Kyle A. Destura
BSCS 3-C

Machine Problem

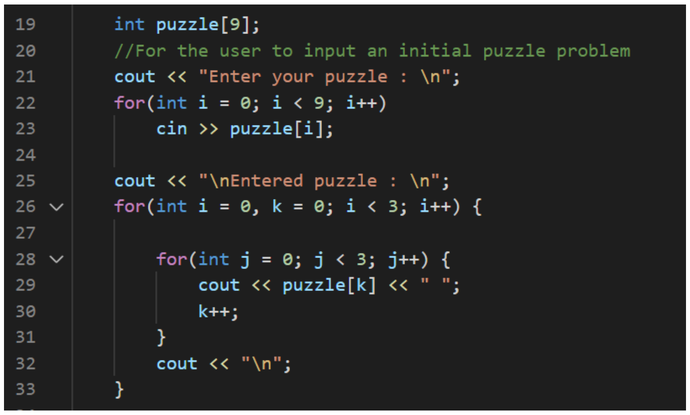
Code Documentation

- Implementation is divided into 3 files, main.cpp, Node.h, and Algo.h

Detailed discussion of the code

main.cpp

Line 19-33

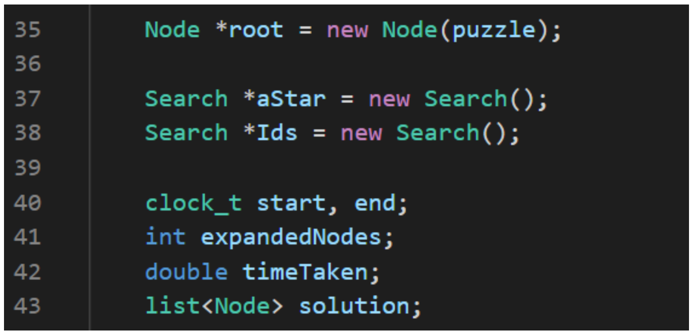


```
19      int puzzle[9];
20      //For the user to input an initial puzzle problem
21      cout << "Enter your puzzle : \n";
22      for(int i = 0; i < 9; i++)
23          cin >> puzzle[i];
24
25      cout << "\nEntered puzzle : \n";
26      for(int i = 0, k = 0; i < 3; i++) {
27
28          for(int j = 0; j < 3; j++) {
29              cout << puzzle[k] << " ";
30              k++;
31          }
32          cout << "\n";
33      }
```

This code is only a basic keyboard input.
It has no error checking such as duplication of number, and out of bound number.

I assume that the user inputs a perfect 8 puzzle problem.

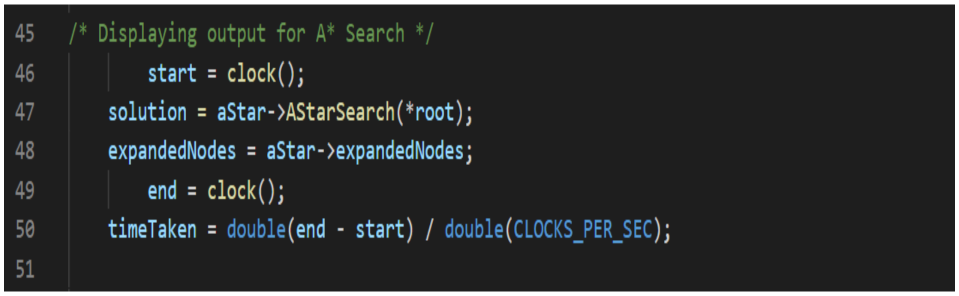
Line 35 - 43



```
35      Node *root = new Node(puzzle);
36
37      Search *aStar = new Search();
38      Search *Ids = new Search();
39
40      clock_t start, end;
41      int expandedNodes;
42      double timeTaken;
43      list<Node> solution;
```

Declaration of variables and Instantiation of Class Objects that is necessary for the required output of the machine problem.

Line 45 - 53



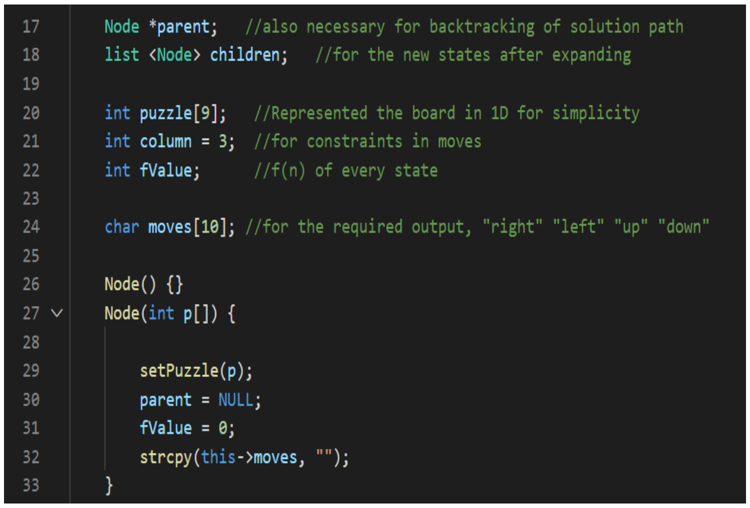
```
45  /* Displaying output for A* Search */
46          start = clock();
47      solution = aStar->AStarSearch(*root);
48      expandedNodes = aStar->expandedNodes;
49          end = clock();
50      timeTaken = double(end - start) / double(CLOCKS_PER_SEC);
51
```

Line 45 – 50, calculates the running time of the algorithm. Result is displayed in seconds unit of time.

Node.h

Node.h is the implementation of the states of the board. This contains the code necessary to create new states, and to get the Manhattan distance heuristics of each state.

Line 17 – Line 33



```
17      Node *parent;   //also necessary for backtracking of solution path
18      list <Node> children;   //for the new states after expanding
19
20      int puzzle[9];   //Represented the board in 1D for simplicity
21      int column = 3;  //for constraints in moves
22      int fValue;      //f(n) of every state
23
24      char moves[10]; //for the required output, "right" "left" "up" "down"
25
26      Node() {}
27      Node(int p[]) {
28
29          setPuzzle(p);
30          parent = NULL;
31          fValue = 0;
32          strcpy(this->moves, "");
33      }
```

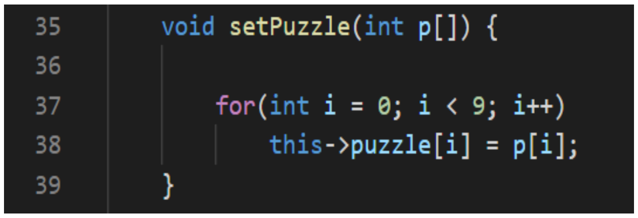
class Node is set to public.

Why the attributes are declared is on their comments

Constructor for class Node,

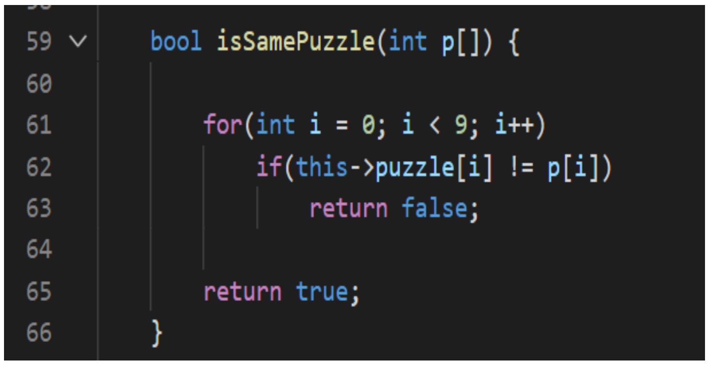
it takes in and sets the puzzle,

fValue is zero, for the manhattan distance



```
35      void setPuzzle(int p[]) {
36
37          for(int i = 0; i < 9; i++)
38              this->puzzle[i] = p[i];
39      }
```

setPuzzle, used in constructor. Basically just copies the inputted puzzle to the object’s own puzzle problem.



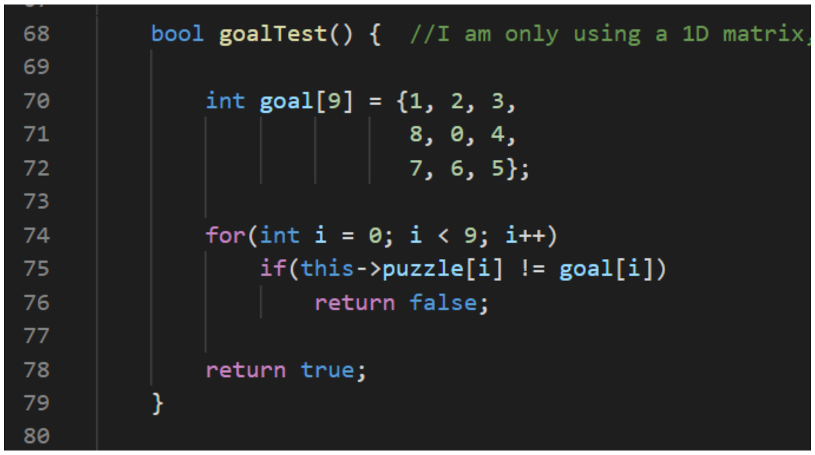
```
59      bool isSamePuzzle(int p[]) {
60
61          for(int i = 0; i < 9; i++)
62              if(this->puzzle[i] != p[i])
63                  return false;
64
65          return true;
66      }
```

isSamePuzzle, takes puzzle problem (state) as parameter and returns a boolean.

It checks whether the input is the same as the objects’s state is the same.

Used for checking if a state already exists in the open list, and closed list (in the A* algorithm)

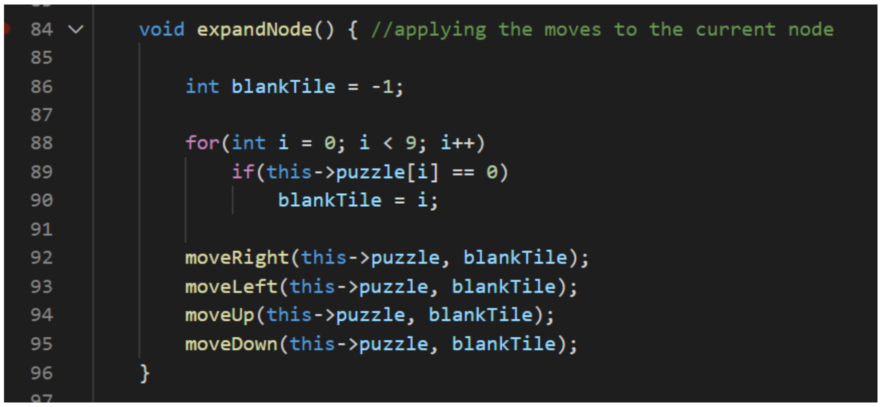
Node.h



```cpp
    bool goalTest() {  //I am only using a 1D matrix,

        int goal[9] = {1, 2, 3,
                       8, 0, 4,
                       7, 6, 5};

        for(int i = 0; i < 9; i++)
            if(this->puzzle[i] != goal[i])
                return false;

        return true;
    }
```

goalTest, I am only using a 1 dimensional array for the board.

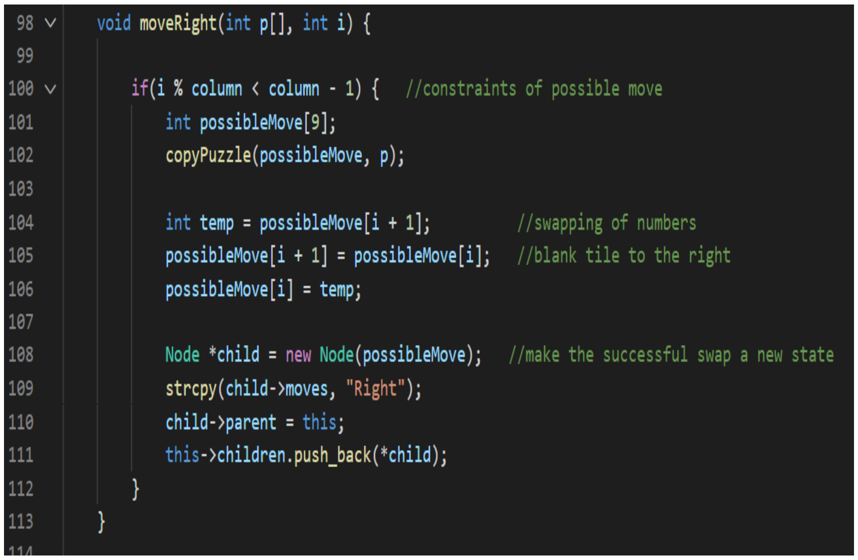
Returns a Boolean, if true then the object's state has reach the goal state



```cpp
    void expandNode() { //applying the moves to the current node

        int blankTile = -1;

        for(int i = 0; i < 9; i++)
            if(this->puzzle[i] == 0)
                blankTile = i;

        moveRight(this->puzzle, blankTile);
        moveLeft(this->puzzle, blankTile);
        moveUp(this->puzzle, blankTile);
        moveDown(this->puzzle, blankTile);
    }
```

expandNode, calls all the possible moves that can be made on the current state.

It locates the blank tile on the state for the parameter on the possible moves.



```cpp
98      void moveRight(int p[], int i) {
99
100         if(i % column < column - 1) {   //constraints of possible move
101             int possibleMove[9];
102             copyPuzzle(possibleMove, p);
103
104             int temp = possibleMove[i + 1];         //swapping of numbers
105             possibleMove[i + 1] = possibleMove[i];  //blank tile to the right
106             possibleMove[i] = temp;
107
108             Node *child = new Node(possibleMove);   //make the successful swap a new state
109             strcpy(child->moves, "Right");
110             child->parent = this;
111             this->children.push_back(*child);
112         }
113     }
```

moveRight, takes the object's current state and the index of the blank tile.

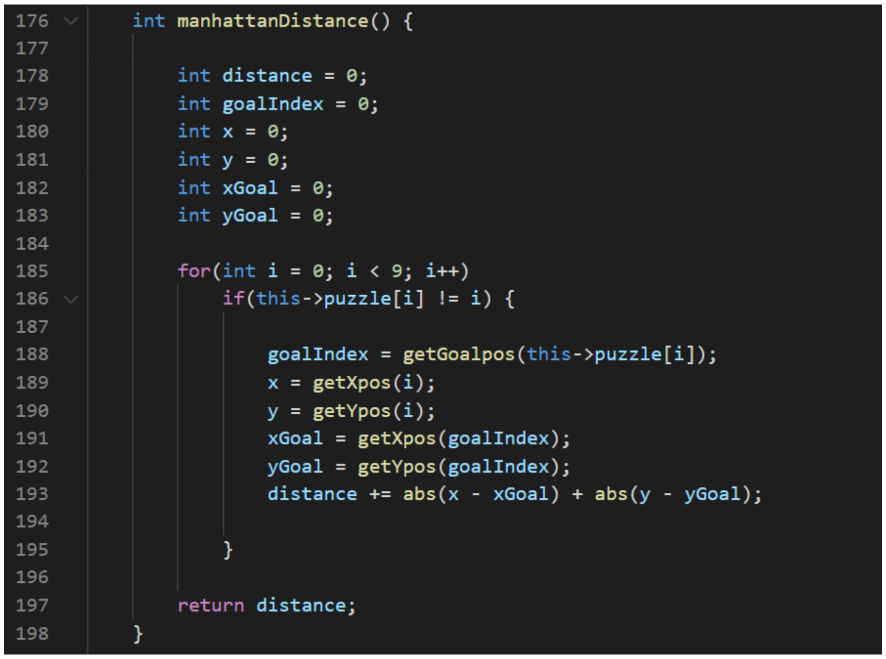
Line 100 is the constraint if moving right is possible.

Line 104 swaps the blank tile to the right

Line 108 – 111, instantiates a child Node (a new state) then points its parent pointer to the current object and add its children list. Also sets its moves to string "Right" for the required output of the machine problem.

This code is similar to moveRight, moveLeft, moveUp, and moveDown

Node.h

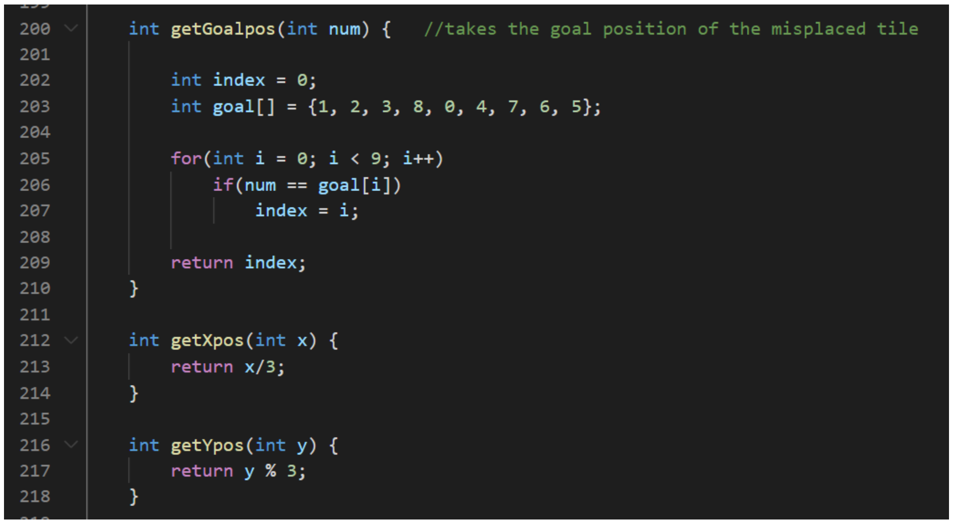


```
176  int manhattanDistance() {
177
178      int distance = 0;
179      int goalIndex = 0;
180      int x = 0;
181      int y = 0;
182      int xGoal = 0;
183      int yGoal = 0;
184
185      for(int i = 0; i < 9; i++)
186          if(this->puzzle[i] != i) {
187
188              goalIndex = getGoalpos(this->puzzle[i]);
189              x = getXpos(i);
190              y = getYpos(i);
191              xGoal = getXpos(goalIndex);
192              yGoal = getYpos(goalIndex);
193              distance += abs(x - xGoal) + abs(y - yGoal);
194
195          }
196
197      return distance;
198  }
```

Line 176 – 198 is the implementation of the Manhattan Distance that is used as the heuristic for A* algorithm.

I used its formula:
abs(x - x of goal) + abs(y - y of Goal)

Being in a 1 dimension array can be tricky so I used getXpos and getYpos functions to get the x and y position of the *misplaced tiles* and its goal index.



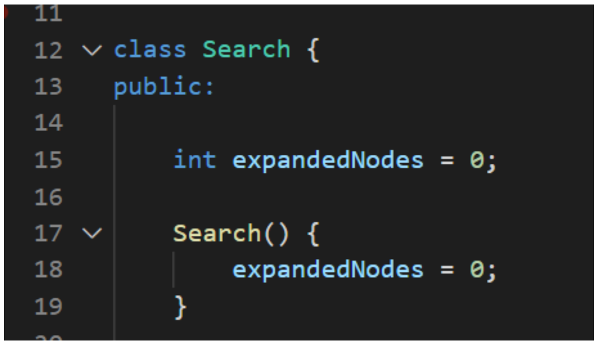
```
200  int getGoalpos(int num) {   //takes the goal position of the misplaced tile
201
202      int index = 0;
203      int goal[] = {1, 2, 3, 8, 0, 4, 7, 6, 5};
204
205      for(int i = 0; i < 9; i++)
206          if(num == goal[i])
207              index = i;
208
209      return index;
210  }
211
212  int getXpos(int x) {
213      return x/3;
214  }
215
216  int getYpos(int y) {
217      return y % 3;
218  }
```

getGoalpos method returns the index of a misplaced tile. it can be made faster but may require more code. I just made it simple enough.

getXpos and *getYpos* are the methods that made it possible to calculate the Manhattan Distance on a 1 dimensional array.

Algo.h

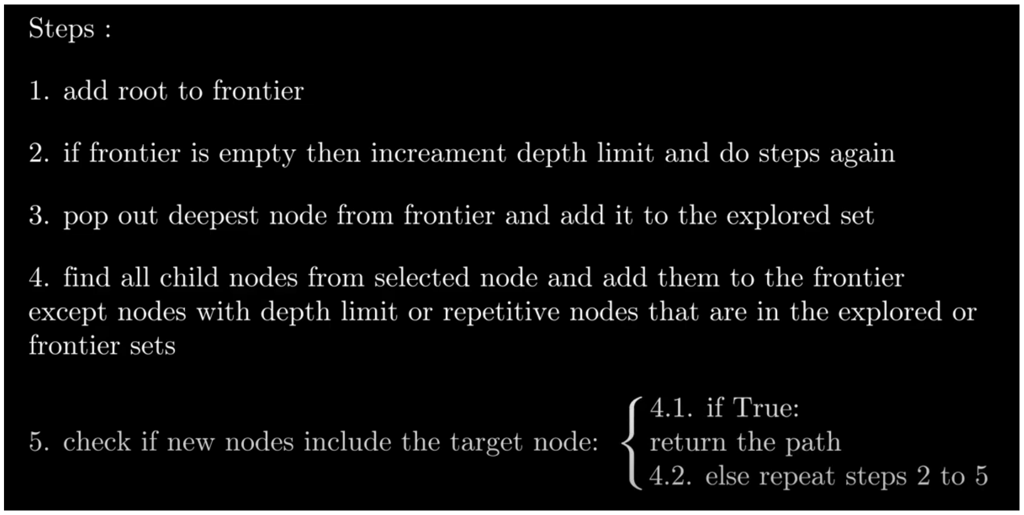
Algo.h contains the implementation for the IDS and A* search algorithm and some other methods to support it.



```cpp
class Search {
public:

    int expandedNodes = 0;

    Search() {
        expandedNodes = 0;
    }
```

I used class *Search* that contains methods *IterativeDeepeningSearch* and *AStarSearch* as methods that returns the solution path if there is any, and takes in the initial state as their parameter.

For the **Iterative Deepening Search**, I used an algorithm from a youtube video. I don't know if it accurate but its simulation shows it is. I also used iterative implementation instead of recursive as I tried recursive and the confusion led me more errors than usual. This showed me that I have to practice my understanding on recursive implementations.



Steps :

1. add root to frontier
2. if frontier is empty then increament depth limit and do steps again
3. pop out deepest node from frontier and add it to the explored set
4. find all child nodes from selected node and add them to the frontier except nodes with depth limit or repetitive nodes that are in the explored or frontier sets
5. check if new nodes include the target node: 4.1. if True: return the path; 4.2. else repeat steps 2 to 5

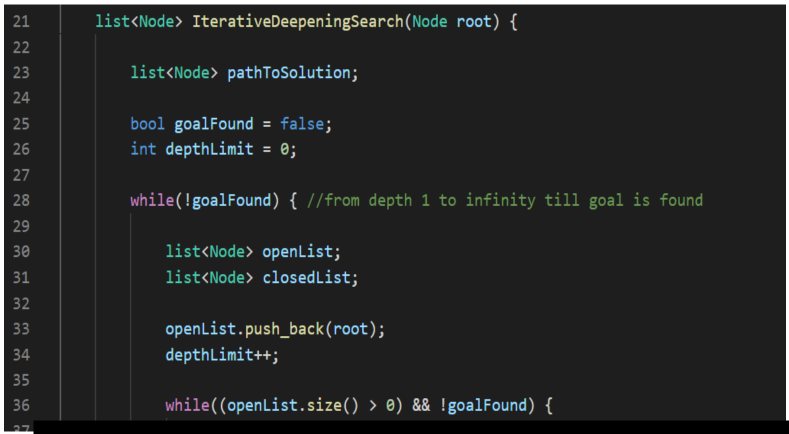
This is the algorithm that I followed.

Based on my understanding and from what I read on it, it is not that good for practical applications. It may be very slow because of its exponential time complexity

Algo.h

IDS implementation

I really had troubles on deciding whether my implementation is correct because it returns a really long execution time and opens far too many nodes when a more complex puzzle is inputted.

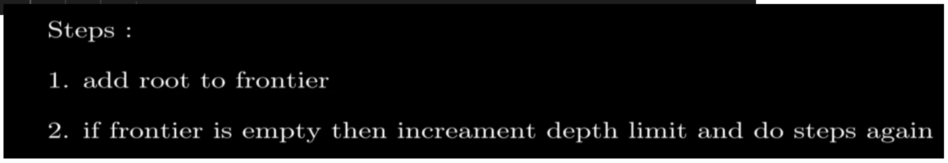


```
21    list<Node> IterativeDeepeningSearch(Node root) {
22
23        list<Node> pathToSolution;
24
25        bool goalFound = false;
26        int depthLimit = 0;
27
28        while(!goalFound) { //from depth 1 to infinity till goal is found
29
30            list<Node> openList;
31            list<Node> closedList;
32
33            openList.push_back(root);
34            depthLimit++;
35
36            while((openList.size() > 0) && !goalFound) {
```

Line 28 – 36

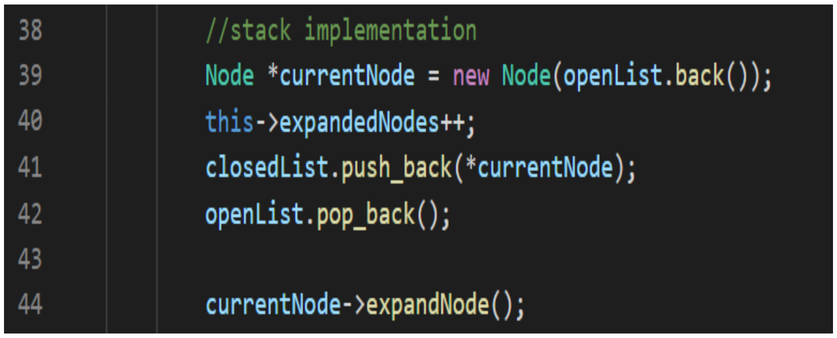
This is the Step 1 and 2 of the algorithm.

If the depth has been reached, it can no longer add new states to the open list therefore making it empty. Step 2 states to start at the initial problem again then increment the depth limit by 1



Steps :

1. add root to frontier
2. if frontier is empty then increament depth limit and do steps again

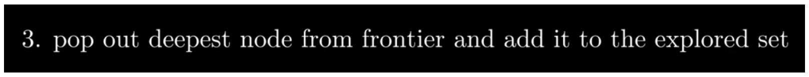


```
38            //stack implementation
39            Node *currentNode = new Node(openList.back());
40            this->expandedNodes++;
41            closedList.push_back(*currentNode);
42            openList.pop_back();
43
44            currentNode->expandNode();
```

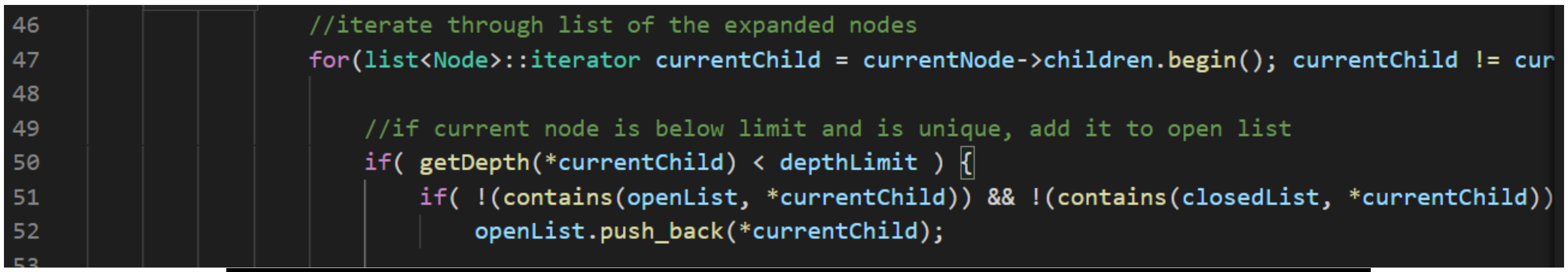
Line 38 – 44 I think is the step 3 in the algorithm.

It is a stack implementation which means it always pop the deepest node that exist inside the open list

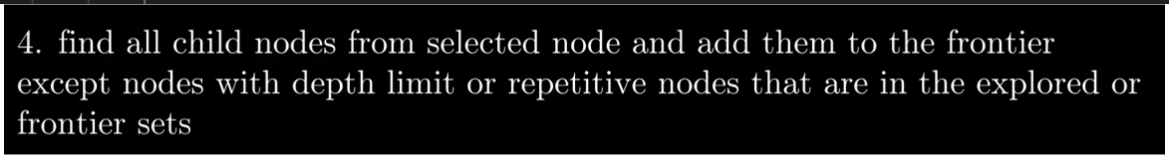
Line 44 expands the node, creating deeper nodes.



3. pop out deepest node from frontier and add it to the explored set



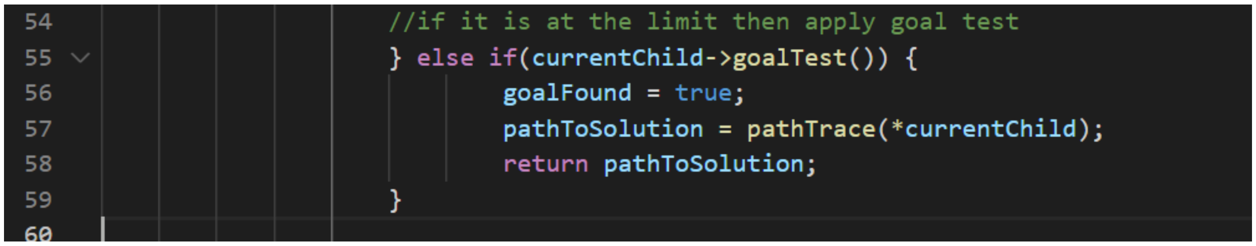
```
46                  //iterate through list of the expanded nodes
47                  for(list<Node>::iterator currentChild = currentNode->children.begin(); currentChild != cur
48
49                      //if current node is below limit and is unique, add it to open list
50                      if( getDepth(*currentChild) < depthLimit ) {
51                          if( !(contains(openList, *currentChild)) && !(contains(closedList, *currentChild))
52                              openList.push_back(*currentChild);
53
```



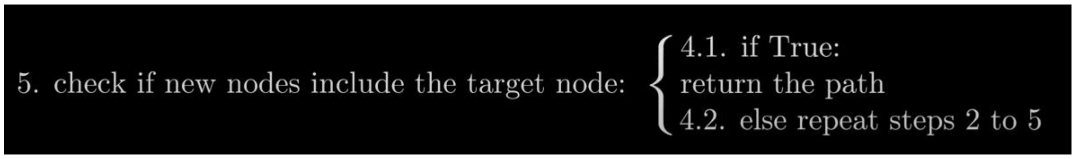
4. find all child nodes from selected node and add them to the frontier except nodes with depth limit or repetitive nodes that are in the explored or frontier sets

Line 47 lets me iterate through the children of the expanded node.

Line 50 – 52 Is my implementation on the step 4, If the child node is below the depth limit and it is unique (meaning no duplicated in the open list and close list)
then add it to the last of the list for the stack implementation



```
54                      //if it is at the limit then apply goal test
55                      } else if(currentChild->goalTest()) {
56                              goalFound = true;
57                              pathToSolution = pathTrace(*currentChild);
58                              return pathToSolution;
59                      }
60
```



$$\text{5. check if new nodes include the target node: } \begin{cases} \text{4.1. if True:} \\ \text{return the path} \\ \text{4.2. else repeat steps 2 to 5} \end{cases}$$

Line 55 – 58 is my implementation on the step 5 of the algorithm. If the previous check if it is below the depth limit fails, then it must mean it would be at the depth limit. If so, apply the goal test on the current child, if the test fails then continue with the while loop.

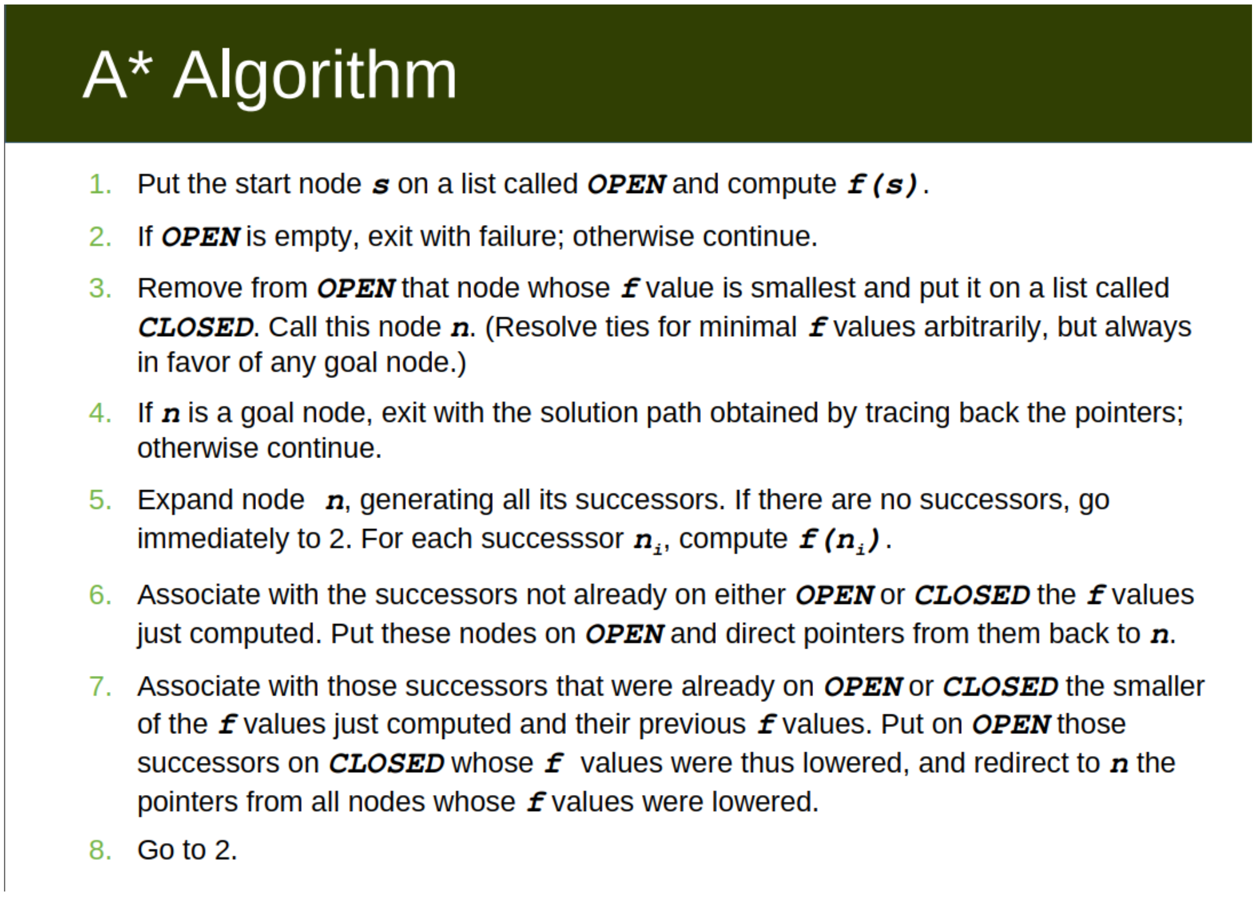
A* Search implementation

I followed the lecture's algorithm for it, I am not sure if it is accurate but it does return far better results than the IDS implementation as expected.

I had some troubles on removing Nodes from the open list and closed lists as the <list> functions on c++ can be quite a hassle.

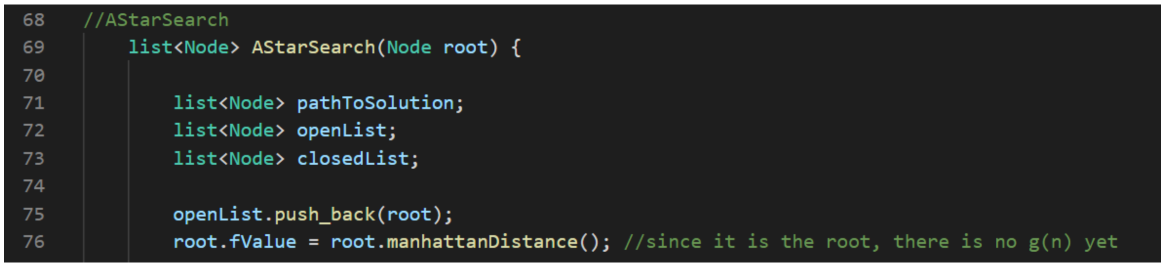
I learned that you can't just remove an element just by the index. I have to create a list::iterator, and keep track of the index of the element I want to remove. I can remove an element successfully using advance() of the iterator and erase() of the list function.

This is the algorithm that I followed.



A* Algorithm

1. Put the start node ***s*** on a list called ***OPEN*** and compute ***f(s)***.
2. If ***OPEN*** is empty, exit with failure; otherwise continue.
3. Remove from ***OPEN*** that node whose ***f*** value is smallest and put it on a list called ***CLOSED***. Call this node ***n***. (Resolve ties for minimal ***f*** values arbitrarily, but always in favor of any goal node.)
4. If ***n*** is a goal node, exit with the solution path obtained by tracing back the pointers; otherwise continue.
5. Expand node ***n***, generating all its successors. If there are no successors, go immediately to 2. For each successsor n_i, compute $f(n_i)$.
6. Associate with the successors not already on either ***OPEN*** or ***CLOSED*** the ***f*** values just computed. Put these nodes on ***OPEN*** and direct pointers from them back to ***n***.
7. Associate with those successors that were already on ***OPEN*** or ***CLOSED*** the smaller of the ***f*** values just computed and their previous ***f*** values. Put on ***OPEN*** those successors on ***CLOSED*** whose ***f*** values were thus lowered, and redirect to ***n*** the pointers from all nodes whose ***f*** values were lowered.
8. Go to 2.

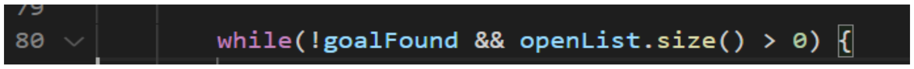


```
68  //AStarSearch
69      list<Node> AStarSearch(Node root) {
70
71          list<Node> pathToSolution;
72          list<Node> openList;
73          list<Node> closedList;
74
75          openList.push_back(root);
76          root.fValue = root.manhattanDistance(); //since it is the root, there is no g(n) yet
```

This is the step 1 of the algorithm from the lecture.

1. Put the start node ***s*** on a list called ***OPEN*** and compute ***f(s)***.

At Line 76, it is still at the root node so the g(n) is still 0. I just use the depth level of each node for the g(n) because it is similar to the steps made to reach it and also because each move cost only 1.

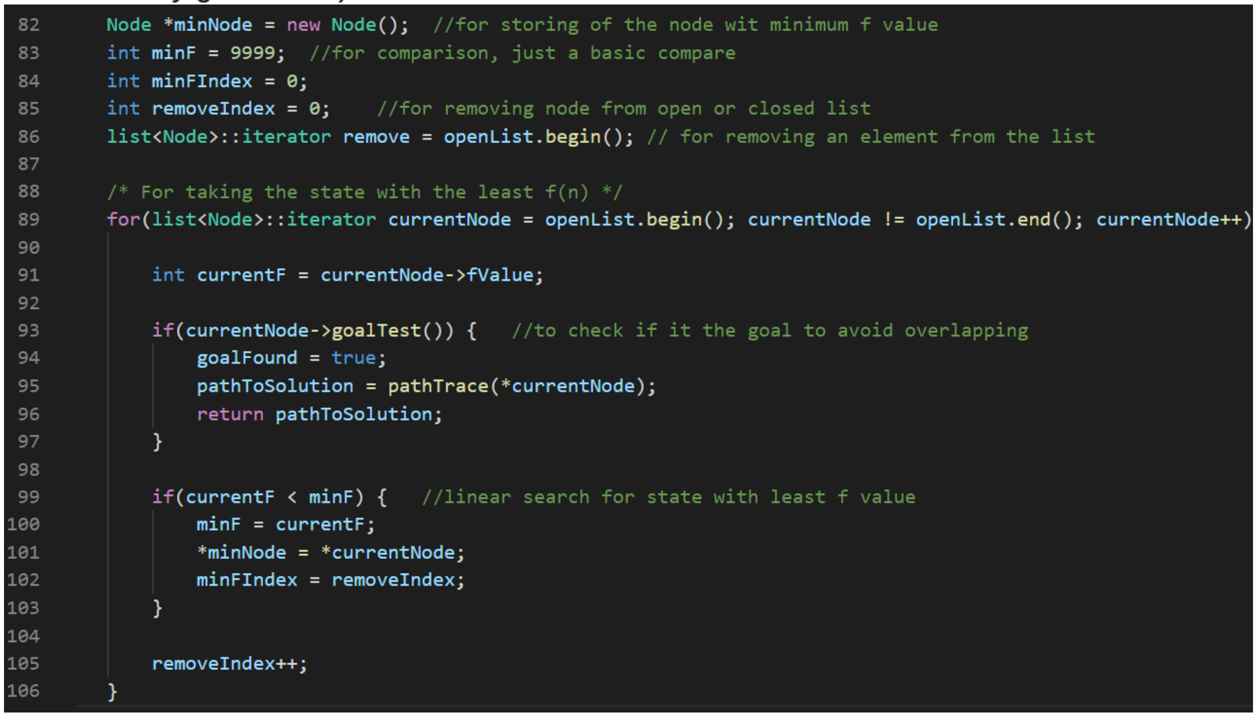


```
80          while(!goalFound && openList.size() > 0) {
```

Line 80 is the implementation of step 2

2. If ***OPEN*** is empty, exit with failure; otherwise continue.

3. Remove from ***OPEN*** that node whose ***f*** value is smallest and put it on a list called ***CLOSED***. Call this node ***n***. (Resolve ties for minimal ***f*** values arbitrarily, but always in favor of any goal node.)



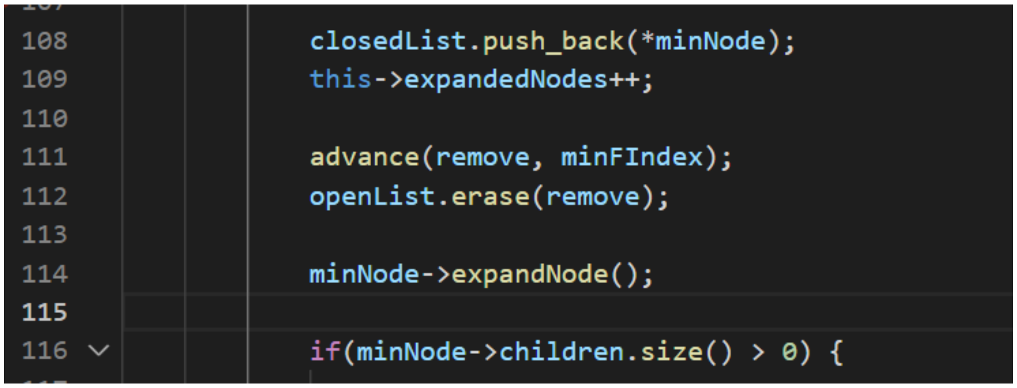
```
Node *minNode = new Node();  //for storing of the node wit minimum f value
int minF = 9999;  //for comparison, just a basic compare
int minFIndex = 0;
int removeIndex = 0;    //for removing node from open or closed list
list<Node>::iterator remove = openList.begin(); // for removing an element from the list

/* For taking the state with the least f(n) */
for(list<Node>::iterator currentNode = openList.begin(); currentNode != openList.end(); currentNode++)

    int currentF = currentNode->fValue;

    if(currentNode->goalTest()) {   //to check if it the goal to avoid overlapping
        goalFound = true;
        pathToSolution = pathTrace(*currentNode);
        return pathToSolution;
    }

    if(currentF < minF) {   //linear search for state with least f value
        minF = currentF;
        *minNode = *currentNode;
        minFIndex = removeIndex;
    }

    removeIndex++;
}
```

This is my implementation of the step 3, I have declared variables to make it possible and I think there is far better implementation for this given more time.

Instead of the step 4, I inserted the goal test inside the for loop so that if ever a goal node appears on the open list, it can end the program faster.

I only used *Linear Search* for comparing of the f(n) values of each node in the open list. I think there are better ways to select better f(n) if there are equal ones.

5. Expand node ***n***, generating all its successors. If there are no successors, go immediately to 2. For each successsor n_i, compute $f(n_i)$.

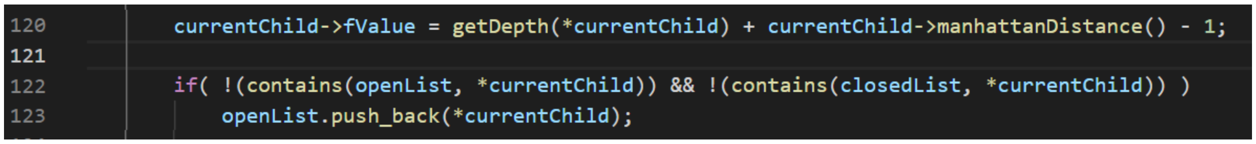


```
108            closedList.push_back(*minNode);
109            this->expandedNodes++;
110
111            advance(remove, minFIndex);
112            openList.erase(remove);
113
114            minNode->expandNode();
115
116            if(minNode->children.size() > 0) {
```

Line 111-112 is the tiring process of removing the node from the list…

Line 114 and Line 116 is my implementation for the step 5 of the algorithm.

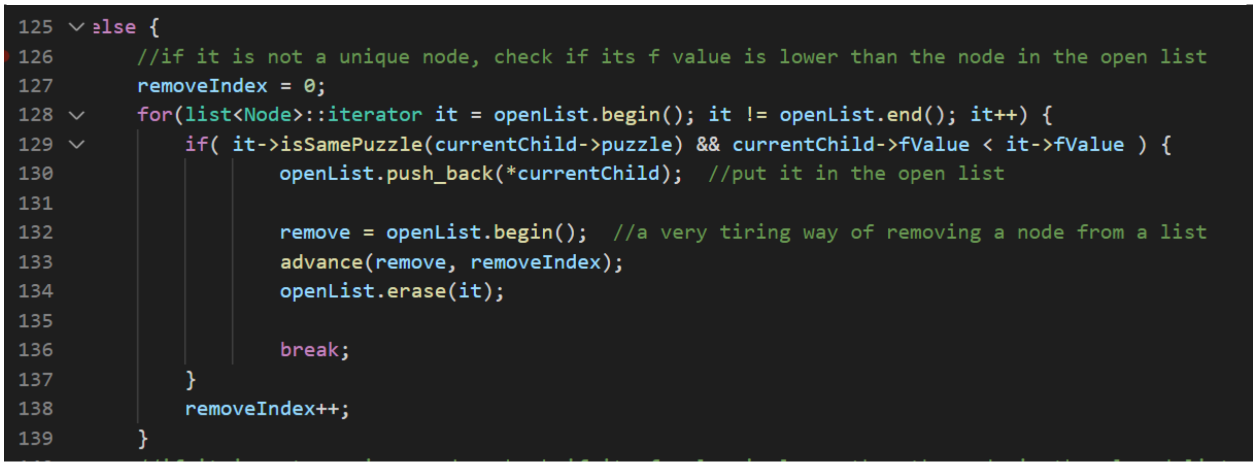
6. Associate with the successors not already on either ***OPEN*** or ***CLOSED*** the ***f*** values just computed. Put these nodes on ***OPEN*** and direct pointers from them back to ***n***.



```
120     currentChild->fValue = getDepth(*currentChild) + currentChild->manhattanDistance() - 1;
121
122     if( !(contains(openList, *currentChild)) && !(contains(closedList, *currentChild)) )
123         openList.push_back(*currentChild);
```

Line 120 – 123 is my implementation for step 6. I don't know if my understanding is correct but I think if it is a unique node, put it on the list with its calculated f(n) value.

7. Associate with those successors that were already on ***OPEN*** or ***CLOSED*** the smaller of the ***f*** values just computed and their previous ***f*** values. Put on ***OPEN*** those successors on ***CLOSED*** whose ***f*** values were thus lowered, and redirect to ***n*** the pointers from all nodes whose ***f*** values were lowered.

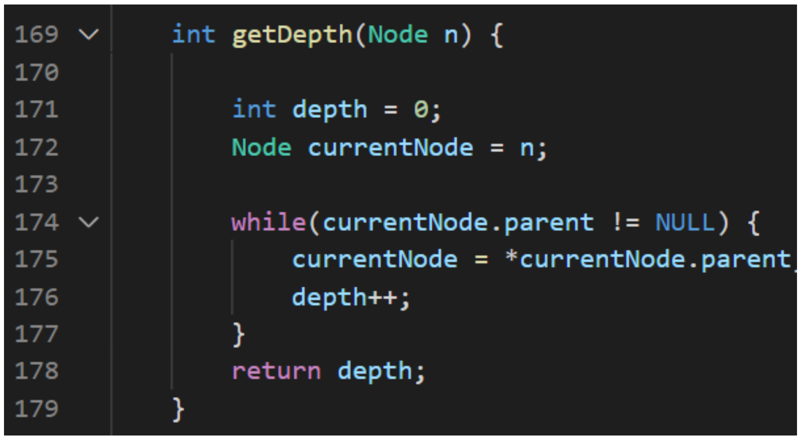


```
125 else {
126     //if it is not a unique node, check if its f value is lower than the node in the open list
127     removeIndex = 0;
128     for(list<Node>::iterator it = openList.begin(); it != openList.end(); it++) {
129         if( it->isSamePuzzle(currentChild->puzzle) && currentChild->fValue < it->fValue ) {
130                 openList.push_back(*currentChild);  //put it in the open list
131
132                 remove = openList.begin();  //a very tiring way of removing a node from a list
133                 advance(remove, removeIndex);
134                 openList.erase(it);
135
136                 break;
137         }
138         removeIndex++;
139     }
```

Line 128 – 139 is my implementation for the step 7. After checking if it unique or not. This implementation says that if it is not unique, check the open list for its duplicate and then compare its f(n) values. If the current child is lower then replace it. I feel like my understanding of step 7 is insufficient, something is off here?

This is similar to Line 140 – 154, difference is that it checks the closed list.

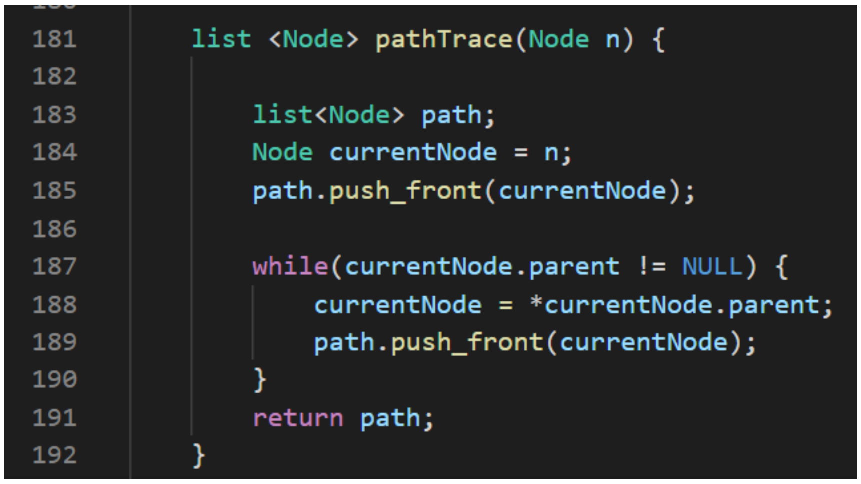
Step 8 says to go back to step 2 which is the while loop. I think the code is done.



```
int getDepth(Node n) {

    int depth = 0;
    Node currentNode = n;

    while(currentNode.parent != NULL) {
        currentNode = *currentNode.parent
        depth++;
    }
    return depth;
}
```

getDepth returns the depth of the entered Node.

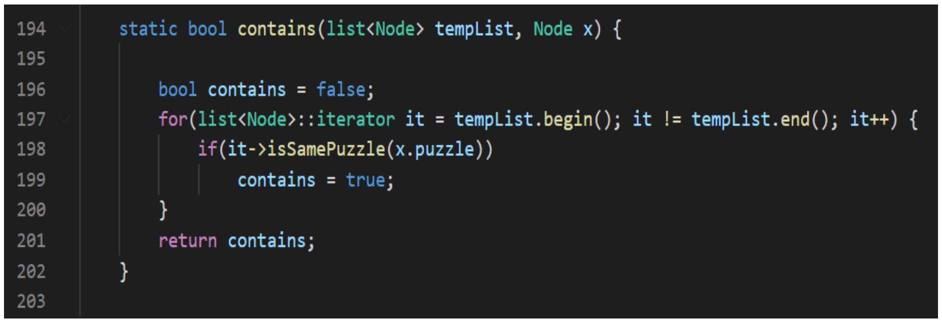
This is useful for both searching algorithm. I used it for the depth limits on IDS and for taking the f(n) values on A* search



```
    list <Node> pathTrace(Node n) {

        list<Node> path;
        Node currentNode = n;
        path.push_front(currentNode);

        while(currentNode.parent != NULL) {
            currentNode = *currentNode.parent;
            path.push_front(currentNode);
        }
        return path;
    }
```

pathTrace returns the path it took from initial to the goal node. This is only used when the *goalTest* has returned true;

It traces back from goal to the initial node using the parent pointer of each created node.



```
    static bool contains(list<Node> tempList, Node x) {

        bool contains = false;
        for(list<Node>::iterator it = tempList.begin(); it != tempList.end(); it++) {
            if(it->isSamePuzzle(x.puzzle))
                contains = true;
        }
        return contains;
    }

```

contains method returns a boolean, this is useful for checking if a new node (state) is unique.

- **Analysis and Comparison**

Initial State				IDS	A*
Easy			Solution Path	U-R-U-L-D	U-R-U-L-D
1	3	4	Solution Cost	5	5
8	6	2	Expanded Nodes	50	5
7		5	Running Time	~0.001 seconds	< 0.000000 seconds

Initial State				IDS	A*
Medium			Solution Path	U-R-R-D-L-L-U-R-D	U-R-R-D-L-L-U-R-D
2	8	1	Solution Cost	9	9
	4	3	Expanded Nodes	560	14
7	6	5	Running Time	~0.10300 seconds	~0.001000 seconds

Initial State				IDS	A*
Hard			Solution Path	U-L-D-R-U-L-D-RU-L-L-U-R-R-D-L-L-U-R-D-D-U	L-U-L-U-R-R-D-L-L-U-R-D
2	8	1	Solution Cost	22	12
4	6	3	Expanded Nodes	41366	28
7	5		Running Time	~117.31900 seconds	~0.002000 seconds

Initial State				IDS	A*
Worst			Solution Path	-	L-D-R-R-U-U-L-L-D-D-R-R-U-U-L-L-D-R-R-U-U-L-L-D-D-R-R-U-L
5	6	7	Solution Cost	-	30
4		8	Expanded Nodes	-	5883
3	2	1	Running Time	>30 minutes	~56 seconds

Initial State				IDS	A*
8	1	2	Solution Path	L-U-R-R-D-D-L-U	L-U-R-R-D-D-L-U
6	0	3	Solution Cost	8	8
			Expanded Nodes	421	16
7	5	4	Running Time	~0.03500 seconds	< 0.000000 seconds