AI assignment 1

Team members:

Ahmed Talaat Nosair (6)
Salma Ragab Gad (31)

Problem statement:

An instance of the 8-puzzle game consists of a board holding 8 distinct movable tiles, plus an empty space. For any such board, the empty space may be legally swapped with any tile horizontally or vertically adjacent to it. In this assignment, the blank space is going to be represented with the number0.
Given an initial state of the board, the search problem is to find a sequence of moves that transitions this state to the goal state; that is, the configuration with all tiles arranged in ascending order 0,1,2,3,4,5,6,7,8.

Our solution:

- Read the input state.
- Reformat the input to use it.
- Use one of the search algorithms (BFS, DFS, A*) to find a path to the goal state.
- Output the path to the goal, its cost, number of nodes expanded, running time and the max. search depth.

Data structure used:

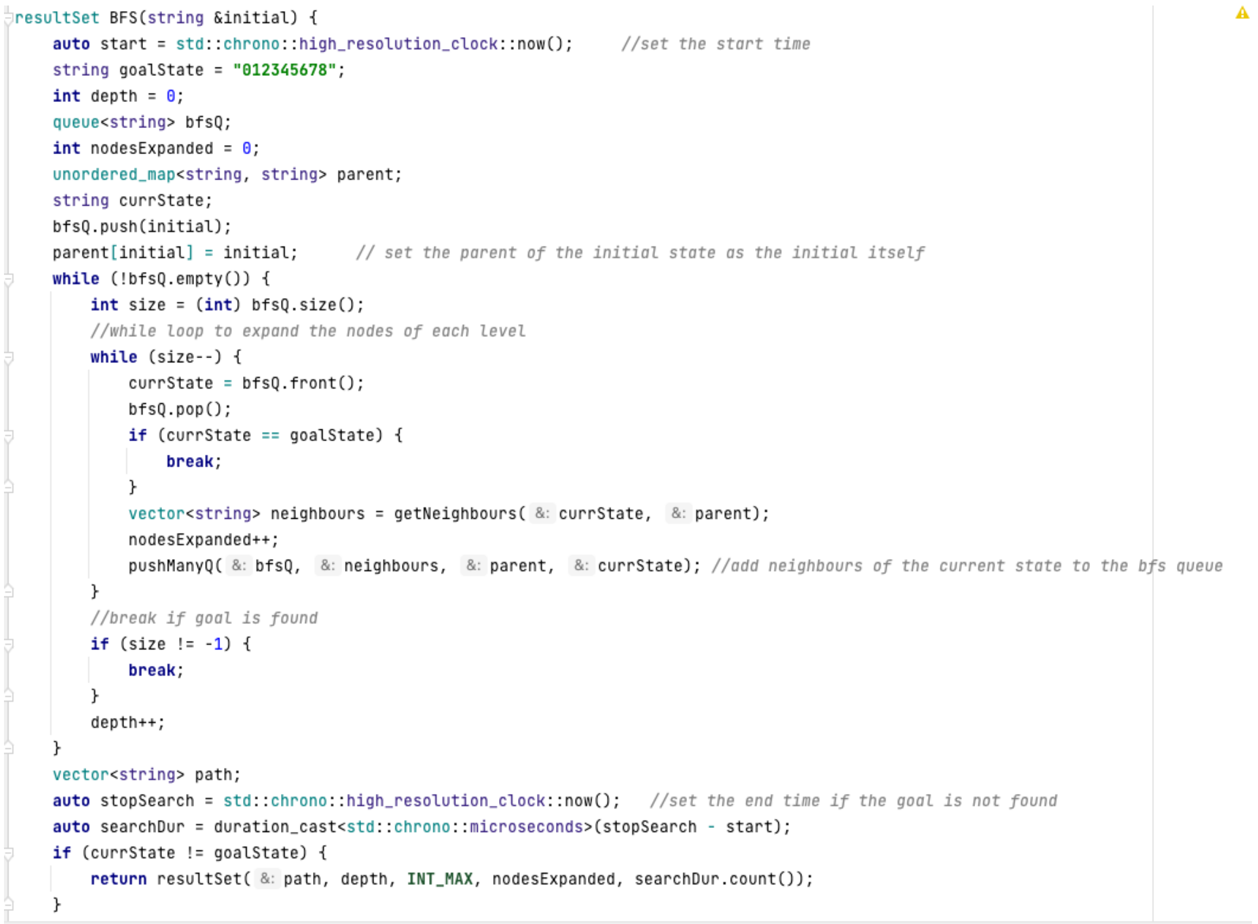
- Queue: to use in BFS search.
- Stack: to use in DFS search.
- Set: to use as priority queue in A* search.
- Unordered map: to associate each node with its parent and find the path.
- Vector of strings: to save the path
- Unordered set: for visited nodes in A* algorithm.

Assumptions:

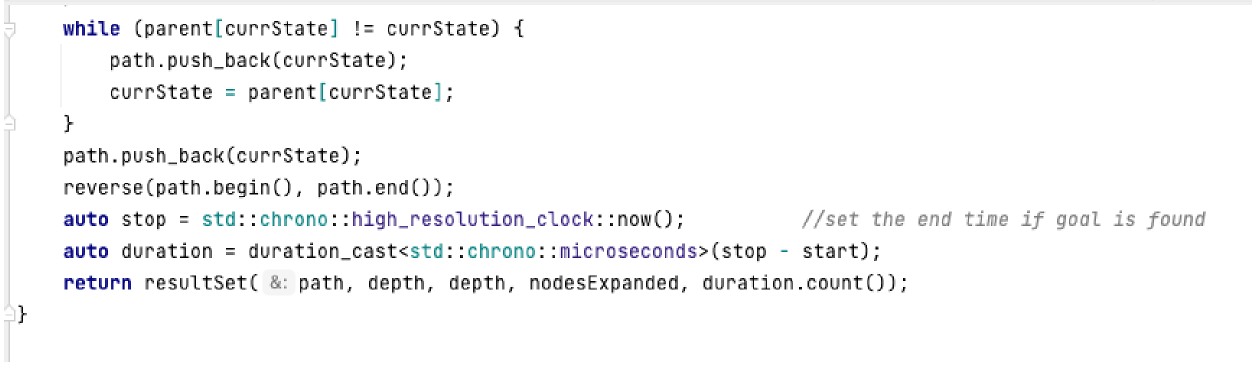
- The input is in the format x1, x2, x3, x4, x5, x6, x7, x8, x9 where x_i is $\in [0,8]$
- When the goal can’t be reached, the cost of path is INT_MAX.

Code snippets:

BFS

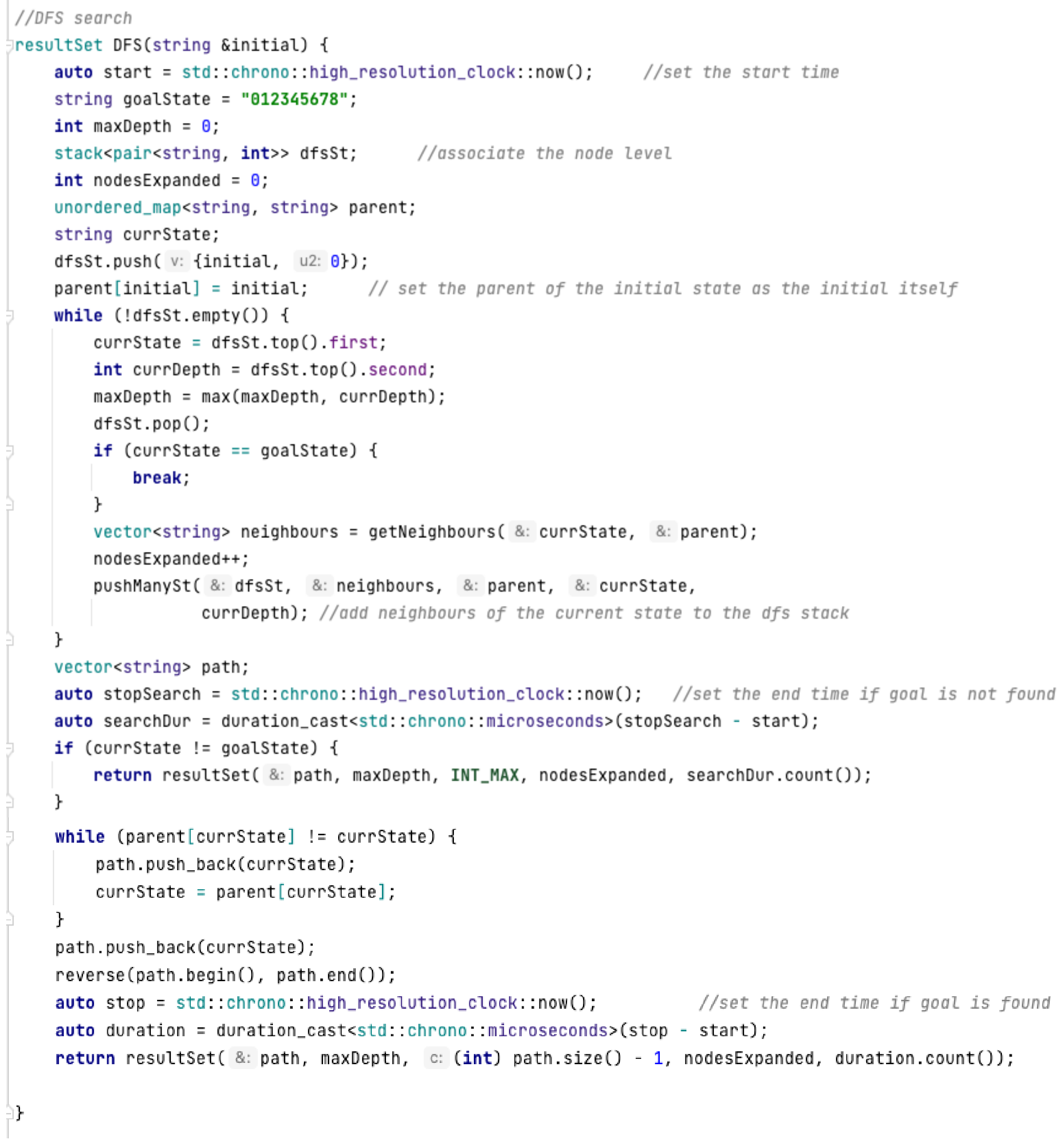


```cpp
resultSet BFS(string &initial) {
    auto start = std::chrono::high_resolution_clock::now();     //set the start time
    string goalState = "012345678";
    int depth = 0;
    queue<string> bfsQ;
    int nodesExpanded = 0;
    unordered_map<string, string> parent;
    string currState;
    bfsQ.push(initial);
    parent[initial] = initial;      // set the parent of the initial state as the initial itself
    while (!bfsQ.empty()) {
        int size = (int) bfsQ.size();
        //while loop to expand the nodes of each level
        while (size--) {
            currState = bfsQ.front();
            bfsQ.pop();
            if (currState == goalState) {
                break;
            }
            vector<string> neighbours = getNeighbours( &: currState,  &: parent);
            nodesExpanded++;
            pushManyQ( &: bfsQ,  &: neighbours,  &: parent,  &: currState); //add neighbours of the current state to the bfs queue
        }
        //break if goal is found
        if (size != -1) {
            break;
        }
        depth++;
    }
    vector<string> path;
    auto stopSearch = std::chrono::high_resolution_clock::now();   //set the end time if the goal is not found
    auto searchDur = duration_cast<std::chrono::microseconds>(stopSearch - start);
    if (currState != goalState) {
        return resultSet( &: path, depth, INT_MAX, nodesExpanded, searchDur.count());
    }
```



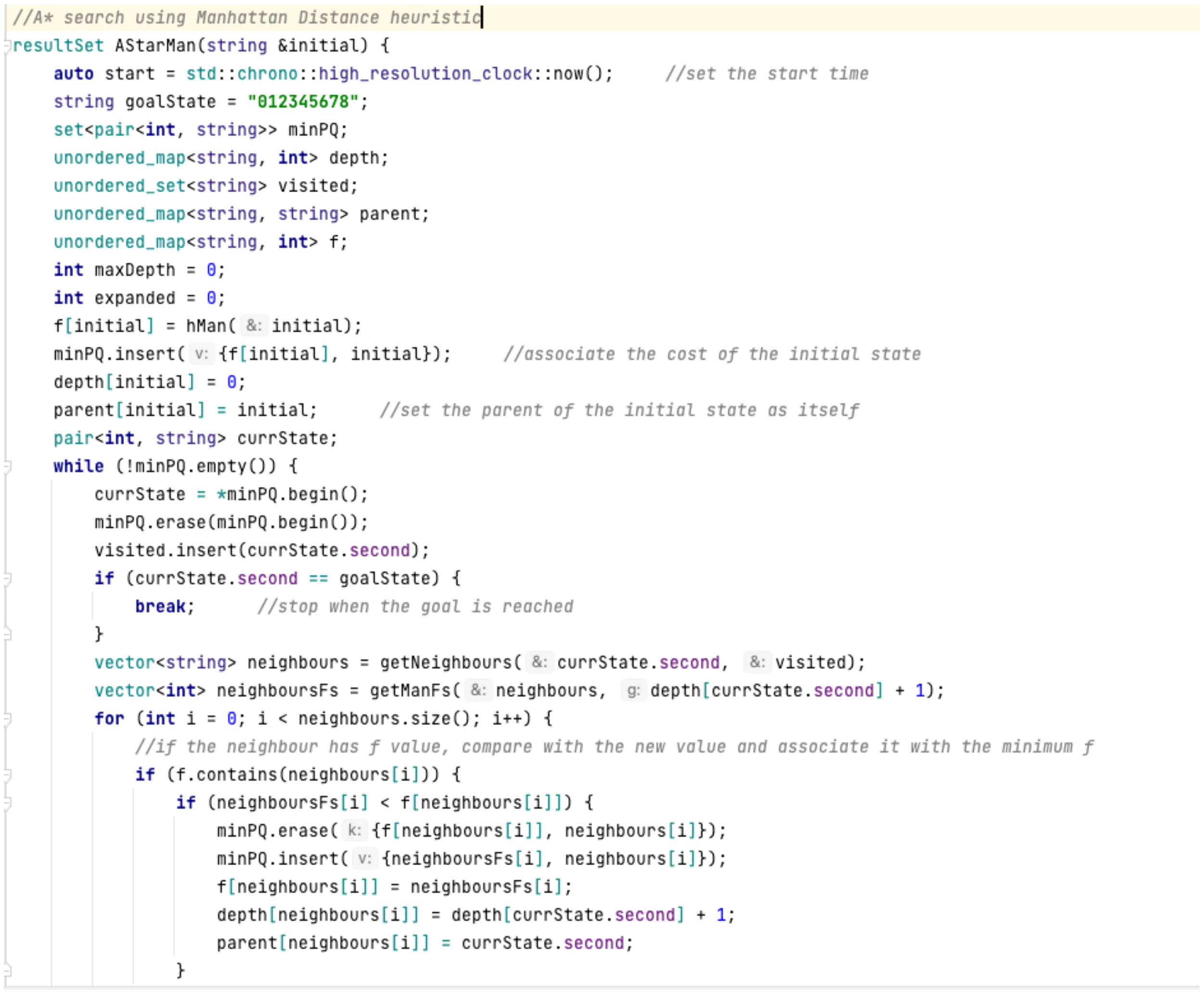
```cpp
    while (parent[currState] != currState) {
        path.push_back(currState);
        currState = parent[currState];
    }
    path.push_back(currState);
    reverse(path.begin(), path.end());
    auto stop = std::chrono::high_resolution_clock::now();          //set the end time if goal is found
    auto duration = duration_cast<std::chrono::microseconds>(stop - start);
    return resultSet( &: path, depth, depth, nodesExpanded, duration.count());
}
```

DFS

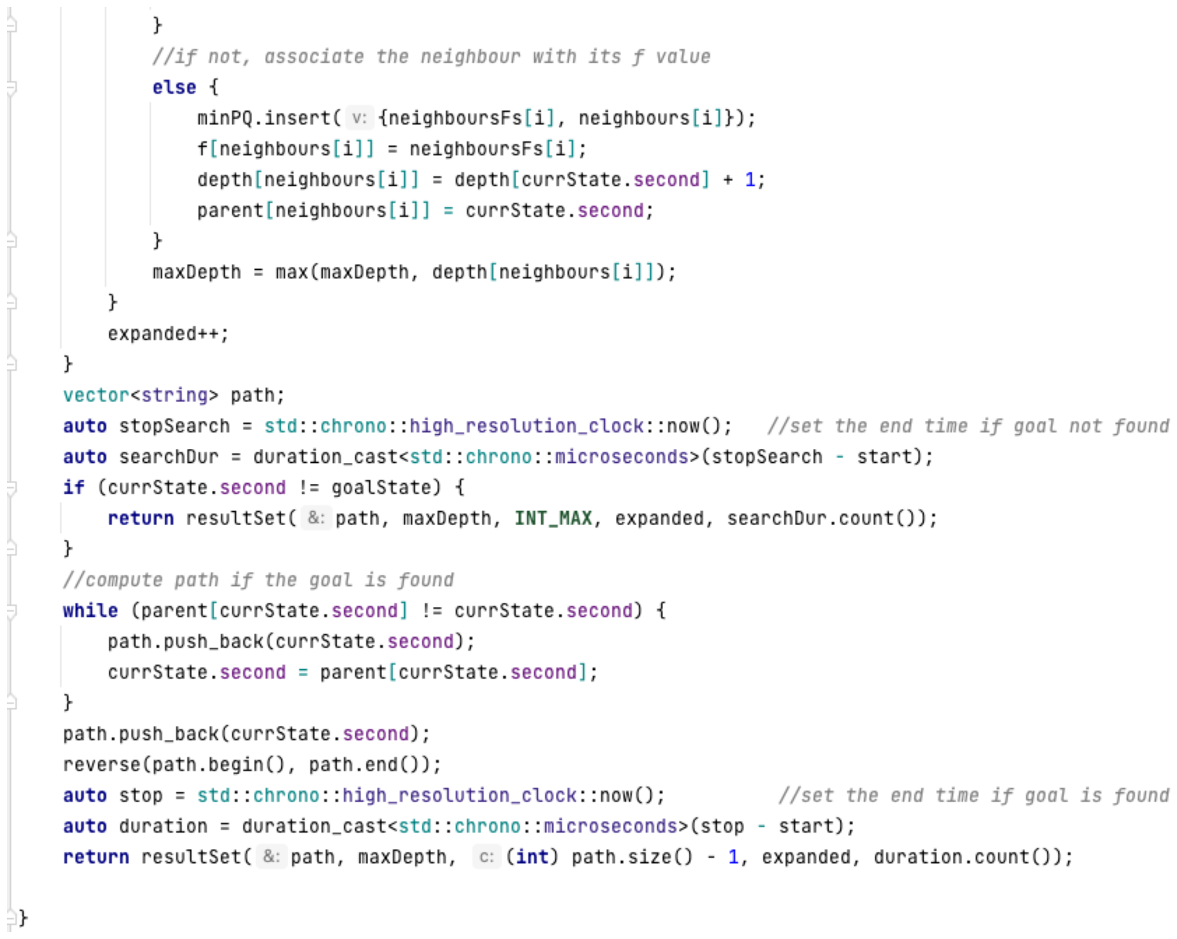


```
//DFS search
resultSet DFS(string &initial) {
    auto start = std::chrono::high_resolution_clock::now();     //set the start time
    string goalState = "012345678";
    int maxDepth = 0;
    stack<pair<string, int>> dfsSt;      //associate the node level
    int nodesExpanded = 0;
    unordered_map<string, string> parent;
    string currState;
    dfsSt.push( v: {initial,  u2: 0});
    parent[initial] = initial;      // set the parent of the initial state as the initial itself
    while (!dfsSt.empty()) {
        currState = dfsSt.top().first;
        int currDepth = dfsSt.top().second;
        maxDepth = max(maxDepth, currDepth);
        dfsSt.pop();
        if (currState == goalState) {
            break;
        }
        vector<string> neighbours = getNeighbours( &: currState,  &: parent);
        nodesExpanded++;
        pushManySt( &: dfsSt,  &: neighbours,  &: parent,  &: currState,
                   currDepth); //add neighbours of the current state to the dfs stack
    }
    vector<string> path;
    auto stopSearch = std::chrono::high_resolution_clock::now();   //set the end time if goal is not found
    auto searchDur = duration_cast<std::chrono::microseconds>(stopSearch - start);
    if (currState != goalState) {
        return resultSet( &: path, maxDepth, INT_MAX, nodesExpanded, searchDur.count());
    }
    while (parent[currState] != currState) {
        path.push_back(currState);
        currState = parent[currState];
    }
    path.push_back(currState);
    reverse(path.begin(), path.end());
    auto stop = std::chrono::high_resolution_clock::now();          //set the end time if goal is found
    auto duration = duration_cast<std::chrono::microseconds>(stop - start);
    return resultSet( &: path, maxDepth,  c: (int) path.size() - 1, nodesExpanded, duration.count());

}
```

A*



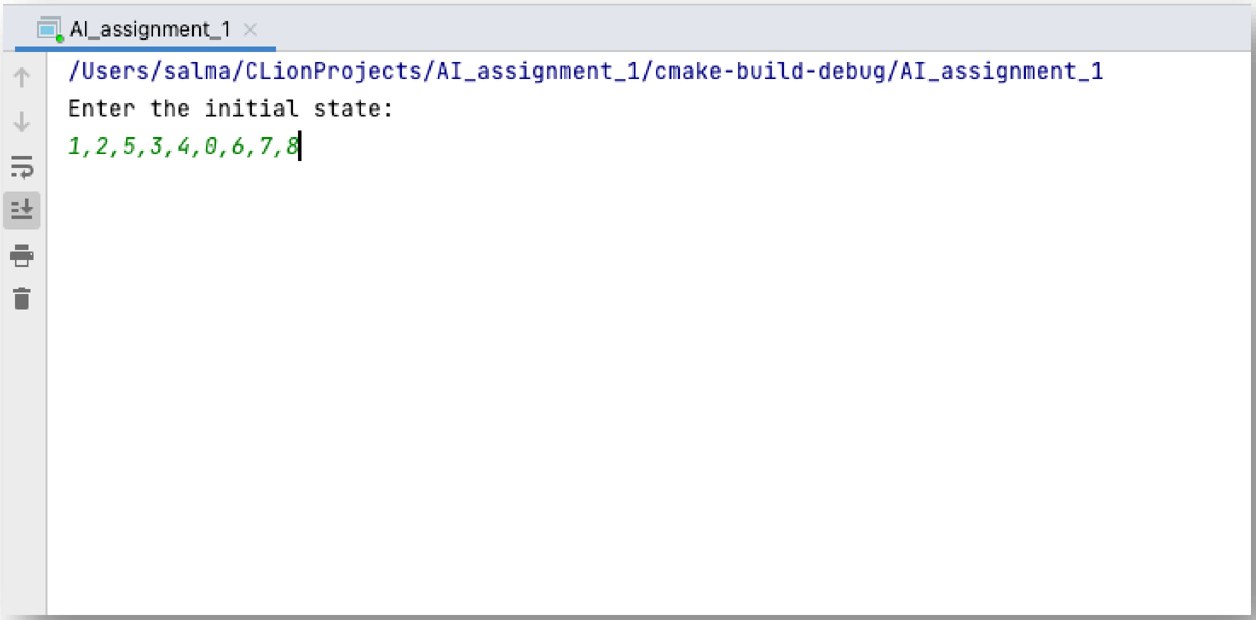
```
//A* search using Manhattan Distance heuristic
resultSet AStarMan(string &initial) {
    auto start = std::chrono::high_resolution_clock::now();     //set the start time
    string goalState = "012345678";
    set<pair<int, string>> minPQ;
    unordered_map<string, int> depth;
    unordered_set<string> visited;
    unordered_map<string, string> parent;
    unordered_map<string, int> f;
    int maxDepth = 0;
    int expanded = 0;
    f[initial] = hMan( &: initial);
    minPQ.insert( v: {f[initial], initial});     //associate the cost of the initial state
    depth[initial] = 0;
    parent[initial] = initial;      //set the parent of the initial state as itself
    pair<int, string> currState;
    while (!minPQ.empty()) {
        currState = *minPQ.begin();
        minPQ.erase(minPQ.begin());
        visited.insert(currState.second);
        if (currState.second == goalState) {
            break;      //stop when the goal is reached
        }
        vector<string> neighbours = getNeighbours( &: currState.second,  &: visited);
        vector<int> neighboursFs = getManFs( &: neighbours,  g: depth[currState.second] + 1);
        for (int i = 0; i < neighbours.size(); i++) {
            //if the neighbour has f value, compare with the new value and associate it with the minimum f
            if (f.contains(neighbours[i])) {
                if (neighboursFs[i] < f[neighbours[i]]) {
                    minPQ.erase( k: {f[neighbours[i]], neighbours[i]});
                    minPQ.insert( v: {neighboursFs[i], neighbours[i]});
                    f[neighbours[i]] = neighboursFs[i];
                    depth[neighbours[i]] = depth[currState.second] + 1;
                    parent[neighbours[i]] = currState.second;
                }
```



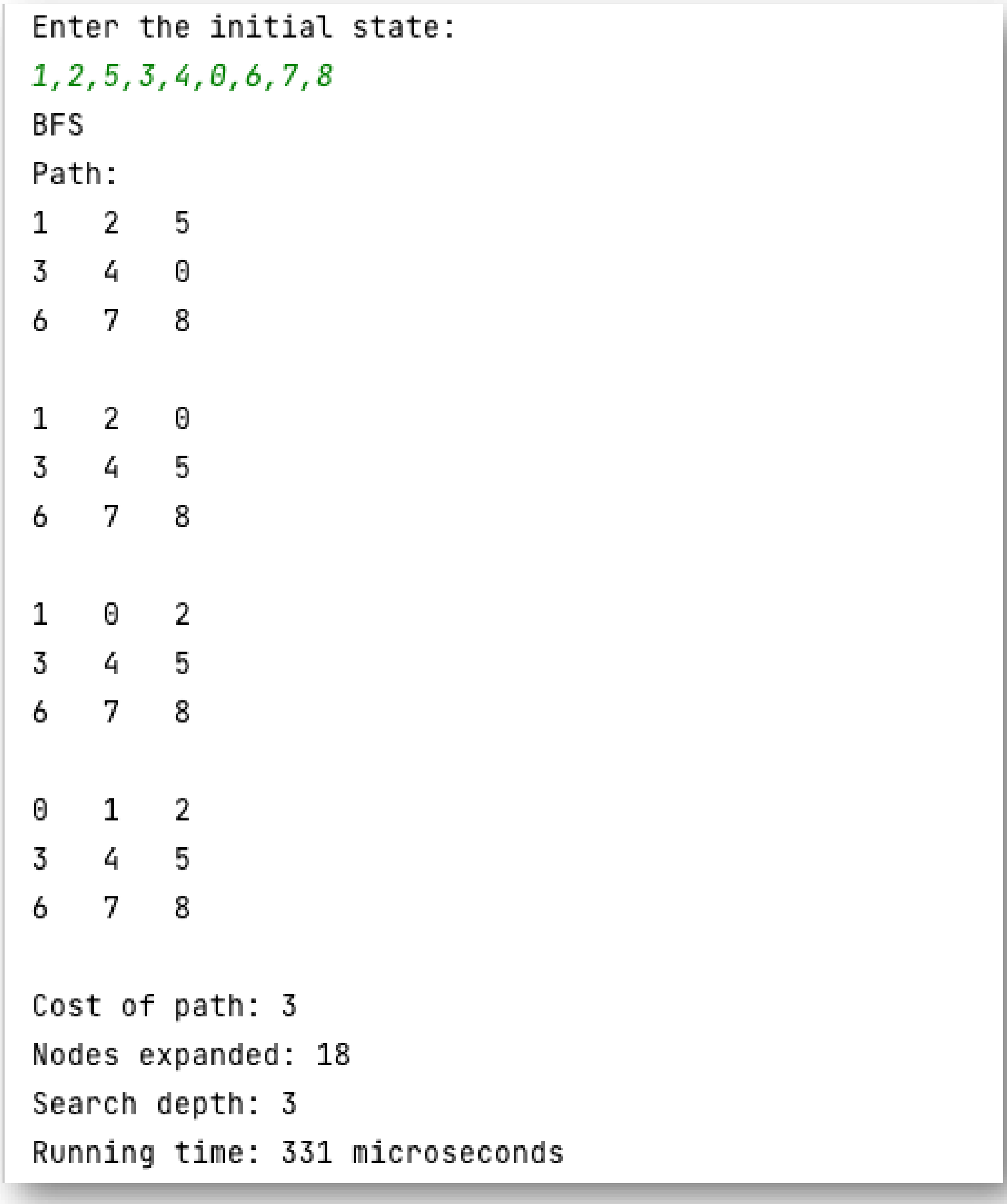
```
            }
            //if not, associate the neighbour with its f value
            else {
                minPQ.insert( v: {neighboursFs[i], neighbours[i]});
                f[neighbours[i]] = neighboursFs[i];
                depth[neighbours[i]] = depth[currState.second] + 1;
                parent[neighbours[i]] = currState.second;
            }
            maxDepth = max(maxDepth, depth[neighbours[i]]);
        }
        expanded++;
    }
    vector<string> path;
    auto stopSearch = std::chrono::high_resolution_clock::now();   //set the end time if goal not found
    auto searchDur = duration_cast<std::chrono::microseconds>(stopSearch - start);
    if (currState.second != goalState) {
        return resultSet( &: path, maxDepth, INT_MAX, expanded, searchDur.count());
    }
    //compute path if the goal is found
    while (parent[currState.second] != currState.second) {
        path.push_back(currState.second);
        currState.second = parent[currState.second];
    }
    path.push_back(currState.second);
    reverse(path.begin(), path.end());
    auto stop = std::chrono::high_resolution_clock::now();          //set the end time if goal is found
    auto duration = duration_cast<std::chrono::microseconds>(stop - start);
    return resultSet( &: path, maxDepth,  c: (int) path.size() - 1, expanded, duration.count());

}
```

Sample runs:

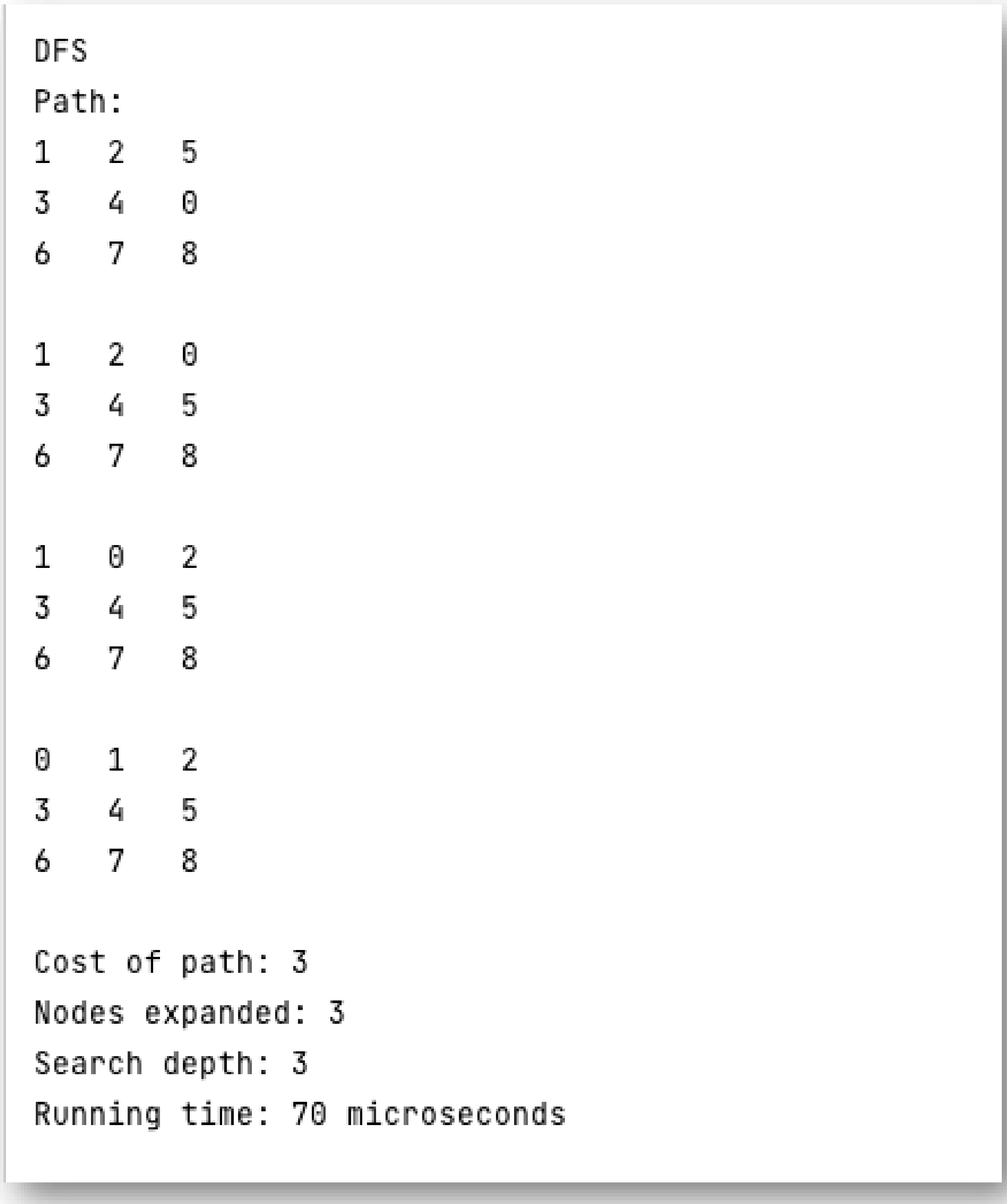
1) 1,2,5,3,4,0,6,7,8

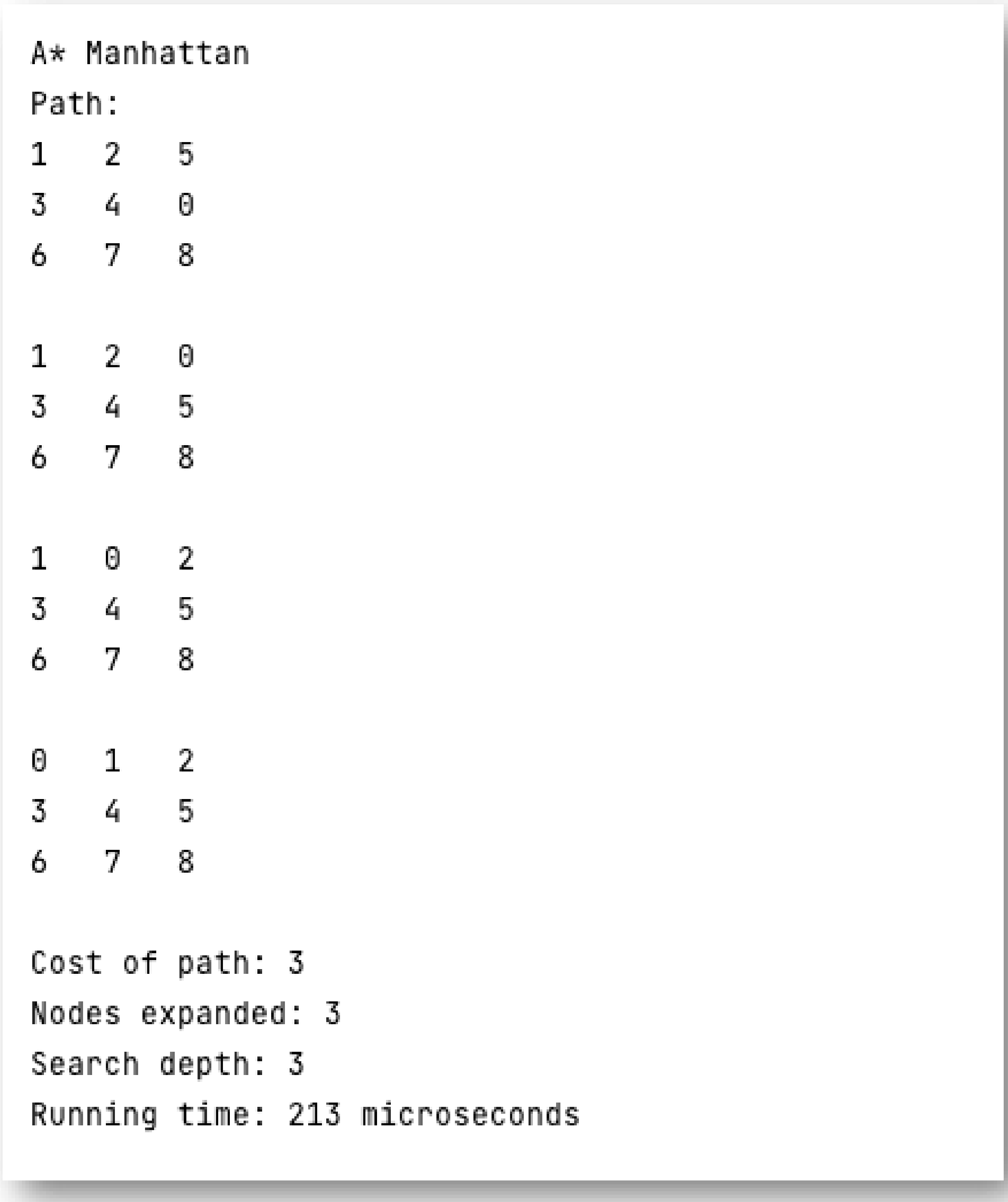


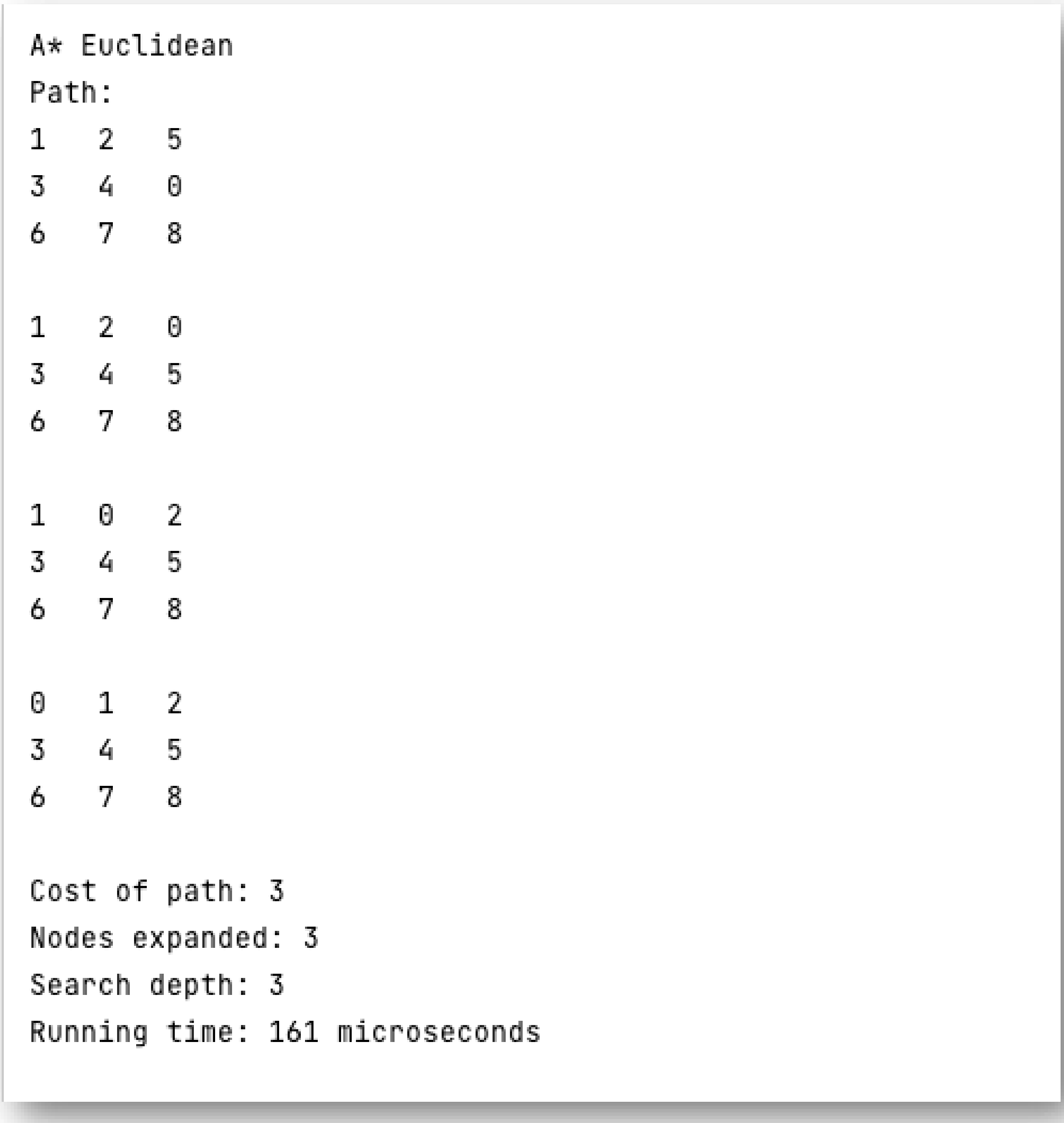
```
/Users/salma/CLionProjects/AI_assignment_1/cmake-build-debug/AI_assignment_1
Enter the initial state:
1,2,5,3,4,0,6,7,8
```



```
Enter the initial state:
1,2,5,3,4,0,6,7,8
BFS
Path:
1   2   5
3   4   0
6   7   8

1   2   0
3   4   5
6   7   8

1   0   2
3   4   5
6   7   8

0   1   2
3   4   5
6   7   8

Cost of path: 3
Nodes expanded: 18
Search depth: 3
Running time: 331 microseconds
```

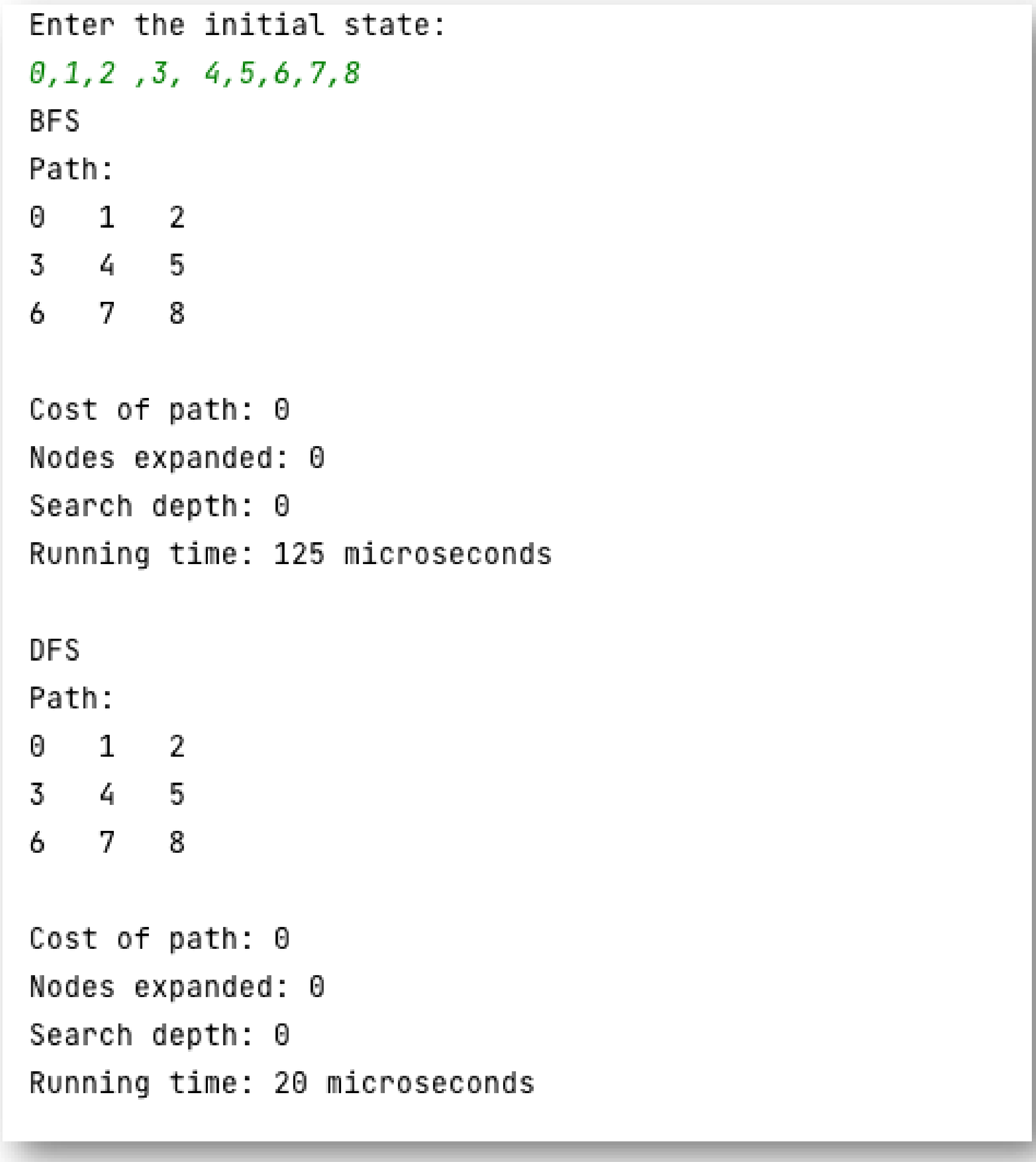

```
DFS
Path:
1   2   5
3   4   0
6   7   8

1   2   0
3   4   5
6   7   8

1   0   2
3   4   5
6   7   8

0   1   2
3   4   5
6   7   8

Cost of path: 3
Nodes expanded: 3
Search depth: 3
Running time: 70 microseconds
```


```
A* Manhattan
Path:
1   2   5
3   4   0
6   7   8

1   2   0
3   4   5
6   7   8

1   0   2
3   4   5
6   7   8

0   1   2
3   4   5
6   7   8

Cost of path: 3
Nodes expanded: 3
Search depth: 3
Running time: 213 microseconds
```

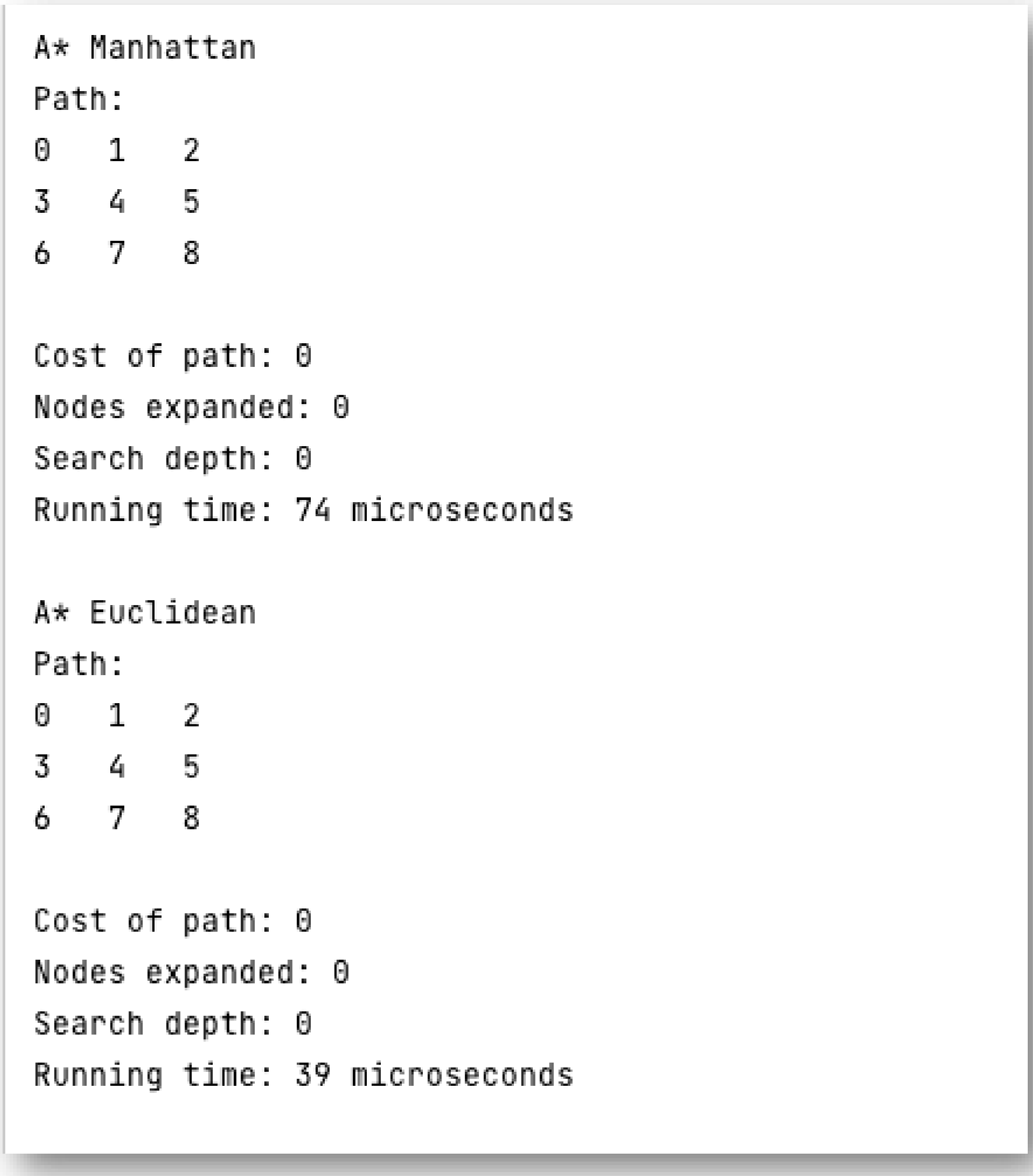


```
A* Euclidean
Path:
1   2   5
3   4   0
6   7   8

1   2   0
3   4   5
6   7   8

1   0   2
3   4   5
6   7   8

0   1   2
3   4   5
6   7   8

Cost of path: 3
Nodes expanded: 3
Search depth: 3
Running time: 161 microseconds
```

2) When the initial is the goal state



```
Enter the initial state:
0,1,2 ,3, 4,5,6,7,8
BFS
Path:
0   1   2
3   4   5
6   7   8

Cost of path: 0
Nodes expanded: 0
Search depth: 0
Running time: 125 microseconds

DFS
Path:
0   1   2
3   4   5
6   7   8

Cost of path: 0
Nodes expanded: 0
Search depth: 0
Running time: 20 microseconds
```

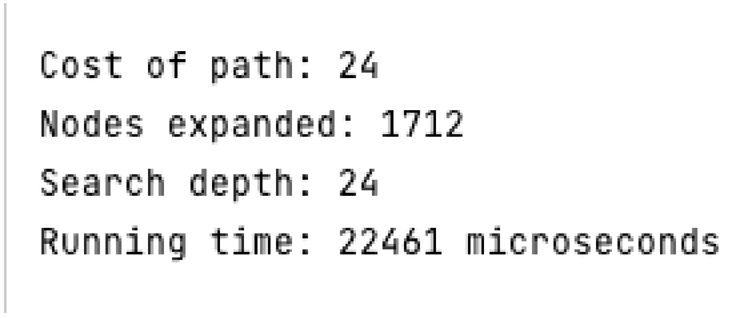


```
A* Manhattan
Path:
0   1   2
3   4   5
6   7   8

Cost of path: 0
Nodes expanded: 0
Search depth: 0
Running time: 74 microseconds

A* Euclidean
Path:
0   1   2
3   4   5
6   7   8

Cost of path: 0
Nodes expanded: 0
Search depth: 0
Running time: 39 microseconds
```

A* heuristics comparison:

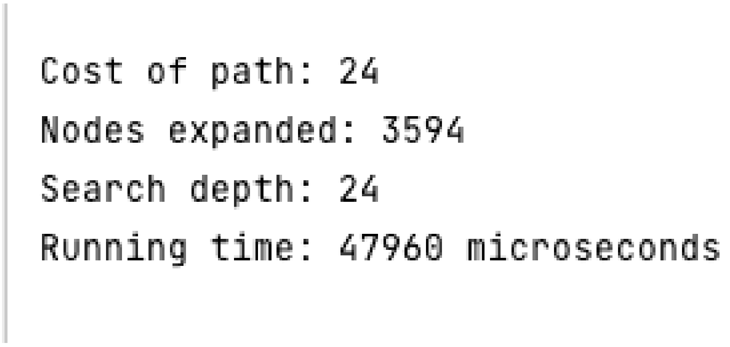
Input: 0,2,3,1,7,6,5,4,8

Manhattan:



```
Cost of path: 24
Nodes expanded: 1712
Search depth: 24
Running time: 22461 microseconds
```

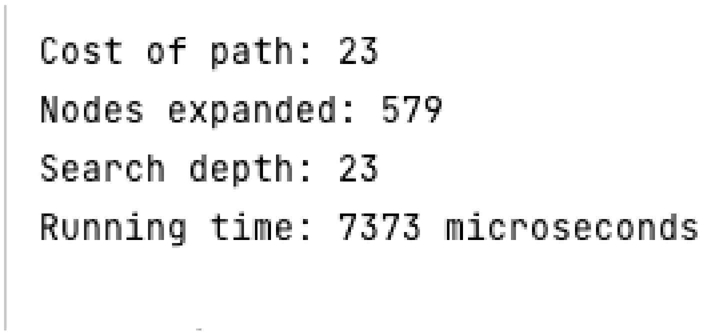
Euclidean:



```
Cost of path: 24
Nodes expanded: 3594
Search depth: 24
Running time: 47960 microseconds
```

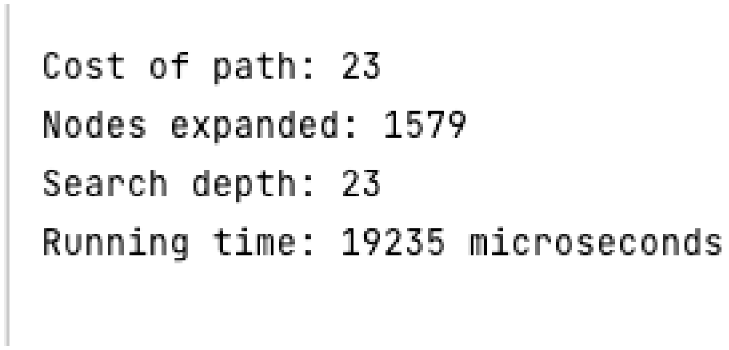
Input: 1,7,3,0,8,6,5,4,2

Manhattan:



```
Cost of path: 23
Nodes expanded: 579
Search depth: 23
Running time: 7373 microseconds
```

Euclidean:



```
Cost of path: 23
Nodes expanded: 1579
Search depth: 23
Running time: 19235 microseconds
```

** We can find that they have the same path but Manhattan has less nodes expanded and less running time than Euclidean heuristic.