MingkuanPang

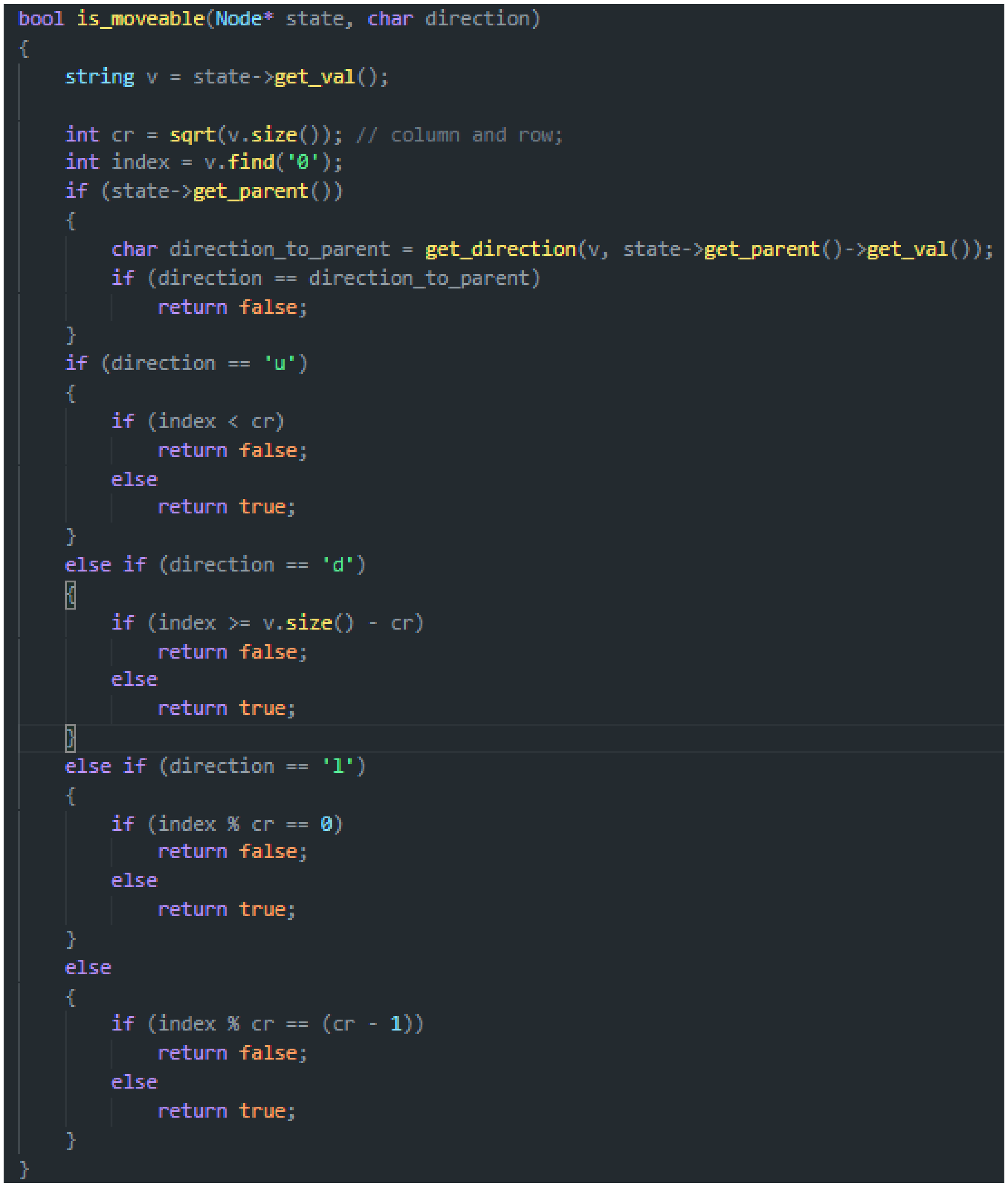
Github link:

https://github.com/2FLing/study/tree/main/CSCI-164(intro%20AI)

Code Overview:

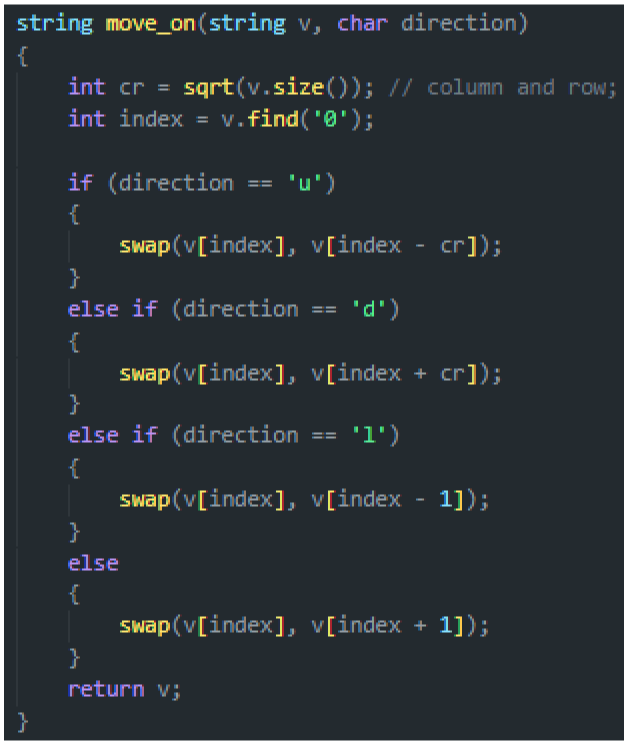
The node expands the way decided by the direction sequences. Before expanding, it will check if the direction is legal to move. Because 0 cannot cross the boundary. Also, it cannot return to its original state, otherwise it will cause an infinite loop. That is how I check if the current state can move to the current direction.

First, it will check if the current direction brings the current state back to its original state, if it will then return false. If it does not, it will check if the current direction makes the ‘0’ cross the boundary. If it will not, then the current state can move on by the current direction.



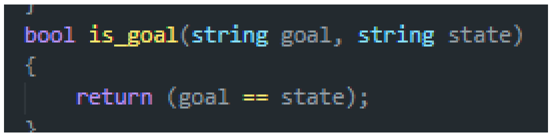
```
bool is_moveable(Node* state, char direction)
{
    string v = state->get_val();

    int cr = sqrt(v.size()); // column and row;
    int index = v.find('0');
    if (state->get_parent())
    {
        char direction_to_parent = get_direction(v, state->get_parent()->get_val());
        if (direction == direction_to_parent)
            return false;
    }
    if (direction == 'u')
    {
        if (index < cr)
            return false;
        else
            return true;
    }
    else if (direction == 'd')
    {
        if (index >= v.size() - cr)
            return false;
        else
            return true;
    }
    else if (direction == 'l')
    {
        if (index % cr == 0)
            return false;
        else
            return true;
    }
    else
    {
        if (index % cr == (cr - 1))
            return false;
        else
            return true;
    }
}
```

The move on function just simply switches the position of ‘0’ to the corresponding position.



```
string move_on(string v, char direction)
{
    int cr = sqrt(v.size()); // column and row;
    int index = v.find('0');

    if (direction == 'u')
    {
        swap(v[index], v[index - cr]);
    }
    else if (direction == 'd')
    {
        swap(v[index], v[index + cr]);
    }
    else if (direction == 'l')
    {
        swap(v[index], v[index - 1]);
    }
    else
    {
        swap(v[index], v[index + 1]);
    }
    return v;
}
```

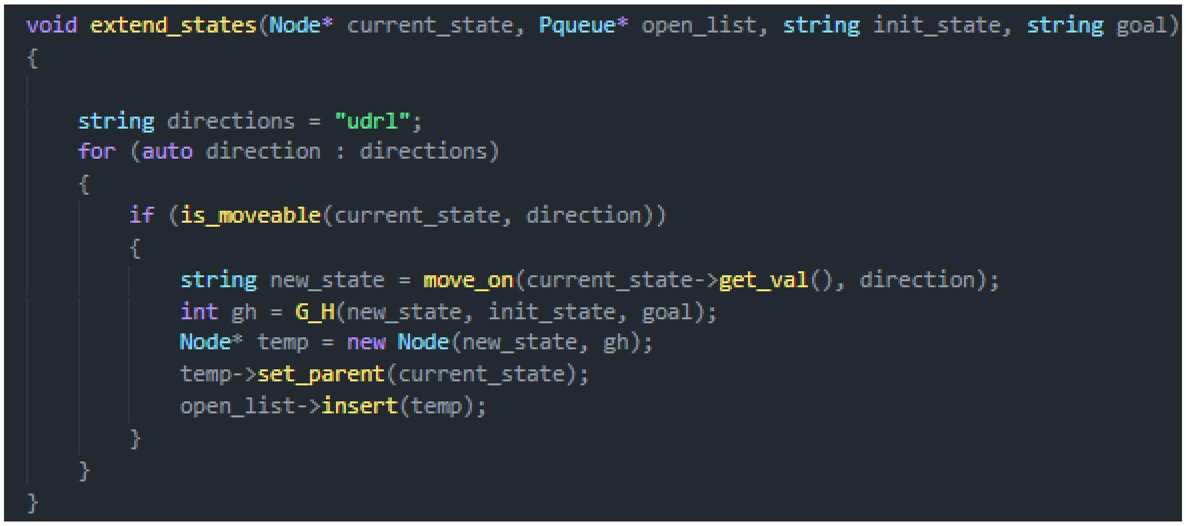
After moving on, it will check if the new state is the goal



```
bool is_goal(string goal, string state)
{
    return (goal == state);
```

If it is not goal yet, then it will keep expanding the current node.

For extending states, first it will read the direction sequences, and then decide if the direction is a legal direction, if it is, then expand the node according to the direction. Then set the parent node of the next node to the current node, then put the new node into open list.



```
void extend_states(Node* current_state, Pqueue* open_list, string init_state, string goal)
{

    string directions = "udrl";
    for (auto direction : directions)
    {
        if (is_moveable(current_state, direction))
        {
            string new_state = move_on(current_state->get_val(), direction);
            int gh = G_H(new_state, init_state, goal);
            Node* temp = new Node(new_state, gh);
            temp->set_parent(current_state);
            open_list->insert(temp);
        }
    }
}
```

To solve the puzzle, it will ask the user to input an 8 size or 15 size of puzzle as an initial state.

Then push the initial state into the open list. First, pop out the node that ready to be expanded from open list. Then check if the popped node is NULL, if it is, that means the open list is empty and there is no way to solve the puzzle. If the node is not NULL, then check if the node is the goal, if it is goal, then it will return a pair with string as its first value, and int as its second value. String will be the path for the initial state to the goal state. Int will be the number of nodes that be expanded. If it is not goal, then it will check if the current node has been expanded before, if it is then getting the next node from open list, if it is not then expanding the current node then push it into close list after being expanded. Keep the process above until the current node is goal.

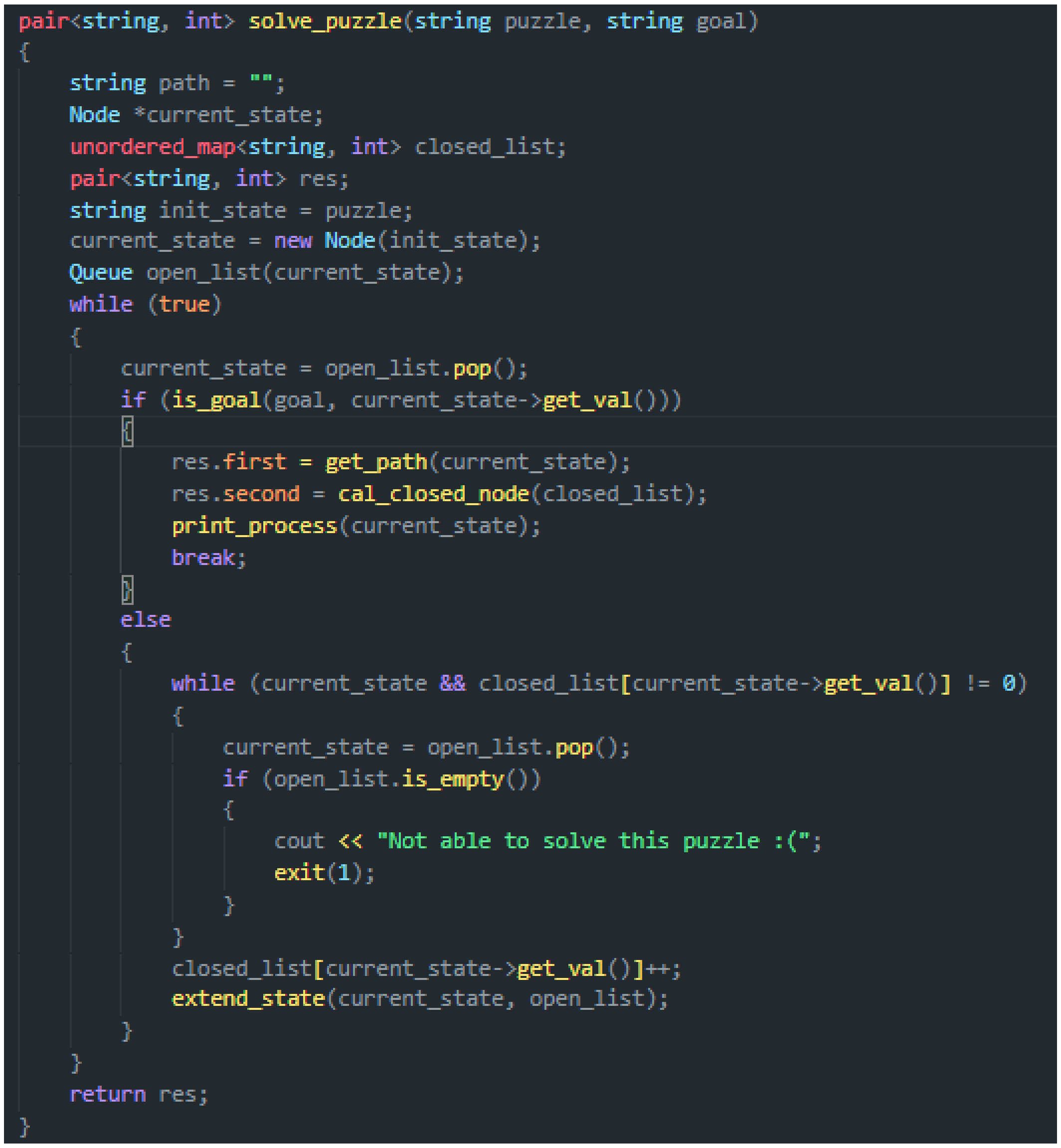
Outputs:

I have the outputs commented on the end of each cpp file.

## For Breadth-First Search

The open list will be queue, which is FIFO. The node come first will be first to expanded.

And it will keep expanding the nodes, until the open list is empty or the current node is the solution.



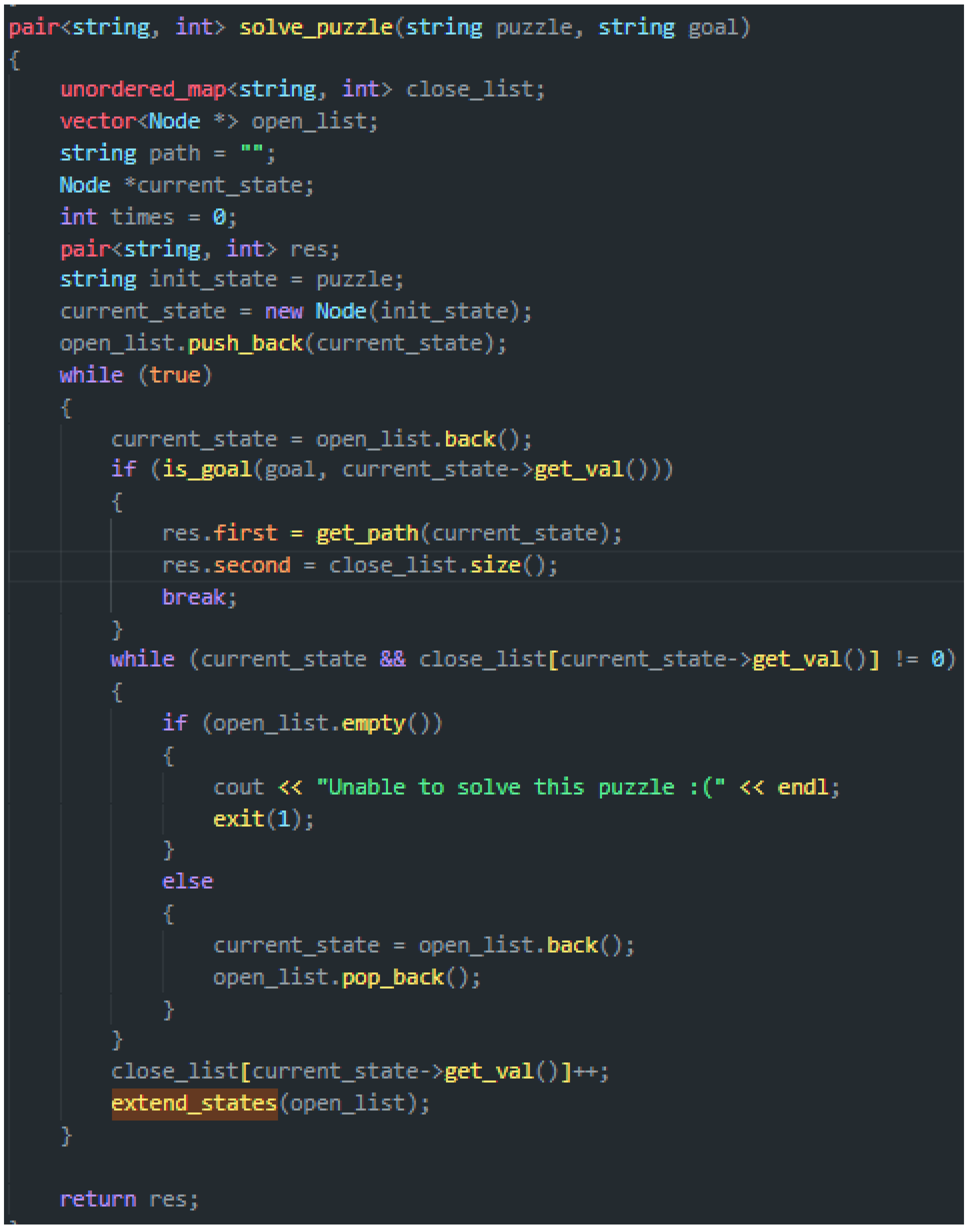
```cpp
pair<string, int> solve_puzzle(string puzzle, string goal)
{
    string path = "";
    Node *current_state;
    unordered_map<string, int> closed_list;
    pair<string, int> res;
    string init_state = puzzle;
    current_state = new Node(init_state);
    Queue open_list(current_state);
    while (true)
    {
        current_state = open_list.pop();
        if (is_goal(goal, current_state->get_val()))
        {
            res.first = get_path(current_state);
            res.second = cal_closed_node(closed_list);
            print_process(current_state);
            break;
        }
        else
        {
            while (current_state && closed_list[current_state->get_val()] != 0)
            {
                current_state = open_list.pop();
                if (open_list.is_empty())
                {
                    cout << "Not able to solve this puzzle :(";
                    exit(1);
                }
            }
            closed_list[current_state->get_val()]++;
            extend_state(current_state, open_list);
        }
    }
    return res;
}
```

Results:

By using Breadth First Search, it takes very long time to solve some of the problems, so I left blank for the answer on those problems. Maybe there is something I did wrong.

Depth First Search:

For the Depth First Search, the type of the open list will be an array, but it adds and pops the elements from its back. That is LIFO.



```
pair<string, int> solve_puzzle(string puzzle, string goal)
{
    unordered_map<string, int> close_list;
    vector<Node *> open_list;
    string path = "";
    Node *current_state;
    int times = 0;
    pair<string, int> res;
    string init_state = puzzle;
    current_state = new Node(init_state);
    open_list.push_back(current_state);
    while (true)
    {
        current_state = open_list.back();
        if (is_goal(goal, current_state->get_val()))
        {
            res.first = get_path(current_state);
            res.second = close_list.size();
            break;
        }
        while (current_state && close_list[current_state->get_val()] != 0)
        {
            if (open_list.empty())
            {
                cout << "Unable to solve this puzzle :(" << endl;
                exit(1);
            }
            else
            {
                current_state = open_list.back();
                open_list.pop_back();
            }
        }
        close_list[current_state->get_val()]++;
        extend_states(open_list);
    }

    return res;
```

Results:

For the Depth First Search, it is even worst than Breadth First Search, it seems like takes forever to get the solution of the problem. So, I left blank for the unsolved problem.

## Iterative-Deepening Depth-First Search

For Iterative-Deepening Depth-First Search, unlike normal DFS, there is a limit for IDDFS. It follows normal DFS process until the limit is reached, and then it goes back and expands the other nodes that are unexpanded yet under the limit.

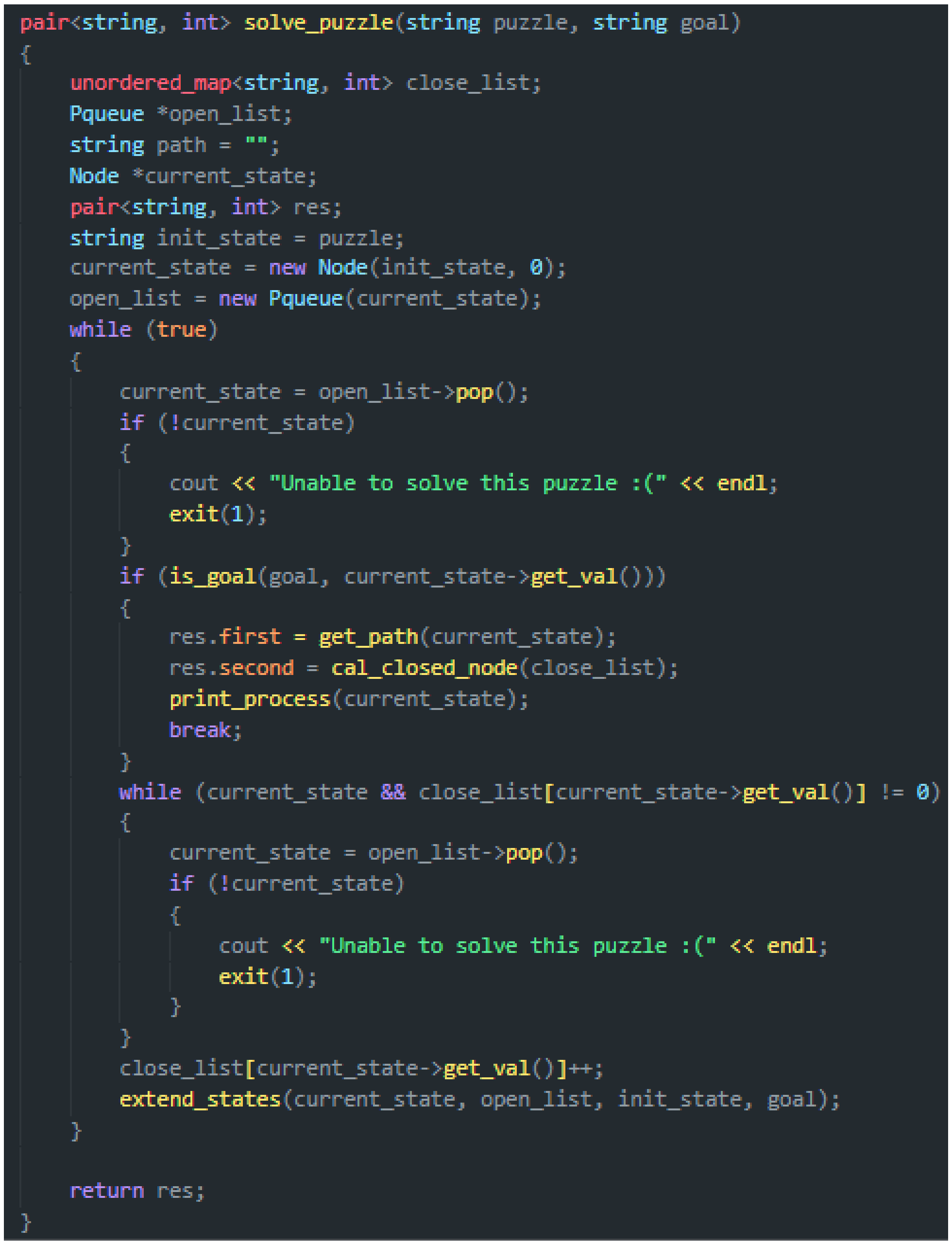
```
pair<string, int> solve_puzzle(string puzzle, string goal)
{
    int limit = 0;
    string init_state, goal_state;
    unordered_map<string, int> close_list;
    vector<Node *> open_list;
    string path = "";
    Node *current_state;
    pair<string, int> res;
    init_state = puzzle;
    goal_state = goal;
    current_state = new Node(init_state);
    open_list.push_back(current_state);
    while (true)
    {
        current_state = open_list.back();
        if (is_goal(goal_state, current_state->get_val()))
        {
            res.first = get_path(current_state);
            res.second = cal_closed_node(close_list);
            print_process(current_state);
            break;
        }
        else
        {
            if (current_state->get_depth() == limit)
            {
                open_list.pop_back();
                if (open_list.empty())
                {
                    Node *node = new Node(init_state);
                    current_state = node;
                    open_list.push_back(current_state);
                    limit++;
                    close_list.clear();
                }
            }
            while (current_state && close_list[current_state->get_val()] != 0)
            {
                if (!open_list.empty())
                {
                    open_list.pop_back();
                    if (open_list.empty())
                    {

                        Node *node = new Node(init_state);
                        current_state = node;
                        open_list.push_back(current_state);
                        limit++;
                        close_list.clear();
                    }
                    current_state = open_list.back();
                }
            }
            if (current_state->get_depth() < limit)
            {
                close_list[current_state->get_val()]++;
                extend_states(open_list);
            }
        }
    }

    return res;
}
```

Results:

IDDFS is much better than the regular DFS, but for those 15-puzzle difficult one, it still seems like takes forever to solve.

A* w/ Out-Of-Place, and Manhattan Distance Heuristics

For AMDH, the type of open list will be Pqueue, it stores the elements according to its priority. Whoever has the lower Manhattan distance (means more closer to the goal) will has the higher priority. The element whoever has the highest priority will be pop out first.



```
pair<string, int> solve_puzzle(string puzzle, string goal)
{
    unordered_map<string, int> close_list;
    Pqueue *open_list;
    string path = "";
    Node *current_state;
    pair<string, int> res;
    string init_state = puzzle;
    current_state = new Node(init_state, 0);
    open_list = new Pqueue(current_state);
    while (true)
    {
        current_state = open_list->pop();
        if (!current_state)
        {
            cout << "Unable to solve this puzzle :(" << endl;
            exit(1);
        }
        if (is_goal(goal, current_state->get_val()))
        {
            res.first = get_path(current_state);
            res.second = cal_closed_node(close_list);
            print_process(current_state);
            break;
        }
        while (current_state && close_list[current_state->get_val()] != 0)
        {
            current_state = open_list->pop();
            if (!current_state)
            {
                cout << "Unable to solve this puzzle :(" << endl;
                exit(1);
            }
        }
        close_list[current_state->get_val()]++;
        extend_states(current_state, open_list, init_state, goal);
    }

    return res;
}
```

Results:

That is almost the best one, but still seems takes forever to solve the last two difficult problems.

## Iterative Deepening A* w/ Out-Of-Place, and Manhattan Distance Heuristics

For Iterative-Deepening A* w/ Out-Of-Place, and Manhattan Distance Heuristics, unlike normal A* w/ Out-Of-Place, and Manhattan Distance Heuristics, there is a limit for Iterative-Deepening A* w/ Out-Of-Place, and Manhattan Distance Heuristics. It follows normal A* w/ Out-Of-Place, and Manhattan Distance Heuristics process until the limit is reached, and then it goes back and expands the other nodes that are unexpanded yet under the limit.

```
pair<string, int> solve_puzzle(string puzzle, string goal)
{
    unordered_map<string, int> close_list;
    Pqueue *open_list;
    string path = "";
    Node *current_state;
    pair<string, int> res;
    string init_state = puzzle;
    int depth_limit = 0;
    current_state = new Node(init_state, 0);
    open_list = new Pqueue(current_state);
    while (true)
    {
        current_state = open_list->pop();
        if (!current_state)
        {
            cout << "Unable to solve this puzzle :(" << endl;
            exit(1);
        }
        if (is_goal(goal, current_state->get_val()))
        {
            res.first = get_path(current_state);
            res.second = cal_closed_node(close_list);
            break;
        }
        while (current_state && close_list[current_state->get_val()] != 0)
        {
            current_state = open_list->pop();
        }
        if (!current_state)
        {
            cout << "Unable to solve this puzzle :(" << endl;
            exit(1);
        }
        close_list[current_state->get_val()]++;
        if (current_state->get_depth() > depth_limit)
        {
            open_list->clear();
            depth_limit++;
            close_list.clear();
            while (current_state->get_parent())
            {
                current_state = current_state->get_parent();
            }
            open_list->insert(current_state);
        }
        else
        {
            extend_states(current_state, open_list, init_state, goal);
        }
    }

    return res;
}
```

Results:

The output is the same as the regular A* w/ Out-Of-Place, and Manhattan Distance Heuristics.