Project 2

Robert Jones & Jonathan Spive

11/25/24

Problem Description:

In this project, we implemented the A* search algorithm on a classic 8-puzzle problem. The game is a sliding puzzle consisting of a 3x3 grid where each cell contains a number 1-8 with one empty space 0. Given an initial state and a final state, the goal is to slide numbered tiles into the empty space until the goal state is given. The A* search algorithm will be implemented at every state to explore the best possible next step. Using the heuristic functions created by the team members, the algorithm will be able to make educated guesses on the best possible next move (or the worst).

Domain:

State Space:

Each state is represented as a 3x3 grid with positions and numbers defined for each spot.

Actions:

The blank space is limited to moving up down left or right one space, at the most. One of these moves will be negated when on a wall, and another will be negated after moving from one position to another (no going back).

Goal & initial state:

Initial 1:
2 8 3
1 6 4
0 7 5

initial 2:
2 1 6
4 0 8
7 5 3

Goal:
1 2 3
8 0 4
7 6 5

Solution:

Our solution will be all the combined moves it takes to get form the initial state, to the final state.

Methodology:

To solve the 8 square we use the A* search best-first search. The algorithm uses path cost and a heuristic value based on the created heuristic functions, to determine the best possible next move numerically.

- f(n) = g(n) + h(n)
- where g(n) is the total cost of the path to our current state n.
- h(n) is the estimated cost from state n to the goal based on the heuristic.

Algorithm Steps:

- Initialize the open list with the root node (initial state).
- Repeat until the open list is empty:
    - Remove the node with the smallest f(n) from the open list (best state to explore).
    - If this node matches the goal state, return the path and metrics.
    - Otherwise, generate all valid child nodes by moving the blank space.
    - Assign f(n) values to child nodes and add them to the open list if they are not already explored.
    - Add the current node to the closed list.
    - Reorder the open list based on f(n) values.
- If the open list becomes empty, return failure (no solution).

Heuristics:

h1(n): Counts the number of tiles out of place compared to the goal state.

h2(n): Calculates the Manhattan distance (sum of vertical and horizontal displacements) for all tiles to their correct positions.

RobertsHueristic: Roberts heuristic calculates the total number of tiles that are in the wrong row and the wrong column compared to their positions in the goal state. It counts the row and column misplacements separately and sums them

JohnathansHeuristic: Jonathan's Heuristic function. Calculates how many tiles of the current state are out of place by 1 move. The first loop iterates through the current state of the puzzle, finds an out of place tile, the second loop iterates through the goal state to look for that tile's spot. Then the distance is calculated using: distance = |current_i - goal_i| + |current_j - goal_j|. If the distance is exactly 1, then the almostInPlace variable is incremented. Returns almostInPlace when all looping in complete.

The heuristics help A* search explores all paths and determine which is the most optimal to reach our goal based off the logic of the heuristic and the knowledge of the already taken path.

Source Code Implementation:

This source code implements an A* search algorithm to solve the 8-puzzle problem using multiple heuristic functions. The code defines the goal state and two initial puzzle configurations as 3x3 grids. Four heuristic functions are provided: **H1** (counts misplaced tiles), **H2** (calculates Manhattan distance), Jonathan's heuristic (counts tiles out of place by one move), and Robert's heuristic (counts row and column misplacements). The A* algorithm is implemented using a priority queue (min-heap) for exploring each state, and a closed list to avoid revisiting nodes. It generates child nodes by exploring valid moves, calculates their heuristic and cost values, and expands nodes until the goal state is reached. The program also computes and outputs key performance metrics, such as execution time, nodes generated/expanded, depth, and effective branching factor. This code showcases heuristic evaluation's role in guiding the A* search for optimal performance.

Source Code:

```
//Introduction to AI, Project 2
//Group Members: Jonathan Spive, Robert Jones

#include <vector>
#include <iostream>
#include <functional>
#include <queue>
#include <chrono>

using namespace std;
using namespace chrono;

//Goal State after the A* search is done. Defined as a 2D Vector.
const vector<vector<int>> goalState = {
    {1, 2, 3},
    {8, 0, 4},
    {7, 6, 5}
};

//First of the initial states of the puzzle, to be sorted by the A*. Defined as a 2D Vector.
const vector<vector<int>> initialState1 = {
    {2, 8, 3},
    {1, 6, 4},
    {0, 7, 5}
};

//Second of the initial states of the puzzle, to be sorted by the A*. Defined as a 2D Vector.
const vector<vector<int>> initialState2 = {
    {2, 1, 6},
    {4, 0, 8},
    {7, 5, 3}
};

// H1(n) function. Takes both the goal, and the current state of the puzzle as parameters.
//  Iterates through both vectors to find how many tiles are out of place, and increments the
//  outOfPlace variable if there is a mismatch. Returns this number as an integer.
int h1 (vector<vector<int>> goal, vector<vector<int>> current)
{
    int outOfPlace = 0;

    for(int i = 0; i < 3; i++){
        for(int j = 0; j < 3; j++) {
            if(current[i][j] != goal[i][j]) {
                outOfPlace++;
            }
        }
    }

    return outOfPlace;
}
```

```
//H2(n) function. Takes both the goal and the current state of the puzzle as parameters.
//  Iterates through the current state vector and then iterates through the goal vector to find the
//  distance each misplaced tile is from it's correct spot. Uses a distance formula in the context
//  of a for loop: distance = |current_i - goal_i| + |current_j - goal_j|, then adds the result to
//  distanceSum. Returns distanceSum when all looping is complete.
int h2 (vector<vector<int>> goal, vector<vector<int>> current)
{
    int distanceSum = 0;

    for(int row = 0; row < 3; row++){
        for(int column = 0; column < 3; column++) {
            int currentTile = current[row][column];

            if(currentTile != 0) {
                for(int goalRow = 0; goalRow < 3; goalRow++) {
                    for(int goalColumn = 0; goalColumn < 3; goalColumn++) {
                        if(goal[goalRow][goalColumn] == currentTile) {
                            distanceSum += abs(row - goalRow) + abs(column - goalColumn);
                            break;
                        }
                    }
                }
            }
        }
    }

    return distanceSum;
}


//Jonathan's Heuristic function. Calclulates how many tiles of the current state are out of place by
//  1 move. The first loop iterates through the current state of the puzzle, finds an out of place
//  tile, the second loop iterates through the goal state to look for that tile's spot. Then it
//  the distance is calculated using: distance = |current_i - goal_i| + |current_j - goal_j|. If the
//  the distance is exactly 1, then the almostInPlace variable is incremented. Returns almostInPlace
//  when all looping in complete.
int jonathanHeuristic (vector<vector<int>> goal, vector<vector<int>> current)
{
    int almostInPlace = 0;

    for(int i = 0; i < 3; i++){
        for(int j = 0; j < 3; j++) {
            if(current[i][j] != goal[i][j]) {
                int currentTile = current[i][j];

                if(currentTile != 0) {
                    for(int goalRow = 0; goalRow < 3; goalRow++) {
                        for(int goalColumn = 0; goalColumn < 3; goalColumn++) {
                            if(goal[goalRow][goalColumn] == currentTile) {
                                int distance = abs(i - goalRow) + abs(j - goalColumn);
                                if(distance == 1) almostInPlace++;
                                break;
                            }
                        }
                    }
                }
            }
        }
    }

    return almostInPlace;
}
```

```cpp
// Roberts heuristic calculates the total number of tiles that are in the wrong row and the wrong
// column compared to their positions in the goal state. It counts the row and column
// misplacements separately and sums them.
int robertHeuristic1(vector<vector<int>> goal, vector<vector<int>> current) {
    int misplacedRows = 0, misplacedColumns = 0;

    for (int row = 0; row < 3; ++row) {
        for (int col = 0; col < 3; ++col) {
            int currentTile = current[row][col];
            if (currentTile != 0) {
                for (int goalRow = 0; goalRow < 3; ++goalRow) {
                    for (int goalCol = 0; goalCol < 3; ++goalCol) {
                        if (goal[goalRow][goalCol] == currentTile) {
                            if (row != goalRow) misplacedRows++;
                            if (col != goalCol) misplacedColumns++;
                        }
                    }
                }
            }
        }
    }

    return misplacedRows + misplacedColumns;
}

//Struct to create nodes loaded with the state of the puzzle (as a child of x), and all the
//  values of that node(f(n), g(n), h(n)). The parent node will be X, the node they were generated
//  from. The operator for node comparison is overloaded to allow the priority queue to operate as
//  a MIN-HEAP instead of the default MAX-HEAP, so the lowest f(n) will be at the top.
struct Node {
    vector<vector<int>> state;
    int totalCost;
    int costFromStart;
    int heuristicValue;
    Node* pNodePointer;

    Node(vector<vector<int>> currentState, int gn, int hn, Node* parent = nullptr) {
        state = currentState;
        costFromStart = gn;
        heuristicValue = hn;
        totalCost = gn + hn;
        pNodePointer = parent;
    }

    bool operator<(const Node& other) const {
        return totalCost > other.totalCost;
    }
};
```

```
//////////////////////////////////////HELPER FUNCTIONS/////////////////////////////////////////////////

//Recursive helper function to print the path of the the A star algorithm. It takes a node as a
//  parameter, then checks if the node is equal to null pointer (meaning it is the parent of the
//  root node) and returns. If that is not the case it calls printPath recursively, based on that
// nodes parent node. The state is printed using a range based loop on the vector for state.
void printPath(Node* goalReached) {
    if (goalReached == nullptr) {
        return;
    }
    printPath(goalReached->pNodePointer);

    for(const auto& row : goalReached->state) {
        for(int tile : row) {
            cout << tile << " ";
        }
        cout << endl;
    }
    cout << endl;
}

//Helper function to generate the children of a node that is being expanded by the aStarSearchFunction
//  Takes a node pointer, the goal, and the hueristic function being used by aStarSearch as params,
//  then finds the blank spot with a nested loop, takes not of that spot, then uses a range based
//  to add those possible moves, validate the possible moves list, and then add the new children to
//  a vector of Node pointers that is returned to the caller.
vector<Node*> generateChildren(Node* currentNode, vector<vector<int>> goal, function<int(vector<vector<int>>, vector<vector<int>>)> heuristicFunction) {
    vector<Node*> childrenOfX;
    vector<vector<int>> stateOfX = currentNode->state;
    int gnOfX = currentNode->costFromStart;

    //Loops to find the position of the blank spot
    int blankRow = -1;
    int blankCol = -1;
    for(int row = 0; row < 3; row++){
        for(int column = 0; column < 3; column++) {
            if(stateOfX[row][column] == 0) {
                blankRow = row;
                blankCol = column;
                break;
            }
        }
        if(blankRow != -1 && blankCol != -1) break;
    }

    //Creating a vector of pairs with the possible moves you could make in the puzzle
    vector<pair<int, int>> possibleMoves = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    for(auto move : possibleMoves) {
        int moveRow = blankRow + move.first;
        int moveCol = blankCol + move.second;

        //Validating the move (making sure it doesn't exceed the bounds of the puzzle)
        if (moveRow >= 0 && moveRow < 3 && moveCol >= 0 && moveCol < 3) {
            vector<vector<int>> moveState = stateOfX;
            int valuePlaceHolder = moveState[moveRow][moveCol];
            moveState[moveRow][moveCol] = 0;
            moveState[blankRow][blankCol] = valuePlaceHolder;

            //Creating the Child Node
            Node* childNode = new Node(moveState, (gnOfX + 1), heuristicFunction(moveState, goal), currentNode);

            //Adding Node to the Vector
            childrenOfX.push_back(childNode);

        }
    }

    return childrenOfX;
}
```

```
235  ////////////////////////////////////////A Star Search/////////////////////////////////////////////////
236
237  //Main function of the project. This function takes the intitial state, goal state, and a pointer to
238  //  a heuristic function as parameters. Creates a node based on the intial state and the heuristic
239  //  function then pushes that node to a priority queue from the queue library (modified by the Node
240  //  struct to be a min heap) and begins the loop.
241  //  The loop is a while based on if OPEN is empty, if not empty then it will begin to check the goal
242  //  state against the items in OPEN, expanding on those items if they are not the goal.
243  //  The generate children function is called if X (the current node being evaluated is not the goal)
244  //  and the children that are generated are evaluated to see if they are already in CLOSED by a
245  //  range based loop over the CLOSED vector. Children not in CLOSED are added to open, and those
246  //  in closed are deleted.
247  //  Once the goal state is reached, the metrics will be printed by calling printPath() on the current
248  //  node and prints the metrics: ET, NG, NE, D, and b*.
249
250  void aStarSearch (vector<vector<int>> initial, vector<vector<int>> goal, function<int(vector<vector<int>>, vector<vector<int>>)> heuristicFunction) {
251      //Start the clock for execution time
252      high_resolution_clock::time_point start = high_resolution_clock::now();
253
254      //Counting Variables (depth, nodes generated, nodes expanded, and effective branching factor)
255      int costFromStart = 0;
256      int nodesGenerated = 1;
257      int nodesExpanded = 0;
258      float effectiveBranchingFactor = 0;
259
260      //Create a priority queue that runs on min-heap rules (puts the lowest f(n) node on the left)
261      //Create a closed data structure (vector most likely for searching in the loop withh the find keyword)
262      priority_queue<Node*, vector<Node*>, greater<>> OPEN;
263      vector<Node*> CLOSED;
264
265
266      //Initialize the first node and add it to the priority queue
267      Node* firstNode = new Node(initial, costFromStart, heuristicFunction(goal, initial));
268      OPEN.push(firstNode);
269
270
271
272      //Search Loop (while the queue is not empty)
273      while (!OPEN.empty()) {
274
275          //Remove the leftmost state from the queue, call it x
276          Node* X = OPEN.top();
277          OPEN.pop();
278
279          //nodesExpanded is incrmented only when a node is popped
280          nodesExpanded++;
281
282          if (X->state == goal) {
283              printPath(X);
284              costFromStart = X->costFromStart;
285
286              //Execution time and effect branching factor calculations
287              high_resolution_clock::time_point done = high_resolution_clock::now();
288              duration<float> executionTime = done - start;
289
290              //Has to static cast on one of the variables to avoid integer division
291              if (costFromStart > 0) {
292                  effectiveBranchingFactor = static_cast<float>(nodesGenerated) / costFromStart;
293              }
294
```

```
290                 //Has to static cast on one of the variables to avoid integer division
291                 if (costFromStart > 0) {
292                     effectiveBranchingFactor = static_cast<float>(nodesGenerated) / costFromStart;
293                 }
294 
295                 //Printing ET, NG, NE, D, and b* in order
296                 cout << "Execution Time: " << executionTime.count()<< " seconds" << endl;
297                 cout << "Nodes Generated: " << nodesGenerated << endl;
298                 cout << "Nodes Expanded: " << nodesExpanded << endl;
299                 cout << "Tree Depth: " << costFromStart << endl;
300                 cout << "Effective Branching Factor: " << effectiveBranchingFactor << endl;
301 
302                 //Delete the X pointer
303                 delete X;
304                 return;
305 
306 
307             } else {
308                 //generate the children of x
309                 vector<Node*> childrenOfX = generateChildren(X, goal, heuristicFunction);
310 
311                 //Range based loop through nodes pointer in CLOSED and add children not on CLOSED to OPEN
312                 for(Node* child : childrenOfX) {
313                     bool inCLOSED = false;
314                     for(auto& closedNode : CLOSED) {
315                         if (child->state == closedNode->state) {
316                             inCLOSED = true;
317                             break;
318                         }
319                     }
320 
321                     if (!inCLOSED) {
322                         OPEN.push(child);
323                         nodesGenerated++;
324                     } else {
325                         //Delete the child node after it has been processed
326                         delete child;
327                     }
328 
329                 }
330                 //Put x on closed
331                 CLOSED.push_back(X);
332             }
333         }
334     }
335 
336     int main(int argc, char const *argv[])
337     {
338         //Initializes the starting state (change to match the starting state that you want to try)
339         vector<vector<int>> startingState = initialState1;
340         //Initializes the heuristic function (change to match the starting state that you want to try)
341         function<int(vector<vector<int>>, vector<vector<int>>)> heuristicFunction = h1;
342 
343         aStarSearch(startingState, goalState, heuristicFunction);
344 
345     }
```

Source Code Run:

InitialState1:

H1:

```
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: 0.00224765 seconds
Nodes Generated: 93
Nodes Expanded: 54
Tree Depth: 6
Effective Branching Factor: 15.5
Program ended with exit code: 0
```

H2:

```
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: 0.00240479 seconds
Nodes Generated: 93
Nodes Expanded: 54
Tree Depth: 6
Effective Branching Factor: 15.5
Program ended with exit code: 0
```

```
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: 0.00235585 seconds
Nodes Generated: 93
Nodes Expanded: 54
Tree Depth: 6
Effective Branching Factor: 15.5
Program ended with exit code: 0
```

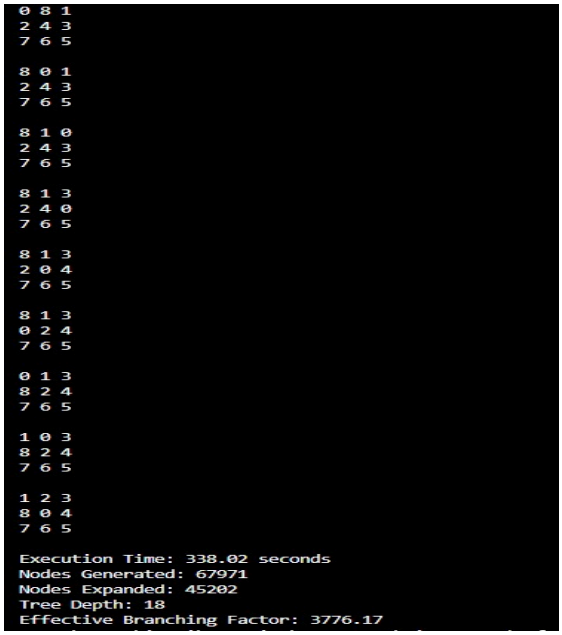
jonathanHeuristic:

```
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: 0.00227285 seconds
Nodes Generated: 93
Nodes Expanded: 54
Tree Depth: 6
Effective Branching Factor: 15.5
Program ended with exit code: 0
```

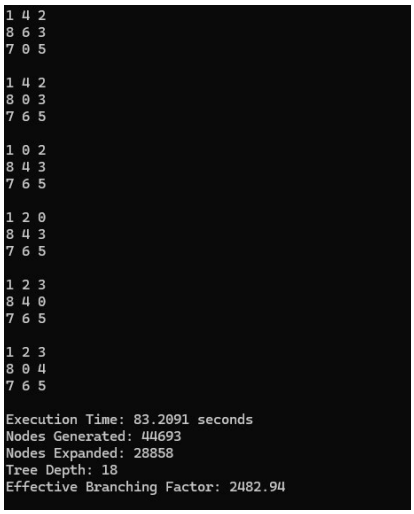
robertHeuristic:

InitialState2:

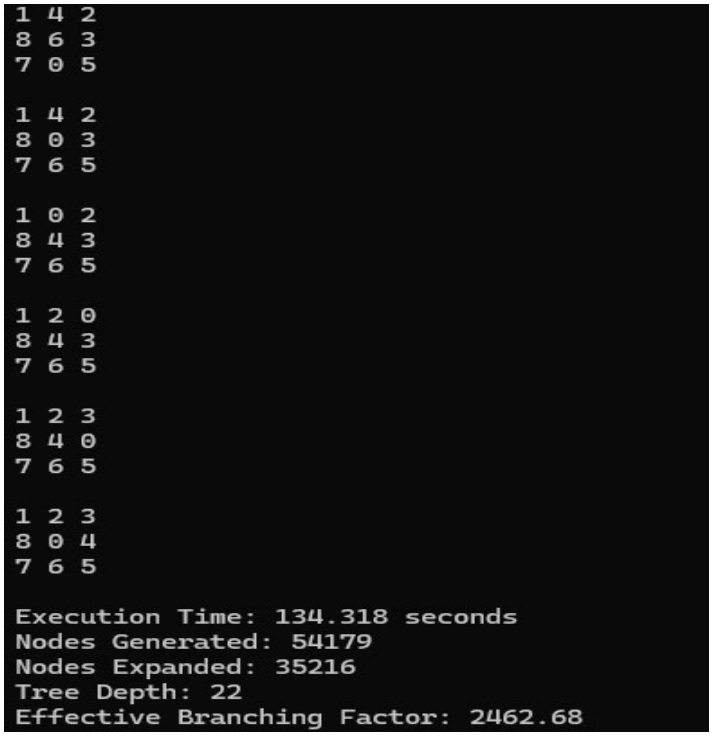
H1:



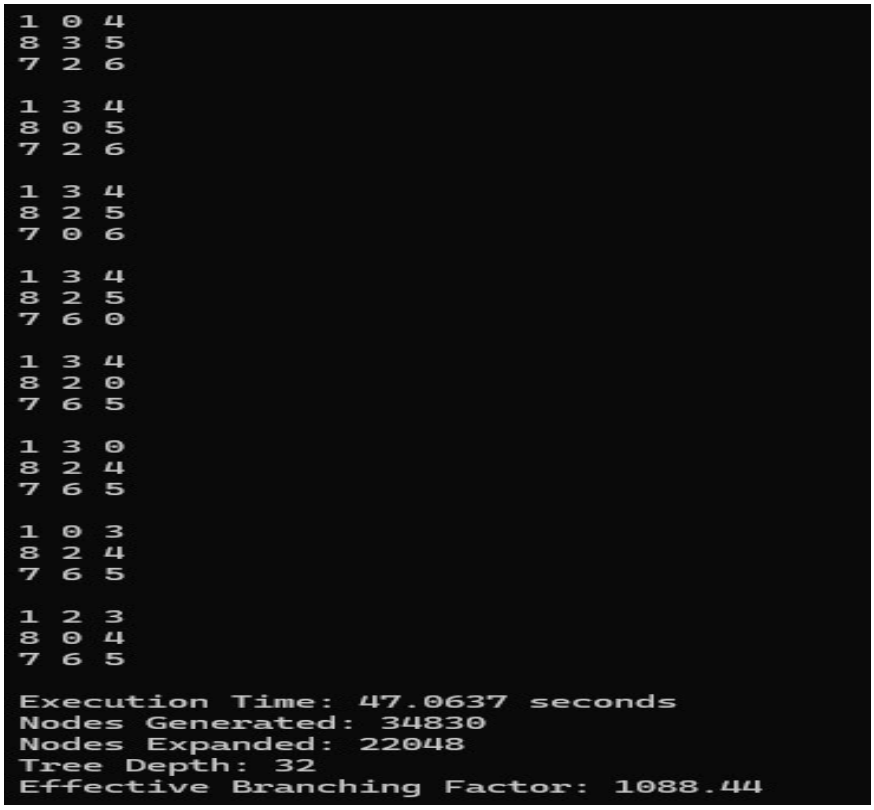
H2:



JonathanHeuristic:



RobertHueristic:



Tabulation of Results:

Initial State #1:

Heuristic Function	ET	NG	NE	D	b*
H1	.00706 seconds	86	48	6	14.333
H2	.01012 seconds	151	88	6	25.166
jonathanHeuristic	.00488 seconds	32	20	6	5.333
robertHeuristic	.02945 seconds	696	417	6	116

Initial State #2:

Heuristic Function	ET	NG	NE	D	b*
H1	338.02 seconds	67971	45202	18	3776.17
H2	83.2091 seconds	44693	28858	18	2482.94

jonathanHeuristic	134.318 seconds	54179	35216	22	2462.68
robertHeuristic	47.063 seconds	34830	22048	32	1088.44

Analysis of Results:

H1:

- Execution Time (ET): Fastest among all heuristic functions for this state, showing simplicity in calculations.
- Nodes Generated (NG) and Nodes Expanded (NE): Moderate values, suggesting that H1's low computation cost trades off with less precise guidance, leading to expanded nodes.
- Tree Depth (D): 6, consistent with other heuristics, as the actual path length is independent of the heuristic.
- *Branching Factor (b):** 14.333, relatively high due to less accurate cost estimation.

Observation: H1 is efficient but sacrifices precision, making it suitable for simpler puzzles.

H2:

- Execution Time (ET): Slightly slower than H1 due to more complex calculations.
- NG and NE: Higher than H1, indicating H2's better guidance reduces unnecessary exploration.
- *Branching Factor (b):** Higher than H1, showing that H2 focuses the search more effectively.

Observation: H2 balances computational cost and accuracy, performing well for this puzzle size.

Jonathan’s Heuristic:

- ET: Fastest overall, indicating minimal computation complexity.

- NG and NE: Lowest values among all heuristics, suggesting a strong ability to focus the search effectively.
- *Branching Factor (b):** Lowest, indicating efficient pruning of the search space.

Observation: Jonathan's heuristic is highly efficient, providing a good trade-off between simplicity and guidance for smaller problems.

Robert's Heuristic:

- ET: Slowest due to its two-step evaluation (row and column mismatches).
- NG and NE: Highest among all heuristics, showing that despite extensive exploration, the heuristic does not prune effectively.
- *Branching Factor (b):** Extremely high, reflecting inefficiency in guiding the search.

Observation: Robert's heuristic is the least optimal and has high computational cost.

Initial State #2

- H1 (Misplaced Tile Heuristic):
- ET: Slowest due to inefficient pruning in this more complex scenario.
- NG and NE: Highest, indicating a significant amount of unnecessary exploration.
- *Branching Factor (b):** Extremely high, showing poor efficiency for complex puzzles.
- Observation: H1's simplicity becomes a limitation in larger search spaces, leading to excessive computation.
- H2 (Manhattan Distance):
- ET: Faster than H1, highlighting its ability to reduce unnecessary node exploration.
- NG and NE: Lower than H1, confirming its better performance in complex puzzles.
- *Branching Factor (b):** Significantly lower than H1, indicating more focused exploration.
- Observation: H2 scales better than H1, demonstrating a solid balance of efficiency and accuracy.
- Jonathan's Heuristic:
- ET: Moderate, indicating its suitability for complex puzzles but not the fastest.
- NG and NE: Comparable to H2 but slightly less efficient.
- *Branching Factor (b):** Similar to H2, showing decent pruning capability.
- Observation: Jonathan's heuristic remains effective but doesn't scale as efficiently as H2 for more complex puzzles.

Robert’s Heuristic:

- ET: Fastest among all heuristics, likely due to higher selectivity in its node expansion.
- NG and NE: Lowest, confirming its ability to guide the search efficiently in larger puzzles.
- *Branching Factor (b):** Lowest, showing strong pruning and focus.

Observation: Robert’s heuristic, despite its inefficiency in simpler puzzles, excels in guiding the search for complex problems.

Conclusion:

This project demonstrated the importance of high order Heuristic functions. While some functions like H1 are optimal for less complex problems, the project highlighted that as complexity changes, so do our heuristic needs. Advance heuristics like h2 and Roberts proved inefficient and excessive in the first initial state, but as complexity rose in initial state 2, the two more complex functions began to outperform. Jonathan's heuristic stood as a good example of a well rounded function, holding its own against both complexities without losing too much efficiency. Overall this project taught us the importance of algorithm optimization based on our domain. This project provided practical experience in implementing and analyzing heuristic-driven algorithms, reinforcing their critical role in artificial intelligence and search optimization.

Member Contributions:

While we worked together on most of the algorithm, Robert (I), contributed the helper functions and RobertsHueristic. Jonathan contributed the rest of the source code.