Jett Tipsword
CS4346

Project 2 Report

Problem Description: Project 2 tasks a team of students to implement the A* algorithm, specifically the A* FINAL algorithm, in C++ and apply it to the 8 tile puzzle game. The team needs to develop the A* algorithm discussed in class with its heuristic. Also, each team member should individually create their own custom heuristic function, that notably avoids using distance calculation like the provided heuristic. The A* algorithm is a search algorithm that uses the actual cost of getting from the start node to the current node and an estimate of the cost of getting from the current node to the goal node to determine which node to expand next. The 8-puzzle game is a sliding puzzle with a 3x3 grid and 9 numbered tiles that must be rearranged into numerical order. Here the 9th number is 0, but for this project 0 is treated as an empty space. The performance of the algorithm will be analyzed by generating tables with execution time, number of nodes generated, number of nodes expanded, depth of the tree, effective branching factor, and total path for two different initial states of the game. Each member should pick a unique heuristic function, test the program with all four functions, and then analyze the results of the program.

Domain: The domain of this project is using artificial intelligence ideologies and search algorithms, namely the A* algorithm. It also involves the application of these concepts to the 8-puzzle game, which is a classic problem in artificial intelligence and involves intelligent search with heuristic functions. These heuristic functions act as rules for our puzzle game, and which move should be made next based on the heuristic value :”h”. The use of the C++ programming language is ideal here for performance at larger scales, and use of pointers in representing the current state, child state, and parent state.

Methodologies: An A* algorithm is an informed search algorithm that uses both heuristic and cost functions to find the shortest path from the initial state to the goal state. Here, I use two different heuristics: the misplaced tile heuristic and a custom heuristic that counts the number of conflicts and inversions.

This heuristic was chosen for its greedy approach, which allows for better results the shorter the depth to find a solution. The section for conflict heuristic counts the number of pairs of tiles that are in conflict with each other, i.e., the number of pairs of tiles that are in the same row or column and not in their final positions. The more conflicts a board has, the more moves it will require to reach the goal state. The inversion section heuristic counts the number of inversions in the board state, which is just the number of pairs of tiles that are in the wrong order. An inversion occurs when a tile precedes another tile with a lower number on it. The higher the number of inversions, the further the board state is from the goal state. Using these two heuristics together can result in a more accurate estimate of the number of moves required to reach the goal state, as they capture different aspects of the puzzle. Combining these heuristics can result in a more informed search algorithm and can reduce the search space by guiding the algorithm towards more promising paths.

Source Code Implementation: Here we have a C++ implementation of the A* algorithm for solving the 8-puzzle game. To start, we first define a node structure to represent the state of the puzzle. This node structure contains the board state, g, h, f values, and a pointer to the parent node. For the values, g is equal to the depth, h is equal to the heuristic value selected, and f is the sum of gand h. The program then defines a priority queue to store the open nodes and a vector to store the closed nodes. The goal state of the puzzle is defined as a global variable so that all over the functions have access to the correct goal state.Several functions for the board functionality are defined, including printBoard(), getFinalPath(), goalReached(), getSuccessor(), and getSuccessors(). These functions allow the board to act according to standardized rules, and generate child nodes for each current state. The printBoard() function prints the board state of a given node. The getFinalPath() function finds the path from the root node to the goal node and prints each board state along the way. The goalReached() function checks if a given node is the goal node. The getSuccessor() function generates a new node by swapping two tiles in the board state. Finally, for the functionality of the game, the getSuccessors() function generates all possible successor nodes of a given node.

Next, there are 2 functions defined called misplacedTiles() and custom heuristic.The misplacedTileHeuristic() function calculates a heuristic value based on the number of tiles out of place in the board state. The customHeuristic() function calculates a heuristic value for the input node, based on the number of conflicts and inversions in the current board state. Conflict count is the number of pairs of tiles that are in the same row or column, and are not in their correct order. A conflict between two tiles increases the number of moves required to solve the puzzle. Inversion count is the number of tiles that are out of order with respect to their position in the goal state. An inversion occurs when a tile with a higher value appears before a tile with a lower value. The heuristic value returned by the function is the sum of the conflict and inversion counts. A higher heuristic value indicates a greater distance from the goal state, and a higher estimated number of moves required to reach the goal state.

The main() function of the program reads in the initial board state from standard input, creates the root node, and adds it to the priority queue. For this program, the user must select which initial state they would like to use, and which heuristic they would like to use. These are globally defined variables so that all functions know which initial state and which heuristic function is being used. The program then enters a loop where it removes the highest priority node from the queue, generates its successors, and adds them to the queue.The program continues this loop until it finds the goal node or until the maximum depth of the search is reached. Maximum depth is set to 30, well over the depth required. This depth limit allows for more precise debugging, however it is not needed here. Finally, the program prints out statistics about the search, including the number of nodes generated and expanded, the depth of the search, and the path from the root to the goal node, all in a tabular manner.

Source Code

```cpp
#include <iostream>
#include<bits/stdc++.h>
#include <cmath>
#include <ctime>

#include <sys/resource.h>

using namespace std;

// node structure
struct node
{
    vector<int> boardState;
    int g, h, f;
    node *parent;

    node()
    {
        g = 0;
        h = 0;
        f = 0;
        parent = NULL;
    }
};

// operator overloading for priority queue
struct comp
{
    bool operator()(const node *node1, const node *node2) const
    {
        return node1->f > node2->f;
    }
};

int numElem; // number of elements
int NG = 0;  // Nodes Generated
int NE = -1; // Nodes Expanded
int D = 0;   // Depth of tree
int bStar = 0;
int heuChoice = 0;
int maxDepth = 30;
vector<int> goalBoard = {1,2,3,8,0,4,7,6,5};//goal state
node* initGoal = new node();

priority_queue<node *, vector<node *>, comp> OPEN; // priority queue
vector<node *> CLOSED;                             // to keep track of closed nodes
vector<node *> totalPath;                          // all the nodes in the path from the root to the goal

// print board
void printBoard(node *n)
{
    int dim = sqrt(numElem);
    int k = 0;
    for (int i = 0; i < dim; ++i){
        for (int j = 0; j < dim; ++j){
            cout << n->boardState[k++] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

```

```
// print path from root to goal node
void getFinalPath(node *n)
{
    node *temp = n;

    while (temp != NULL){
        totalPath.push_back(temp);
        temp = temp->parent;
    }

    int totalSize = totalPath.size();

    // show the moves one by one
    cout << "Total Path" << endl;
    for (int i = totalSize - 1; i >= 0; --i){
        printBoard(totalPath[i]);
    }
}

// return true if goal nonde
bool goalReached(node *n)
{
    if (n->h == 0){
        return 1;
    }
    else{
        return 0;
    }
}
```

```


//customHeuristic
int customHeuristic(node* n) {

    int conflicts = 0;
    int inversions = 0;
    initGoal->boardState = goalBoard;
    int check = 0;
    for(int i = 0; i < 9; i++){
        if(n->boardState[i] == initGoal->boardState[i] ){
            check++;
        }
    }


    if(check == 9){
        return 0;
    }

    // Count the number of conflicts
    for (int i = 0; i < n->boardState.size() - 1; i++) {
        for (int j = i + 1; j < n->boardState.size(); j++) {
            if (n->boardState[i] > n->boardState[j]) {
                int row_i = i / 3;
                int col_i = i % 3;
                int row_j = j / 3;
                int col_j = j % 3;
                if (row_i == row_j || col_i == col_j) {
                    conflicts++;
                }
            }
        }
    }

    // Count the number of inversions
    for (int i = 0; i < n->boardState.size() - 1; i++) {
        for (int j = i + 1; j < n->boardState.size(); j++) {
            if (n->boardState[i] && n->boardState[j] && n->boardState[i] > n->boardState[j]) {
                inversions++;
            }
        }
    }

    return conflicts + inversions;
}


//calculate heuristic
int misplacedTileHeuristic(node* n){
    int count = 0;
    for(int i = 0; i < numElem; ++i){
        if(goalBoard[i] != n->boardState[i]){
            ++count;//tiles out of place
        }
    }
    return count;
}
```

```
// build successor node
node *getSuccessor(node *state, int pos1, int pos2)
{
    NG++; // increment nodes generated
    node *newState = new node();
    newState->boardState = state->boardState;//copy the board state
    swap(newState->boardState[pos1], newState->boardState[pos2]);
    switch(heuChoice) {//decide which heuristic to use
        case 1:
            newState->h = misplacedTileHeuristic(newState);
            break;
        case 2:
            newState->h = customHeuristic(newState);
            break;
        default:
            std::cout << "Invalid Choice\n";
            break;
    }
    newState->g = state->g + 1; // increment depth
    newState->f = newState->h + newState->g; // apply formula
    newState->parent = state;

    return newState;
}

// generate successors
vector<node *> getSuccessors(node *n)
{
    NE++;
    int pos;
    int row;
    int col;
    int dim;//dimension of the game
    for (int i = 0; i < numElem; ++i)
    {
        if (n->boardState[i] == 0)
        {
            pos = i;
            break;
        }
    }
    dim = sqrt(numElem);
    row = pos / dim;
    col = pos % dim;

    vector<node *> successors;

    if (col != 0){//move left
            successors.push_back(getSuccessor(n, pos, pos - 1));
    }
    if (col != dim - 1){//move right
        successors.push_back(getSuccessor(n, pos, pos + 1));
    }
    if (row != 0){//move up
        successors.push_back(getSuccessor(n, pos, pos - dim));
    }
    if (row != dim - 1){//mode down
        successors.push_back(getSuccessor(n, pos, pos + dim));
    }

    return successors;
}
```

```
// check if node has been previously generated
bool checkOLD(node *n)
{
    int totalSize = CLOSED.size(), j;
    for (int i = 0; i < totalSize; ++i){
        for (j = 0; j < numElem; ++j){
            //check if the node is inside CLOSED
            if (n->boardState[j] != CLOSED[i]->boardState[j])
                break;
        }
        if (j == numElem){//node was found
            return 1;
        }
    }
    return 0;//node was not found
}

void A_star(node *n)
{
    switch(heuChoice) {//decide which heuristic to use
        case 1:
            n->h = misplacedTileHeuristic(n);
            break;
        case 2:
            n->h = customHeuristic(n);
            break;
        default:
            std::cout << "Invalid Choice\n";
            break;
    }
    n->f = n->h;
    n->parent = NULL;
    OPEN.push(n);

    bool done;
    int totalGCost, totalSize, k;
    node *current, *temp;
    vector<node *> currentSuccessors;
    priority_queue<node *, vector<node *>, comp> Pqueue;

```

```
    while (!OPEN.empty()){//loop as long as there is boards in OPEN
        current = OPEN.top();//best f value board
        OPEN.pop();
        CLOSED.push_back(current);

        if (goalReached(current)){
            getFinalPath(current);
            return;
        }

        currentSuccessors.clear();
        currentSuccessors = getSuccessors(current);

        totalSize = currentSuccessors.size();
        for (int i = 0; i < totalSize; ++i){
            if (checkOLD(currentSuccessors[i])){
                continue;
            }

            totalGCost = current->g + 1;

            while (!Pqueue.empty()){
                Pqueue.pop();
            }
            while (!OPEN.empty()){
                temp = OPEN.top();
                OPEN.pop();
                done = 0;

                for (k = 0; k < numElem; ++k){//loop through elements
                    if (currentSuccessors[i]->boardState[k] != temp->boardState[k]){
                        break;
                    }
                }
                if (k == numElem){
                    done = 1;
                }

                if (done && totalGCost < temp->g){//if done with successors, and total
                    temp->parent = current;
                    temp->g = totalGCost;
                    temp->f = temp->g + temp->h;// apply f = g + h
                }
                Pqueue.push(temp);
            }
            if (!done){
                Pqueue.push(currentSuccessors[i]);
            }
            OPEN = Pqueue;

            // update the depth
                if (currentSuccessors[i]->g > D) {
                    D = currentSuccessors[i]->g;
                }

                // check if the depth limit has been reached
                if (D == maxDepth) {
                    getFinalPath(currentSuccessors[i]);
                    cout << "GOAL NOT REACHED" << endl << endl;
                    return;
                }
        }
    }

    return;
}
```

```cpp

int main()
{

    struct rusage usage;

    node *newNode = new node();
    numElem = 9;

    node* initNode1 = new node();
    node* initNode2 = new node();
    node* initGoal = new node();

    cout << "Goal State" << endl;
    initGoal->boardState = goalBoard;
    printBoard(initGoal);

    cout << "State 1" << endl;
    vector<int> puzzle1 = {2,8,3,1,6,4,0,7,5};//initial state 1
    initNode1->boardState = puzzle1;
    printBoard(initNode1);

    cout << "State 2" << endl;
    vector<int> puzzle2 = {2,1,6,4,0,8,7,5,3};//initial state 2
    initNode2->boardState = puzzle2;
    printBoard(initNode2);


    cout << "Which Initial State?: ";
    vector<int> puzzle = {0};
    int puzzleChoice = 0;
    cin >> puzzleChoice;
    cout << endl;

    cout << "Which Heuristic would you like to use?: " << endl;
    cout << "(1) Misplaced Tiles or (2) Custom Heuristic: ";
    cin >> heuChoice;
    cout << endl;

    if (puzzleChoice == 1){ // initial state 1 selected
        puzzle = puzzle1;
    }
    else if (puzzleChoice == 2){ // initial state 2 selected
        puzzle = puzzle2;
    }
    else{ // wrong initial state
        cout << "invalid choice" << endl;
        return 0;
    }
    cout << endl;

```

```
    //start time 1
    getrusage(RUSAGE_SELF, &usage);
    clock_t start_r = usage.ru_utime.tv_sec * 1000000 + usage.ru_utime.tv_usec;

    newNode->boardState = puzzle;
    cout << "Initial State" << endl;
    printBoard(newNode);

    A_star(newNode);
    D = totalPath.size() - 1;
    bStar = ((float)NG / (float)D);

    //end time
    getrusage(RUSAGE_SELF, &usage);
    clock_t end_r = usage.ru_utime.tv_sec * 1000000 + usage.ru_utime.tv_usec;
    double cpu_time_used_r = ((double) (end_r - start_r)) / 1000000;
    float ET = cpu_time_used_r;
    stringstream ss;
    ss << fixed << setprecision(6) << ET;
    string Exact_ET = ss.str();


    cout << "METRICS: Execution Time(ET), Nodes Generated (NG), " << endl
    << "Nodes Expanded (NE), Depth (D), and Branching Factor (b*)" << endl << endl;
    const int col_width = 6;
    cout << left << setw(col_width + 5) << " ET"
                 << setw(col_width) << "NG"
                 << setw(col_width) << "NE"
                 << setw(col_width) << "D"
                 << setw(col_width) << "b*" << endl;

    cout << left << setw(col_width + 5) << Exact_ET
                 << setw(col_width) << NG
                 << setw(col_width) << NE
                 << setw(col_width) << D
                 << setw(col_width) << bStar << endl;
}
```

Copy of program run(Jett's Custom heuristic):

```
[jbt71@eros P2]$ g++ -o Proj2 -std=c++11 Project2_jbt71.cpp
[jbt71@eros P2]$ ./Proj2
Goal State
1 2 3
8 0 4
7 6 5

State 1
2 8 3
1 6 4
0 7 5

State 2
2 1 6
4 0 8
7 5 3

Which Initial State?: 1

Which Heuristic would you like to use?:
(1) Misplaced Tiles or (2) Custom Heuristic: 2


Initial State
2 8 3
1 6 4
0 7 5

Total Path
2 8 3
1 6 4
0 7 5

2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

METRICS: Execution Time(ET), Nodes Generated (NG),
Nodes Expanded (NE), Depth (D), and Branching Factor (b*)

 ET         NG     NE     D     b*
0.000091    20     6      6     3
```

```
Goal State
1 2 3
8 0 4
7 6 5

State 1
2 8 3
1 6 4
0 7 5

State 2
2 1 6
4 0 8
7 5 3

Which Initial State?: 2

Which Heuristic would you like to use?:
(1) Misplaced Tiles or (2) Custom Heuristic: 2


Initial State
2 1 6
4 0 8
7 5 3

Total Path
2 1 6
4 0 8
7 5 3

2 0 6
4 1 8
7 5 3

0 2 6
4 1 8
7 5 3

4 2 6
0 1 8
7 5 3

4 2 6
1 0 8
7 5 3

4 2 6
1 8 0
7 5 3

4 2 0
1 8 6
7 5 3
```

```
4 0 2
1 8 6
7 5 3

0 4 2
1 8 6
7 5 3

1 4 2
0 8 6
7 5 3

1 4 2
8 0 6
7 5 3

1 4 2
8 6 0
7 5 3

1 4 2
8 6 3
7 5 0

1 4 2
8 6 3
7 0 5

1 4 2
8 0 3
7 6 5

1 0 2
8 4 3
7 6 5

1 2 0
8 4 3
7 6 5

1 2 3
8 4 0
7 6 5

1 2 3
8 0 4
7 6 5

METRICS: Execution Time(ET), Nodes Generated (NG),
Nodes Expanded (NE), Depth (D), and Branching Factor (b*)

 ET         NG    NE    D     b*
80.563606  34617 12756 18    1923
```

Analysis of the program: The program implementation and modified features can have a significant impact on the results of the program. In this case, the program's layout follows good programming practices, including encapsulation, modularity, and separation of concerns, which makes it optimal. The usage of global variables is minimized, and each function performs a single task, which contributes to the program's efficiency and maintainability.One of the most critical features of the program is dynamic memory allocation for nodes, which reduces memory usage and enables efficient use of memory resources. This feature is especially crucial in resource-constrained environments, where memory usage needs to be optimized. It also facilitates future modifications and upgrades to the program.

The time complexity of the program is O(b^(h+1)), where b is the branching factor, and h is the depth of the optimal solution. The branching factor is the number of possible choices at each node, and the depth of the optimal solution is the number of levels in the search tree required to reach the optimal solution. This complexity provides insight into the program's performance and scalability, and it can help identify areas for optimization.The space complexity of the program is O(b^h), where b is the branching factor, and h is the maximum depth of the tree. This complexity determines the maximum amount of memory required to store the search tree and can also help identify areas for optimization.

Modified features included in the program can also have a significant impact on the results. For example, the custom heuristics developed by Jett, Rafael, and Rayyan are modified features that can significantly affect the program's performance. The custom heuristics take into account specific properties of the game state and can provide a faster and more accurate solution.

In conclusion, the implementation and modified features of the program can have a significant impact on the program's performance and results. By following good programming practices, minimizing memory usage, and developing custom heuristics, the program can be optimized for efficiency, scalability, and accuracy.

Misplaced Tiles State 1:

```
Goal State
1 2 3
8 0 4
7 6 5

State 1
2 8 3
1 6 4
0 7 5

State 2
2 1 6
4 0 8
7 5 3

Which Initial State?: 1

Which Heuristic would you like to use?:
(1) Misplaced Tiles or (2) Custom Heuristic: 1


Initial State
2 8 3
1 6 4
0 7 5

Total Path
2 8 3
1 6 4
0 7 5

2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

METRICS: Execution Time(ET), Nodes Generated (NG),
Nodes Expanded (NE), Depth (D), and Branching Factor (b*)

 ET         NG    NE    D     b*
0.000000    27    8     6     4
```

Misplaced Tiles State 2:

```
Goal State
1 2 3
8 0 4
7 6 5

State 1
2 8 3
1 6 4
0 7 5

State 2
2 1 6
4 0 8
7 5 3

Which Initial State?: 2

Which Heuristic would you like to use?:
(1) Misplaced Tiles or (2) Custom Heuristic: 1


Initial State
2 1 6
4 0 8
7 5 3

Total Path
2 1 6
4 0 8
7 5 3

2 1 6
4 8 0
7 5 3

2 1 0
4 8 6
7 5 3

2 0 1
4 8 6
7 5 3

2 8 1
4 0 6
7 5 3

2 8 1
4 6 0
7 5 3

2 8 1
4 6 3
7 5 0

2 8 1
4 6 3
7 0 5

2 8 1
4 0 3
7 6 5
```

```
2 8 1
0 4 3
7 6 5

0 8 1
2 4 3
7 6 5

8 0 1
2 4 3
7 6 5

8 1 0
2 4 3
7 6 5

8 1 3
2 4 0
7 6 5

8 1 3
2 0 4
7 6 5

8 1 3
0 2 4
7 6 5

0 1 3
8 2 4
7 6 5

1 0 3
8 2 4
7 6 5

1 2 3
8 0 4
7 6 5

METRICS: Execution Time(ET), Nodes Generated (NG),
Nodes Expanded (NE), Depth (D), and Branching Factor (b*)

 ET         NG    NE    D     b*
2.112283    5219  1873  18    289
```

Rafael Custom Heuristic State 1:

```
Which Initial State?
1

Initial State
2 8 3
1 6 4
0 7 5

Total Path
2 8 3
1 6 4
0 7 5

2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

Execution time: 1323021 microseconds
Nodes Generated: 27
Nodes Expanded: 8
Depth: 6
b* Factor: 4.5
```

Rafael's Custom Heuristic State 2:

```
Which Initial State?
2

Initial State
2 1 6
4 0 8
7 5 3

Total Path
2 1 6
4 0 8
7 5 3

2 1 6
4 8 0
7 5 3

2 1 0
4 8 6
7 5 3

2 0 1
4 8 6
7 5 3

2 8 1
4 0 6
7 5 3

2 8 1
4 6 0
7 5 3

2 8 1
4 6 3
7 5 0

2 8 1
4 6 3
7 0 5

2 8 1
4 0 3
7 6 5

2 8 1
0 4 3
7 6 5

0 8 1
2 4 3
7 6 5
```

```
0 8 1
2 4 3
7 6 5

8 0 1
2 4 3
7 6 5

8 1 0
2 4 3
7 6 5

8 1 3
2 4 0
7 6 5

8 1 3
2 0 4
7 6 5

8 1 3
0 2 4
7 6 5

0 1 3
8 2 4
7 6 5

1 0 3
8 2 4
7 6 5

1 2 3
8 0 4
7 6 5

Execution time: 5011527 microseconds
Nodes Generated: 5649
Nodes Expanded: 2008
Depth: 18
b* Factor: 313.833
```

Rayyan’s Custom Heuristic State 1

```
Goal State
1 2 3
8 0 4
7 6 5

State 1
2 8 3
1 6 4
0 7 5

State 2
2 1 6
4 0 8
7 5 3

Which Initial State?: 1

Which Heuristic would you like to use?:
(1) Misplaced Tiles or (2) Custom Heuristic: 2


Initial State
2 8 3
1 6 4
0 7 5

Total Path
2 8 3
1 6 4
0 7 5

2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

METRICS: Execution Time(ET), Nodes Generated (NG),
Nodes Expanded (NE), Depth (D), and Branching Factor (b*)

 ET         NG    NE    D     b*
0.000000    30    10    6     5
```

Rayyan Custom Heuristic State 2:

```
Goal State
1 2 3
8 0 4
7 6 5

State 1
2 8 3
1 6 4
0 7 5

State 2
2 1 6
4 0 8
7 5 3

Which Initial State?: 2

Which Heuristic would you like to use?:
(1) Misplaced Tiles or (2) Custom Heuristic: 2


Initial State
2 1 6
4 0 8
7 5 3

Total Path
2 1 6
4 0 8
7 5 3

2 1 6
4 8 0
7 5 3

2 1 0
4 8 6
7 5 3

2 0 1
4 8 6
7 5 3

2 8 1
4 0 6
7 5 3

2 8 1
4 6 0
7 5 3

2 8 1
4 6 3
7 5 0

2 8 1
4 6 3
7 0 5

2 8 1
4 0 3
7 6 5

2 8 1
0 4 3
7 6 5
```

```
0 8 1
2 4 3
7 6 5

8 0 1
2 4 3
7 6 5

8 1 0
2 4 3
7 6 5

8 1 3
2 4 0
7 6 5

8 1 3
2 0 4
7 6 5

8 1 3
0 2 4
7 6 5

0 1 3
8 2 4
7 6 5

1 0 3
8 2 4
7 6 5

1 2 3
8 0 4
7 6 5

METRICS: Execution Time(ET), Nodes Generated (NG),
Nodes Expanded (NE), Depth (D), and Branching Factor (b*)

 ET        NG    NE    D    b*
0.396633   2385  863   18   132
```

Tabulation of Results (Total Path located at Page number)

Initial State 1

Heuristic	ET	NG	NE	D	b*	TP
Misplaced Tiles	0.000104	27	8	6	4	Pg: 15
Jett's Custom Heuristic	0.000000	20	6	6	3	Pg: 12
Rafael's custom Heuristic	1.323021	827	8	6	4.5	Pg: 17
Rayyan custom Heuristic	.000086	30	10	6	5	Pg: 19

Initial State 2

Heuristic	ET	NG	NE	D	b*	TP
Misplaced Tiles	2.107693	5219	1873	18	289	Pg: 16
Jett's Custom Heuristic	80.67483	34617	12756	18	1923	Pg: 13
Rafael's custom Heuristic	5.811527	5649	2008	18	313.833	Pg: 18
Rayyan custom Heuristic	.398507	2385	863	28	132	Pg: 20

Analysis of Results: The results present the performance of four different heuristics on two different initial states of a puzzle. The heuristics are evaluated based on several metrics, including execution time, nodes generated, nodes expanded, depth, and branching factor.

For Initial State 1, the best performing heuristic in terms of execution time is Jett's Custom Heuristic, which took only 0.000000 seconds to execute. LIkely was the case that this heuristic didn't actually take 0 execution time, it was just faster than the specified cutoff for time. However, Rafael's custom heuristic generated the most nodes (827) and had the longest execution time (1.323021 seconds) among all heuristics on this initial state. Misplaced Tiles and Rayyan's custom heuristic took less than a second to execute and generated fewer nodes than the other two heuristics.

For Initial State 2, Jett's Custom Heuristic took the longest execution time (80.67483 seconds) and generated the most nodes (34617). Rafael's custom heuristic and Misplaced Tiles also performed relatively poorly on this initial state, as they generated more nodes and had a longer execution time compared to Rayyan's custom heuristic. The maximum depth of any heuristic was (18) for all heuristics on both initial states. The branching factor for Initial State 1 was on average lower (between 3 and 5) compared to Initial State 2 (between 132 and 1923).

Overall, the heuristics' performance on Initial State 2 was worse than on Initial State 1, as indicated by the higher number of nodes generated and longer execution times. Some initial states will always take more steps to complete than others. Properly tuning heuristic parameters and ensuring that the program follows good programming practices such as modularity and encapsulation can help optimize the program's performance.

Conclusion: In this project, we have developed a program to solve the 8-tile puzzle game using different heuristic functions. We have implemented three different heuristic functions, including misplaced tiles, custom heuristics by Jett, Rafael, and Rayyan. We have also modified some features in the program to optimize its performance and produce better results for the solution towards the game. This project has shown me how artificial intelligence methods and algorithms can be used in games or other real life scenarios. I have come to realize that the choice of heuristic can drastically change the results of the algorithm, even if the same algorithm is used. Overall, this was an enjoyable and informative assignment, in which I have learned much from.

Team Member contributions: Jett's main contribution to the project was the development of the misplaced tiles heuristic, and implementing the 8-Tile Puzzle game. This heuristic involves counting the number of tiles that are in the wrong position on the board and using that as an estimate of how far the current state is from the goal state. Jett worked on implementing this heuristic into the A* search algorithm and fine-tuning it to improve its accuracy and speed.

Rafael contributed significantly to the structure of the nodes and the priority queue used in the A* search algorithm. He worked on designing a node structure that efficiently stores the board state, the cost of the current state, and the estimated cost to the goal state. Rafael also implemented a priority queue that sorts the nodes based on their total cost, which is a combination of the current cost and the estimated cost to the goal state. These contributions are what optimized our use of the puzzle game and the board states.

Rayyan focused on the A* search algorithm. He worked making sure that the algorithm followed the examples provided. He also gave us some helpful resources in understanding the algorithm, and how it works with varying heuristics. Rayyan also contributed to the testing and validation of the algorithm, ensuring that it provides accurate and efficient solutions to the puzzle.

Overall, each team member made significant contributions to the project, and their contributions complemented each other's strengths. Jett's focus on the heuristic function and the Puzzle game’s functionality , Rafael's work on the node structure and priority queue, and Rayyan's optimization of the A* search algorithms and finding resources all played critical roles in the project's success. The team worked together equally, and their collaboration and communication were key to the project's successful completion.

References:

- Lecture Slides SCP#5 on canvas
- CS 4346 Project #2 Spring 2023.doc on canvas
- GeeksforGeeks. (2023, March 8). A* search algorithm. GeeksforGeeks. Retrieved April 2, 2023, from https://www.geeksforgeeks.org/a-search-algorithm/