Programming Project 1

Written Report.

Submitted by:

Daphne Julienne Agao
Jade Airin Judin
Judith Balatinsayo
Neslie Marie Colasito
Juliet Clarisse Bertulfo

BS Computer Science 3A

Table of Contents

I. Introduction

The “15 Puzzle Solver” is a C-based command-line program that tackles the popular 15-puzzle game. The 15-puzzle is a classic sliding tile puzzle where the objective is to rearrange a 15 tile within a 4x4 grid, aiming to reach a predefined target configuration by sliding the tiles into an empty space.

Key Features:

- **Reconfiguration:** You can reconfigure the puzzle board by specifying your own initial state, setting the stage for the puzzle-solving adventure.
- **Iterative Deepening Search (IDS):** The program employs Iterative Deepening Search, a depth-limited search strategy, to find a solution. It explores various depth levels, incrementally seeking the most efficient path to victory.
- **A* Search:** A popular informed search algorithm is utilized to optimize the solution-finding process. It prioritizes states based on an estimated cost to reach the goal state.
- **User Interaction:** the program offers a console-based interface where you can make choices, observe the solution process, and receive detailed information about the solution path, cost and execution time.

Experience the challenge and satisfaction of solving the 15-puzzle with the “15-Puzzle Solver” program. Try different initial configurations, explore the depths of Iterative Deepening Search, and optimize your strategy with A* Search. Enjoy the journey of mastering this classic puzzle!

II. Header file and predefined Functions

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include <stdbool.h>
#define BLANK_CHARACTER '0'
```

1. `#include` Directives:
   - `#include <stdio.h>`: This directive includes the standard input/output library, which provides functions like “`printf`” and “`scanf`” for reading and writing data to and from the console.

- `#include <stdlib.h>`: This directive includes the standard library, which contains functions for memory allocation (`malloc`, `free`), random number generation (`rand`, `srand`), and other utility functions.
- `#include <time.h>`: This directive includes the time and date library, which is used for working with time-related functions and structures.
- `#include <stdbool.h>`: This directive includes a standard header that defines the "`bool`" type and values "`true`" and "`false`" to work with boolean logic.

2. `#define` Directive:
    - `#define BLANK_CHARACTER '0'`: This directive defines a macro named "`BLANK_CHARACTER`" with the character value `'0'`. It essentially assigns the character '0' to the identifier "`BLANK_CHARACTER`", making it easier to reference and modify throughout the code.

III. User-Defined Functions

```
int main() {
    int choice;
    int limit;
    State initial;       //initial board state
    State goal;          //goal board configuration

    SolutionPath *ids; //solution path of each search method
    SolutionPath *astar;

    do {
        system("cls||clear"); // Clear the console screen.
        printf("  _  ___        ___                _        ___       _   
\n");
        printf(" / || __| ___ | _ \\ _  _  ___ ___| | ___  / __| ___ | |__
__ ___  _ _ \n");
        printf(" | ||__ \\|___||  _/| || ||_ /|_ /| |/ -_) \\__ \\/ _ \\|
|\\ V // -_)| '_|\n");
        printf(" |_||___/      |_|   \\_,_|/__|/__||_|\\___| |___/\\___/|_|
\\_/ \\___||_|  \n");
        printf("
\n");

printf("------------------------------------------------------------------\n"
);
```

```c
        printf("|0.) Reconfigure the Board\t\t\t\t\t|\n");
        printf("|1.) Iterative Deepening Search\t\t\t\t\t|\n");
        printf("|2.) A* Search\t\t\t\t\t\t\t|\n");
        printf("|3.) End Program\t\t\t\t\t\t|\n");

printf("-----------------------------------------------------------------\n"
);
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 0:
                  printf("\nReconfiguring the puzzle board...");
                inputInitialState(&initial);
                inputgoalState(&goal);
                printf("Board reconfigured successfully.\n");
                printf("\n");
                system("pause");
                break;
            case 1:
                if (statesMatch(&goal, &initial)){
                    printf("No moves needed. The initial state is already
the goal state.\n");
                }
                        for (limit = 1; limit < 1000000; limit++) {
                    nodesExpanded = 0;
                    nodesGenerated = 0;
                    solnLength = 0;
                    runtime = 0;

                    SolutionPath *ids = IDSearch(&initial, &goal, limit);
                    printf("\n----------- USING IDS ALGORITHM DEPTH LIMIT =
%d -----------\n", limit);

                    printSolution(ids);
                    if (solnLength) {
                        limit = 1000000;
                        printf("IDS completed successfully.\n");
                    }
                    else {
                        printf("IDS did not find a solution.\n");
                    }
                    destroySolution(&ids);
```

```
                    }
                system("pause");
                break;

                  case 2:
                if (statesMatch(&goal, &initial)){
                    printf("No moves needed. The initial state is already
the goal state.\n");
                }
                        printf("\n------------------ USING A* ALGORITHM
---------------------- \n");
                  nodesExpanded = 0;
                  nodesGenerated = 0;
                  solnLength = 0;
                  runtime = 0;
                  SolutionPath *astar = AStarSearch(&initial, &goal);

                  printSolution(astar);

                  if (solnLength) {
                        printf("A* Search completed successfully.\n");
                  } else {
                        printf("A* Search did not find a solution.\n");
                  }

                  destroySolution(&astar);
                  system("pause");
                  break;
            case 3:
                  printf("\nYey! You're done! :)\n");
                return 0;  // Exit the program
            default:
                printf("Invalid choice. Please enter a valid menu
option.\n");
                system("pause");
                break;
        }
    } while (choice != 3);

    return 0;
}
```

The `main()` function:

1. **Initialization:** The main function begins by declaring several variables, including "`choice`" for user input, "`depth_limit`" for controlling the maximum depth in IDS, "`initial`" and "`goal`" to store the puzzle's initial and goal states, and a pointer "idds" for the solution path of IDS.
2. **Menu and User Input:** The program enters a "`do-while`" loop where it displays a menu on the console, presenting the user with various options.
3. **User Choice**: Based on the user's choice, the program executes different actions.
    - **Choice 0:** Reconfigures the puzzle board. It clears the screens, calls "`inputInitialState`" and "`inputgoalState`" function to input the initial and goal states, and informs the user about the successful reconfiguration.
    - **Choice 1:** Initiates the IDS algorithm. It runs IDS with increasing depth limits until a solution is found(or a predefined limit is reached). It displays information about the current depth limit, prints the solution if found, and provides a message indicating whether IDS completed successfully or didn't find a solution.
    - **Choice 2:** Similar to IDS, this section initiates the A* Search algorithm. It tracks the same performance metrics and displays the results, indicating whether A* Search successfully found a solution or not.
    - **Choice 3:** Exits the program with a farewell message.
4. **Loop Continuation:** The program keeps running as long as the user's choice is not equal to 3(to exit the program).
5. **Return and Termination:** Once the user selects the option to exit (choice 3), the program displays a closing message and return 0, indicating successful termination.

```
unsigned int nodesExpanded;  //number of expanded nodes
unsigned int nodesGenerated; //number of generated nodes
unsigned int solnLength; //number of moves in solution
double runtime;              //elapsed time (in milliseconds)

typedef struct Node Node;
typedef struct NodeList NodeList;
typedef struct State State;
typedef enum Move Move;
typedef struct SolutionPath SolutionPath;
```

This part of the code defines several global variables and data structures that are used in the 15-puzzle solving program:

1. `unsigned int nodesExpanded`: This variable is used to keep track of the number of nodes expanded during the search process.
2. `unsigned int nodesGenerated`: This variable is used to count the number of nodes generated during the search process.

3. `unsigned int solnLength`: This variable is used to store the number of moves in the solution path once a solution is found.
4. `double runtime`: This variable measures the elapsed time in milliseconds for the execution of specific search algorithms.
5. `typedef struct Node Node`: This line defines a structure named “Node’, which represents a node in the search tree. It contains information about the depth of the node, its heuristic cost, the current state of the puzzle, a reference to its parent node, and a list of a child nodes.
6. `typedef struct NodeList NodeList`: This line defines a structure named “NodeList”, which represents a list of nodes. It includes information about the number of nodes in the list and pointers to the head and tail nodes in the linked list.
7. `typedef struct State State`: This structure represents the state of the 15-puzzle. It includes the action that led to this state and the configuration of the puzzle board.
8. `typedef enum Move Move`: This enumeration defines possible moves in the 15-puzzle, including UP, DOWN, LEFT, RIGHT, nad NOT_APPLICABLE. These moves are used to update the puzzle’s state during the search.
9. `typedef struct SolutionPath SolutionPath`: This structure represents the solution path, storing the sequence of moves required to solve the puzzle. It includes information about the action taken and a reference to the next move in the path.

```
void inputInitialState(State * const state);
void inputgoalState(State * const state);
void printBoard(char const board[][4]);
void printSolution(struct SolutionPath *path) ;
void destroySolution(SolutionPath **list);
char pushNode(Node *node, NodeList** const list);
Node* popNode(NodeList** const list);
Node* popNode_head(NodeList** const list);
void pushList(NodeList **toAppend, NodeList *list);
Node* createNode(unsigned int d, unsigned int h, State *s, Node *p);
void destroyTree(Node *node);
NodeList* getChildren(Node *parent, State *goalState);
State* createState(State *state, Move move);
void destroyState(State **state);
int manhattanDist(State * const curr, State * const goal);
void pushListInOrder(NodeList **toAppend, NodeList *list);
char statesMatch(State const *testState, State const *goalState);
```

This part of the code defines a set of functions and utilities used for various purposes within the 15-puzzle solving program:

1. `void inputInitialState(State * const state);`: This function is responsible for taking user input to configure the initial state of the 15-puzzle. It updates the “`State`” structure with the user-defined puzzle configuration.
2. `void inputgoalState(State * const state);`: Similar to the “`inputInitialState`” function, this one is used to configure the goal state of the puzzle. It allows the user to specify the desired end configuration.
3. `void printBoard(char const board[][4]);`: This function is used to print the current state of the puzzle board. It takes a 2D character array representing the board configuration and displays it on the console.
4. `void printSolution(struct SolutionPath *path);`: This function is responsible for printing the solution path, which is a sequence of moves required to solve the puzzle. It takes a “`SolutionPath`” structure as input and displays the moves to the console.
5. `void destroySolution(SolutionPath **list);`: This function is used to release the memory occupied by the solution path. It deallocates the memory used for the linked list of moves.
6. `char pushNode(Node *node, NodeList** const list);`: This function is used to push a node onto a node list (linked list of nodes). It returns a character indicating success or failure(usually ‘0’ for success and ‘1’ for failure).
7. `Node* popNode(NodeList** const list);`: This function pops a node from the end of a node list. It returns a pointer to the popped node and updates the list accordingly.
8. `Node* popNode_head(NodeList** const list);`: This function pops a node from the head (beginning) of a node list. It returns a pointer to the popped node and updates the list accordingly.
9. `void pushList(NodeList **toAppend, NodeList *list);`: This function is used to merge two node lists. It takes a pointer to the destination list(‘to Append’) and a source list(‘list’) and appends the nodes from the source list to the destination list.
10. `Node* createNode(unsigned int d, unsigned int h, State *s, Node *p);`: This function is responsible for creating a new node in the search tree. It takes parameters for depth, heuristic value, state, and parent node and returns a pointer to the newly created node.
11. `void destroyTree(Node *node);`: This function is used to release the memory occupied by the entire search tree starting from the specified node. It recursively deallocates the memory.
12. `NodeList* getChildren(Node *parent, State *goalState);`: This function generates child nodes of a parent node based on the possible moves from the current state. It returns a node list containing the child nodes.
13. `State* createState(State *state, Move move);`: This function is used to create a new state by applying a move to an existing state. It returns a pointer to the new state.
14. `void destroyState(State **state);`: This function deallocates the memory used by a state structure.

15. `int manhattanDist(State * const curr, State * const goal);`: This function calculates the Manhattan distance between the current state and the goal state. The Manhattan distance is a heuristic used to estimate the cost of reaching the goal from the current state.
16. `void pushListInOrder(NodeList **toAppend, NodeList *list);`: This function pushes nodes into a node list while maintaining a specific order, often based on their heuristic values.
17. `char statesMatch(State const *testState, State const *goalState);`: This function is used to check if two states match, including that current state has reached the goal state.

```
typedef enum Move {
    UP, DOWN, LEFT, RIGHT, //values for moving up, down, left, right,
respectively
    NOT_APPLICABLE         //value assigned for initial and goal input
states
} Move;

typedef struct State {
    Move action;           //action that resulted to `this` board state
   char board[4][4];      //resulting board configuration after applying
action
} State;

typedef struct ListNode {
    Node *currNode;
    struct ListNode *prevNode; //the node before `this` instance
    struct ListNode *nextNode; //the next node in the linked list
} ListNode;

typedef struct SolutionPath {
    Move action;
    struct SolutionPath *next;
} SolutionPath;
```

The program defines the following user-defined data structures that represent the board's information.

1. `typedef enum Move` uses typedef enum which defines a new data type named move. This represents the different actions a state can have and will be used in generating the solution path.

- `UP, DOWN, LEFT, RIGHT,`: constants that indicates the values for moving up, down, left, and right
- `NOT_APPLICABLE`: constant that indicates the value assigned for initial and goal input states.

2. `typedef struct State` represents the state of the board in the puzzle game. It involves the following:
   - `Move action`: an enum that indicates the action performed to transition to the current or resulting state of the board.
   - `Char board[4][4]`: a two-dimensional character array of size 4x4 that represents the resulting board configuration.
3. `Typedef struct ListNode` represents a list of nodes that will also indicate the number of nodes in the list. The structure involves three pointers:
   - `Node* currNode`: which is a pointer to a Node structure that represents the current node in the list
   - `struct ListNode *prevNode`: which is a pointer to another ListNode structure that represents the previous node in the list
   - `Struct ListNode *nextNode`: which is a pointer to another ListNode structure that represents the next node in the list
4. `Typedef struct SolutionPath` represents the solution path for the given puzzle problem. This includes the following:
   - `Move action`: contains the correct actions performed to reach the goal state
   - `Struct SolutionPath *next`: a pointer that references the next action taken in the path.

```
struct Node {
    unsigned int depth; //depth of the node from the root. For A* search,
                        //this will also represent the node's path cost
    unsigned int hCost; //heuristic cost of the node
    State *state;       //state designated to a node
    Node *parent;       //parent node
    NodeList *children; //list of child nodes
};

struct NodeList {
    unsigned int nodeCount;    //the number of nodes in the list
    ListNode *head;            //pointer to the first node in the list
    ListNode *tail;            //pointer to the last node in the list
};
```

These defines two essential data structures and two important functions used in 15-puzzle program:

1. `struct Node`: This structure represents a node in the search tree used for search algorithms like ID-DFS and A*. It contains the following fields:
   - `unsigned int depth`: Represents the depth of the node from the root of the search tree. In A* search, this also represents the node's path cost.
   - `unsigned int hCost`: Represents the heuristic cost of the node. The heuristic cost is an estimate of the cost to reach the goal state from the current node.
   - `State *state`: Points to the state associated with this node.
   - `Node *parent`: Points to the parent node of the current node in the search tree.
   - `NodeList *children`: Represents a list of child nodes derived from this node.
2. `struct NodeList`: This structure is used to manage a list of nodes within the search tree. It includes the following fields:
   - `unsigned int nodeCount`: Specifies the number of nodes in the list.
   - `ListNode *head`: Points to the first node in the list.
   - `ListNode *tail`: Points to the last node in the list.
3. `SolutionPath* IDSearch(State *, State *, int depth_limit);`: This function performs Iterative Deepening Search (ID-DFS) to find a solution path from the initial state to the goal state. It takes three parameters: the initial state, the goal state, and a depth limit. It returns a "`SolutionPath`" structure, which represents the path from the initial state to the goal state or a failure indicator if no solution is found within the specified depth limit.
4. `SolutionPath* AStarSearch(State *, State *);`: This function implements the A* search algorithm to find a solution path from the initial state to the goal state. It takes two parameters: the initial state and the goal state. It returns a "`SolutionPath`" structure representing the optimal path from the initial state to the goal state.

```
SolutionPath* IDSearch(State *initial, State *goal,int depth_limit) {
   NodeList *queue = NULL;
    NodeList *children = NULL;
    Node *node = NULL;

    //start timer
    clock_t start = clock();

    //initialize the queue with the root node of the search tree
    pushNode(createNode(0, manhattanDist(initial, goal), initial, NULL),
&queue);
    Node *root = queue->head->currNode; //for deallocating the generated
tree
```

```
while(queue->nodeCount > 0) {
    //pop the last node (tail) of the queue
    node = popNode_head(&queue);

    //if the state of the node is the goal state
    if(statesMatch(node->state, goal))
        break;

    if(node->depth< depth_limit){
        children = getChildren(node, goal);
        ++nodesExpanded;
        pushList(&children, queue);
    }
}

//determine the time elapsed
runtime = (double)(clock() - start) / CLOCKS_PER_SEC;

SolutionPath *pathHead = NULL;
SolutionPath *newPathNode = NULL;
if(statesMatch(node->state, goal)){

    while(node) {
        newPathNode = malloc(sizeof(SolutionPath));
        newPathNode->action = node->state->action;
        newPathNode->next = pathHead;
        pathHead = newPathNode;

        //update the solution length and move on the next node
        ++solutionLength;
        node = node->parent;
    }

    --solnLength; //uncount the root node
    }
    //deallocate the generated tree
    destroyTree(root);

    return pathHead;
```

The `IDSearch` function is responsible for performing Iterative Deepening Search (IDS) to solve the 15-puzzle problem.

1. `SolutionPath* IDSearch(State *initial, State *goal, int depth_limit)`: This function takes three parameters: the initial state, the goal state, and the depth limit for the IDS algorithm. It returns a pointer to a `SolutionPath`, which represents the solution path or sequence of moves to solve the puzzle.
2. `NodeList *queue = NULL;` and `NodeList *children = NULL;`: These are pointers to `NodeList` structures. `queue` represents the list of nodes to be explored, and `children` is used to store the children of the currently explored node.
3. `Node *node = NULL;`: This is a pointer to a `Node` structure, which represents the currently explored node in the search.
4. `clock_t start = clock();`: This line records the current clock time as the starting point for measuring the runtime of the search algorithm.
5. `pushNode(createNode(0, manhattanDist(initial, goal), initial, NULL), &queue);`: This line initializes the search by creating the root node and pushing it into the `queue`. Here's what's happening in this line:
    - `createNode(0, manhattanDist(initial, goal), initial, NULL)`: This function creates a new node with a depth of 0, a heuristic cost calculated using the Manhattan distance heuristic, the initial state, and a `NULL` parent (since this is the root node).
    - `pushNode(...)`: This function adds the newly created root node to the `queue`. It manages the queue of nodes to be explored in the search process.

```
SolutionPath* AStarSearch(State *initial, State *goal){
  NodeList *openList = NULL;
    NodeList *closedList = NULL;
    Node *node = NULL;

    //start timer
    clock_t start = clock();

    pushNode(createNode(0, manhattanDist(initial, goal), initial, NULL),
&openList);
    Node *root = openList->head->currNode; //for deallocating generated
tree


    while(openList->nodeCount > 0) {
        node = popNode(&openList);
```

```
        //if the state of the node is the goal state
        if(statesMatch(node->state, goal))
            break;

        //else, expand the node and update the expanded-nodes counter
        closedList = getChildren(node, goal);
        ++nodesExpanded;

        //add the node's closedList to the openList
        pushListInOrder(&closedList, openList);
    }

    //determine the time elapsed
    runtime = (double)(clock() - start) / CLOCKS_PER_SEC;

    //get solution path in order from the root, if it exists
    SolutionPath *pathHead = NULL;
    SolutionPath *newPathNode = NULL;

    while(node) {
        newPathNode = malloc(sizeof(SolutionPath));
        newPathNode->action = node->state->action;
        newPathNode->next = pathHead;
        pathHead = newPathNode;

        //update the solution length and move on the next node
        ++solutionLength;
        node = node->parent;
    }

    --solutionLength; //uncount the root node

    //deallocate the generated tree
    destroyTree(root);

    return pathHead;
```

This `AStarSearch` function is responsible for A* Search to solve the 15-puzzle problem.

1. `SolutionPath* AStarSearch(State *initial, State *goal)`: This function takes two parameters: initial state and goal state for A* search algorithm. It returns a pointer to a `SolutionPath` representing the solution path if one is found.

2. `NodeList *openList = NULL;` and `NodeList *closedList = NULL;`: These are pointers to `NodeList` structures. `openList` an open list that holds nodes to be explored. `closedList` an closed list for nodes that have already been explored.
3. `Node *node = NULL;` This is a pointer to a `Node` structures, which represents a temporary variable to hold the current node being processed.
4. `clock_t start = clock();`: This line starts a timer to measure the runtime of the algorithm.
5. `pushNode(createNode(0, manhattanDist(initial, goal), initial, NULL), openList);`pushes the initial node onto the `openList`. This node represents the starting state and has a cost of 0, a heuristic estimate of the remaining cost (usually the Manhattan distance between the current state and the goal state), and a reference to the parent node.
6. `Node *root = openList->head->currNode;` : This node creates a reference to the root node for deallocating generated tree later.
7. `while(openList->nodeCount > 0)`:This while loop continues as long as there are nodes in the `openList` to explore.
8. `node = popNode(&openList);`: Within the loop, the next node with the lowest total cost is removed from the openList and stored in the `node` variable.
9. `if(statesMatch(node->state, goal)) break;`: If the state of the current node matches the goal state, the search is terminated, and the loop is exited.
10. `closedList = getChildren(node, goal);`: This closedList call the `getChildren` function to expand the current node and generate a list of child nodes.
11. `++nodesExpanded;` : This nodes incremented expanded during the search.
12. `pushListInOrder(&closedList, openList);` : The child nodes generated in the `closedList` are pushed into the `openList`.
13. `runtime = (double)(clock() - start) / CLOCKS_PER_SEC;`: This line used to calculate the elapsed time (in seconds) of the A* search algorithm.
14. `SolutionPath *pathHead = NULL;` and `SolutionPath *newPathNode = NULL;` This function declare two pointers, `pathHead` and `newPathNode`, both initially set to `NULL`. These pointers will be used to construct a linked list that represents the solution path.
15. Inside the `while` loop as long as the node pointer is not `NULL`:
    a. `newPathNode = malloc(sizeof(SolutionPath));` : This `newPathNode` allocates memory for a new `SolutionPath` structure.
    b. `newPathNode->action = node->state->action;` This `newPathNode` clone the `action` from the `state` of the current `node` into the `newPathNode`.
    c. `newPathNode->next = pathHead;` links the `newPathNode` to the previous node in the linked list by setting its `next` pointer to the current `pathHead`.
    d. `pathHead = newPathNode`; updates the `pathHead` pointer to point to the newly created `newPathNode`.

16. `++solutionLength;`:This variable increment to keeps track of the length of the solution path.
17. `node = node->parent;` This `node` pointer updates the to point to its parent node, effectively moving up the tree to the next node in the solution path.
18. `--solutionLength;` :subtracts 1 to uncount the root node.
19. `destroyTree(root);` This function call that destroys the tree structure rooted at `root`, to clean up memory and prevent memory leaks.
20. `return pathHead;`:The `pathHead` pointer returns to the head of the linked list that represents the solution path in order from the root.

```
void printSolution(struct SolutionPath *path) {
      //check if solution exists
    if(!path) {
        printf("No solution found.\n");
        return;
    }

      //if the initial state is already the goal state
      if(!path->next) {
            printf("No moves needed. The initial state is already the
goal state.\n");
            return;
      }

    printf("SOLUTION PATH: (Relative to the space character)\n");

    //will use hash map to speed up the proccess a bit
    char *move[4] = { "UP", "DOWN", "LEFT", "RIGHT" };
    int counter = 1;

    //will be skipping the first node since it represents the initial
state with no action
    for(path = path->next; path; path = path->next, ++counter) {
        printf("%i. Move %s\n", counter, move[path->action]);
    }

    printf(
        "\nDETAILS:\n"
        " - Solution cost    : %i\n"
        " - Nodes expanded   : %i\n"
        " - Running time     : %g milliseconds\n\n",
```

```
        solutionLength, nodesExpanded, runtime);
}
```

The printSolution function is responsible for printing the solution path of the 15-puzzle problem, along with some additional details.

1. `if (!path)`: This condition checks if a valid solution path exists. If `path` is `NULL`, it means there is no solution, and the function prints "No solution found."
2. `if (!path->next)`: If the next node inhe solution path is `NULL`, it means that the initial state is already the goal state, and no moves are needed. In this case, the function prints "No moves needed. The initial state is already the goal state."
3. `printf("SOLUTION PATH: (Relative to the space character)\n")`: This line prints a header indicating that the following lines will display the solution path. It also mentions that the moves are relative to the space character (blank tile).
4. `char *move[4] = { "UP", "DOWN", "LEFT", "RIGHT" };`: This array `move` stores strings representing the four possible moves: UP, DOWN, LEFT, and RIGHT. These strings will be used to display the moves in the solution path.
5. `int counter = 1;`: The `counter` variable is initialized to 1 and will be used to number the moves in the solution path.
6. The `for` loop iterates through the solution path starting from the second node (skipping the first node, which represents the initial state). For each node in the path, it prints the move number and the corresponding move direction based on the `move` array.
7. After printing the solution path, the function provides additional details:
    - Solution cost: The length of the solution path, which represents the number of moves required to reach the goal state.
    - Nodes expanded: The number of nodes expanded during the search process.
    - Running time: The time taken to find the solution in milliseconds, measured using the `clock` function.

```
void destroySolution(SolutionPath **list) {
    SolutionPath *next;
    while(*list) {
        next = (*list)->next;
        free(*list);
        *list = next;
    }
    *list = NULL;
}
```

The `destroySolution` function is responsible for freeing the memory associated with the linked list of solution paths.

1. `SolutionPath **list`: The function takes a pointer to a pointer to a `SolutionPath` list as an argument. This allows it to modify the original pointer, ensuring that it is set to `NULL` after all memory has been freed.
2. `SolutionPath *next;`: This line declares a pointer `next` of type `SolutionPath`. It will be used to temporarily store the next node in the linked list while the current node is being freed.
3. `while (*list) { ... }`: This is a `while` loop that continues as long as the pointer `*list` is not `NULL`, indicating that there are more nodes in the linked list to be processed.
4. `next = (*list)->next;`: Inside the loop, the `next` pointer is assigned the address of the next node in the linked list. This is done to preserve the reference to the next node before the current node is freed.
5. `free(*list);`: The current node pointed to by `*list` is freed using the `free` function. This releases the memory allocated for this node.
6. `*list = next;`: After the current node has been freed, the pointer `*list` is updated to point to the `next` node, which is the next node in the linked list.
7. The loop continues to the next iteration, where it repeats steps 4 to 6 for the new current node (now pointed to by `*list`). This process continues until all nodes in the linked list have been freed.
8. `*list = NULL;`: After the loop exits, the function sets the original pointer `*list` to `NULL` to indicate that the entire linked list has been destroyed. This is important to prevent any accidental access to the now-freed memory.

SOURCE CODE

```
/*
    MEMBERS:
    Agao, Daphne Julienne
    Balatinsayo, Judith
    Bertulfo, Juliet Clarisse
    Colasito, Neslie
    Judin, Jade Airin
*/

#include<stdio.h>
#include<stdlib.h>
#include<time.h>
```

```c
#include <stdbool.h>
#define BLANK_CHARACTER '0'

unsigned int nodesExpanded;  //number of expanded nodes
unsigned int nodesGenerated; //number of generated nodes
unsigned int solnLength; //number of moves in solution
double runtime;              //elapsed time (in milliseconds)

typedef struct Node Node;
typedef struct NodeList NodeList;
typedef struct State State;
typedef enum Move Move;
typedef struct SolutionPath SolutionPath;

void inputInitialState(State * const state);
void inputgoalState(State * const state);
void printBoard(char const board[][4]);
void printSolution(struct SolutionPath *path) ;
void destroySolution(SolutionPath **list);
char pushNode(Node *node, NodeList** const list);
Node* popNode(NodeList** const list);
Node* popNodehead(NodeList** const list);
void pushList(NodeList **toAppend, NodeList *list);
Node* createNode(unsigned int d, unsigned int h, State *s, Node *p);
void destroyTree(Node *node);
NodeList* getChildren(Node *parent, State *goalState);
State* createState(State *state, Move move);
void destroyState(State **state);
int manhattanDist(State * const curr, State * const goal);
void pushListInOrder(NodeList **toAppend, NodeList *list);
char statesMatch(State const *testState, State const *goalState);
void checkStates(State * const initial, State * const goal);

typedef enum Move {
    UP, DOWN, LEFT, RIGHT, //values for moving up, down, left, right,
respectively
    NOT_APPLICABLE         //value assigned for initial and goal input
states
} Move;

typedef struct State {
    Move action;           //action that resulted to `this` board state
   char board[4][4];      //resulting board configuration after applying
```

```c
action
} State;

typedef struct ListNode {
    Node *currNode;
    struct ListNode *prevNode; //the node before `this` instance
    struct ListNode *nextNode; //the next node in the linked list
} ListNode;

typedef struct SolutionPath {
    Move action;
    struct SolutionPath *next;
} SolutionPath;

struct Node {
    unsigned int depth; //depth of the node from the root. For A* search,
                        //this will also represent the node's path cost
    unsigned int hCost; //heuristic cost of the node
    State *state;       //state designated to a node
    Node *parent;       //parent node
    NodeList *children; //list of child nodes
};

struct NodeList {
    unsigned int nodeCount;    //the number of nodes in the list
    ListNode *head;            //pointer to the first node in the list
    ListNode *tail;            //pointer to the last node in the list
};


SolutionPath* IDSearch(State *, State *,int depth_limit);
SolutionPath* AStarSearch(State *, State *);

int main() {
    int choice;
    int limit;
    State initial;       //initial board state
    State goal;         //goal board configuration

    SolutionPath *ids; //solution path of each search method
    SolutionPath *astar;

    do {
```

```
        system("cls||clear"); // Clear the console screen.
        printf("  _  ___      ___               _       ___      _
\n");
        printf(" / || __| ___ | _ \\ _  _  ___ ___| | ___  / __| ___ |
|__ __ ___  _ _ \n");
        printf(" | ||__ \\|___||  _/| || ||_ /|_ /| |/ -_) \\__ \\/ _ \\|
|\\ v // -_)| '_|\n");
        printf(" |_||___/     |_|   \\_,_|/__|/__||_|\\___|
|___/\\___/|_| \\_/ \\___||_|  \n");
        printf("
\n");

printf("-----------------------------------------------------------------
\n");
        printf("|0.) Reconfigure the Board\t\t\t\t\t|\n");
        printf("|1.) Iterative Deepening Search\t\t\t\t\t|\n");
        printf("|2.) A* Search\t\t\t\t\t\t\t|\n");
        printf("|3.) End Program\t\t\t\t\t\t|\n");

printf("-----------------------------------------------------------------
\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 0:
                  printf("\nReconfiguring the puzzle board...");
                inputInitialState(&initial);
                inputgoalState(&goal);
                printf("Board reconfigured successfully.\n");
                printf("\n");
                system("pause");
                break;
            case 1:
                if (statesMatch(&goal, &initial)){
                    printf("No moves needed. The initial state is already
the goal state.\n");
                }
                        for (limit = 1; limit < 1000000; limit++) {
                    nodesExpanded = 0;
                    nodesGenerated = 0;
                    solnLength = 0;
                    runtime = 0;
```

```
                    SolutionPath *ids = IDSearch(&initial, &goal, limit);
                    printf("\n----------- USING IDS ALGORITHM DEPTH LIMIT
= %d -----------\n", limit);

                    printSolution(ids);
                    if (solnLength) {
                        limit = 1000000;
                        printf("IDS completed successfully.\n");
                    }
                    else {
                        printf("IDS did not find a solution.\n");
                    }
                    destroySolution(&ids);
                      }
                system("pause");
                break;

                  case 2:
                if (statesMatch(&goal, &initial)){
                    printf("No moves needed. The initial state is already
the goal state.\n");
                }
                        printf("\n------------------ USING A* ALGORITHM
---------------------- \n");
                  nodesExpanded = 0;
                  nodesGenerated = 0;
                  solnLength = 0;
                  runtime = 0;
                  SolutionPath *astar = AStarSearch(&initial, &goal);

                  printSolution(astar);

                  if (solnLength) {
                        printf("A* Search completed successfully.\n");
                  } else {
                        printf("A* Search did not find a solution.\n");
                  }

                  destroySolution(&astar);
                  system("pause");
                  break;
            case 3:
```

```
                printf("\nYey! You're done! :)\n");
               return 0;  // Exit the program
           default:
               printf("Invalid choice. Please enter a valid menu
option.\n");
               system("pause");
               break;
       }
    } while (choice != 3);

    return 0;
}

SolutionPath* IDSearch(State *initial, State *goal,int depth_limit) {
    NodeList *queue = NULL;
    NodeList *children = NULL;
    Node *node = NULL;

    clock_t start = clock(); //start timer


    pushNode(createNode(0, manhattanDist(initial, goal), initial, NULL),
&queue);   //initialize the queue with the root node of the search tree
    Node *root = queue->head->currNode; //for deallocating the generated
tree


    while(queue->nodeCount > 0) {
        node = popNodehead(&queue);//pop the last node (tail) of the
queue

        if(statesMatch(node->state, goal)) //if the state of the node is
the goal state
            break;

        if(node->depth< depth_limit){
            children = getChildren(node, goal);
            ++nodesExpanded;
            pushList(&children, queue);
        }
    }

    runtime = (double)(clock() - start) / CLOCKS_PER_SEC; //determine the
```

```
time elapsed

    SolutionPath *pathHead = NULL;
    SolutionPath *newPathNode = NULL;
    if(statesMatch(node->state, goal)){

        while(node) {
            newPathNode = malloc(sizeof(SolutionPath));
            newPathNode->action = node->state->action;
            newPathNode->next = pathHead;
            pathHead = newPathNode;

            ++solnLength; //update the solution length and move on the
next node
            node = node->parent;
        }

        --solnLength; //uncount the root node
        }

        destroyTree(root);  //deallocate the generated tree

        return pathHead;
}

SolutionPath* AStarSearch(State *initial, State *goal){
    NodeList *openList = NULL;
    NodeList *closedList = NULL;
    Node *node = NULL;

    clock_t start = clock();

    pushNode(createNode(0, manhattanDist(initial, goal), initial, NULL),
&openList);
    Node *root = openList->head->currNode;


    while(openList->nodeCount > 0) {
        node = popNode(&openList);

        if(statesMatch(node->state, goal)) //if the state of the node is
the goal state
            break;
```

```
        closedList = getChildren(node, goal); //else, expand the node and
update the expanded-nodes counter
        ++nodesExpanded;

        pushListInOrder(&closedList, openList); //add the node's
closedList to the openList
    }

    runtime = (double)(clock() - start) / CLOCKS_PER_SEC;

    SolutionPath *pathHead = NULL;
    SolutionPath *newPathNode = NULL;

    while(node) {
        newPathNode = malloc(sizeof(SolutionPath));
        newPathNode->action = node->state->action;
        newPathNode->next = pathHead;
        pathHead = newPathNode;

        ++solnLength;
        node = node->parent;
    }

    --solnLength;

    destroyTree(root);

    return pathHead;
}

void inputInitialState(State * const state){
    state->action=NOT_APPLICABLE;

      printf("\nInput the initial state of 15 puzzle separated with space
(enter 0 for the blank tile):\n");
    int i,j;
    int input;
    for(i = 0; i < 4; ++i) {
        for(j = 0; j < 4; ++j) {
            scanf("%d", &input);
            state->board[i][j] = input + '0';
        }
```

```
    }
}

void inputgoalState(State * const state){
    state->action=NOT_APPLICABLE;
    int i,j;

    int num=0;
    for(i = 0; i < 4; ++i) {
        for(j = 0; j < 4; ++j) {
            state->board[i][j] = num + '0';
            num++;
        }
    }
}

void printSolution(struct SolutionPath *path) {
    if(!path) {//check if solution exists
        return;
    }

      if(!path->next) { //if the initial state is already the goal state
            printf("No moves needed. The initial state is already the
goal state.\n");
            return;
      }

    printf("SOLUTION PATH: (Relative to the space character)\n");

    char *move[4] = { "UP", "DOWN", "LEFT", "RIGHT" }; //will use hash
map to speed up the proccess a bit
    int counter = 1;

    for(path = path->next; path; path = path->next, ++counter) { //will
be skipping the first node since it represents the initial state with no
action
        printf("%i. Move %s\n", counter, move[path->action]);
    }

    printf(
        "\nDETAILS:\n"
        " - Solution cost   : %i\n"
        " - Nodes expanded  : %i\n"
```

```
        " - Running time    : %g milliseconds\n\n",
        solnLength, nodesExpanded, runtime);
}

void destroySolution(SolutionPath **list) {
    SolutionPath *next;
    while(*list) {
        next = (*list)->next;
        free(*list);
        *list = next;
    }
    *list = NULL;
}

char pushNode(Node *node, NodeList** const list) {
    if(!node)
        return 0;

    ListNode *doublyNode = malloc(sizeof(ListNode));
    if(!doublyNode)
        return 0;

    doublyNode->currNode = node;

    if(*list && !(*list)->nodeCount) {
        (*list)->head = doublyNode;
        (*list)->tail = doublyNode;
        doublyNode->nextNode = NULL;
        doublyNode->prevNode = NULL;
        ++(*list)->nodeCount;
        return 1;
    }

    if(*list == NULL) {
        *list = malloc(sizeof(NodeList));
        if(*list == NULL)
            return 0;

        (*list)->nodeCount = 0;
        (*list)->head = NULL;
        (*list)->tail = doublyNode;
    }
    else {
```

```
        (*list)->head->prevNode = doublyNode;
    }

    doublyNode->nextNode = (*list)->head;
    doublyNode->prevNode = NULL;
    (*list)->head = doublyNode;

    ++(*list)->nodeCount;

    return 1;
}

Node* popNode(NodeList** const list) {
    if(!*list || (*list)->nodeCount == 0)
        return NULL;

    Node *popped = (*list)->tail->currNode;
    ListNode *prevNode = (*list)->tail->prevNode;

    free((*list)->tail); //free the list node pointing to node to be
popped

    if((*list)->nodeCount == 1) {
        (*list)->head = NULL;
    }
      else {
            prevNode->nextNode = NULL;
      }

    (*list)->tail = prevNode;
    --(*list)->nodeCount;
    return popped;
}

Node* popNodehead(NodeList** const list){
    if(!*list || (*list)->nodeCount == 0)
        return NULL;

    Node *popped = (*list)->head->currNode;
    ListNode *nextNode = (*list)->head->nextNode;

    free((*list)->head);
```

```
    if((*list)->nodeCount == 1) {
        (*list)->tail = NULL;
    }
      else {
            nextNode->prevNode = NULL;
      }

    (*list)->head = nextNode;
    --(*list)->nodeCount;
    return popped;
}

void pushList(NodeList **toAppend, NodeList *list) {
    //if either of the list is NULL, the head of the list to be appended
is NULL,
    //or the list points to the same starting node
    if(!*toAppend || !list || !(*toAppend)->head || (*toAppend)->head ==
list->head) {
        return;
    }

    if(!list->nodeCount) { //if the list to append to has currently no
element
        list->head = (*toAppend)->head;
        list->tail = (*toAppend)->tail;
    }
    else { //connect the lists
        (*toAppend)->tail->nextNode = list->head;
        list->head->prevNode = (*toAppend)->tail;
            list->head = (*toAppend)->head;
    }

    list->nodeCount += (*toAppend)->nodeCount; //update list information

    free(*toAppend);
    *toAppend = NULL;
}

Node* createNode(unsigned int d, unsigned int h, State *s, Node *p) {
    Node *newNode = malloc(sizeof(Node));
    if(newNode) {
        newNode->depth = d;
        newNode->hCost = h;
```

```
        newNode->state = s;
        newNode->parent = p;
        newNode->children = NULL;
        ++nodesGenerated; //update counter
    }
    return newNode;
}

void destroyTree(Node *node) {
    if(node->children == NULL) {
        free(node->state);
        free(node);
        return;
    }

    ListNode *listNode = node->children->head;
    ListNode *nextNode;

    while(listNode) {
        nextNode = listNode->nextNode;
        destroyTree(listNode->currNode);
        listNode = nextNode;
    }

    //free(node->state);
    free(node->children);
    free(node);
}

NodeList* getChildren(Node *parent, State *goalState) {
    NodeList *childrenPtr = NULL;
    State *testState = NULL;
    Node *child = NULL;

    if(parent->state->action != DOWN && (testState =
createState(parent->state, UP))) { //attempt to create states for each
moves, and add to the list of children if true
        child = createNode(parent->depth + 1, manhattanDist(testState,
goalState), testState, parent);
        pushNode(child, &parent->children);
        pushNode(child, &childrenPtr);
    }
    if(parent->state->action != UP && (testState =
```

```
createState(parent->state, DOWN))) {
        child = createNode(parent->depth + 1, manhattanDist(testState,
goalState), testState, parent);
        pushNode(child, &parent->children);
        pushNode(child, &childrenPtr);
    }
    if(parent->state->action != RIGHT && (testState =
createState(parent->state, LEFT))) {
        child = createNode(parent->depth + 1, manhattanDist(testState,
goalState), testState, parent);
        pushNode(child, &parent->children);
        pushNode(child, &childrenPtr);
    }
    if(parent->state->action != LEFT && (testState =
createState(parent->state, RIGHT))) {
        child = createNode(parent->depth + 1, manhattanDist(testState,
goalState), testState, parent);
        pushNode(child, &parent->children);
        pushNode(child, &childrenPtr);
    }

    return childrenPtr;
}

State* createState(State *state, Move move) {
    State *newState = malloc(sizeof(State));

    //copy the board configuration of `state` to `newState`
    //while searching for the row and column of the blank character
    int i, j;        //used for traversing the 3x3 arrays
    int row=0;
    int col=0;    //coordinates of the blank character

    for(i = 0; i < 4; ++i) {
        for(j = 0; j < 4; ++j) {
            if(state->board[i][j] == BLANK_CHARACTER) {
                row = i;
                col = j;
            }

            newState->board[i][j] = state->board[i][j];
        }
    }
```

```
     //test if the coordinates are valid after translation based on the
move
    //if it is, swap the concerned board values to reflect the move
    if(move == UP && row - 1 >= 0) {
        char temp = newState->board[row - 1][col];
        newState->board[row - 1][col] = BLANK_CHARACTER;
        newState->board[row][col] = temp;
        newState->action = UP;
        return newState;
    }
    else if(move == DOWN  && row + 1 < 4) {
        char temp = newState->board[row + 1][col];
        newState->board[row + 1][col] = BLANK_CHARACTER;
        newState->board[row][col] = temp;
        newState->action = DOWN;
        return newState;
    }
    else if(move == LEFT  && col - 1 >= 0) {
        char temp = newState->board[row][col - 1];
        newState->board[row][col - 1] = BLANK_CHARACTER;
        newState->board[row][col] = temp;
        newState->action = LEFT;
        return newState;
    }
    else if(move == RIGHT && col + 1 < 4) {
        char temp = newState->board[row][col + 1];
        newState->board[row][col + 1] = BLANK_CHARACTER;
        newState->board[row][col] = temp;
        newState->action = RIGHT;
        return newState;
    }

    free(newState);
    return NULL;
}

void destroyState(State **state) {
    free(*state);
    state = NULL;
}

int manhattanDist(State * const curr, State * const goal) {
```

```
    int x0, y0; //used for indexing each symbol in `curr`
    int x1, y1; //correspoinding row and column of symbol from curr[y0,
x0] at `goal`
    int dx, dy; //change in x0 and x1, and y0 and y1, respectively
    int sum = 0;

    //for each symbol in `curr`
    for(y0 = 0; y0 < 4; ++y0) {
        for(x0 = 0; x0 < 4; ++x0) {
            //find the coordinates of the same symbol in `goal`
            for(y1 = 0; y1 < 4; ++y1) {
                for(x1 = 0; x1 < 4; ++x1) {
                    if(curr->board[y0][x0] == goal->board[y1][x1]) {
                        dx = (x0 - x1 < 0)? x1 - x0 : x0 - x1;
                        dy = (y0 - y1 < 0)? y1 - y0 : y0 - y1;
                        sum += dx + dy;
                    }
                }
            }
        }
    }
    return sum;
}

int totalCost(Node * const node) {
    return node->depth + node->hCost;
}
void pushListInOrder(NodeList **toAppend, NodeList *list){
    if(!*toAppend || !list || !(*toAppend)->head || (*toAppend)->head ==
list->head) {
        return;
    }

    if(!list->nodeCount) {
        pushNode(popNode(toAppend), &list);
    }

    ListNode *toAppendNode;
    ListNode *listNode;
    Node *node;

     while((toAppendNode = (*toAppend)->head)) {
        listNode = list->head;
```

```
        while(listNode && totalCost(toAppendNode->currNode) <
totalCost(listNode->currNode)) {
            listNode = listNode->nextNode;
        }

        ListNode *temp = toAppendNode->nextNode;

        if(!listNode) {
            list->tail->nextNode = toAppendNode;
            toAppendNode->prevNode = list->tail;
            toAppendNode->nextNode = NULL;
            list->tail = toAppendNode;
        }
        else {
            if(listNode->prevNode) {
                toAppendNode->prevNode = listNode->prevNode;
                toAppendNode->nextNode = listNode;
                listNode->prevNode->nextNode = toAppendNode;
                listNode->prevNode = toAppendNode;
            }
            else {
                toAppendNode->nextNode = list->head;
                toAppendNode->prevNode = NULL;
                list->head->prevNode = toAppendNode;
                list->head = toAppendNode;
            }
        }

        (*toAppend)->head = temp;
        --(*toAppend)->nodeCount;
        ++list->nodeCount;
    }
    free(*toAppend);
    *toAppend = NULL;
}

char statesMatch(State const *testState, State const *goalState) {
    int row = 4, col;

    while(row--) {
        col = 4;
        while(col--) {
```

```c
            if(testState->board[row][col] != goalState->board[row][col])
                return 0;
        }
    }
    return 1;
}
```

This C code is an implementation of two search algorithms, namely Iterative Deepening Search (IDS) and A* Search, to solve the 15-puzzle problem. The 15-puzzle problem is a sliding puzzle where a 4x4 grid is filled with numbered tiles, except for one blank tile. The goal is to reorder the tiles from an initial configuration to a goal configuration using a minimal number of moves.

Here's an explanation of the code as a whole:

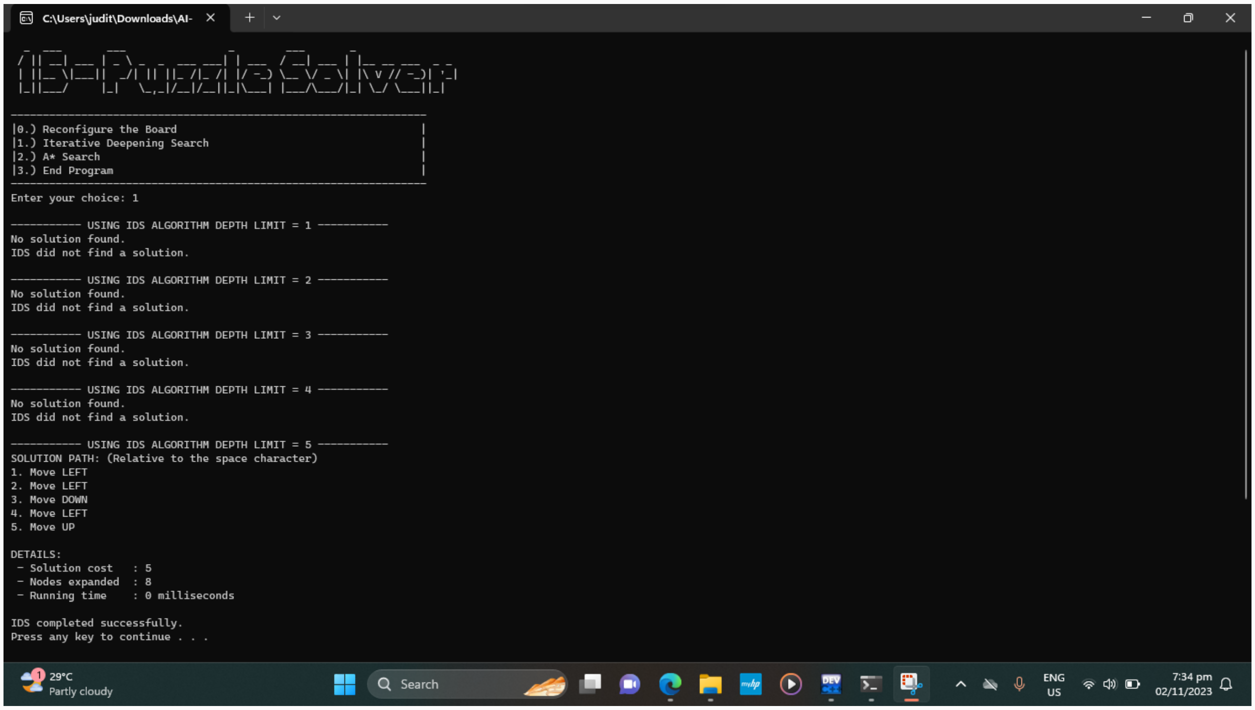
1. **Header Files:** The code includes various standard C library and header files, including `<stdio.h>`, `<stdlib.h>`, `<time.h>`, and `<stdbool.h>`. These are used for input/output, memory management, timing, and boolean data types.
2. **Preprocessor Directives:** The code defines a constant `BLANK_CHARACTER` for the blank tile, which is set to '0'. It also defines global variables to keep track of the number of expanded and generated nodes, solution length, and runtime.
3. **Type Definitions:** Several custom data structures are defined, including `Node`, `NodeList`, `State`, `Move`, and `SolutionPath`. These structures are used to manage the search process, store puzzle states, and represent solution paths.
4. **Function Prototypes:** The code declares the prototypes of various functions that are defined later in the code. These functions are responsible for different aspects of the puzzle solving process, including input, output, memory management, search algorithms, and more.
5. **Main Function:** The `main` function serves as the entry point for the program. It displays a menu for the user to choose various options:
    - Option 0: Reconfigure the puzzle board with an initial and goal state.
    - Option 1: Perform Iterative Deepening Search (IDS) with varying depth limits.
    - Option 2: Perform A* Search.
    - Option 3: Exit the program.
6. **Search Algorithms:**
    - Iterative Deepening Search (IDS): The `IDSearch` function implements the IDS algorithm. It takes the initial and goal states, along with a depth limit as parameters, and returns a solution path.
    - *A Search**: The `AStarSearch` function implements the A* search algorithm. It takes the initial and goal states as parameters and returns a solution path.

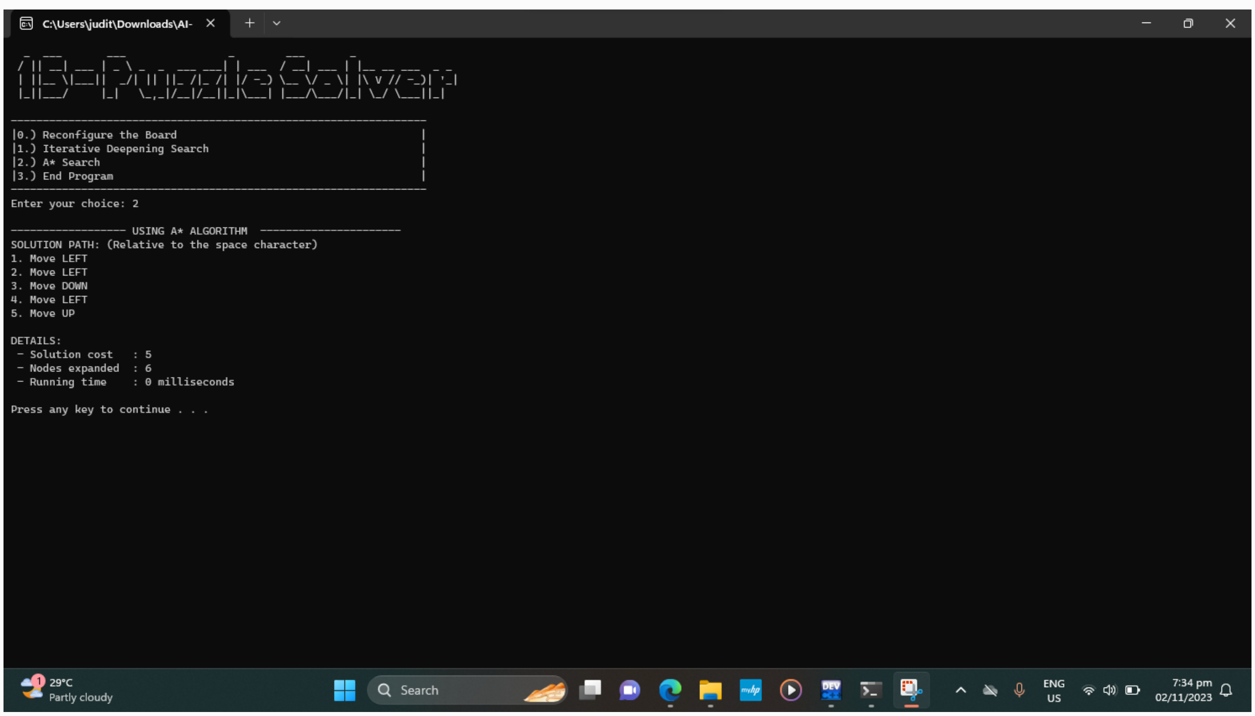
7. **User Input Functions:** The code defines functions to input the initial and goal states, including the `inputInitialState` and `inputgoalState` functions.
8. **Output Functions:** The code defines functions to print the puzzle board configuration and the solution path. The "`printBoard`" and "`printSolution`" functions are responsible for this.
9. **Memory Management Functions:** The code provides functions to manage memory and data structures, including `destroySolution` for freeing solution paths, `pushNode` for adding nodes to a list, and `popNode` for retrieving nodes from a list.
10. **Node and State Functions:** The code contains functions for creating, destroying, and manipulating nodes and states. These functions handle generating child nodes and managing the search tree.
11. **Heuristic Functions:** The code includes the `manhattanDist` function, which calculates the Manhattan distance heuristic for a given state.
12. **List Sorting Functions:** The `pushListInOrder` function is responsible for pushing nodes onto a list in a sorted order based on their costs. This is used in the A* algorithm.
13. **State Comparison Function:** The `statesMatch` function checks if two states match, which is used to determine if a solution has been found.

Overall, this code provides a comprehensive implementation of two search algorithms for solving the 15-puzzle problem, with a menu-driven interface for user interaction. It uses various data structures and functions to manage the search process, track nodes, and find optimal solutions.

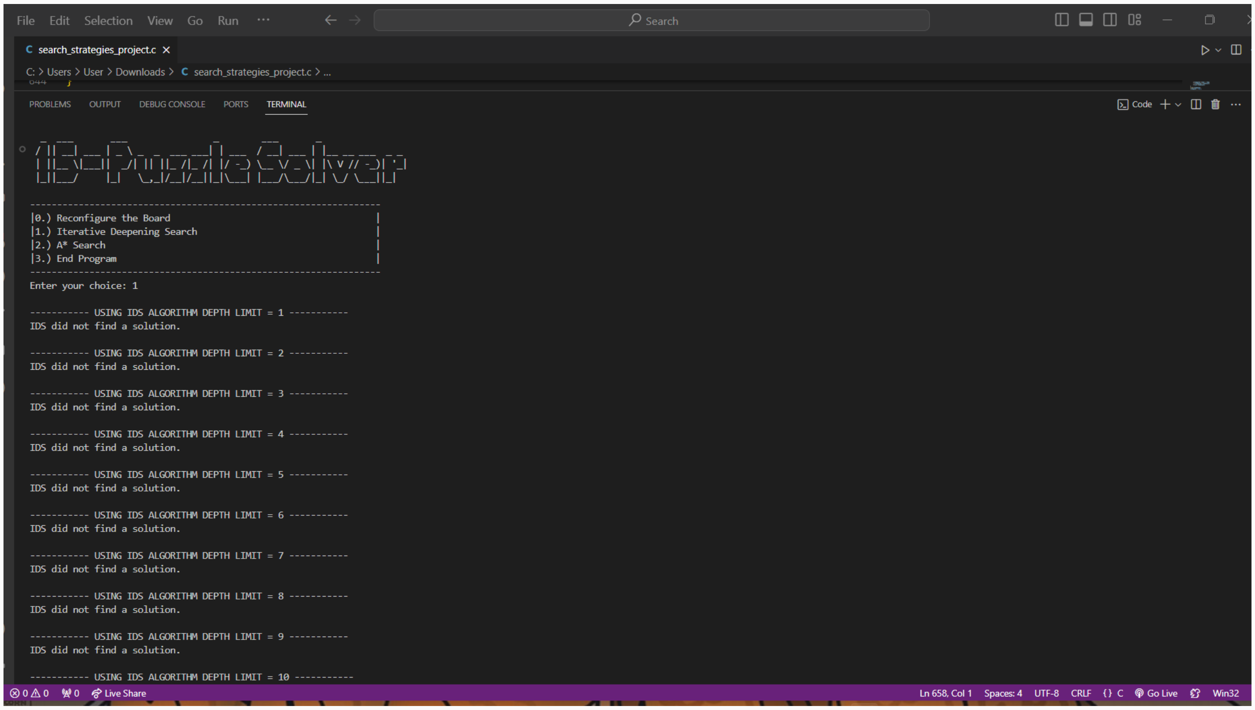
IV. Screenshots of the Actual Execution of the Program

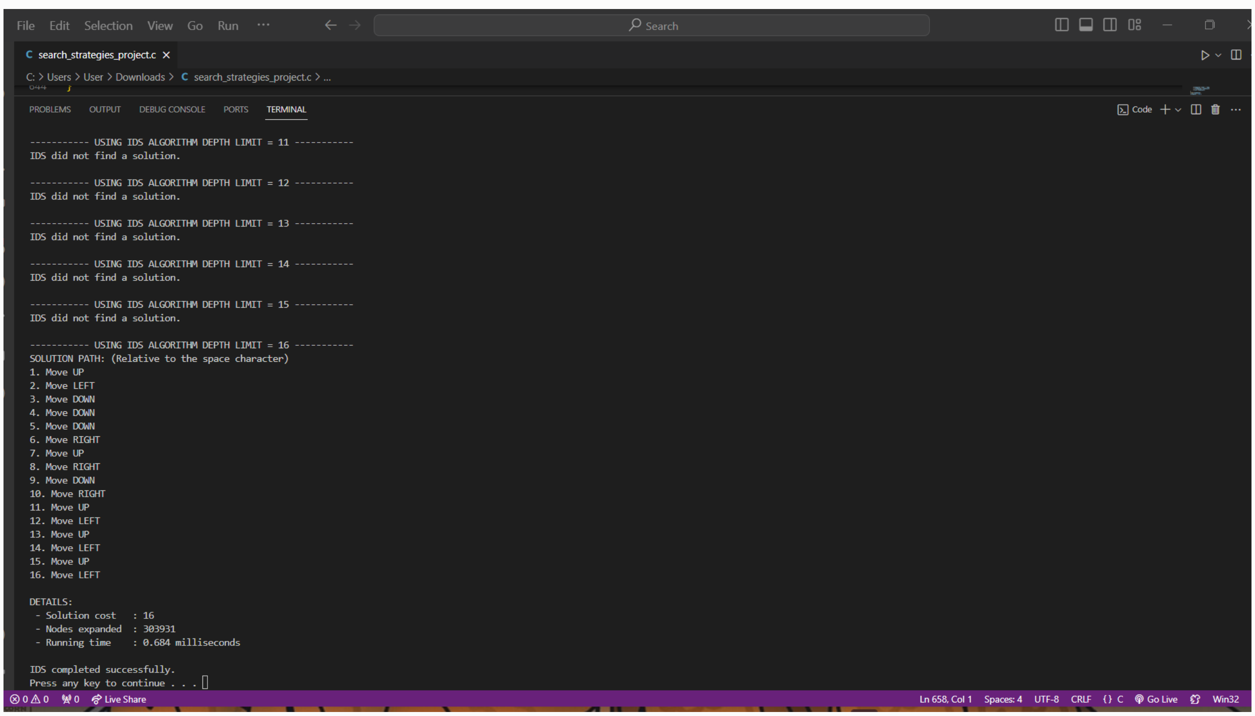
EASY PUZZLE

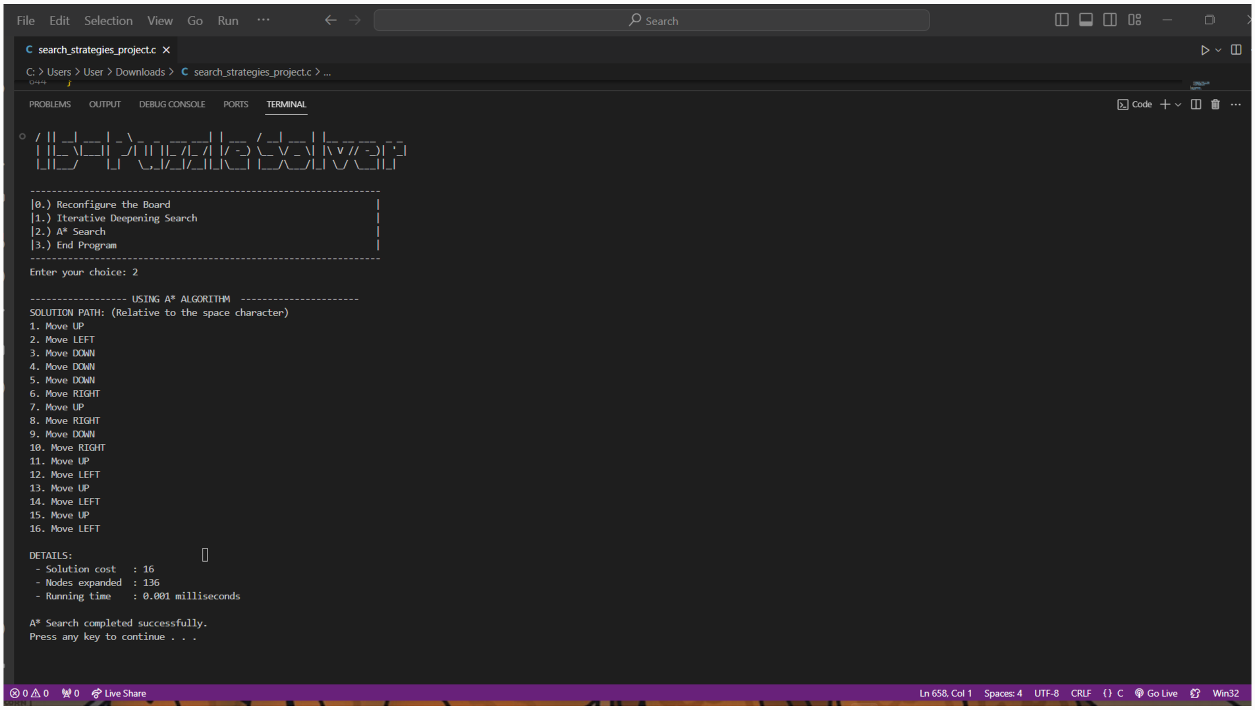




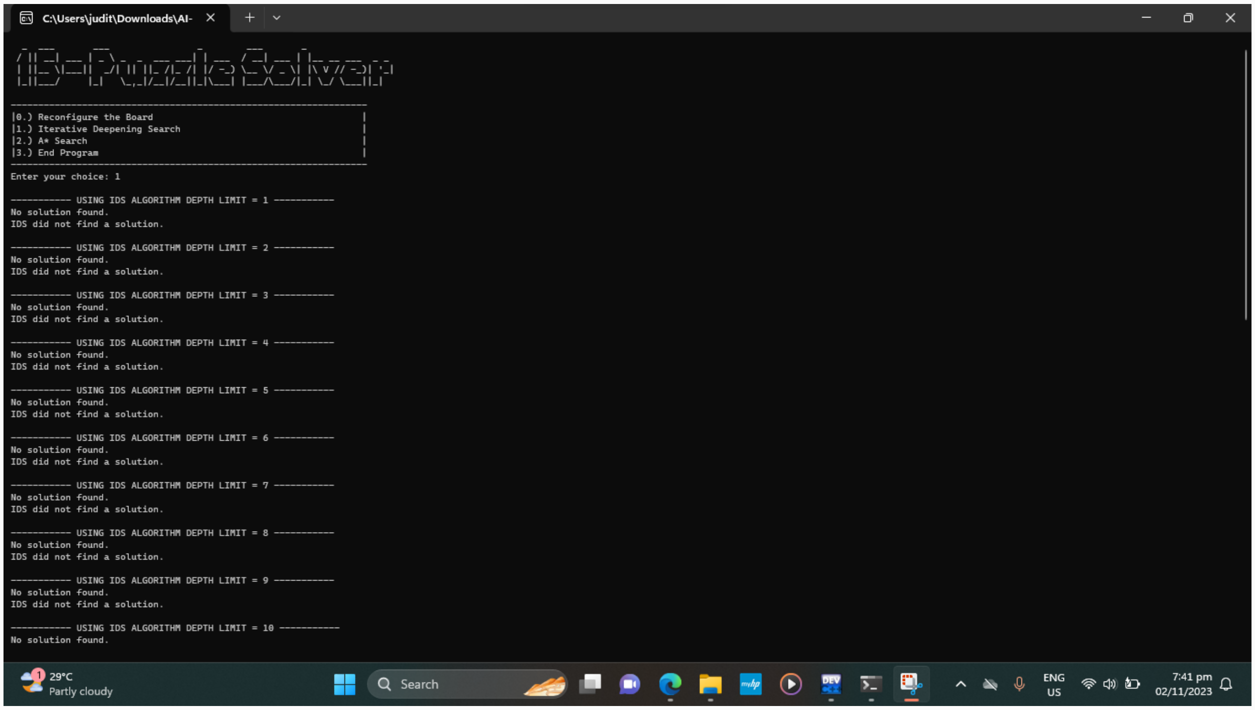
MEDIUM PUZZLE



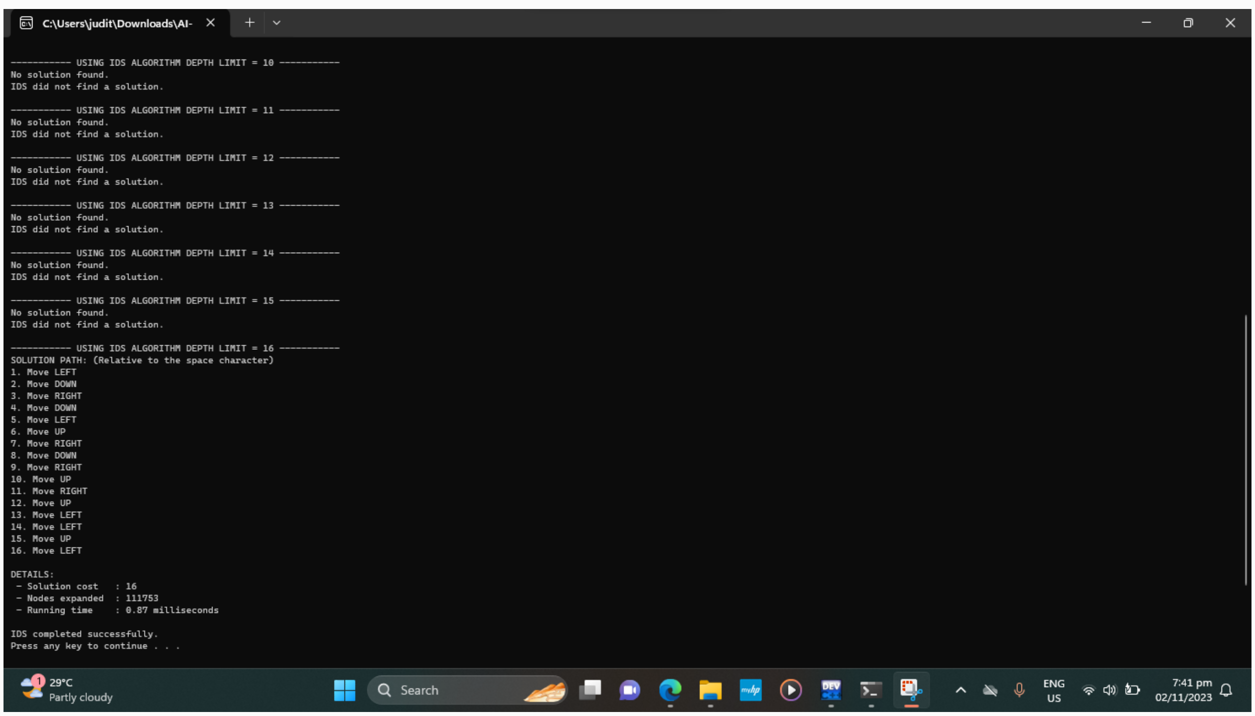



File Edit Selection View Go Run
Search
search_strategies_project.c
C: > Users > User > Downloads > search_strategies_project.c > ...
PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL
Code
|0.) Reconfigure the Board |
|1.) Iterative Deepening Search |
|2.) A* Search |
|3.) End Program |
Enter your choice: 2
----------------- USING A* ALGORITHM ---------------------
SOLUTION PATH: (Relative to the space character)
1. Move UP
2. Move LEFT
3. Move DOWN
4. Move DOWN
5. Move DOWN
6. Move RIGHT
7. Move UP
8. Move RIGHT
9. Move DOWN
10. Move RIGHT
11. Move UP
12. Move LEFT
13. Move UP
14. Move LEFT
15. Move UP
16. Move LEFT
DETAILS:
- Solution cost : 16
- Nodes expanded : 136
- Running time : 0.001 milliseconds
A* Search completed successfully.
Press any key to continue . . .
0 0 0 Live Share
Ln 658, Col 1 Spaces: 4 UTF-8 CRLF {} C Go Live Win32

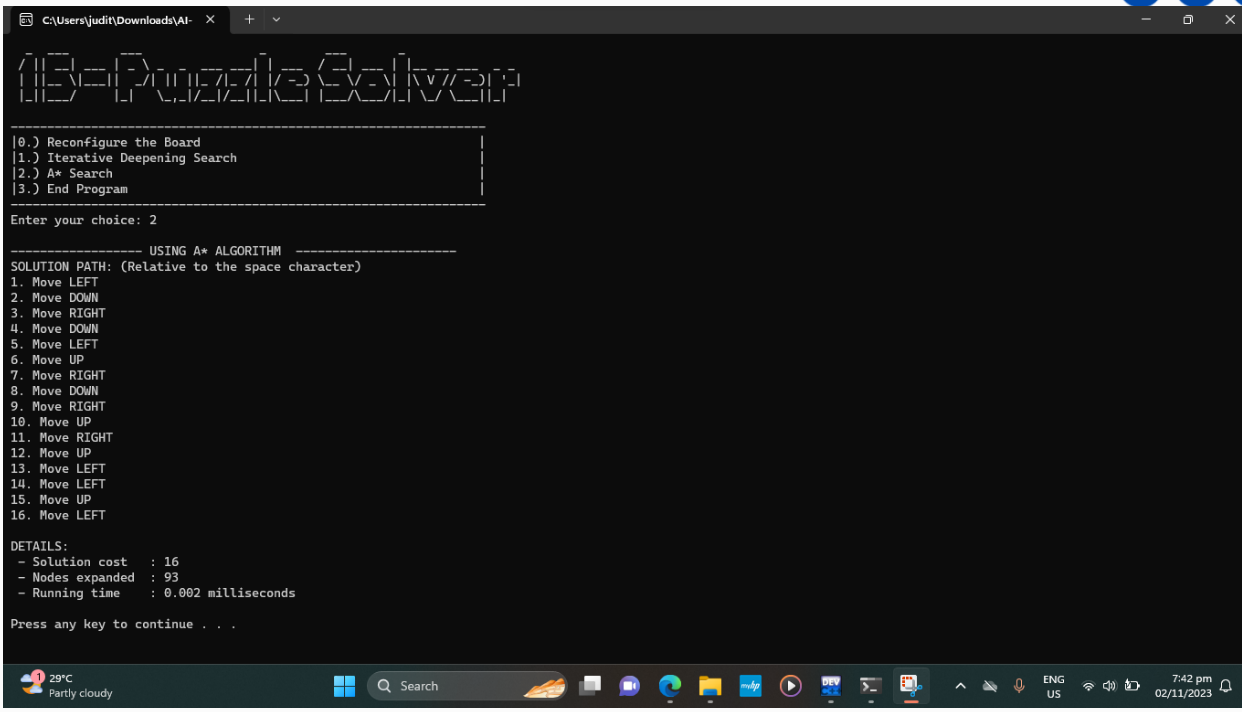
HARD PUZZLE



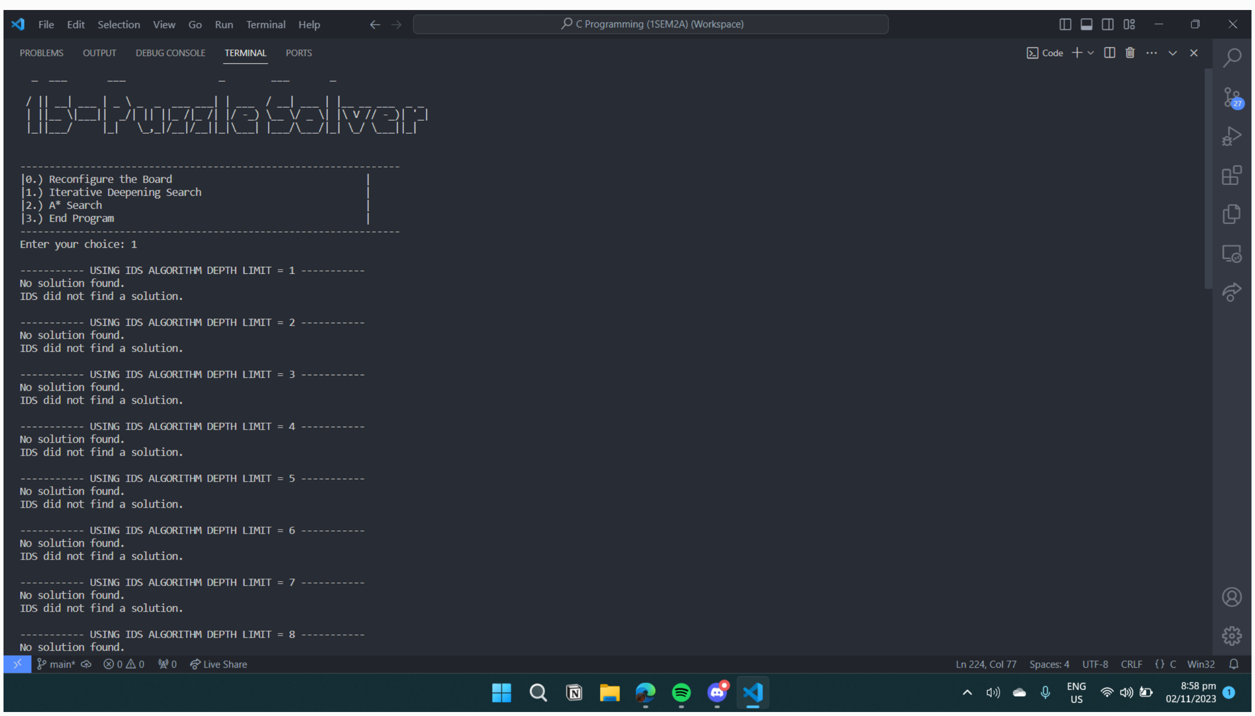
C:\Users\judit\Downloads\AI-
|0.) Reconfigure the Board
|1.) Iterative Deepening Search
|2.) A* Search
|3.) End Program
Enter your choice: 1
----------- USING IDS ALGORITHM DEPTH LIMIT = 1 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 2 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 3 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 4 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 5 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 6 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 7 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 8 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 9 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 10 -----------
No solution found.
29°C
Partly cloudy
Search
ENG
US
7:41 pm
02/11/2023

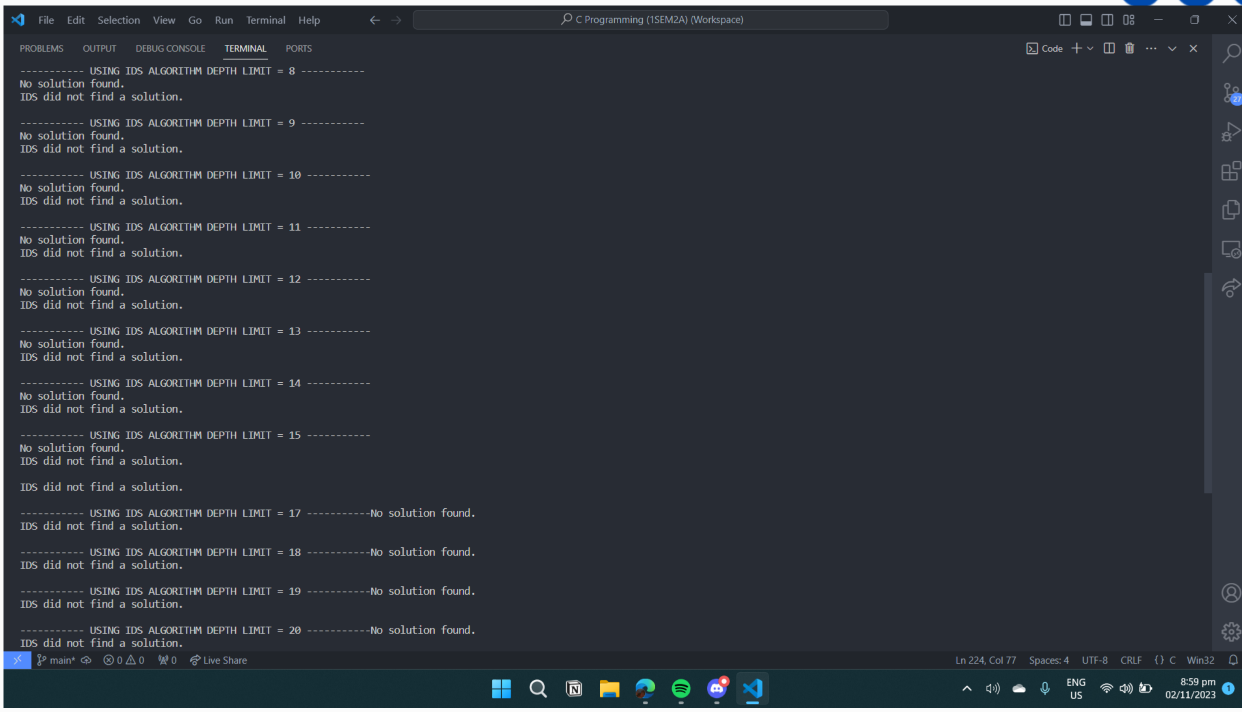


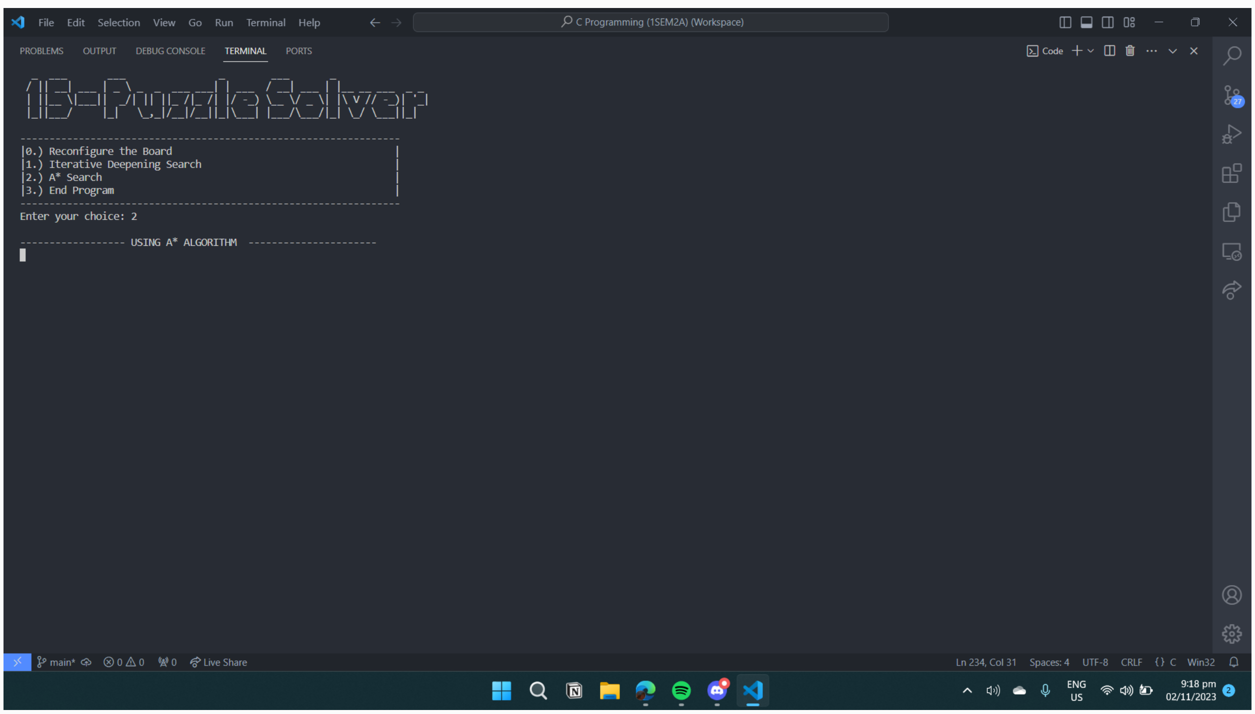
C:\Users\judit\Downloads\AI-
----------- USING IDS ALGORITHM DEPTH LIMIT = 10 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 11 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 12 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 13 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 14 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 15 -----------
No solution found.
IDS did not find a solution.
----------- USING IDS ALGORITHM DEPTH LIMIT = 16 -----------
SOLUTION PATH: (Relative to the space character)
1. Move LEFT
2. Move DOWN
3. Move RIGHT
4. Move DOWN
5. Move LEFT
6. Move UP
7. Move RIGHT
8. Move DOWN
9. Move RIGHT
10. Move UP
11. Move RIGHT
12. Move UP
13. Move LEFT
14. Move LEFT
15. Move UP
16. Move LEFT
DETAILS:
- Solution cost : 16
- Nodes expanded : 111753
- Running time : 0.87 milliseconds
IDS completed successfully.
Press any key to continue . . .
29°C
Partly cloudy
Search
ENG
US
7:41 pm
02/11/2023



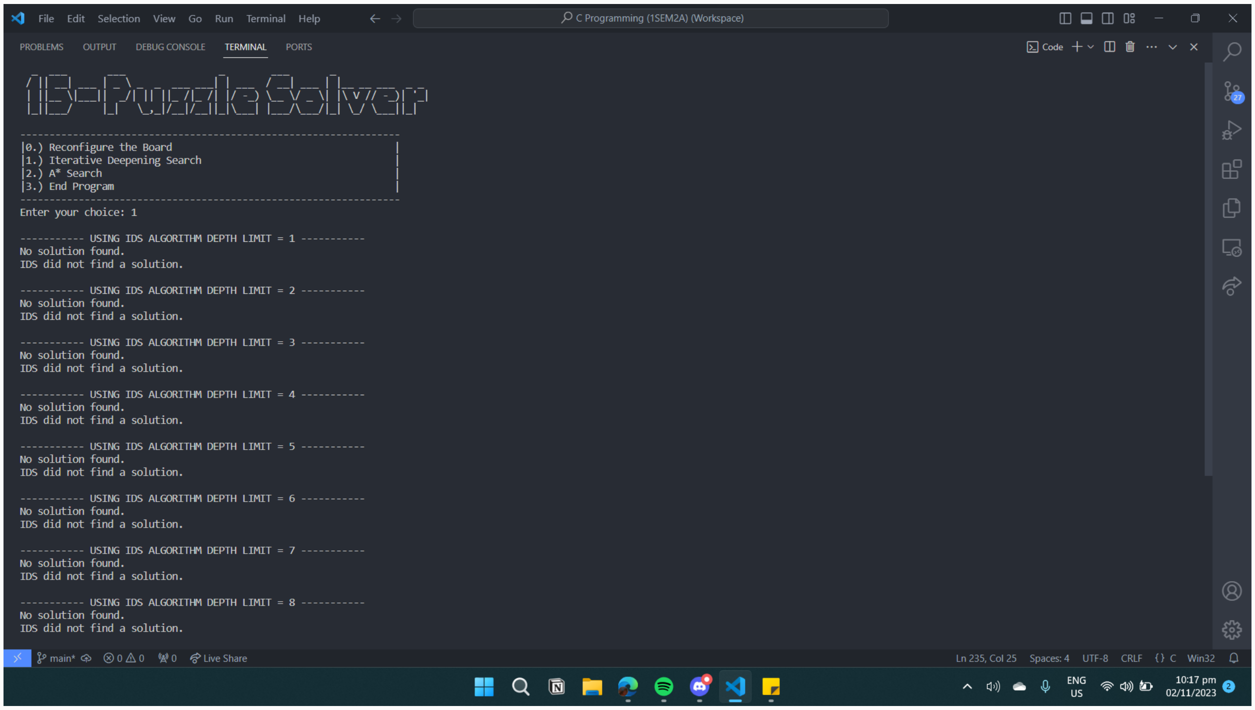
WORST PUZZLE

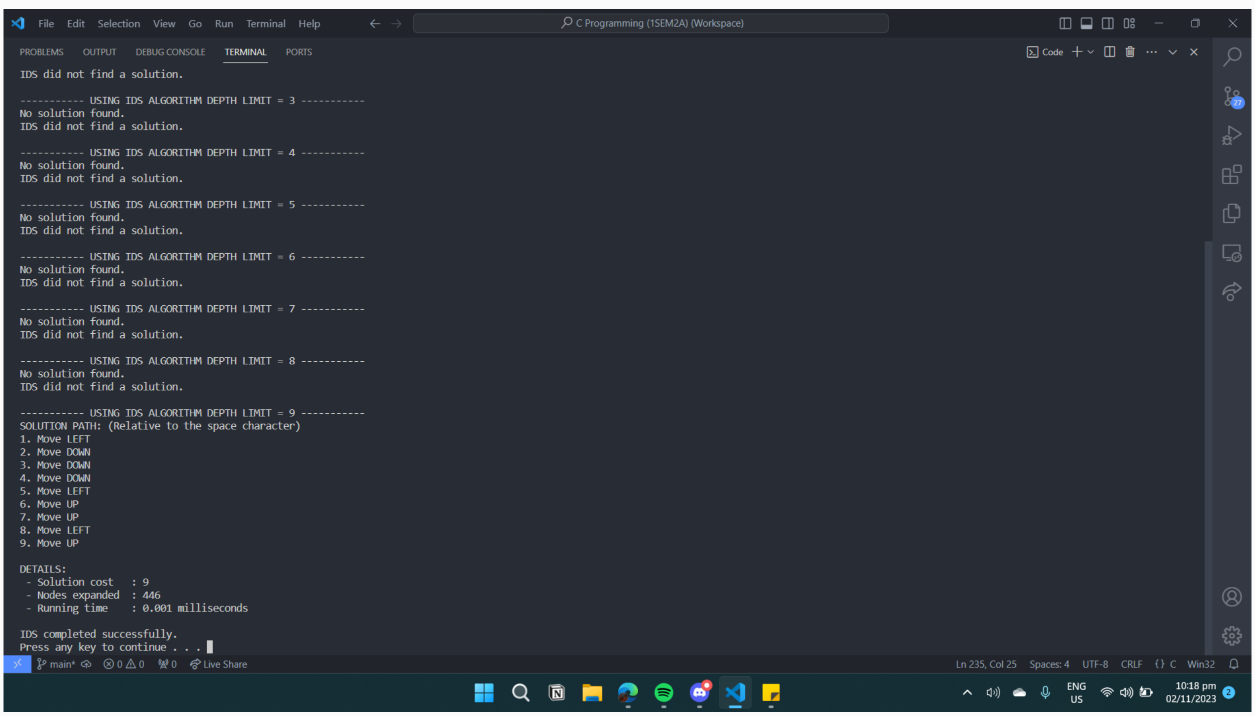


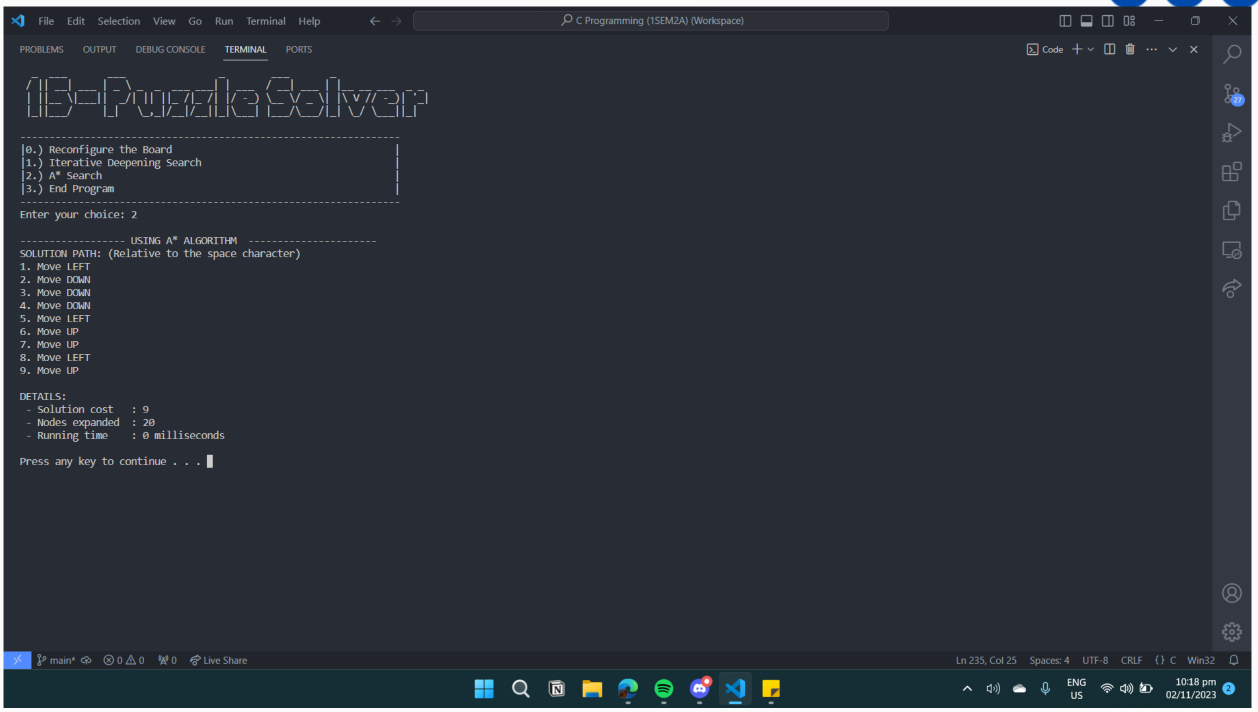

File Edit Selection View Go Run Terminal Help
C Programming (1SEM2A) (Workspace)
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
---------- USING IDS ALGORITHM DEPTH LIMIT = 8 ----------
No solution found.
IDS did not find a solution.
---------- USING IDS ALGORITHM DEPTH LIMIT = 9 ----------
No solution found.
IDS did not find a solution.
---------- USING IDS ALGORITHM DEPTH LIMIT = 10 ----------
No solution found.
IDS did not find a solution.
---------- USING IDS ALGORITHM DEPTH LIMIT = 11 ----------
No solution found.
IDS did not find a solution.
---------- USING IDS ALGORITHM DEPTH LIMIT = 12 ----------
No solution found.
IDS did not find a solution.
---------- USING IDS ALGORITHM DEPTH LIMIT = 13 ----------
No solution found.
IDS did not find a solution.
---------- USING IDS ALGORITHM DEPTH LIMIT = 14 ----------
No solution found.
IDS did not find a solution.
---------- USING IDS ALGORITHM DEPTH LIMIT = 15 ----------
No solution found.
IDS did not find a solution.
IDS did not find a solution.
---------- USING IDS ALGORITHM DEPTH LIMIT = 17 ----------No solution found.
IDS did not find a solution.
---------- USING IDS ALGORITHM DEPTH LIMIT = 18 ----------No solution found.
IDS did not find a solution.
---------- USING IDS ALGORITHM DEPTH LIMIT = 19 ----------No solution found.
IDS did not find a solution.
---------- USING IDS ALGORITHM DEPTH LIMIT = 20 ----------No solution found.
IDS did not find a solution.
main* 0 0 0 Live Share
Ln 224, Col 77 Spaces: 4 UTF-8 CRLF {} C Win32
ENG US 8:59 pm 02/11/2023


File Edit Selection View Go Run Terminal Help
C Programming (1SEM2A) (Workspace)
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
|0.) Reconfigure the Board
|1.) Iterative Deepening Search
|2.) A* Search
|3.) End Program
Enter your choice: 2
----------------- USING A* ALGORITHM ---------------------
main* 0 0 0 Live Share
Ln 234, Col 31 Spaces: 4 UTF-8 CRLF {} C Win32
ENG US 9:18 pm 02/11/2023

PREFERRED INITIAL CONFIGURATION






File Edit Selection View Go Run Terminal Help
C Programming (1SEM2A) (Workspace)
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Code
|0.) Reconfigure the Board |
|1.) Iterative Deepening Search |
|2.) A* Search |
|3.) End Program |
Enter your choice: 2
----------------- USING A* ALGORITHM ---------------------
SOLUTION PATH: (Relative to the space character)
1. Move LEFT
2. Move DOWN
3. Move DOWN
4. Move DOWN
5. Move LEFT
6. Move UP
7. Move UP
8. Move LEFT
9. Move UP
DETAILS:
- Solution cost : 9
- Nodes expanded : 20
- Running time : 0 milliseconds
Press any key to continue . . .
main* 0 0 0 Live Share
Ln 235, Col 25 Spaces: 4 UTF-8 CRLF {} C Win32
ENG US 10:18 pm 02/11/2023

V. Analysis and comparison of the performance of the IDS and A* search algorithms

<table>
<tr><th>Initial State</th><th></th><th>IDS</th><th>A*</th></tr>
<tr><td rowspan="4">Easy
<table><tr><td>4</td><td>2</td><td>3</td><td></td></tr><tr><td>5</td><td>1</td><td>6</td><td>7</td></tr><tr><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>12</td><td>13</td><td>14</td><td>15</td></tr></table></td><td>Solution path</td><td>Move [Left-Left-Down-Left-Up]</td><td>Move [Left-Left-Down-Left-Up]</td></tr>
<tr><td>Solution cost</td><td>5</td><td>5</td></tr>
<tr><td>Number of nodes expanded</td><td>8</td><td>6</td></tr>
<tr><td>Running Time</td><td>0 millisecond</td><td>0 millisecond</td></tr>
<tr><td rowspan="4">Medium
<table><tr><td>5</td><td>6</td><td>2</td><td>3</td></tr><tr><td>1</td><td></td><td>10</td><td>7</td></tr><tr><td>4</td><td>13</td><td>9</td><td>15</td></tr><tr><td>8</td><td>12</td><td>11</td><td>14</td></tr></table></td><td>Solution path</td><td>Move [Up-Left-Down-Down-Down-Right-Up-Right-Down-Right-Up-Left-Up-Left-Up-Left]</td><td>Move [Up-Left-Down-Down-Down-Right-Up-Right-Down-Right-Up-Left-Up-Left-Up-Left]</td></tr>
<tr><td>Solution cost</td><td>16</td><td>16</td></tr>
<tr><td>Number of nodes expanded</td><td>303931</td><td>136</td></tr>
<tr><td>Running Time</td><td>0.684 milliseconds</td><td>0.001 milliseconds</td></tr>
<tr><td rowspan="4">Hard
<table><tr><td>1</td><td>5</td><td>2</td><td>3</td></tr><tr><td>6</td><td></td><td>7</td><td>11</td></tr><tr><td>4</td><td>12</td><td>14</td><td>10</td></tr><tr><td>9</td><td>8</td><td>13</td><td>15</td></tr></table></td><td>Solution path</td><td>Move [Left-Down-Right-Down-Left-Up-Right-Down-Right-Up-Right-Up-Left-Left-Up-Left]</td><td>Move [Left-Down-Right-Down-Left-Up-Right-Down-Right-Up-Right-Up-Left-Left-Up-Left]</td></tr>
<tr><td>Solution cost</td><td>16</td><td>16</td></tr>
<tr><td>Number of nodes expanded</td><td>111753</td><td>93</td></tr>
<tr><td>Running Time</td><td>0.07 milliseconds</td><td>0.002 milliseconds</td></tr>
</table>

<table>
<tr><td>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

Solution cost

Number of nodes expanded

Running Time</td><td>Unsolvable</td><td>Unsolvable</td></tr>
<tr><td>Your preferred initial configuration
<table><tr><td>4</td><td>1</td><td>3</td><td>0</td></tr><tr><td>5</td><td>9</td><td>2</td><td>7</td></tr><tr><td>8</td><td>13</td><td>6</td><td>11</td></tr><tr><td>12</td><td>14</td><td>10</td><td>15</td></tr></table></td><td>Solution path

Solution cost

Number of nodes expanded

Running Time</td><td>Move [Left-Down-Down-Down-Left-Up-Up-Left-Up]

9

446

0.001 milliseconds</td><td>Move [Left-Down-Down-Down-Left-Up-Up-Left-Up]

9

2

0 millisecond</td></tr>
</table>

Analysis:

The analysis of the puzzle-solving algorithms, Iterative Deepening Search (IDS) and A*, revealed their strengths and limitations when applied to puzzles of varying complexities. These findings provide important insights into the algorithms' performance under different conditions.

1. **Easy Puzzle Configuration**
   - For both IDS and A*, the algorithms perform exceptionally well with easy puzzle configurations.
   - These puzzles are solved swiftly and without any issues, which is expected due to the algorithm's efficiency in finding solutions quickly.
2. **Medium Puzzle Configuration**
   - The IDS and A* algorithms continue to exhibit reliable and efficient performance with medium-difficulty puzzles.
   - Both algorithms efficiently find solutions, demonstrating their versatility in handling puzzles of moderate complexity.

3. **Hard Puzzle Configuration**
    - IDS experiences a minor pause but ultimately succeeds in solving hard puzzles.
    - A* operates smoothly and effectively finds solutions for hard puzzles as well.
    - The increased solution times are reasonable given the puzzles' complexity, indicating that the algorithms can tackle challenging scenarios.
4. **Worst Puzzle Configuration**
    - The worst puzzle configuration, which is assumed to be unsolvable, presents significant challenges for both IDS and A*.
    - Both algorithms struggle to handle this puzzle, and they may take an impractical amount of time to run. In some cases, they might lead to system crashes due to excessive memory usage.
5. **Preferred initial configuration**
    - We create a preferred initial configuration for 15-puzzle to test or experiment with solvability and algorithmic approaches. In this context, we consider that both Iterative Deepening Search (IDS) and A* search algorithms are capable of solving and reaching the goal state as specified.
    - Both algorithms efficiently solve the 15-puzzle, and they encounter no issues in finding solution quickly.

The analysis emphasizes the need for implementing solvability checks to prevent algorithms from running on unsolvable puzzles. Additionally, it highlights the importance of setting time or memory limits to avoid system unresponsiveness in extremely challenging scenarios.

In conclusion, the investigation underscores the significance of selecting appropriate algorithms and conducting thorough performance analysis to ensure efficient problem-solving in different puzzle scenarios. These insights can guide the development of more robust and efficient puzzle-solving systems, balancing algorithmic power with resource management to tackle a wide range of puzzle complexities.

VI. Challenges Encountered

During the development of our program, we faced a notable challenge related to debugging the code, with a particular focus on implementing two essential search algorithms: Iterative Deepening Search (IDS) and A*. These algorithms form the core of our program's functionality, and addressing the associated issues requires careful consideration.

1. **Debugging Complexity:** Debugging the code proved to be a complex task, primarily due to the intricacies of IDS and A*. These algorithms involve intricate logic, and even small errors can lead to unexpected behaviors. This challenge compelled us to dive deep into the codebase to diagnose and rectify issues effectively.
2. **Search Cost Computation:** A critical challenge we encountered revolved around accurately computing the search cost. Properly tracking and quantifying the cost incurred during the search process is essential for evaluating the algorithm's efficiency and performance. This required a meticulous examination of the code to ensure accurate cost calculations.
3. **Solution Path in IDS:** In the case of IDS, solving the solution path for relatively easy puzzles presented a conundrum. We observed that IDS sometimes generated the 'up' action even when the zero tile had already reached the top of the puzzle board. Solving this issue required a deep dive into the code to identify the root cause and implement a solution that avoids unnecessary actions. It was a test of patience and problem-solving skills.
4. **Precise Timing:** Measuring the running time of each algorithm accurately was vital for performance assessment. Placing the clock's start and end points in the code at the right locations was pivotal. Incorrect placement could skew the time measurements, potentially leading to misleading performance evaluations. Achieving precise timing required careful consideration and testing.
5. **Worst-case Puzzle:** These puzzles push the boundaries of the computational capabilities of our laptop and can lead to system crashes and sudden stops. The reason behind these crashes lies in the enormous computational load imposed by solving such complex puzzles. These puzzles require a substantial amount of computational resources, including memory and processing power, to explore and analyze the numerous potential moves and states. As a result, the program's execution can become extremely resources-intensive, leading to high memory consumption and long processing times.

To tackle these challenges effectively, we engaged in rigorous code reviews, implemented systematic debugging strategies, and fine-tuned the algorithms to ensure their correctness and efficiency. These experiences not only improved the program's reliability but also deepened our understanding of algorithmic problem-solving and performance evaluation.

In the end, the process of iterative development, where we changed and improved our code repeatedly until it ran smoothly, was a valuable lesson in perseverance and the art of crafting robust software.

VII. Members Participation

Daphne Julienne Agao

- Created a GitHub repository to compile all the codes assigned for every member
- Assist in resolving certain coding bugs
- Helped with coding the Iterative Deepening Search (IDS)
- Helped in designing the written report
- Give suggestions
- Helps on designing, adjusting and aligning the output of the program

Jade Airin Judin

- Helped with coding the Iterative Deepening Search (IDS)
- Assist in the resolution of some issues in the code
- Helped with the documentation of the written report
- Give suggestions
- Helps on designing, adjusting and aligning the output of the program

Judith Balatinsayo

- Helped with coding the A* Search
- Created the doc file
- Assist in the resolution of some issues in the code
- Helped with the documentation of the written report
- Give suggestions

Neslie Marie Colasito

- Helped with coding the A* Search
- Help fix some problem in the code
- Helped with the documentation of the written report
- Give suggestions

Juliet Clarisse Bertulfo

- Helped with coding the A* Search
- Help fix some problem in the code
- Helped with the documentation of the written report
- Give suggestions