CS-531
PA 2: Puzzle

Tommy Hollenberg
Matthew Phillips
Michael Slater

February 6, 2019

Abstract

This document describes two types of search algorithms, IDA* and RBFS. The search algorithms explore the space of possible solutions to a 4x4 puzzle using three different heuristics. The performance of the search algorithms and the heuristics are judged upon the nodes expanded, number of steps to reach the goal, and total computational time.

Contents

1 Introduction

In this experiment, a 4x4 puzzle board containing the numbers 1 through 15 and a blank space is solved. The blank space may be swapped with any adjacent number on the board. The goal board state has the numbers ordered from least to greatest starting in the top left corner and continuing left to right for each row. The goal state also has the blank state in the bottom right corner of the board.

For the search algorithms, each board state represents a node. As the algorithm explores the space, new nodes are created as the board state changes. The nodes are evaluated using a heuristic which generates an estimated cost to the solution from the current node.

2 Memory-bounded Heuristic Search Algorithms

2.1 Iterative Deepening A*

Iterative Deepening A* (IDA*) is similar to iterative deepening depth-limited search, but an f-limit is used to limit the search instead of the depth. This algorithm explores all possible paths until their f-values exceed the current f-value. Then the algorithm begins again at the start node with a new f-limit which was taken from the previous iteration. The new f-limit is the smallest f-value larger than the old f-limit from the paths explored in the previous iteration.

2.2 Recursive Best First

Recursive best-first search (RBFS) is to similar recursive depth-first search, however RBFS chooses the successor state with the best f-value and remembers the best alternative f-value path. If the current path's f-value exceeds the best alternative f-value, the algorithm unwinds and continues down the alternative path. As the algorithm unwinds, the best f-value from a node's successors is backed-up in node. This allows the algorithm to remember the cost of the node's path and if the node is worth exploring later. The f-value is created by adding the heuristic cost of the current state to number of nodes in the path, also known as the g-value. Our algorithm is modified slightly from the standard RBFS because it checks if successor state is in the current path. This prevents expansion of duplicate nodes thus saving computational time.

3 Heuristics

3.1 Inversion Distance

We can flatten the board into a 1-dimensional array by concatenating the rows (ignoring the empty square). In the goal state, the tiles will be in ascending

order in the resulting array. Therefore, the number of inversions in this array gives a measure of how far the puzzle is from being solved.

Each horizontal move in the puzzle has no impact on the ordering of this horizontally flattened ordering. Vertical moves, on the other hand, "jumps" the moved tile 3 positions left (if moved up) or right (if moved down) in the flattened ordering. This changes the number of inversions by ± 3 (if all or none of the jumped over tiles formed inversion pairs with the jumping tile) or ± 1 (if two formed inversion pairs with the jumping tile and one didn't, or vice versa). Therefore, we can lower bound the number of vertical moves to the goal state by $\lceil n/3 \rceil$, where n is the number of inversions in the horizontally flattened ordering.

We can similarly lower bound the number of horizontal moves to the goal state by vertically flattening the board by concatenating the columns. The goal state is not in ascending order, so we need to map the tiles to their position in the goal state ordering before computing the number of inversions. Opposite from the horizontal ordering, vertical moves do not affect this vertically flattened ordering and horizontal moves change the number of inversions by ± 1 or ± 3. Thus, the number of horizontal moves to the goal state be lower bounded by $\lceil n/3 \rceil$, where n is the number of inversions in the vertically flattened ordering.

Since these two measures are orthogonal (vertical moves do not impact the estimate of remaining horizontal moves, and horizontal moves do not impact the estimate of remaining vertical moves), these two lower bound estimates can be added together to form an admissible heuristic.

3.2 Manhattan Distance

The Manhattan Distance, L1 distance, is the sum of the number of rows and number of columns that each number is away from its location in the goal state.

$\sum_{v \neq 0} (|r_v - r_{v,goal}| + |c_v - c_{v,goal}|)$

This heuristic is the solution size of the relaxed problem where any tile can be moved one square in any direction, regardless of the positions of other tiles. As the solution to a relaxed version of the problem, it is guaranteed to be admissible.

3.3 Manhattan Distance with Linear Conflict Correction

4 Experimental Setup

Each experimental setup had three parameters: scramble size, heuristic used, and algorithm selected. The scramble size determined the number of random moves the board state experienced before the search algorithm attempted to solve the puzzle. The different values used for scramble size were: 10, 20, 30, 40, and 50. Overall, we ran the simulation approximately 300,000 times, with 150,000 simulations per algorithm and each setup 10,000 times.

5 Results

An overall summary of results can be seen in 1. There are obvious increases in computation time and node expansion as the problem complexity (the number of scrambles) increases. The performance of the heuristics and algorithms will be explored in greater detail below.

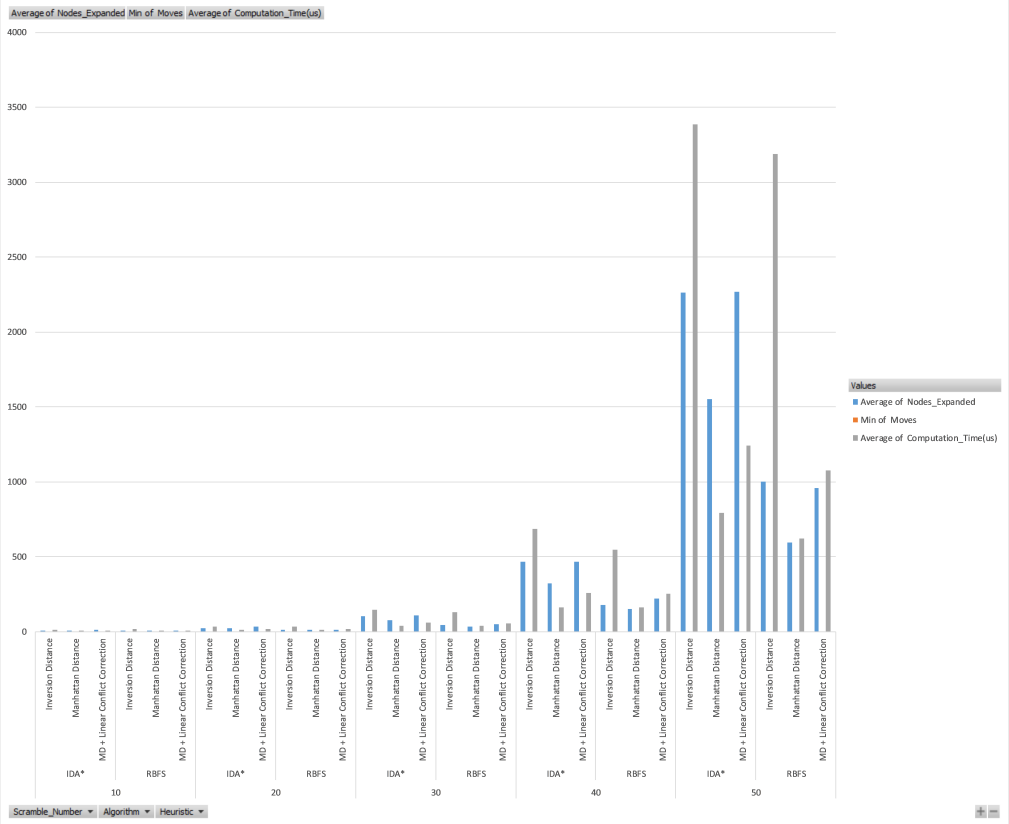

Figure 1: Summary of Performance Results

5.1 Heuristic Performance

Examining the performance of the puzzle solver with our three different heuristics, we find distinct differences between the heuristics. In 2, we can see, on average, that the Inversion Distance heuristic uses the most CPU time and expands the most nodes. The Linear Conflict Correction variant of Manhattan Distance is more CPU efficient than the Inversion Distance, but expands about same number of nodes. Somewhat surprisingly, the base Manhattan Distance heuristic with its reliance on the relaxed problem, performs the best of all our heuristics.

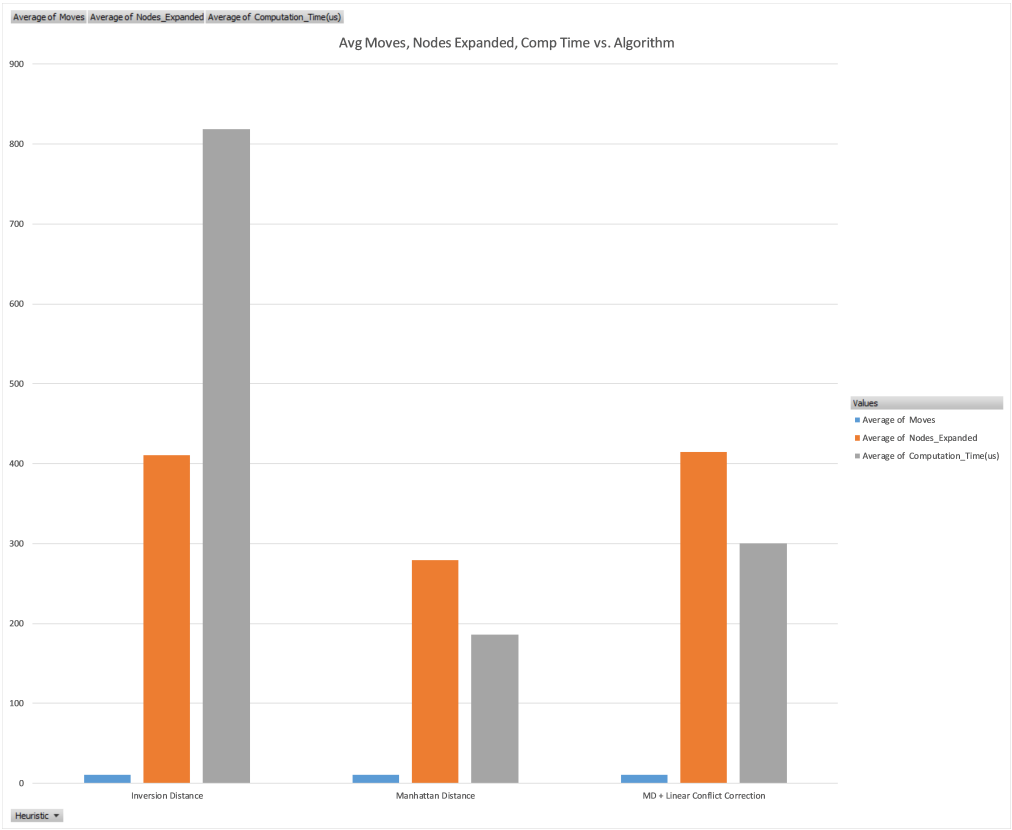

Figure 2: Average Performance of Puzzle Solver vs. Heuristic Choice

To further examine the heuristic performance, we can look at the total sum of the performance metrics as well as the average. In figure 3 we see that the Manhattan Distance is clearly the most efficient heuristic, for both CPU time and node expansion.

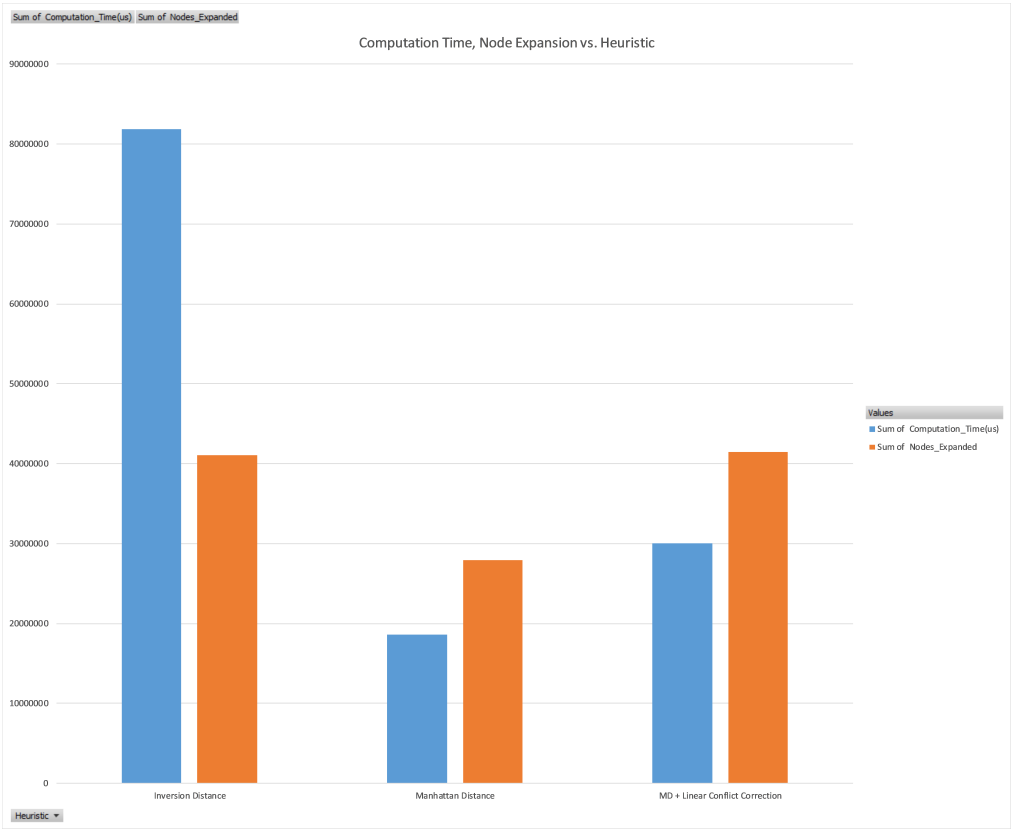

Figure 3: Sum of Performance Metrics of Puzzle Solver vs. Heuristic Choice

5.2 Optimality versus Computational Cost

If we consider a solution to a smaller scramble size problem as a non-optimal solution to a large scramble size problem, then it is clear that a non-optimal solution does not require as many nodes to be expanded, nor does it use as much CPU time as an optimal solution. If we examine the results in Figure X, we can see that as the scramble size increases, the computational costs also increase quite rapidly. Hence, for some problems where a fully optimal solution is not within our cost budget of memory, CPU time, or both, we may wish to sacrifice some acceptable level of optimality in order to stay within our budget.

In Figure 4 we can see the largest two scramble sizes require more than linear increases in computation time to solve.

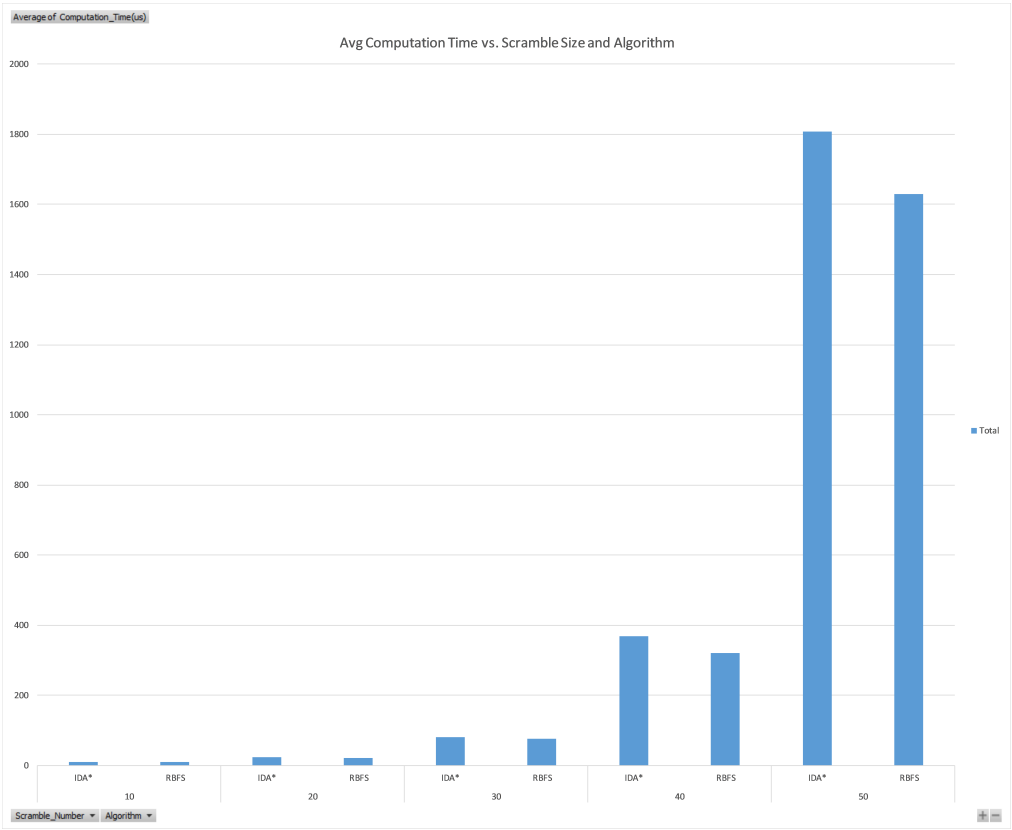

Figure 4: Average Computational Cost vs. Scramble Size

Likewise, in Figure 5 we also see the largest two scramble sizes require more than linear increases in node expansion to solve.

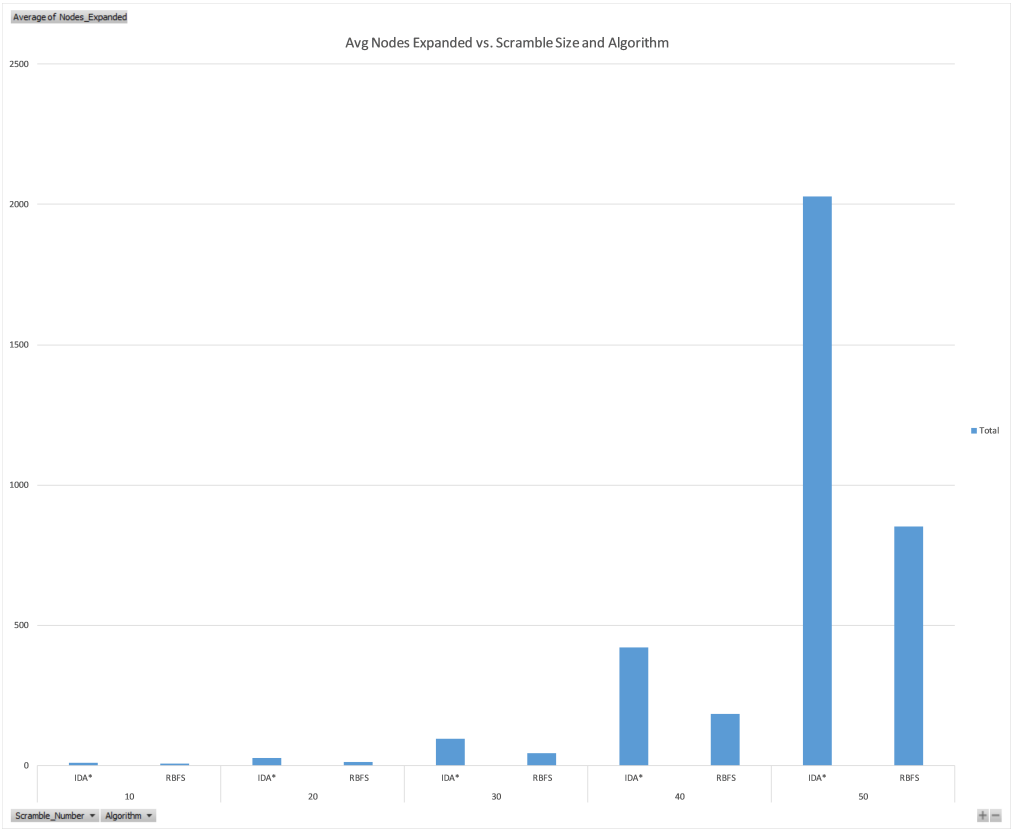

Figure 5: Average Computational Cost vs. Scramble Size

5.3 Heuristic Cost

To examine the impact of heuristic choice on the total problem-solving time, we computed the average computation time required per expanded node. Using this computed metric, we can see that the unit cost of the inversion distance heuristic is over twice that of either the Manhattan distance metric or the Manhattan distance metric with linear conflict correction. Hence, we can conclude that for more complex problems that require many nodes to be expanded, the heuristic code will be a significant component of the overall computation time. In figure 6, we can see the obvious differences between the heuristics, as previously described.

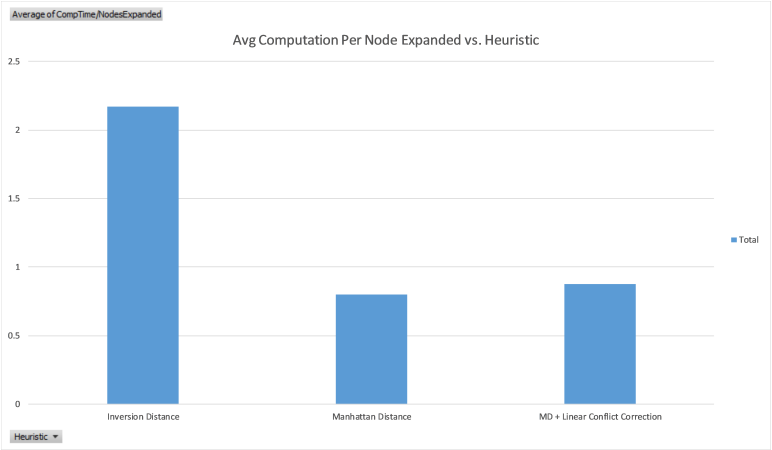

Figure 6: Average Computational Time Per Node Expansion vs. Heuristic Choice

5.4 Heuristic Selection and Characterization

For any given problem domain, there will often be a range of admissible heuristics. Ideally, we would like to be able to construct a heuristic, $h(s)$, such that $h(s)$ is within some small ϵ of $h^*(s)$. For the 15-Puzzle problem, we know that there are heuristics that may be closer to $h^*(s)$ than the commonly used heuristic, Manhattan Distance. Two similar heuristics were considered for this work, both of which examine the number of out-of-order pairs of tiles in the scrambled puzzle. These heuristics reflect the fact that the Manhattan Distance operates on a relaxed problem and that by considering how the problem is relaxed, we can choose heuristics that directly address the relaxation.

5.4.1 Linear Conflict Correction

In the domain of 15-Puzzle, if we have two tiles in the same row, both of which are are supposed to end up in this row of the solved puzzle, but which are currently in the wrong order, we know that the Manhattan distance is an underestimate of the true cost, in that we need to account for at least two extra moves to position the tiles past each other. Hence, to compute the total linear conflict correction, we need to look at every line of the puzzle, and search for out-of-order tile pairs that are in the correct row of the solved puzzle. As we find such out-of-order tile pairs, we will increment a count that corresponds to the extra moves required to reposition the tile pairs.

5.4.2 Inversion distance

Thinking further about out-of-order tile pairs, another admissible heuristic is inversion distance. There are two components to inversion distance. For hori-

zontal inversion distance, we concatenating the rows end-to-end so we are effectively treating the $4x4$ puzzle grid as a single row. In this single row, we then count the number of tile pairs that are out of order. These are horizontal inversions. In counting inversions, we do not count any tile pairs that include the empty square. Horizontal moves cannot affect horizontal inversions. A vertical move in the square puzzle corresponds to one tile jumping over three other tiles in the single-row format puzzle. This changes the number of inversions by ± 1 or ± 3. Therefore a lower bound for the number of vertical moves in a solution is given by $\lceil n/3 \rceil$, where n is the number of inversions in our long row. The number of horizontal moves needed can be computed by considering the vertical inversions. The total inversion distance is the sum of these two inversion counts, both of which are lower bounds.

5.4.3 Algorithm versus Node Expansion and Computation Time

We found from running 10,000 trials for each of the scramble sizes, heuristics, and algorithms, we found that the IDA* algorithm requires about 10-15% more computation time (the included data reflects a 11.3% increase). Of course, IDA* ends up expanding many more nodes than RBFS due to its iterative nature which ends up repeating work that was done in previous iterations. Figure 7 shows these differences between the algorithms.

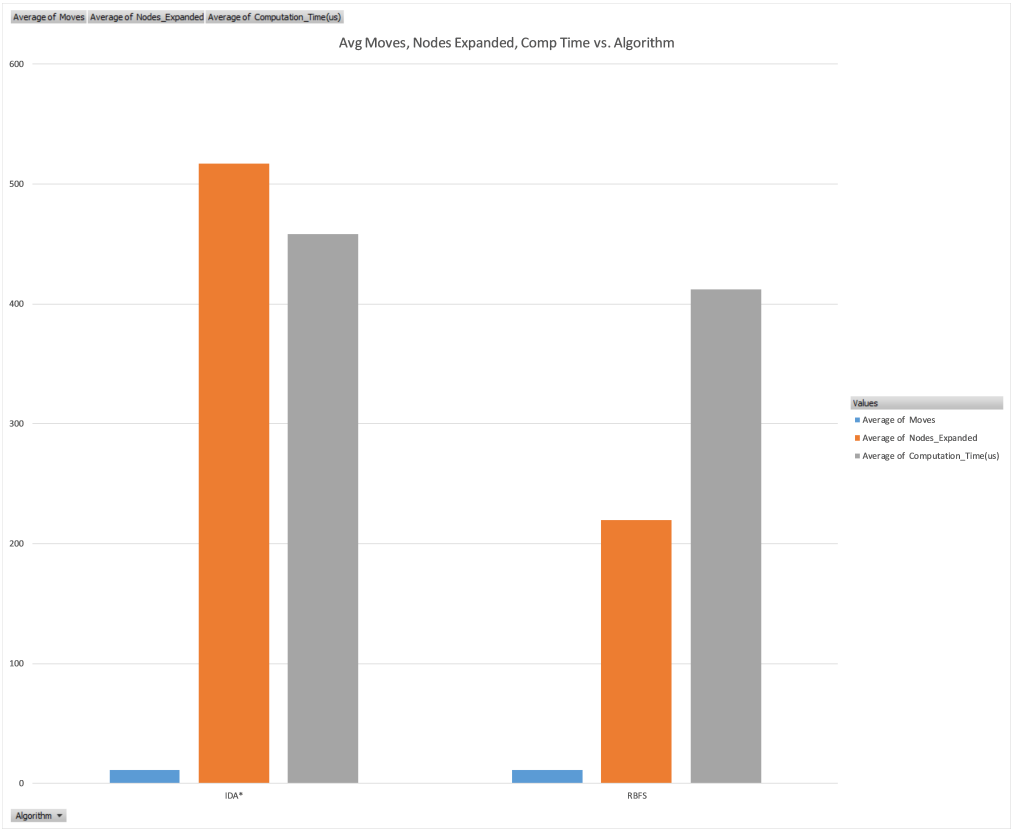

Figure 7: Computation Time and Node Expansion vs. Algorithm Choice

6 Discussion

The experimental results provided us with several key findings. First, we found that there were marked differences in the performance of the three heuristics. Surprisingly, the simplest heuristic that we might otherwise think is farthest from $h^*(s)$ performed the best in terms of computational cost and node expansion. The second key finding was that the problem difficulty did not scale linearly with the increase in the scramble size. The puzzle solver took much longer for the two more complicated initial puzzle states (Scramble-40 and Scramble-50). From this finding we might infer that for problems that do not need a fully optimal solution, we would be able to trade computation time and/or memory usage for solution optimality. In terms of the computational cost of the heuristic themselves, our third finding is that only one heuristic (the Inversion Distance) added significantly to the overall computational cost of solving the puzzles. In later analysis, we determined that more efficient code and/or data structures would most likely have reduced this computational cost. Finally, we found that

RBFS, not having to iterate through increasing depths, was generally more efficient than IDA*.

While not reflected directly in the various figures, certain initial board states needed an enormous amount of node expansion to find a solution. These difficult configurations point toward future improvements in algorithm and/or heuristic choice. Other potential future efforts include testing on larger versions of the puzzle, low level code and data structure optimizations, composite or ensemble type heuristics, and finer resolution code timing to help us identify slow portions of the algorithms and/or heuristics.

7 Code

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <string>
#include <cmath>
#include <random>
#include <chrono>
#include <utility>
#include <unordered_set>
#include <climits>
#include "MurmurHash3.h"

using namespace std;


/*************************************
 * Board State class
 *
 * Be wary, action functions do not check
 * whether the action is applicable
 * to the board state.
 *************************************/
class Board {
public:
    static const uint32_t HASHSEED = 0x9747b28c;
    static const int ROWS = 4;
    static const int COLS = 4;
    int board[ROWS][COLS];
    int i_cord;
    int j_cord;
    int F;

    enum Action { UP, DOWN, LEFT, RIGHT };
```

```
Board() : i_cord(3), j_cord(3) {
    for (int i = 0; i < 15; ++i)
        board[i / 4][i % 4] = i + 1;
    board[3][3] = 0;
}

Board(const Board &obj)
{
    for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 4; ++j)
            board[i][j] = obj.board[i][j];
    i_cord = obj.i_cord;
    j_cord = obj.j_cord;
    F = obj.F;
}


Board(Board &b, Action a) : i_cord(b.i_cord), j_cord(b.j_cord) {
    for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 4; ++j)
            board[i][j] = b.board[i][j];

    switch (a) {
    case UP:   up();   break;
    case DOWN: down(); break;
    case LEFT: left(); break;
    case RIGHT: right(); break;
    }
}

bool operator==(const Board &b) const {
    for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 4; ++j)
            if (board[i][j] != b.board[i][j])
                return false;
    return true;
}

Board& up() {
    swap(board[i_cord][j_cord], board[i_cord - 1][j_cord]);
    --i_cord;
    return *this;
}

Board& down() {
    swap(board[i_cord][j_cord], board[i_cord + 1][j_cord]);
    ++i_cord;
    return *this;
}
```

```
    Board& left() {
        swap(board[i_cord][j_cord], board[i_cord][j_cord - 1]);
        --j_cord;
        return *this;
    }

    Board& right() {
        swap(board[i_cord][j_cord], board[i_cord][j_cord + 1]);
        ++j_cord;
        return *this;
    }

    void print() {
        for (int i = 0; i < 4; ++i) {
            for (int j = 0; j < 4; ++j) {
                int v = board[i][j];
                if (v < 10) cout << ' ';
                cout << ' ' << v;
            }
            cout << endl;
        }
    }
};

// Implement std::hash<Board> so we can use std::unordered_set<Board>
namespace std
{
    template<>
    struct hash<Board> {
        size_t operator()(const Board &b) const {
            uint32_t hash;
            MurmurHash3_x86_32(&b.board, sizeof(b.board),
                Board::HASHSEED, &hash);
            return hash;
        }
    };
}


/*****************************
 * Abstract Heuristic class
 *****************************/
class Heuristic {
public:
    virtual int operator()(Board &b) = 0;
    virtual string get_name() = 0;
};
```

```
/*****************************
 * Heuristic Implementations
 *****************************/
class ManhattanDistance : public Heuristic {
public:
    virtual int operator()(Board &b) {
        int MD = 0;
        for (int i = 0; i < 16; ++i) {
            int x = i / 4;
            int y = i % 4;
            int v = b.board[x][y];
            if (v > 0) {
                int x_dest = (v - 1) / 4;
                int y_dest = (v - 1) % 4;
                MD += abs(x - x_dest) + abs(y - y_dest);
            }
        }
        return MD;
    }

    virtual string get_name() {
        return "Manhattan Distance";
    }
};

//
// Linear conflict correction:
// Look at every line of the puzzle. If you find two tiles there which
    are supposed to end up in this line,
// but which are currently in the wrong order, then you know that the
    Manhattan distance is too optimistic
// and you actually need at least 2 more moves to get the two tiles past
    each other. One can prove that the
// heuristic function remains admissible (in fact monotone) even if you
    add 2 for every pair with this problem
// in any row. The same applies to every pair with the analogous problem
    in any column.
//
class LinearConflictMD : public ManhattanDistance {
    Board solved = Board();

    inline bool isValidForRow(int row, int x)
    {
        return (x >= solved.board[row][0] && x <=
            solved.board[row][Board::COLS - 1]);
    }

    int getRowCount(Board &b)
    {
        int count = 0;
```

```
        for (int row = 0; row < Board::ROWS; row++)
        {
            for (int column = 0; column < Board::COLS - 2; column++)
            {
                int left = b.board[row][column];
                int right = b.board[row][column + 1];
                int correct_right = solved.board[row][column + 1];
                if (left == solved.board[row][column] &&
                    isValidForRow(row, right))
                {
                    if (right != correct_right)
                        count++;
                }
            }
        }
        return count;
    }

public:
    virtual int operator()(Board &b) {
        return ManhattanDistance::operator()(b) + getRowCount(b) * 2;
    }

    virtual string get_name() {
        return "MD + Linear Conflict Correction";
    }
};


class InversionDistance : public Heuristic {
public:
    virtual int operator()(Board &b) {
        int h_inv = 0;
        int v_inv = 0;
        for (int i = 0; i < 16; ++i) {
            for (int j = i + 1; j < 16; ++j) {
                int x = b.board[i / 4][i % 4];
                int y = b.board[j / 4][j % 4];
                if (x * y > 0) {  // ignore inversions with empty square
                    if (x > y) ++h_inv;

                    // map to column major ordering
                    int vi = 4 * (i % 4) + i / 4;
                    int vj = 4 * (j % 4) + j / 4;
                    int vx = 4 * ((x - 1) % 4) + (x - 1) / 4;
                    int vy = 4 * ((y - 1) % 4) + (y - 1) / 4;
                    if ((vx > vy) != (vi > vj)) ++v_inv;
                }
            }
        }
```

```
        return ceil(h_inv / 3.0) + ceil(v_inv / 3.0);
    }

    virtual string get_name() {
        return "Inversion Distance";
    }
};


/***************************************
 * Problem class
 *
 * Contains the heuristic, successor, and
 * goal_test functions. Also contains a
 * scramble function, that generates
 * solvable starting states, and print
 * function that prints a sequence of board
 * states.
 ***************************************/
class Problem {
    mt19937 randgen;
public:
    Heuristic &h;

    Problem(Heuristic &h) : h(h),
        randgen(mt19937(chrono::system_clock::now().time_since_epoch().count()))
        {}

    vector<Board> successors(Board &b) {
        vector<Board> succ;
        if (b.i_cord > 0) succ.emplace_back(b, Board::UP);
        if (b.i_cord < 3) succ.emplace_back(b, Board::DOWN);
        if (b.j_cord > 0) succ.emplace_back(b, Board::LEFT);
        if (b.j_cord < 3) succ.emplace_back(b, Board::RIGHT);
        return succ;
    }

    bool goal_test(Board &b) {
        return h(b) == 0;
    }

    Board scramble(int m) {
        Board b;
        for (int i = 0; i < m; ++i) {
            vector<Board> succ = successors(b);
            int r = randgen() % succ.size();
            b = succ[r];
        }
        return b;
    }
```

```
    void print(vector<Board> &path) {
        for (Board &b : path) {
            b.print();
            cout << endl;
        }
    }
};

// Debugging helper function
void pause(Board &b, Problem &p, int f) {
    b.print();
    cout << "h: " << p.h(b) << ", f: " << f << endl << "paused..." <<
        endl << endl;
    string s;
    getline(cin, s);
}

/*************************
 * IDA* Search Algorithm *
 *************************/
int DL_A_star(vector<Board> &path, unordered_set<Board> &pathSet,
    Problem &p, int g, int f_limit, int &nodes_expanded) {
    Board &b = path.back();
    int f = g + p.h(b);
    ++nodes_expanded;
    //pause(b, p, f);
    if (f > f_limit) return f;
    if (p.goal_test(b)) return f;
    int f_min = INT_MAX;
    for (Board &s : p.successors(b)) {
        if (pathSet.find(s) == pathSet.end()) {
            path.push_back(s);
            pathSet.insert(s);
            f = DL_A_star(path, pathSet, p, g + 1, f_limit,
                nodes_expanded);
            if (f <= f_limit) return f; // if goal is found, return length
            if (f < f_min) f_min = f; // if smallest over limit, update
                f_min
            path.pop_back();
            pathSet.erase(s);
        }
    }
    return f_min;                   // return smallest over limit
}

vector<Board> ID_A_star(Board &start, Problem &p, int &nodes_expanded) {
    int f_limit = p.h(start);
    vector<Board> path{ start };
    unordered_set<Board> pathSet{ start };
```

```
    while (1) {
        int f_min = DL_A_star(path, pathSet, p, 0, f_limit,
            nodes_expanded);
        if (f_min <= f_limit) return path;          // if goal is found,
            return path
        if (f_min == INT_MAX) return vector<Board>(); // if failure,
            return empty path
        f_limit = f_min;
    }
}

/**********************
 *   RBFS Algorithm *     // with path checking
 **********************/
bool ascF(Board &b1, Board &b2) { return b1.F < b2.F; }

int RBFS(vector<Board> &path, unordered_set<Board> &pathSet, Problem &p,
     int g, int f_limit, int &nodes_expanded) {
    Board &b = path.back();
    ++nodes_expanded;
    if (p.goal_test(b)) return b.F;
    vector<Board> successors;
    for (Board &s : p.successors(b)) {
        if (pathSet.find(s) == pathSet.end()) { // only consider states
            not already visited on current path
            s.F = max(g + p.h(s), b.F);      // (and their siblings,
                which are stored in memory as well)
            successors.push_back(s);
        }
    }
    if (successors.empty()) return INT_MAX;
    if (successors.size() == 1) { // If there is only one successor,
        successors.emplace_back();   // add a dummy element so we can use
            same logic in loop,
        successors.back().F = INT_MAX; // but make sure it never gets
            expanded.
    }
    while (1) {
        sort(successors.begin(), successors.end(), ascF);
        Board &best = successors[0];
        if (best.F > f_limit) return best.F;
        int new_f_limit = min(f_limit, successors[1].F);
        path.push_back(best);
        pathSet.insert(best);
        best.F = RBFS(path, pathSet, p, g + 1, new_f_limit,
            nodes_expanded);
        if (best.F <= new_f_limit) return best.F;
        path.pop_back();
        pathSet.erase(best);
    }
```

```
}

vector<Board> RecursiveBestFirst(Board &start, Problem &p, int
    &nodes_expanded) {
    start.F = p.h(start);
    vector<Board> path{ start };
    unordered_set<Board> pathSet{ start };
    RBFS(path, pathSet, p, 1, INT_MAX, nodes_expanded);
    return path;
}


void csv_write_headers(std::ofstream& f) {
    f << "Board_ID, Scramble_Number, Algorithm, Heuristic, Moves,
        Nodes_Expanded, Computation_Time(us)" << endl;
}

void csv_write_row(std::ofstream& f, int board_id, int scramble_num,
    string algo, string heuristic, size_t moves, int nodes_exp, long
    microseconds) {
    long micros = std::max(1L, microseconds);
    f << board_id << "," << scramble_num << "," << algo << "," <<
        heuristic << "," << moves << "," << nodes_exp << "," << micros
        << endl;
}

int main()
{
    const int TOTAL_TRIALS = 10000;
    LinearConflictMD lc;
    ManhattanDistance md;
    InversionDistance id;
    vector<Heuristic*> heuristics = { &md, &lc, &id };

    std::ofstream csv_file;
    string filename = "pa2-" + std::to_string(TOTAL_TRIALS) + ".csv";
    csv_file.open(filename);
    csv_write_headers(csv_file);
    int b_id = 0;

    for (int scramble_size = 10; scramble_size <= 50; scramble_size +=
        10) {
        cout << "Scramble size: " << scramble_size << " trial: ";
        for (int num_trials = 0; num_trials < TOTAL_TRIALS;
            num_trials++) {
            if (num_trials % (TOTAL_TRIALS / 10) == 0)
                cout << num_trials << " ";
            for (Heuristic* heuristic : heuristics) {
                Problem p_rbfs(*heuristic);
                Board start_rbfs = p_rbfs.scramble(scramble_size);
```

```
            ++b_id;

            // RBFS Algorithm
            int nodes_expanded = 0;
            chrono::time_point<chrono::high_resolution_clock> t0, t1;
            t0 = chrono::high_resolution_clock::now();
            vector<Board> solution_RBFS =
                RecursiveBestFirst(start_rbfs, p_rbfs,
                nodes_expanded);
            t1 = chrono::high_resolution_clock::now();
            chrono::microseconds duration =
                chrono::duration_cast<chrono::microseconds>(t1 - t0);
            //p.print(solution);
            csv_write_row(csv_file, b_id, scramble_size, "RBFS",
                heuristic->get_name(), solution_RBFS.size() - 1,
                nodes_expanded, duration.count());
        }
    }
    cout << endl;
}

for (int scramble_size = 10; scramble_size <= 50; scramble_size +=
     10) {
    cout << "Scramble size: " << scramble_size << " trial: ";
    for (int num_trials = 0; num_trials < TOTAL_TRIALS;
         num_trials++) {
        if (num_trials % (TOTAL_TRIALS / 10) == 0)
            cout << num_trials << " ";
        for (Heuristic* heuristic : heuristics) {
            Problem p_ida(*heuristic);
            Board start_ida = p_ida.scramble(scramble_size);
            ++b_id;

            // IDA* Algorithm
            int nodes_expanded = 0;
            chrono::time_point<chrono::high_resolution_clock> t0, t1;
            t0 = chrono::high_resolution_clock::now();
            vector<Board> solution_A_star = ID_A_star(start_ida,
                p_ida, nodes_expanded);
            t1 = chrono::high_resolution_clock::now();
            chrono::microseconds duration =
                chrono::duration_cast<chrono::microseconds>(t1 - t0);
            //p.print(solution);
            csv_write_row(csv_file, b_id, scramble_size, "IDA*",
                heuristic->get_name(), solution_A_star.size() - 1,
                nodes_expanded, duration.count());
        }
    }
    cout << endl;
}
```

```
    csv_file.close();
}
```