Programming Project 1

In Partial Fulfillment for CS Elec 1

Submitted by:

Herrera, Cy Jay

Deduque, Lean Jedfrey

Cleofe, Erjay

Tabug, Ignacio III

Bron, Greg

Table of Contents

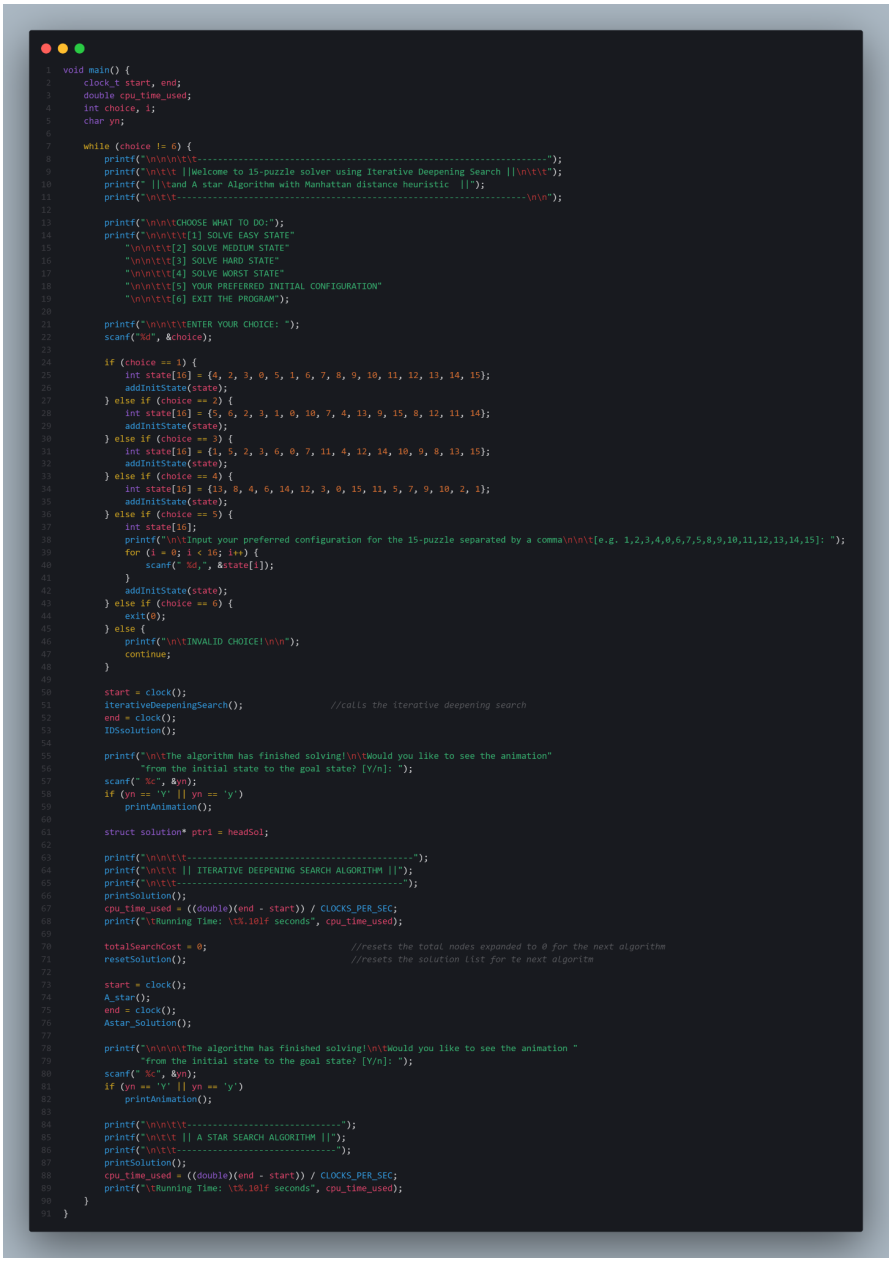
Code Structure

Variable Declarations

```
struct Node{
    int state[16];
    struct Node* Parent;
    int moves; //0 for down, 1 for up, 2 for right, 3 for left
    int depth;
    int fn; //f(n) = g(n) + manhattan distance
    int gn;
    struct Node* next;
};

struct Fringe{
    struct Node* nodeAddress;
    struct Fringe* next;
};

struct closedList{
    struct Node* nodeAddress;
    struct closedList* next;
};

struct solution{
    struct Node* nodeAddress;
    struct solution* next;
    int totalSearchCost;
};

int goalState[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
int initState[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

struct Node* headNode;
struct Fringe* head;
struct closedList* headCl;
struct solution* headSol;

int totalSearchCost;
```

struct Node: This structure represents a node in the search space of the 15-puzzle. It contains an array state to store the configuration of the puzzle, a reference to the parent node Parent, an integer moves to track the move taken to reach this node (0 for down, 1 for up, 2 for right, 3 for left), depth to store the depth of the node in the search tree, fn and gn to hold the total cost function values for A* search (f(n) = g(n) + h(n)), and a reference to the next node next in a linked list.
struct Fringe: This structure represents a linked list of nodes in the fringe, which are nodes that have been generated but not yet expanded. It contains a reference to a node nodeAddress and a reference to the next node in the fringe next.
struct closedList: This structure represents a linked list of nodes that have been expanded and added to the closed list to avoid revisiting them. It contains a reference to a node nodeAddress and a reference to the next node in the closed list next.
struct solution: This structure represents a linked list of nodes that form the solution path from the initial state to the goal state. It contains a reference to a node nodeAddress, a reference to the next node in the solution.
goalState: An array representing the desired configuration of the 15-puzzle, which is used as the goal state in the search algorithms.
initState: An array to store the initial state of the 15-puzzle, which is typically set based on user input or selected scenarios.
headNode, head, headCl, headSol: Pointers that reference the heads of linked lists for nodes, fringe, closed list, and the solution path, respectively.
totalSearchCost: An integer used to keep track of the total search cost during the execution of the search algorithms.

main()



```
void main() {
    clock_t start, end;
    double cpu_time_used;
    int choice, i;
    char yn;

    while (choice != 6) {
        printf("\n\n\n\t\t-----------------------------------------------------------------------");
        printf("\n\t\t ||Welcome to 15-puzzle solver using Iterative Deepening Search ||\n\t\t");
        printf(" ||\tand A star Algorithm with Manhattan distance heuristic  ||");
        printf("\n\t\t-----------------------------------------------------------------------\n\n");

        printf("\n\n\tCHOOSE WHAT TO DO:");
        printf("\n\n\t\t[1] SOLVE EASY STATE"
            "\n\n\t\t[2] SOLVE MEDIUM STATE"
            "\n\n\t\t[3] SOLVE HARD STATE"
            "\n\n\t\t[4] SOLVE WORST STATE"
            "\n\n\t\t[5] YOUR PREFERRED INITIAL CONFIGURATION"
            "\n\n\t\t[6] EXIT THE PROGRAM");

        printf("\n\n\t\tENTER YOUR CHOICE: ");
        scanf("%d", &choice);

        if (choice == 1) {
            int state[16] = {4, 2, 3, 0, 5, 1, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
            addInitState(state);
        } else if (choice == 2) {
            int state[16] = {5, 6, 2, 3, 1, 0, 10, 7, 4, 13, 9, 15, 8, 12, 11, 14};
            addInitState(state);
        } else if (choice == 3) {
            int state[16] = {1, 5, 2, 3, 6, 0, 7, 11, 4, 12, 14, 10, 9, 8, 13, 15};
            addInitState(state);
        } else if (choice == 4) {
            int state[16] = {13, 8, 4, 6, 14, 12, 3, 0, 15, 11, 5, 7, 9, 10, 2, 1};
            addInitState(state);
        } else if (choice == 5) {
            int state[16];
            printf("\n\tInput your preferred configuration for the 15-puzzle separated by a comma\n\n\t[e.g. 1,2,3,4,0,6,7,5,8,9,10,11,12,13,14,15]: ");
            for (i = 0; i < 16; i++) {
                scanf(" %d,", &state[i]);
            }
            addInitState(state);
        } else if (choice == 6) {
            exit(0);
        } else {
            printf("\n\tINVALID CHOICE!\n\n");
            continue;
        }

        start = clock();
        iterativeDeepeningSearch();               //calls the iterative deepening search
        end = clock();
        IDSsolution();

        printf("\n\tThe algorithm has finished solving!\n\tWould you like to see the animation"
               "from the initial state to the goal state? [Y/n]: ");
        scanf(" %c", &yn);
        if (yn == 'Y' || yn == 'y')
            printAnimation();

        struct solution* ptr1 = headSol;

        printf("\n\n\t\t-----------------------------------------");
        printf("\n\t\t || ITERATIVE DEEPENING SEARCH ALGORITHM ||");
        printf("\n\t\t-----------------------------------------");
        printSolution();
        cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("\tRunning Time: \t%.10lf seconds", cpu_time_used);

        totalSearchCost = 0;                        //resets the total nodes expanded to 0 for the next algorithm
        resetSolution();                            //resets the solution list for te next algoritm

        start = clock();
        A_star();
        end = clock();
        Astar_Solution();

        printf("\n\n\n\tThe algorithm has finished solving!\n\tWould you like to see the animation "
               "from the initial state to the goal state? [Y/n]: ");
        scanf(" %c", &yn);
        if (yn == 'Y' || yn == 'y')
            printAnimation();

        printf("\n\n\t\t-----------------------------");
        printf("\n\t\t || A STAR SEARCH ALGORITHM ||");
        printf("\n\t\t-----------------------------");
        printSolution();
        cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("\tRunning Time: \t%.10lf seconds", cpu_time_used);
    }
}
```

It implements a menu-driven user interface within a while loop, continuously running until the user selects the "Exit" option. The menu offers choices to solve the 15-puzzle using various methods, including easy, medium, hard, or worst-case scenarios, as well as an option to input a custom initial configuration. Depending on the user's choice, specific puzzle configurations are defined and used as the initial state for the 15-puzzle. The program then measures the execution time for both the iterative deepening search and A* algorithms, displays the solutions, and optionally provides an

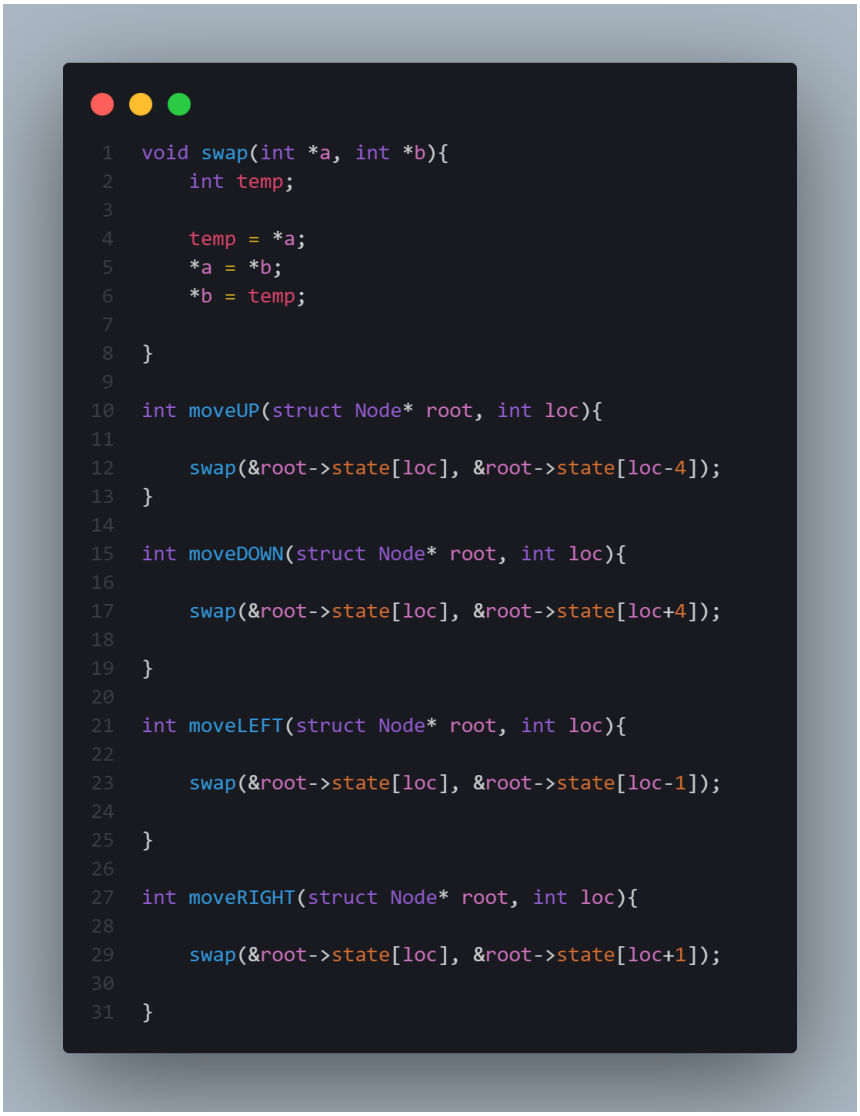
animation of the puzzle solving process. It efficiently manages the user interface, puzzle configuration, algorithm execution, and result presentation, making it a versatile tool for solving 15-puzzle instances with different complexities.

createNode()

```c
struct Node* createNode(int state[], struct Node* parent, int moves, int depth, int fn, int gn){

                                        //creation of the node and its corresponding details

    int i, j;
    struct Node* tempNode = (struct Node*)malloc(sizeof(struct Node));

    if(tempNode == NULL){
        exit(1);
    }
    for(i=0; i<16; i++){
        tempNode->state[i] = state[i];
    }
    tempNode->Parent = parent;
    tempNode->moves = moves;
    tempNode->depth = depth;
    tempNode->fn = fn;
    tempNode->gn = gn;

    return tempNode;

}
```

The createNode function is a critical component in constructing the search tree for the 15-puzzle solving algorithms. It is responsible for generating a new node to represent a specific puzzle state. This function takes several parameters, including the current puzzle state, denoted as the state array, a reference to the parent node from which the new node is derived, the move executed to transition to this state, the depth of the node in the search tree, and the heuristic values fn and gn. The fn value is particularly important; it combines the cost gn to reach the current state and the estimated heuristic value based on the Manhattan distance. By creating and linking nodes through the parent-child relationship, the search tree forms, allowing the algorithm to systematically explore and evaluate various puzzle states. The createNode function essentially encapsulates the puzzle's current configuration and its relationship to the previous state, playing a pivotal role in guiding the search and heuristic evaluation processes in the quest to find the optimal solution.

swap(), moveUP(), moveDOWN(), moveLEFT(), moveRIGHT()



```c
void swap(int *a, int *b){
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;

}

int moveUP(struct Node* root, int loc){

    swap(&root->state[loc], &root->state[loc-4]);
}

int moveDOWN(struct Node* root, int loc){

    swap(&root->state[loc], &root->state[loc+4]);

}

int moveLEFT(struct Node* root, int loc){

    swap(&root->state[loc], &root->state[loc-1]);

}

int moveRIGHT(struct Node* root, int loc){

    swap(&root->state[loc], &root->state[loc+1]);

}
```

The swap function is a fundamental utility function within the 15-puzzle solving algorithm that plays a pivotal role in swapping the positions of two tiles (or values) in the puzzle grid. This function takes two integer pointers, a and b, as arguments and facilitates the exchange of values they point to. In the context of the 15-puzzle, this function is instrumental for implementing moves within the puzzle. When the swap function is called, it temporarily stores the value pointed to by one pointer in a temporary variable, allowing it to be safely overwritten with the value from the other pointer. Subsequently, the value in the temporary variable is written to the second location. While the moveUP, moveDOWN, moveLEFT, and moveRIGHT functions refer to the movement of the blank tile, which means it would swap the locations of the blank tile and the tile depending on the action.

validActions()

```
int validActions(int state[], int prevMove, int possibleMoves[]){

    int loc = 0;
    int i;

    for(i=0; i<16; i++){                //finds the location of the blank tile (zero)
        if(state[i] == 0){
            loc = i;
            break;
        }
    }

                                        //the possibleMoves array would contain 2 values:
                                        // 0 if the action is valid, -1 if otherwise

    for(i=0; i<4; i++){
        possibleMoves[i] = 0;
    }

    switch(prevMove){                   //disregards the inverse of the parent's move to the child's valid actions
                                        //e.g. if the parent's move was move UP, the child will not move DOWN
        case 0: possibleMoves[1] = -1;
                break;
        case 1: possibleMoves[0] = -1;
                break;
        case 2: possibleMoves[3] = -1;
                break;
        case 3: possibleMoves[2] = -1;
                break;
    }                                     //mathematically calculates the impossible move depending on the location
    if((loc-4)< 0){                       // of the blank tile
        possibleMoves[1] = -1;
    }
    if((loc+4)>15){
        possibleMoves[0] = -1;
    }
    if((loc%4) == 0){
        possibleMoves[3] = -1;
    }
    if(((loc+1) % 4) == 0){
        possibleMoves[2] = -1;
    }

    return loc;


}
```

The validActions function serves as a critical piece in the 15-puzzle solving algorithm, primarily responsible for evaluating the permissibility of each potential move within the current puzzle state. It takes three essential parameters: the current state of the puzzle represented as a 1D array (state[]), the previous move executed (prevMove), and an array (possibleMoves[]) to record the feasibility of each move. Initially, it initializes the possibleMoves array, designating all moves as valid (0). It then identifies the current location of the blank tile (commonly represented as 0) within the puzzle by iterating through the state array. To prevent the reversal of the prior move, it marks the opposite move as invalid by setting the corresponding element in possibleMoves to -1.

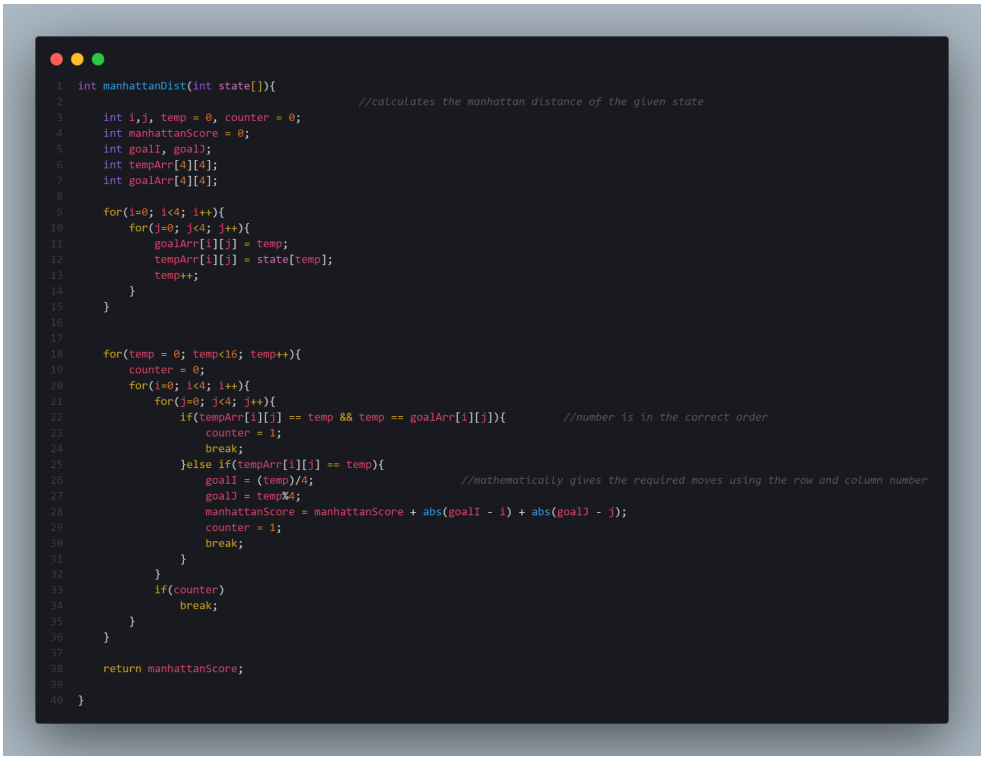
Furthermore, it considers boundary cases where moving outside the puzzle grid is disallowed. For instance, attempting to move up when the blank tile is in the top row is marked as invalid. The function ultimately returns the location of the blank tile within the puzzle, furnishing crucial information for subsequent moves. In essence, validActions guarantees that the search algorithm explores only legal, legitimate moves, thus preventing unlawful or superfluous actions during the puzzle-solving process.

resetclosedList()

```
void resetClosedList(){
                                    //The function deletes all the nodes in the closed lists
    struct closedList* ptr;         // once the search is completed

    ptr = headCl;

    while(headCl!=NULL){
        if(ptr->next == NULL){
            free(ptr);
            headCl = NULL;
        }else{
            headCl = headCl->next;
            ptr->next = NULL;
            free(ptr);
        }
        ptr = headCl;
    }
}
```

The primary purpose of resetClosedList is to efficiently clear and deallocate the memory occupied by these nodes, ensuring that system resources are not wasted, once the search (A* search) is finished. It progressively releases memory for each node while ensuring that the linked list structure is correctly maintained. Once the entire closed list has been processed and all nodes have been deallocated, the closed list is effectively reset, and the memory is freed, making it ready for future use in subsequent search iterations.

manhattanDist()



```
int manhattanDist(int state[]){
                                        //calculates the manhattan distance of the given state
    int i,j, temp = 0, counter = 0;
    int manhattanScore = 0;
    int goalI, goalJ;
    int tempArr[4][4];
    int goalArr[4][4];

    for(i=0; i<4; i++){
        for(j=0; j<4; j++){
            goalArr[i][j] = temp;
            tempArr[i][j] = state[temp];
            temp++;
        }
    }


    for(temp = 0; temp<16; temp++){
        counter = 0;
        for(i=0; i<4; i++){
            for(j=0; j<4; j++){
                if(tempArr[i][j] == temp && temp == goalArr[i][j]){        //number is in the correct order
                    counter = 1;
                    break;
                }else if(tempArr[i][j] == temp){
                    goalI = (temp)/4;                   //mathematically gives the required moves using the row and column number
                    goalJ = temp%4;
                    manhattanScore = manhattanScore + abs(goalI - i) + abs(goalJ - j);
                    counter = 1;
                    break;
                }
            }
            if(counter)
                break;
        }
    }

    return manhattanScore;

}
```

The manhattanDist function calculates the Manhattan distance for a given state in the 15-puzzle problem. The Manhattan distance is a heuristic used in search algorithms to estimate the minimum number of moves required to transform the current state into the goal state. In the context of the 15-puzzle, it measures the sum of the horizontal and vertical distances of each tile from its current position to its desired position in the goal state.

The function first initializes two 4x4 arrays, tempArr and goalArr, to represent the current state and the goal state, respectively. It then iterates through the elements of these arrays, comparing the positions of each number in the current state to their corresponding positions in the goal state. If a number is in the correct position, the function increments a counter. If a number is not in the correct position, it calculates the Manhattan distance by taking the absolute difference between the current row and the goal row and adding it to the absolute difference between the current column and the goal column. This process is repeated for all the numbers in the state, and the sum of these distances is returned as the Manhattan distance for that state. The higher the Manhattan distance, the farther the state is from the goal state, making it a suitable heuristic for guiding search algorithms like A* towards finding an optimal solution.

addNode()

```
void addNode(struct Node* state){

                            //specifically exclusive to the IDS
    totalSearchCost++;      //increments the total nodes expanded every time the IDS call the function

    if(headNode == NULL){
        headNode = state;
        state->next = NULL;
    }else{
        state->next = headNode;
        headNode = state;
    }
}

int checkGoal(struct Node* state){
                            //checks if the state is the goal state
    int i;

    for(i=0; i<16; i++){
        if(state->state[i] != goalState[i]){
            return 0;
        }
    }
    return 1;
}
```

This function is responsible for adding a node to a linked list, specifically designed to maintain the nodes generated and explored during the iterative deepening search. The function puts the recent node to the head of the list.

addLists()

```
void addLists(struct Node* state, int change){
                                                //adds nodes to the fringe or to the closed list

struct closedList* ptr = (struct closedList*)malloc(sizeof(struct closedList));
    struct Fringe* ptr1 = (struct Fringe*)malloc(sizeof(struct Fringe));

    switch(change){
        case 0:
                totalSearchCost++;                  //increments the total nodes expanded once it's added to the closed list
                if(headCl==NULL){
                    headCl = ptr;
                    ptr->nodeAddress = state;
                    ptr->next = NULL;
                }else{
                    ptr->nodeAddress = state;
                    ptr->next = headCl;
                    headCl = ptr;
                }
                break;
        case 1:
                if(head==NULL){
                    head = ptr1;
                    ptr1->nodeAddress = state;
                    ptr1->next = NULL;
                }else{
                    ptr1->nodeAddress = state;
                    ptr1->next = head;
                    head = ptr1;
                }
                break;
    }
}
```

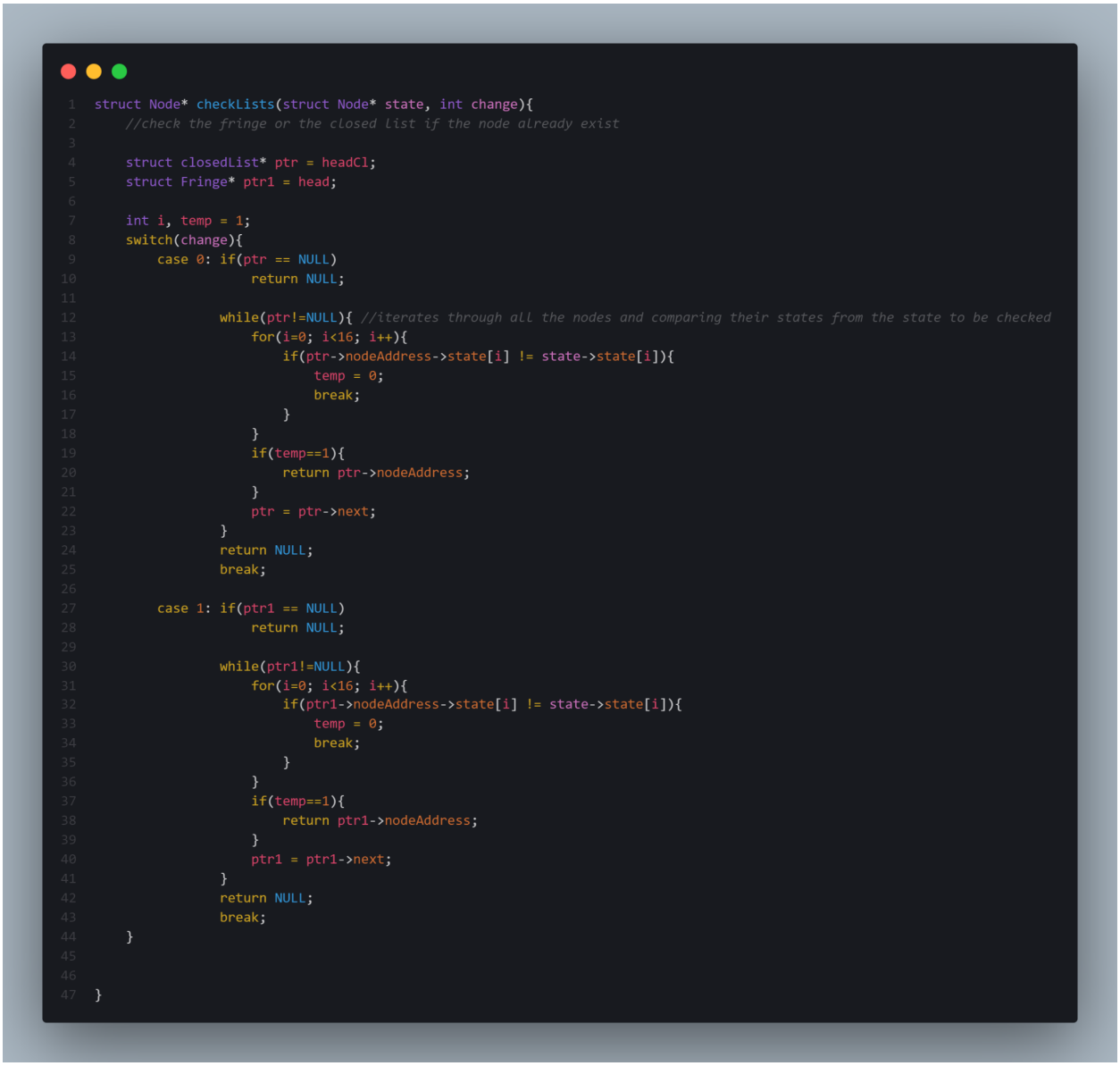
This function is responsible for adding a given node or state to one of two lists: the closed list or the fringe (open) list. These lists are integral to A* search. The closed list keeps track of states that have been explored and should not be re-explored to avoid revisiting the same states. The fringe, on the other hand, stores states that are candidates for future exploration. The addLists function receives a pointer to a state and an integer to determine whether to add it to the closed list or the fringe. If the state is added to the closed list, it indicates that the state has been explored and should not be revisited. If added to the fringe, the state becomes a candidate for future exploration. Proper management of these lists is crucial for the A* algorithm to efficiently search for the optimal solution by considering the least costly paths first and avoiding unnecessary re-exploration of states.

checkGoal()

```
int checkGoal(struct Node* state){
                              //checks if the state is the goal state
    int i;

    for(i=0; i<16; i++){
        if(state->state[i] != goalState[i]){
            return 0;
        }
    }
    return 1;
}
```

The checkGoal function compares the elements of the provided state with a predefined goal state, typically consisting of the numbers 1 through 15 (or 0 for the empty tile) arranged in ascending order. If the elements of the input state match those of the goal state, the function returns 1, indicating a successful match, and signaling that the current state is the puzzle's solved configuration. On the other hand, if any element in the state differs from the goal state, the function returns 0, indicating that the puzzle has not been solved in the current state.

checkLists()



```
struct Node* checkLists(struct Node* state, int change){
    //check the fringe or the closed list if the node already exist

    struct closedList* ptr = headCl;
    struct Fringe* ptr1 = head;

    int i, temp = 1;
    switch(change){
        case 0: if(ptr == NULL)
                    return NULL;

                while(ptr!=NULL){ //iterates through all the nodes and comparing their states from the state to be checked
                    for(i=0; i<16; i++){
                        if(ptr->nodeAddress->state[i] != state->state[i]){
                            temp = 0;
                            break;
                        }
                    }
                    if(temp==1){
                        return ptr->nodeAddress;
                    }
                    ptr = ptr->next;
                }
                return NULL;
                break;

        case 1: if(ptr1 == NULL)
                    return NULL;

                while(ptr1!=NULL){
                    for(i=0; i<16; i++){
                        if(ptr1->nodeAddress->state[i] != state->state[i]){
                            temp = 0;
                            break;
                        }
                    }
                    if(temp==1){
                        return ptr1->nodeAddress;
                    }
                    ptr1 = ptr1->next;
                }
                return NULL;
                break;
    }


}
```

The function ‘checkLists’ is used to identify whether a node/state exists in either the closed list or the fringe. This helps manage the open and closed lists used in the algorithm A* and Iterative Deepening Search. The function iterates this through respective lists and compares the states configuration with the states stored in the list. If there are matching states found within the lists, the function will return a pointer to that state, otherwise it will return ‘NULL’. This function helps prevent duplicate exploration of states and efficiently manages the progression of the search algorithm.

removeLists()

```
void removeLists(struct Node* state, int change) {
    //removes node from the fringe or the closed list
    switch (change) {
        case 0: {
            struct closedList* ptr = headCl;
            struct closedList* prev = NULL;

            if (ptr == NULL) {
                return; // Nothing to remove.
            }

            // If the head is the target state.
            if (ptr->nodeAddress == state) {
                headCl = ptr->next;
                free(ptr);
                return;
            }

            while (ptr != NULL) {
                if (ptr->nodeAddress == state) {
                    prev->next = ptr->next;
                    free(ptr);
                    return;
                }
                prev = ptr;
                ptr = ptr->next;
            }
            break;
        }
        case 1: {
            struct Fringe* ptr1 = head;
            struct Fringe* prev1 = NULL;

            if (ptr1 == NULL) {
                return; // Nothing to remove.
            }

            // If the head is the target state.
            if (ptr1->nodeAddress == state) {
                head = ptr1->next;
                free(ptr1);
                return;
            }

            while (ptr1 != NULL) {
                if (ptr1->nodeAddress == state) {
                    prev1->next = ptr1->next;
                    free(ptr1);
                    return;
                }
                prev1 = ptr1;
                ptr1 = ptr1->next;
            }
            break;
        }
    }
}
```

The ‘removeList’ function serves to remove a specified state from either a closed list or the fringe. It takes input from a state to be removed and an indicator to remove it from. The function iterates through the selected list, finds the target state, and removes it from the list, while ensuring that the data structure is updated correctly

getLowestFn()

```
struct Node* getLowestFn() {
    //gets the node that has the lowest f(n) value in the fringe
    if (head == NULL) {
        return NULL; // Return NULL if the list is empty.
    }

    struct Fringe* ptr = head;
    int temp = head->nodeAddress->fn; //initializes the first node's f(n) to be the smallest f(n)
    struct Node* state = head->nodeAddress;

    if (head->next == NULL) {
        return state;
    }

    ptr = ptr->next;
    while (ptr != NULL) {                   //iterates through the fringe to check each node's f(n)
        if (ptr->nodeAddress->fn < temp) {  // if the node's f(n) is less than the temp variable, the node's f(n)
            state = ptr->nodeAddress;       // will be the next temp value
            temp = ptr->nodeAddress->fn;
        }
        ptr = ptr->next;
    }

    return state;                           //returns the state with the lowest f(n)
}
```

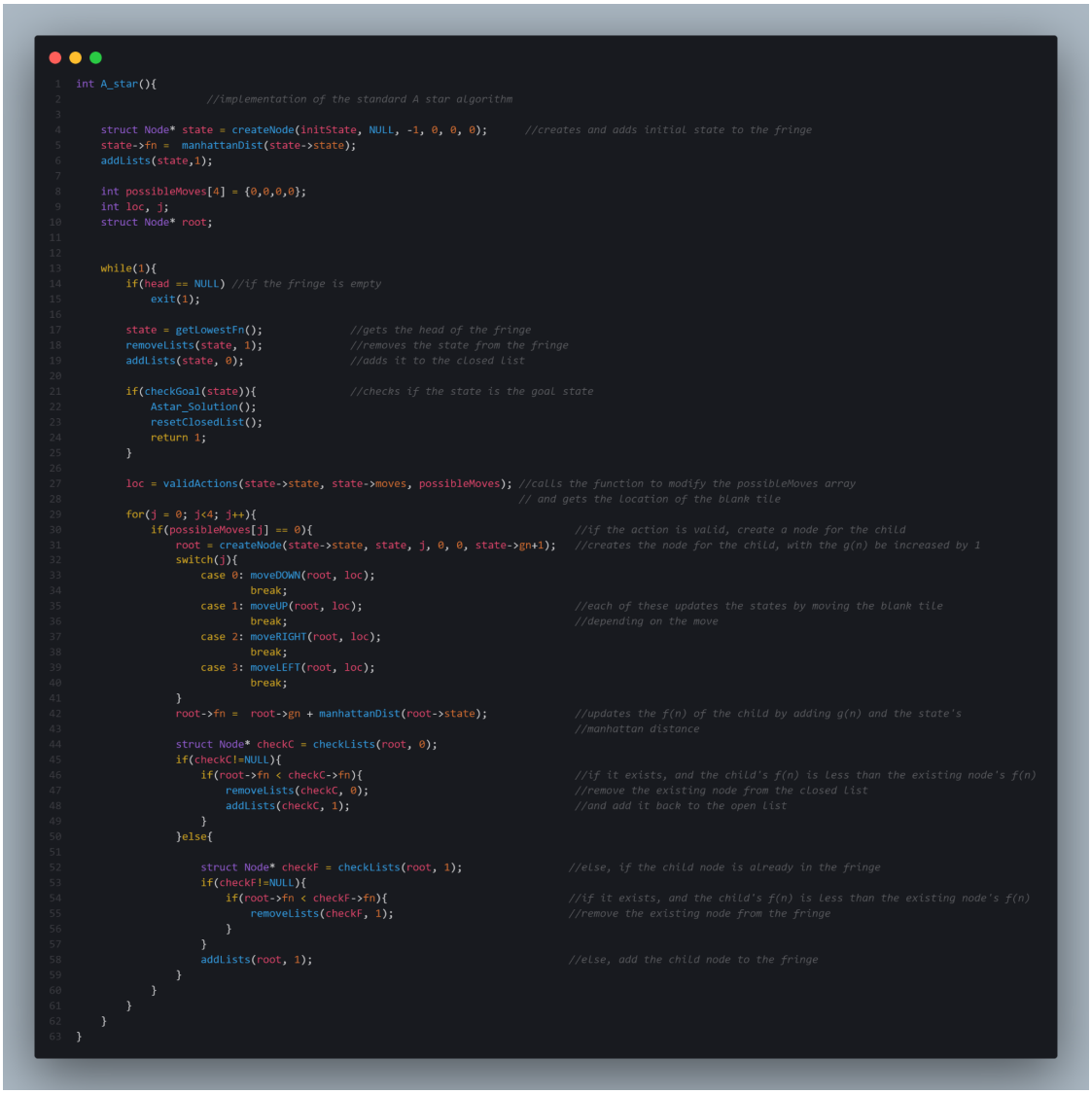
The ‘getLowestFn’ function is responsible for identifying and returning the state/node with the lowest f(n) value. In the A* search algorithm, f(n) represents the total estimated cost of reaching the goal state through the current state. This function iterates through the nodes in the fringe, comparing their f(n) values and selecting the one with the lowest f(n). By selecting the node with the lowest f(n), the A* algorithm prioritizes exploring the nodes that are expected to lead to the goal state efficiently.

Astar_Solution()

```
void Astar_Solution(){

                            //gets the solution of the A star algorithm from the closed list

    struct closedList* ptr = headCl;
    struct closedList* prev = ptr->next;

    struct solution* ptrSol = (struct solution*)malloc(sizeof(struct solution));

    if(ptrSol == NULL)
        exit(1);

    ptrSol->nodeAddress = ptr->nodeAddress;
    ptrSol->next = NULL;
    headSol = ptrSol;

    while(prev!=NULL){
        while(ptr->nodeAddress->Parent != prev->nodeAddress){       //tracks the nodes from the goal node to its parents
            prev = prev->next;
        }
        struct solution* ptrSol = (struct solution*)malloc(sizeof(struct solution));

        ptrSol->nodeAddress = prev->nodeAddress;
        ptrSol->next = headSol;
        headSol = ptrSol;

        ptr = prev;
        prev = ptr->next;
    }

}
```

The ‘Astar_Solution()’ function is responsible for generating the solution path for the A* search algorithm. It operates by reconstructing the path from the goal state back to the initial state after the A* algorithm has found the goal. The function starts by traversing the closed list, where nodes explored during the A* search are stored. It uses two pointers, ‘ptr’ and ‘prev’, to navigate the list. ‘ptr’ points to the current node, while ‘prev’ points to the next node in the closed list. As it traverses the closed list, Astar_Solution() identifies parent-child relationships between nodes by comparing the Parent attribute of nodes in the closed list. It continues this comparison until it finds the parent of the current node pointed to by ‘prev’’. For each parent-child relationship the function creates a solution node and adds it to the solution path. The process repeats until it builds a solution path from the goal state to the initial state effectively providing a step by step sequence of the moves that lead to the solution

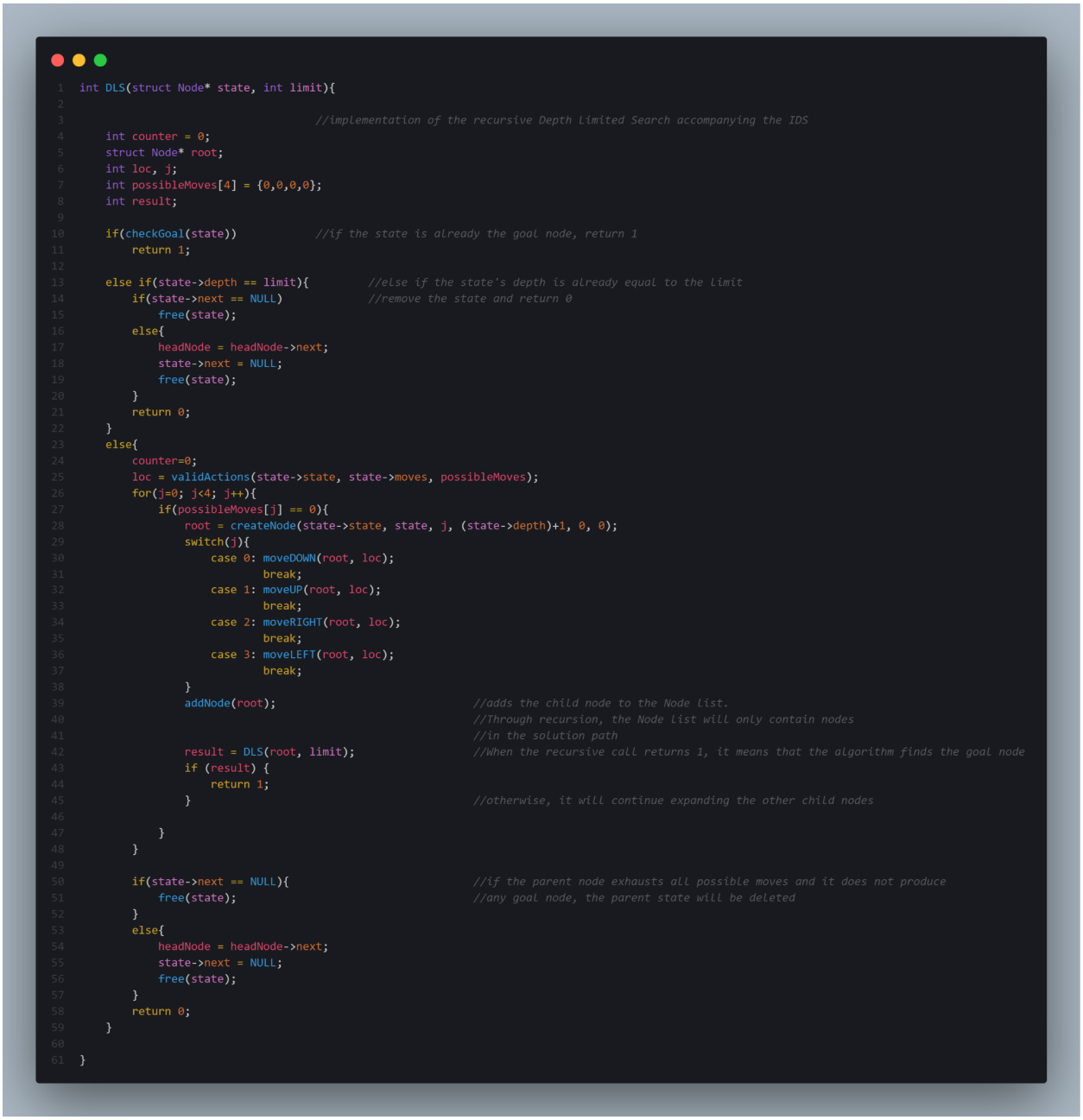
A_star()



```
int A_star(){
                    //implementation of the standard A star algorithm

    struct Node* state = createNode(initState, NULL, -1, 0, 0, 0);      //creates and adds initial state to the fringe
    state->fn =  manhattanDist(state->state);
    addLists(state,1);

    int possibleMoves[4] = {0,0,0,0};
    int loc, j;
    struct Node* root;


    while(1){
        if(head == NULL) //if the fringe is empty
            exit(1);

        state = getLowestFn();              //gets the head of the fringe
        removeLists(state, 1);              //removes the state from the fringe
        addLists(state, 0);                 //adds it to the closed list

        if(checkGoal(state)){               //checks if the state is the goal state
            Astar_Solution();
            resetClosedList();
            return 1;
        }

        loc = validActions(state->state, state->moves, possibleMoves); //calls the function to modify the possibleMoves array
                                                                       // and gets the location of the blank tile
        for(j = 0; j<4; j++){
            if(possibleMoves[j] == 0){                                  //if the action is valid, create a node for the child
                root = createNode(state->state, state, j, 0, 0, state->gn+1);   //creates the node for the child, with the g(n) be increased by 1
                switch(j){
                    case 0: moveDOWN(root, loc);
                            break;
                    case 1: moveUP(root, loc);                          //each of these updates the states by moving the blank tile
                            break;                                      //depending on the move
                    case 2: moveRIGHT(root, loc);
                            break;
                    case 3: moveLEFT(root, loc);
                            break;
                }
                root->fn =  root->gn + manhattanDist(root->state);              //updates the f(n) of the child by adding g(n) and the state's
                                                                                //manhattan distance
                struct Node* checkC = checkLists(root, 0);
                if(checkC!=NULL){
                    if(root->fn < checkC->fn){                                  //if it exists, and the child's f(n) is less than the existing node's f(n)
                        removeLists(checkC, 0);                                 //remove the existing node from the closed list
                        addLists(checkC, 1);                                    //and add it back to the open list
                    }
                }else{

                    struct Node* checkF = checkLists(root, 1);                 //else, if the child node is already in the fringe
                    if(checkF!=NULL){
                        if(root->fn < checkF->fn){                             //if it exists, and the child's f(n) is less than the existing node's f(n)
                            removeLists(checkF, 1);                            //remove the existing node from the fringe
                        }
                    }
                    addLists(root, 1);                                         //else, add the child node to the fringe
                }
            }
        }
    }
}
```

The ‘A_star()’ function implements the A* search algorithm to find the optimal solution for a given problem or puzzle. It begins by creating the initial state node, setting up the initial values and adding it to the fringe, then it enters a loop that continues to search until a solution is found . In each iteration it selects the node with the lowest f(n) from the fringe and removes it from the fringe marking it as closed. The function checks if the selected node is the goal state, if so it then constructs a solution path, if not, it generates child nodes for the current state, calculates their costs, and adds them to the fringe if they are not already in the closed list. The function also uses the Manhattan distance as a heuristic to estimate the cost to reach the goal state from a given state. This continues on until an optimal solution and goal state is found

DLS()



```
int DLS(struct Node* state, int limit){

                                    //implementation of the recursive Depth Limited Search accompanying the IDS
    int counter = 0;
    struct Node* root;
    int loc, j;
    int possibleMoves[4] = {0,0,0,0};
    int result;

    if(checkGoal(state))            //if the state is already the goal node, return 1
        return 1;

    else if(state->depth == limit){         //else if the state's depth is already equal to the limit
        if(state->next == NULL)             //remove the state and return 0
            free(state);
        else{
            headNode = headNode->next;
            state->next = NULL;
            free(state);
        }
        return 0;
    }
    else{
        counter=0;
        loc = validActions(state->state, state->moves, possibleMoves);
        for(j=0; j<4; j++){
            if(possibleMoves[j] == 0){
                root = createNode(state->state, state, j, (state->depth)+1, 0, 0);
                switch(j){
                    case 0: moveDOWN(root, loc);
                            break;
                    case 1: moveUP(root, loc);
                            break;
                    case 2: moveRIGHT(root, loc);
                            break;
                    case 3: moveLEFT(root, loc);
                            break;
                }
                addNode(root);                          //adds the child node to the Node list.
                                                        //Through recursion, the Node list will only contain nodes
                                                        //in the solution path
                result = DLS(root, limit);              //When the recursive call returns 1, it means that the algorithm finds the goal node
                if (result) {
                    return 1;
                }                                       //otherwise, it will continue expanding the other child nodes

            }
        }

        if(state->next == NULL){                        //if the parent node exhausts all possible moves and it does not produce
            free(state);                                //any goal node, the parent state will be deleted
        }
        else{
            headNode = headNode->next;
            state->next = NULL;
            free(state);
        }
        return 0;
    }

}
```

The recursive function 'DLS()' is to perform a depth limited search on the state space of the puzzle. The function takes two parameters: a state node representing the current state being explored and a limit parameter that restricts the search depth. The depth limit is increased incrementally during the iterative deepening search (IDS) process. If the current state is the goal state, the function returns 1, indicating success. This means that the goal state has been found within the depth limit. If the depth of the current state reaches the specified limit without finding the goal state, the function returns 0, indicating that the search failed at this depth. In this case, the function may

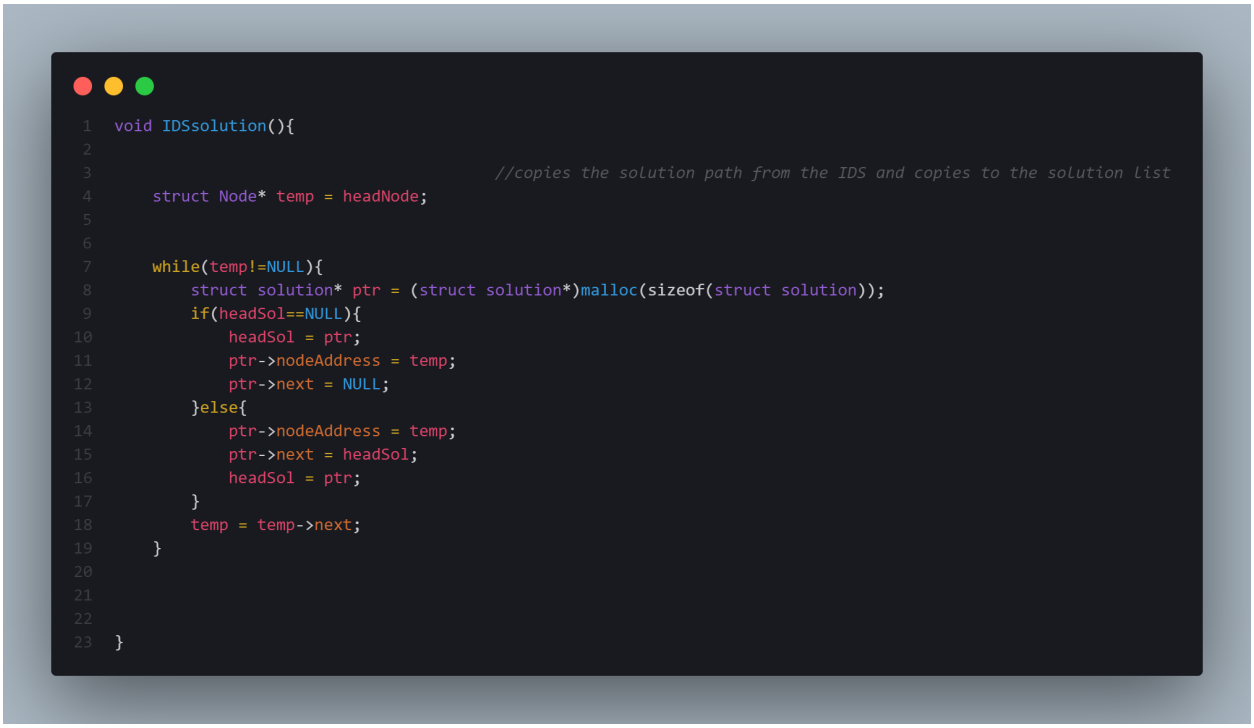
remove the current state from the list of nodes being explored to free up memory. When neither of the conditions are met, the function explores all possible moves from the current state within the depth limit. It creates child nodes for each valid move, adds them to the list of nodes to be explored, and then recursively calls DLS() on each child node to explore deeper into the state space.

iterativeDeepeningSearch()

```
void iterativeDeepeningSearch(){

                                //implements the Iterative Deepening Search
    int limit = 0;
    int counter = 1;
    int result;

    while(counter){                 //depth limit will start by 0

        struct Node* root = createNode(initState, NULL, -1, 0, 0, 0);
        addNode(root);
        result = DLS(root, limit);      //calls the DLS
        if(result)
            counter = 0;
        else{                           //if the DLS does not find a goal, the limit will increase by one
            headNode = NULL;
            limit++;
        }
    }
}
```

The 'iterativeDeepeningSearch()' is a combination of the benefits depth-first and breadth-first search. It starts with a depth limit of 0 and performs a depth-limited search at each iteration, gradually increasing the depth limit. In each DLS, the algorithm explores nodes up to the specified depth limit, searching for a solution. If a solution is not found at the current depth limit, the depth limit is increased, and the search continues. This process guarantees that the algorithm explores the entire search space while systematically increasing the depth of its search. When a solution is found, the function constructs the solution path and stores it in the ‘headSol’ linked list, providing the optimal solution to the problem. IDS ensures that it finds the most cost-effective solution while maintaining memory efficiency.

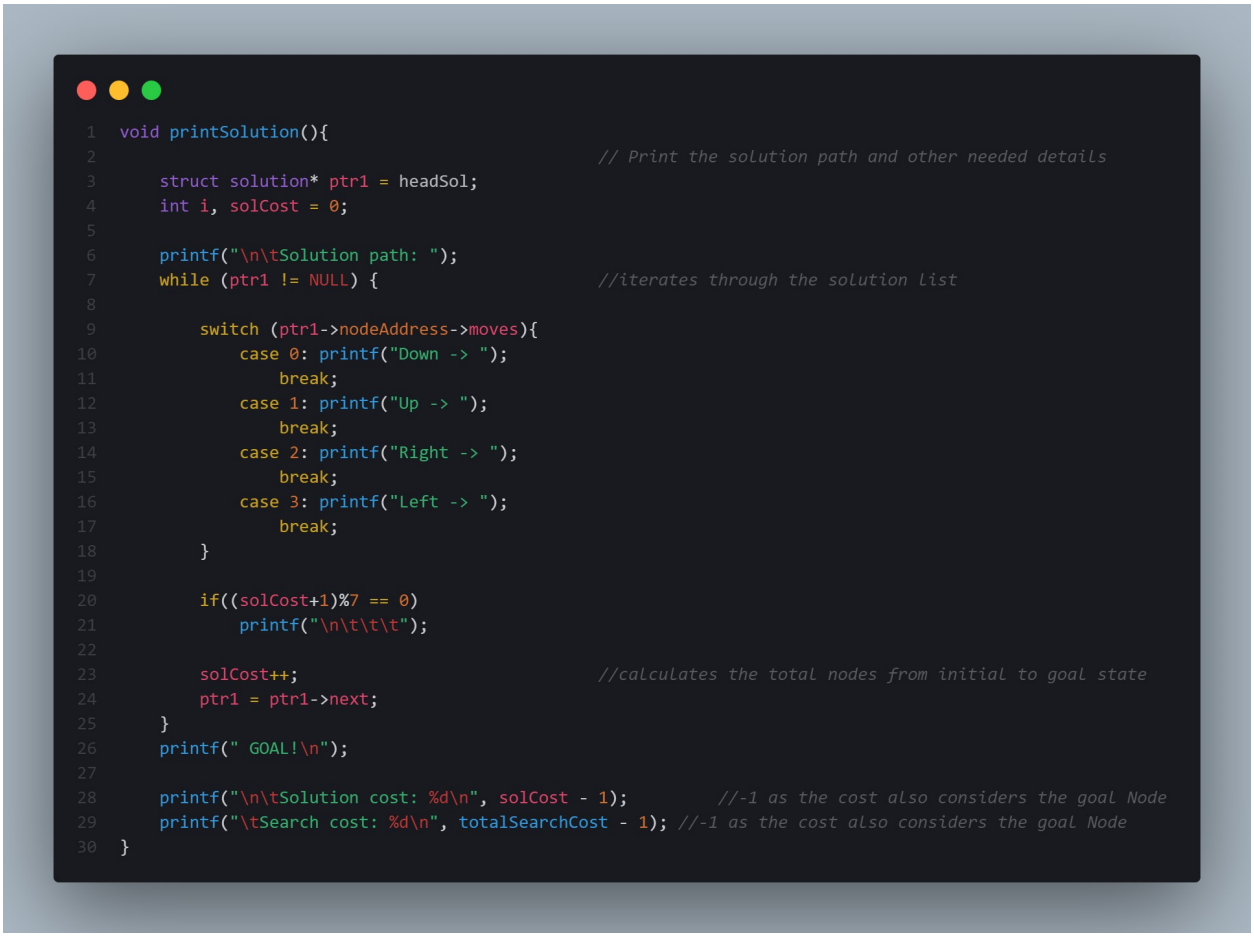
IDSsolution()



```
void IDSsolution(){

                                        //copies the solution path from the IDS and copies to the solution list
    struct Node* temp = headNode;


    while(temp!=NULL){
        struct solution* ptr = (struct solution*)malloc(sizeof(struct solution));
        if(headSol==NULL){
            headSol = ptr;
            ptr->nodeAddress = temp;
            ptr->next = NULL;
        }else{
            ptr->nodeAddress = temp;
            ptr->next = headSol;
            headSol = ptr;
        }
        temp = temp->next;
    }


}
```

The function 'IDSsolution()' is responsible for copying the solution path found by the IDS algorithm and storing it in a linked list called 'headSol' It first initializes a pointer called 'ptr1' to traverse the linked list of nodes generated by the IDS process, where each node is representing a state in the solution path. Inside a loop, it allocates memory for a new solution node 'ptr', and copies the address of the node from the IDS solution path into 'ptr'. This effectively duplicates the solution path nodes into the 'headSol' linked list. The new solution node, 'ptr', is then added to the 'headSol' linked list. The loop continues to copy each node from the IDS solution path into the headSol linked list until all nodes in the path are duplicated.

printSolution()



```
void printSolution(){
                                                // Print the solution path and other needed details
    struct solution* ptr1 = headSol;
    int i, solCost = 0;

    printf("\n\tSolution path: ");
    while (ptr1 != NULL) {                      //iterates through the solution list

        switch (ptr1->nodeAddress->moves){
            case 0: printf("Down -> ");
                break;
            case 1: printf("Up -> ");
                break;
            case 2: printf("Right -> ");
                break;
            case 3: printf("Left -> ");
                break;
        }

        if((solCost+1)%7 == 0)
            printf("\n\t\t\t");

        solCost++;                              //calculates the total nodes from initial to goal state
        ptr1 = ptr1->next;
    }
    printf(" GOAL!\n");

    printf("\n\tSolution cost: %d\n", solCost - 1);         //-1 as the cost also considers the goal Node
    printf("\tSearch cost: %d\n", totalSearchCost - 1); //-1 as the cost also considers the goal Node
}
```

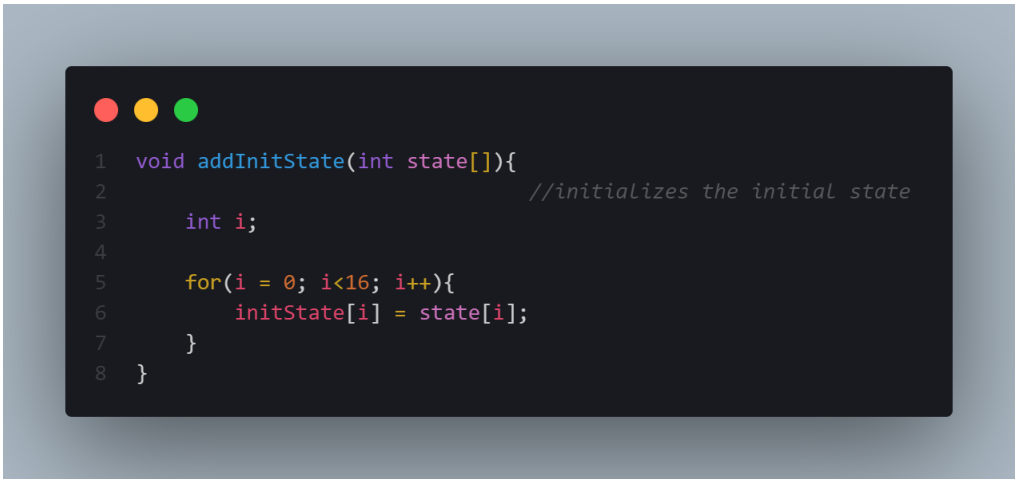
It begins by initializing a pointer ptr1 to the head of a linked list of solution nodes. The code then declares and initializes an integer variable solCost to 0, which will represent the total cost of the solution path, i.e., the number of moves required to reach the goal state from the initial state. Inside a loop that iterates through the linked list of solution nodes, the code uses a switch statement to determine and print the move direction associated with each solution node, such as "Down," "Up," "Right," or "Left." To enhance readability, the code checks if the number of moves (solCost + 1) is a multiple of 7 and adds extra indentation and a newline when necessary. Finally, it prints the solution cost (minus 1, as it considers the goal node) and the search cost (also minus 1, as it includes the goal node) as well as indicating the completion of the solution path with "GOAL!" at the end.

resetSolution()

```
void resetSolution(){
                                        //the function removes all the node from the solution of IDS
                                        //so that it can be used again by the A*
    struct solution* ptr = headSol;

    while(headSol!=NULL){
        if(ptr->next == NULL){
            free(ptr);
            headSol = NULL;
        }else{
            headSol = headSol->next;
            ptr->next = NULL;
            free(ptr);
        }
        ptr = headSol;
    }

}
```

As with the resetClosedList() function, this function resets or deallocates all the nodes stored in the solution list. So that it would be used again by the next search algorithm.

addInitState()



```
void addInitState(int state[]){
                                    //initializes the initial state
    int i;

    for(i = 0; i<16; i++){
        initState[i] = state[i];
    }
}
```

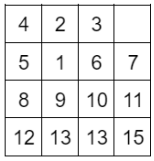
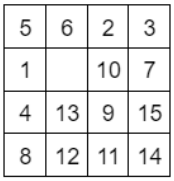
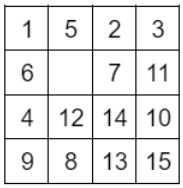
This function takes an integer array, state, as its argument and initializes the initial state of the puzzle. It does so by using a for loop to iterate through the state array, which represents the current state of the puzzle, with 16 elements corresponding to the 16 tiles on the 4x4 grid. Inside the loop, it copies the value of each element in the state array into a separate array called initState, which is used to store the initial state of the puzzle.

printAnimation()

```
void printAnimation(){
                                //prints a sample animation movement from initial state to goal state
    int i;
    struct solution* ptr1 = headSol;
        while(ptr1!=NULL){

        system("clear");
        printf("\n\n\t\t");
        printf(" ------------------------------\n\t\t");
        for(i=0; i<16; i++){
            if((i)%4 == 0 && i!=0)
                printf(" \n\t\t ------------------------------\n\t\t");
            if(ptr1->nodeAddress->state[i]==0)
                printf(" | __ |");
            else
                printf(" | - | ", ptr1->nodeAddress->state[i]);
        }
        printf("\n\t\t ------------------------------\n\t\t");

        printf("\n\n");
        ptr1 = ptr1->next;
        sleep(1);  // Sleep for 1 second

    }
}
```

It operates by iterating through a linked list of solution steps (struct solution*) stored in the program, and for each step, it clears the console screen using the "clear" command, presenting a visually formatted representation of the puzzle grid. It uses nested loops to iterate through the puzzle's 4x4 grid, displaying the tile numbers or an empty space. The grid is separated into cells with borders, making it visually clear and easy to follow. The function updates the screen after each puzzle move by using the system("clear") command and introduces a brief pause with sleep(1) to create an animation effect, allowing users to follow the step-by-step solution process. This function aids in visualizing and understanding the puzzle-solving algorithm's progress by presenting an animated demonstration of how the puzzle evolves from its initial state to the final solved state.

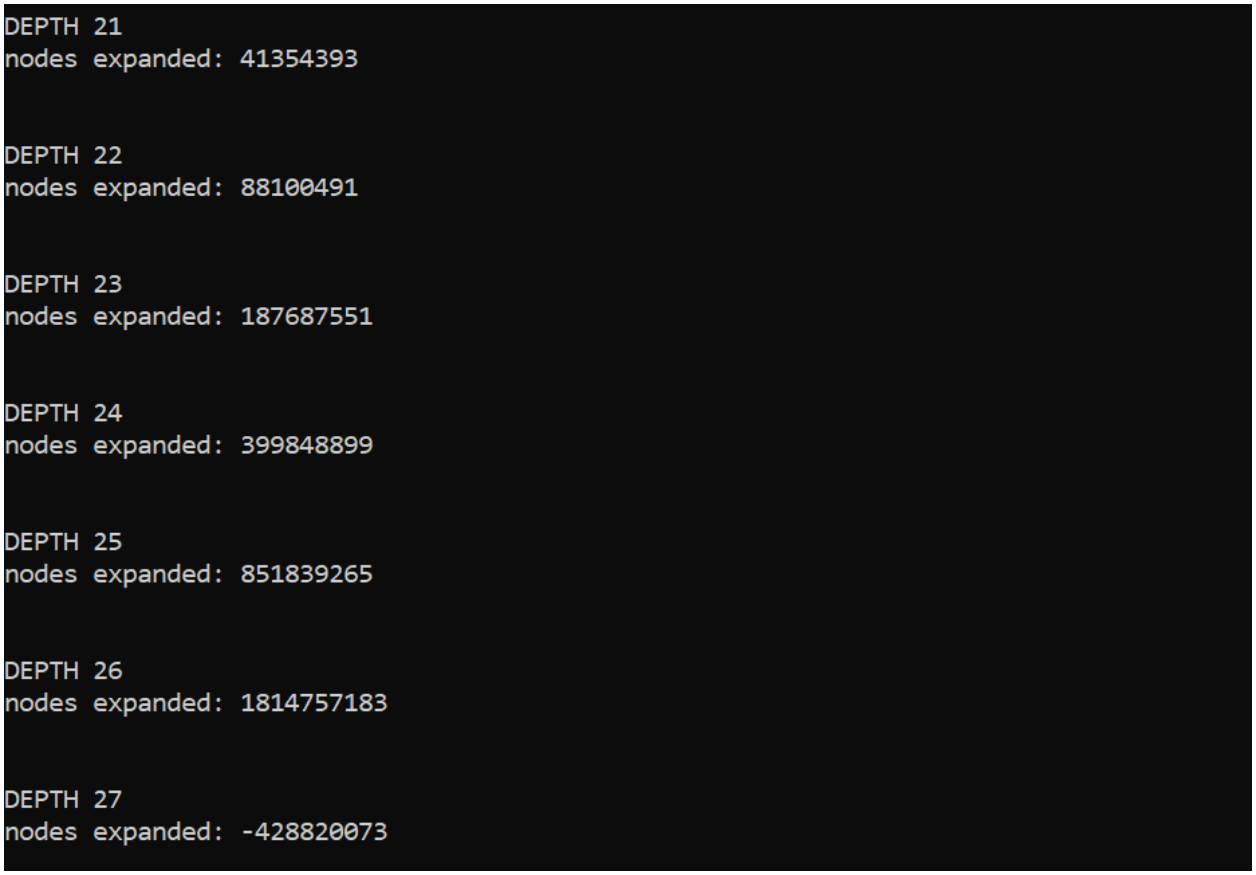
Analysis and Comparison

Initial State		IDS	A*
Easy 	Solution Path	Left - Left - Down - Left - Up	Left - Left - Down - Left - Up
	Solution Cost	5	5
	Number of Nodes Expanded	159	6
	Running Time	<1 millisecond*	<1 millisecond*
Medium 	Solution Path	Up - Left - Down - Down - Down - Right - Up - Right - Down - Right - Up - Left - Up - Left - Up - Left	Up - Left - Down - Down - Down - Right - Up - Right - Down - Right - Up - Left - Up - Left - Up - Left
	Solution Cost	16	16
	Number of Nodes Expanded	858,668	138
	Running Time	0.14 seconds	0.001 seconds
Hard 	Solution Path	Left - Down - Down - Right - Up - Left - Down - Right - Right - Up - Right - Up - Left - Left - Up - Left	Left - Down - Right - Down - Left - Up - Right - Down - Right - Up - Right - Up - Left - Left - Up - Left
	Solution Cost	16	16
	Number of Nodes Expanded	1,117,627	43
	Running Time	0.171 seconds	0.000274 seconds

<table>
<tr><td rowspan="4">Worst
<table><tr><td>13</td><td>8</td><td>4</td><td>6</td></tr><tr><td>14</td><td>12</td><td>3</td><td></td></tr><tr><td>15</td><td>11</td><td>5</td><td>7</td></tr><tr><td>9</td><td>10</td><td>2</td><td>1</td></tr></table></td><td>Solution Path</td><td>**</td><td>**</td></tr>
<tr><td>Solution Cost</td><td>**</td><td>**</td></tr>
<tr><td>Number of Nodes Expanded</td><td>**</td><td>**</td></tr>
<tr><td>Running Time</td><td>>2 days**</td><td>>2 days**</td></tr>
<tr><td rowspan="4">Preferred Initial Configuration
<table><tr><td>1</td><td>4</td><td>5</td><td>2</td></tr><tr><td>14</td><td>6</td><td>9</td><td>3</td></tr><tr><td>8</td><td>12</td><td>11</td><td>7</td></tr><tr><td>13</td><td>15</td><td></td><td>10</td></tr></table></td><td>Solution Path</td><td>Right - Up - Up - Left - Left - Down -Right - Up - Up - Right - Right - Down - Down - Down - Left - Left - Up - Right - Right - Down - Left - Left - Left - Up - Up - Right - Up - Left</td><td>Right - Up - Up - Left - Left - Down -Right - Up - Up - Right - Right - Down - Down - Down - Left - Left - Up - Right - Right - Down - Left - Left - Left - Up - Up - Right - Up - Left</td></tr>
<tr><td>Solution Cost</td><td>28</td><td>28</td></tr>
<tr><td>Number of Nodes Expanded</td><td>2,121,123,535</td><td>9,127</td></tr>
<tr><td>Running Time</td><td>1204.603762 seconds</td><td>3.781968 seconds</td></tr>
</table>

*On the terminal output, the running time was 0.000000 seconds. As the algorithm technically does not consume time when solving the puzzle, where it is just too fast to be registered, the <1 milliseconds was chosen to be more appropriate.

The worst state proves to be unsolvable using our code. The testing was the last thing we did, and we did it on October 30th. The worst case was tested on the Iterative Deepening Search on the same day at approximately 1 in the afternoon. We got anxious as midnight came and the program had not finished. With debug printing statements*, we saw that the IDS was already at depth 33. The program was left until the morning and at 3 in the afternoon, November 1, the program was just at depth 35. As it has been two days, of which the laptop used was also at 100% CPU utilization, we decided to terminate the program. A star algorithm was next to be tested in the afternoon, but it also took more than a day, November 2, the program was yet to be finished. Thus, we collectively decided that the program cannot solve the problem within the limited time we had.



*** sample debug statements in IDS solving the worst state, where at depth 26, the total nodes expanded were 1,814,757,183.

Contributions of Each Member

Research for Construction of the Program: Herrera, Tabug
Code Implementation: Herrera, Tabug, Cleofe, Deduque
Debugger: Herrera, Tabug, Cleofe
Code Tester: Herrera, Tabug, Cleofe
Explained the functions: Deduque, Bron, Herrera