15 Puzzle

Wadood Alam, Matthew Pacey, Joe Nguyen

April 27, 2022

Abstract

In this paper we explore the comparison of two types of informed searches, A* and Recursive Best-First Search (RBFS). We implement these search algorithms when solving a 4x4 15 puzzle game. We use two different admissible heuristic functions: 1)Ccity Block Distance and 2) Linear Conflict Heuristic for both our algorithms and collect data on each of their respective performances. After implementing the code and collecting the data, we have concluded that 1) Linear Conflict Heuristic outperforms City Block Distance in terms of the number of nodes expanded and the percentage of solutions produced in both algorithms while guaranteeing optimality; 2) heuristic run-time accounts for a significant part of the search's total run-time and 3) A* Search has a higher percentage of solutions produced than RBFS given time constraints but at the cost of more CPU-Time.

1 Introduction

Our goal is to solve a 15-puzzle using two different search algorithms and two different heuristics to support those searches. The 15 puzzle is a 4x4 grid with tiles numbered 1 through 15 and one empty space. Legal moves for the numbered tiles are up, down, left, or right into the empty space (only one tile may move at a time). The goal state is to have the tiles ordered starting in the top left (i.e. 1, 2, 3, 4 on the first row, etc). We implement our algorithms in Python and collect data across various metrics. We plot the data for better understanding of the results and discuss the implications of the results.

2 Experimental Setup

To implement everything, we used Python to program and solve the 4x4 15-puzzle game. Our code (via the Scramble function) takes in 10 solved puzzles and scrambles them with 10, 20, 30, 40, and 50 random legal move sequences. Then, our code attempts to solve the puzzles using both search algorithms (A* and RBFS) and records the data. Each search algorithm uses 2 different heuristic functions.

2.1 Heuristic Functions

We use two admissible heuristic functions for our program which are

City Block Heuristic The City Block Distance (a.k.a Manhattan Distance) calculates the number of spaces each tile needs to travel from its current position to the target position in a solved puzzle. This is calculated with only that single tile on the board and the tile is free to move to any space. This greatly relaxes the problem and ignores the cost of "going around" other numbers in order to let the current number go to its target destination. For example, if the number 1 tile was in the bottom right corner its City Block Distance would be 6 (3 moves to the left and 3 moves up to the top left corner where it belongs). The sum of all tile's movements is returned as the City Block Heuristic estimate. This heuristic is admissible because it always underestimates the number of moves needed to get a tile to its goal position (since it calculates the distance as if there were no other tiles on the board).

Linear Conflict Heuristic Our Linear Conflict Heuristic builds on the City Block Distance and takes into account tiles that are "blocking" one another in the same row or column. For example, suppose we have this first row configuration:

4	X	X	1

and we have its target configuration:

1	X	X	4

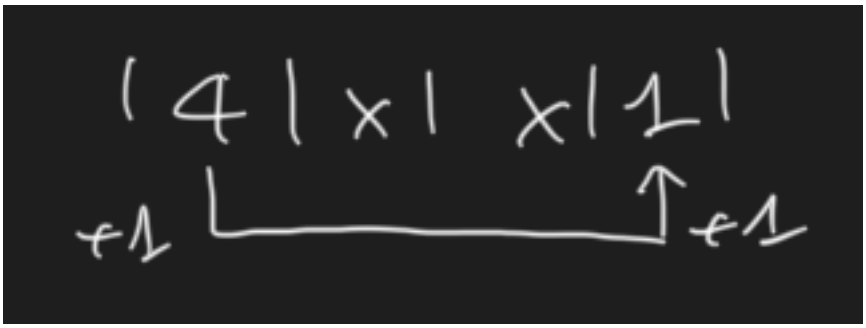


Figure 1: Linear conflict example

In here, the target positions of cell 4 and cell 1 are both in the same row with the configuration, and at the opposite side to those in the configuration. This implies that either cell 4 and cell 1 has to move to a new line to move around the other cell. Specifically, if we want to move cell 4, in order to move 4 to $(1, 4)$, which is the target position of 4 and the position of 1, we have to 1) either move up or down to a new line and "go around" cell 1 and 2) move both cell 1 along the row to $(1, 1)$, cell 4 to $(1, 4)$.
The first action takes at least 2 actions to 1) move to a new line and 2) move back to the current, as illustrated in Figure 1. The second action takes Manhattan distance.
To make all cells in each row not linear conflict with each other, we have to remove cell 4. Therefore the additional heuristic value for the current row row_i is two times the least number of cells $|cell^i_{conflict}|$ in each row such that the rest of the cells are not linear conflict: $f_{row_i} = 2 * |cell^{row_i}_{conflict}|$
Note that the same logic is applied to each column $column_i$. In short we have the heuristic value for the current configuration is:

$$f = Manhattan + \sum_{i=1}^{N} 2 * |cell^{row_i}_{conflict}| + 2 * |cell^{column_i}_{conflict}|$$

where N is the size of the grid.
This heuristic function f is admissible since we only find the least number of cells removed so it never over-estimates the actual cost.

2.2 A* Search

A* search operates by generating a parent node and expanding its child nodes. The parent node evaluates the f-value (heuristic) and then the node (estimate heuristic value, count for solution length, and the node data itself) is stored in a priority queue as it operates on first-in-first-out (FIFO) data structure. Then the children for parent node are generated and then expanded (we keep track of the expanded nodes using an expanded list). Then we update the estimated f-value by adding the child cost plus its corresponding heuristic value. Then we store the new node(s) in the same priority queue. This process is repeated until the goal state (solved puzzle) is reached and the solution length, number of nodes expanded are returned.

2.3 Recursive Best-First Search (RBFS)

Recursive Best-First Search operates by expanding the child nodes of the given parent node. The algorithm maintains an f limit (cost plus heuristic estimate) for nodes that it explores on its frontier. For the child nodes, it selects the best node (lowest f value) and expands that node while also keeping track of the estimate of the best alternative. If the f limit of a node exceeds the search f limit, the search unwinds back to the alternative path.

3 Results

The data recorded includes Average Time Consumed, Number of Nodes Expanded during search, the length of the solution(s) and the percentage of the times the solution is produced, for 10 different puzzles per each move sequence. Then we plot the graphs for move sequence on y-axis vs Average Time Consumed, Average # of Nodes Expanded, Average Solution Length on x-axis on different graphs respectively. The plotted graphs are for each algorithm with blue using the city block heuristic function and red being the linear conflict heuristic function.

3.1 Table

Search Algorithm	Move Sequence m	Average Time Consumed (sec)	# of Nodes Expanded	Average Solution Length for Solved	% of Solutions Produced	Heuristic % Run Time
A*	10	0.0013	35	10	100	13.3000
	20	0.0166	4656	20	100	11.3460
	30	1.2772	11289	28	100	6.5851
	40	5.8670	28028	27	50	5.2937
	50	9.5943	45651	23	10	2.7554
RBFS	10	0.0008	15	10	100	22.0289
	20	0.1192	2161	20	100	21.6843
	30	2.2653	39486	28	50	21.5568
	40	1.8341	32458	26	40	21.5926
	50	2.6700	47098	32	10	21.7928

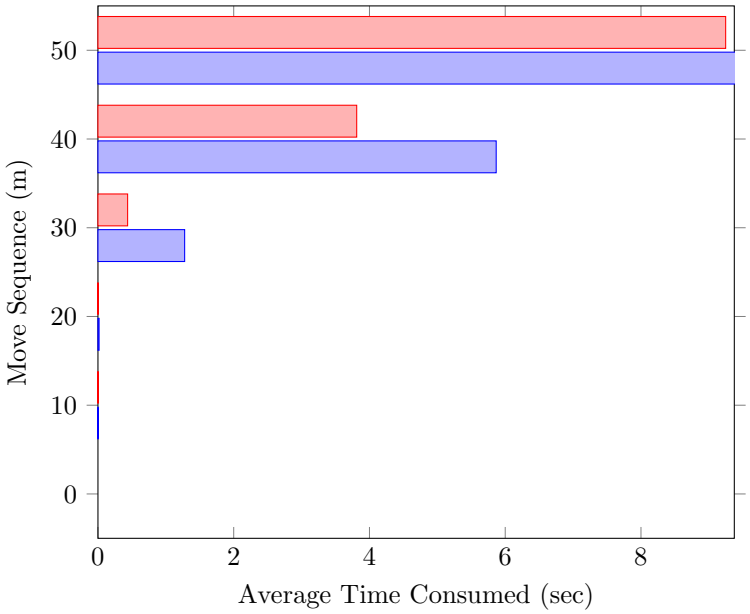
Table 1: Data for City Block Heuristic

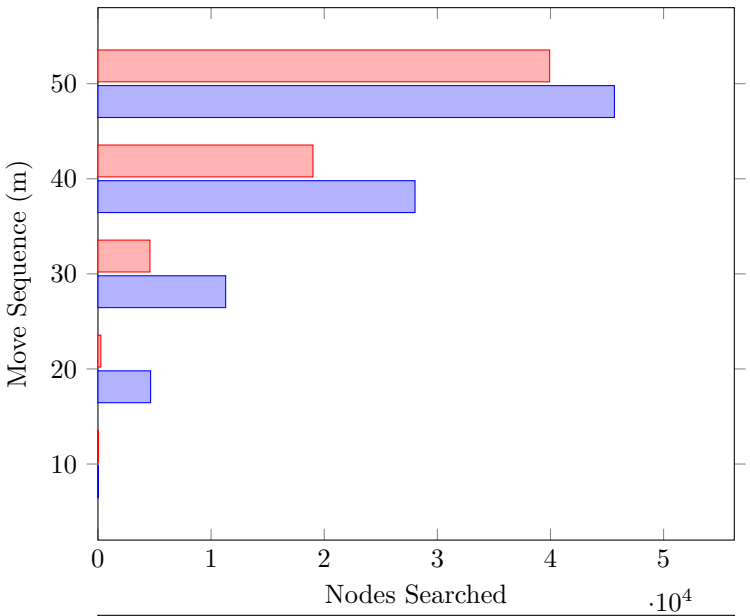
Search Algorithm	Move Sequence m	Average Time Consumed (sec)	# of Nodes Expanded	Average Solution Length for Solved	% of Solutions Produced	Heuristic % Run Time
A*	10	0.0017	35	10	100	58.0209
	20	0.0014	259	20	100	50.0850
	30	0.4371	4595	28	100	42.2862
	40	3.8120	19004	27	70	32.7600
	50	9.2476	39926	23	40	18.9070
RBFS	10	0.0018	15	10	100	74.6375
	20	0.0547	459	20	100	72.9310
	30	3.6482	28631	28	60	72.1275
	40	3.7285	32023	26	40	72.4587
	50	5.0885	45072	32	10	72.5811

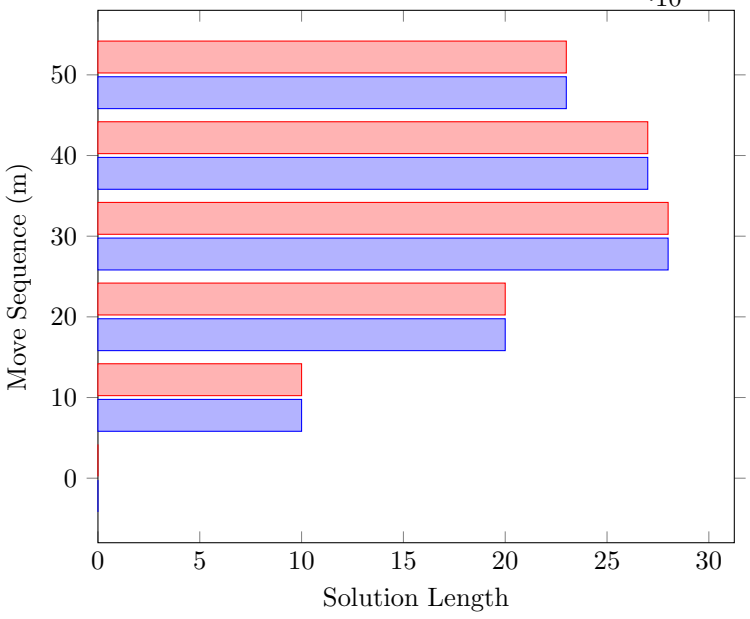
Table 2: Data for Linear Conflict Heuristic

Note for plots: Blue = City Block Heuristic , Red = Linear Conflict Heuristic

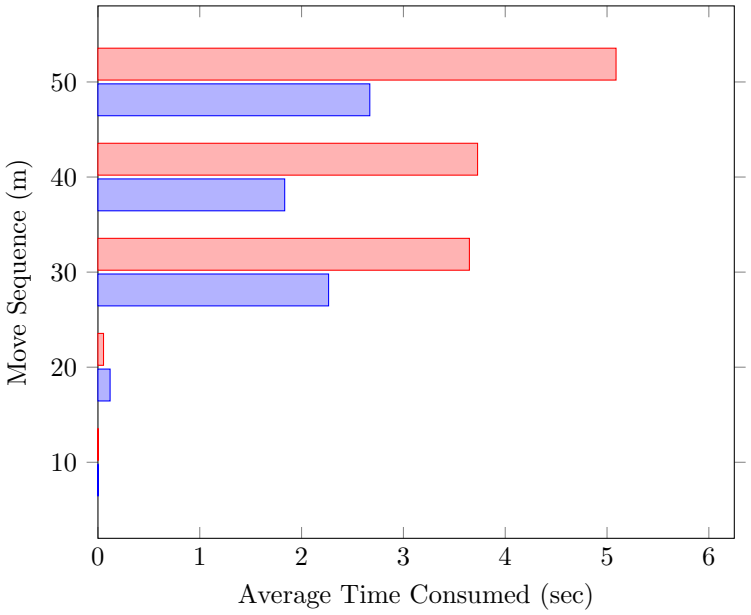
3.2 Plots for A*

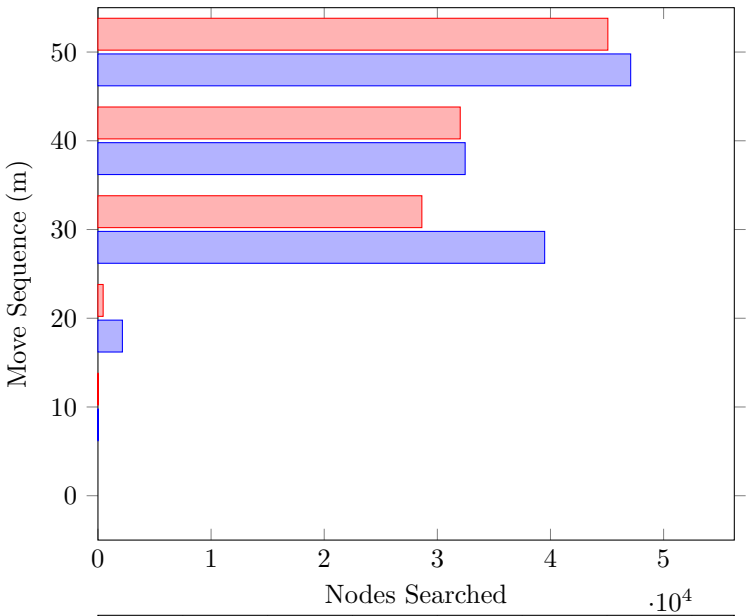


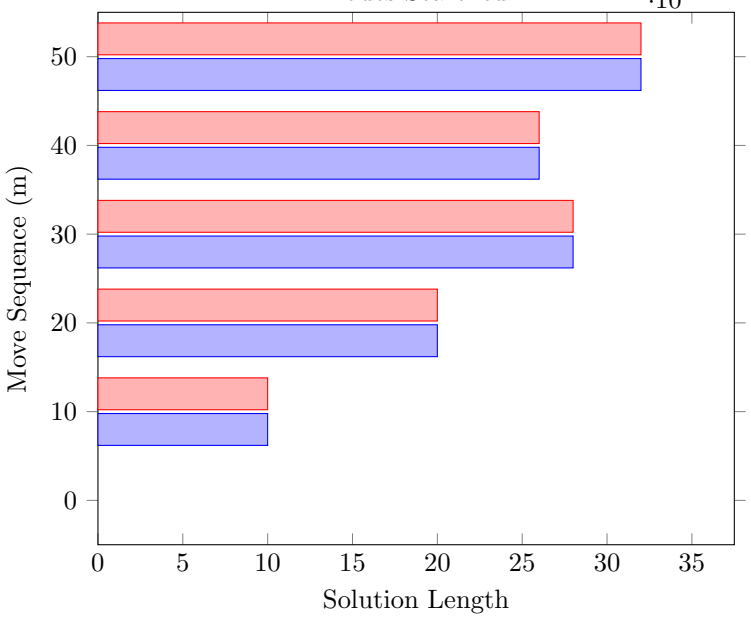




3.3 Plots for RBFS







4 Discussion

Two heuristic functions The Linear Conflict Heuristic was able to significantly reduce the number of nodes expanded in both searches. This resulted in a slight increase in the proportion of puzzles solved compared to City Block, and thus reducing the overall time consumed in A*. However, due to the extra time in evaluating the Linear Conflict Heuristic the RBFS had longer run times/higher CPU demands.
The Linear Conflict Heuristic is admissible so it always guarantees optimality. Therefore, the solution lengths using the two heuristic functions in each algorithms are the same.

Heuristic Run-Time The heuristic run-time can make a significant impact on a search's total run-time. For example, the Linear Conflict Heuristic made up roughly 70% of the total RBFS run-time. The reason for such a large percentage is because of the nature of RBFS using such small memory and potentially regenerating nodes many times.
Before creating our final heuristic, we created one that built on the city block heuristic by incorporating the distance distance to the empty tile for each tile. The assumption was that the empty tile would need to move to the tile's position (if the tile was not in its target location) to get the tile to move. This was tested against the City Block Heuristic for both searches and it performed worse in both dimensions: it explored more nodes and did not find optimal solutions, so we did not use it.

Two search algorithms The RBFS search was able to out perform the A* Search in terms of CPU-Time (lower average time consumed). The A* Search was able to solve more puzzles (higher % of solutions produced) compared to RBFS.
The A* and RBFS searches are both consistent and optimal so they are always able to produce the optimal solution given no time constraints. In our case, we limited the maximum number of nodes searched to 50,000 before stopping the search, hence that is why our data does not show 100% of solutions produced for the searches.