]:
```
from IPython.display import Image
Image("notebook_image/p1.png")
```

[1]:



To solve this, I use getline to handle the problem and it successfully solved the problem and can read the whole line of input with space.

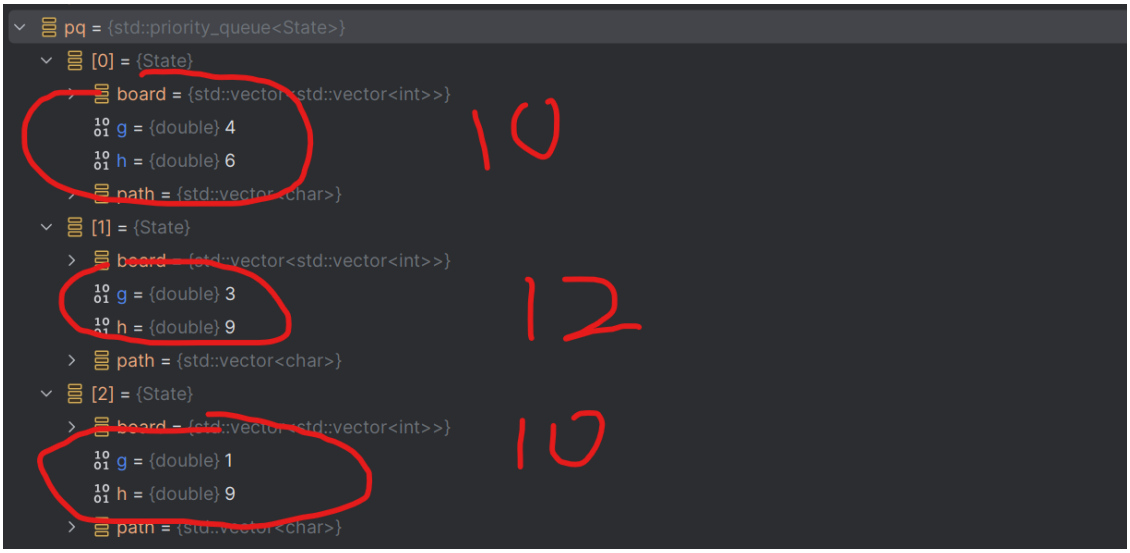
[ ]:
```
string input;
getline(cin, input);
```

2.2 Priorty Queue

During coding, I found that the priority queue does not work properly since it cannot correctly arrange the right state, the state with higher cost will line before low cost as follows.

[2]:
```
Image("notebook_image/p2.png")
```

[2]:



Then it will lead to incorrect path like below: the path are not the optimal

[3]:
```
Image("notebook_image/p5.png")
```

[3]:

```
C:\Users\Quan\GitHub\CUHKSZ-CSC3180\AS_122090007.exe
[[3, 1, 4], [0, 5, 2]]
16
ruldrurdluldrurd

Process finished with exit code 0
```

To solve this problem, I went back to struct, and implemented operator overloading.

[ ]:
```
bool operator<(const State& other_state) const {
        if (g + h == other_state.g + other_state.h) {
            string path1, path2;
            for (auto i: path) {
                path1 += get_priority(i);
            }
            for (auto i: other_state.path) {
                path2 += get_priority(i);
            }
            return path1 > path2;
        } else {
            return (g + h) > (other_state.g + other_state.h);
        }
    }
```

In this way the priority queue can correctly determine the queue of the state and implementing A* search and the path is the optimal.

[4]:
```
Image("notebook_image/p4.png")
```

[4]:
```
C:\Users\Quan\GitHub\CUHKSZ-CSC3180\AS_122090007.exe
[[3, 1, 4], [0, 5, 2]]
16
urdlurrdluldrurd

Process finished with exit code 0
```

2.3 Visted States

During coding, I first used an unordered set to record the state that has been visited to avoid infinite loop for searching, later on I found that this implementation may prematurely mark certain

states as visited, causing shorter paths that should have been explored later to be ignored. By using an unordered set for visited states, some suboptimal paths might be recorded first, preventing the algorithm from updating to a more optimal path when it is discovered later. Below are the wrong steps and path:

[5]:
```
Image("notebook_image/p3.png")
```

[5]:
```
C:\Users\Quan\GitHub\CUHKSZ-CSC3180\AS_122090007.exe
[[3, 1, 4], [0, 5, 2]]
18
ruldrruldrulldrurd

Process finished with exit code 0
```

Instead, using an unordered map to store the g value (cost from the start) allows the algorithm to properly update and prioritize shorter paths when a better route is found. Below are the correct path:

[ ]:
```
for (auto neighbor : get_next_state(current)) {
            string state_str = board_to_str(neighbor.board);
            if (visited.find(state_str) == visited.end() or visited[state_str]␣
 ↪> neighbor.g) {
                visited[state_str] = neighbor.g;
                pq.push(neighbor);
            }
        }
```

[6]:
```
Image("notebook_image/p4.png")
```

[6]:
```
C:\Users\Quan\GitHub\CUHKSZ-CSC3180\AS_122090007.exe
[[3, 1, 4], [0, 5, 2]]
16
urdlurrdluldrurd

Process finished with exit code 0
```