Luke Garrigan

Registration number 100086495

2017

AI search algorithms for the solution of puzzles

Supervised by Dr Pierre Chardaire

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

Acknowledgements

Contents

1 Introduction

A search algorithm is a series of steps that can be used to find the location or the path of a desired state. In most scenarios there will be additional constraints that will need to be fulfilled such as the time taken to reach the desired state, memory availability, maximum number of moves. This paper shows experimental results of numerous search algorithm for specific problem domains such as the sliding-tile puzzle, Towers of Hanoi and the Rubik's cube.

A classic example in the AI literature of pathfinding problems are the sliding-tiles puzzles such as the 3×3 Eight Puzzle, the 4×4 fifteen puzzle and the 5×5 twenty-four puzzle. The Eight Puzzle consists of a 3×3 grid with eight numbered square tiles and one blank. The blank is used to slide other tiles in which are horizontally or vertically adjacent into that position in an attempt to reach the goal state. The objective is to rearrange the tiles from some random configuration to a specified goal configuration usually with the numbers in ascending order.

The Towers of Hanoi puzzle consists of 3 pegs and n discs all of varying sizes. The discs are initially stacked on the left peg in decreasing order of size with the largest on the bottom. The task is to move all discs from the initial peg to the goal peg without violating two constraints: Only the top disc of any peg can be moved and a larger disc can never be placed on top of a smaller disc. For the 3 peg problem a simple recursive algorithm can be used to find the solution in the minimum number of moves, however, for the 4 peg problem (TOH4) the recursive algorithm doesn't find the goal state optimally. The recursive algorithm works by initially moving the $n-1$ smallest discs to the intermediate peg, then moving the $n-1$ largest disc to the goal peg, then move the $n-1$ smallest discs from the intermediate peg to the goal peg. Finding shortest path with this algorithm is not possible with TOH4 because there are two intermediate pegs rather than one and without priori there is no way to determine which peg should be used over the other. This implies the use of systematic search to find the optimal path.

2 Relevant Literature

3 Program Architecture

3.1 Initial Implementation

When a node is being expanded it is necessary to find its neighbours. Neighbours for the sliding-tile puzzle represent the possible moves from the current configuration, each state has a minimum of 2 neighbours and a maximum of 4. Figure 1 shows the possible moves from current state and how the direction variable would be stored ready for output.

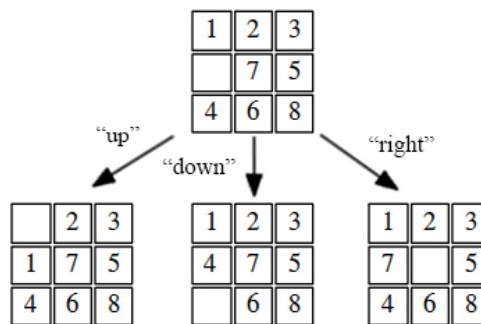Figure 1: Finding possible moves

4 Brute-Force Search

Brute-force search is a general problem-solving technique that consists of systematically enumerating all the possible options for a given solution and checking to see whether that given option satisfies the problem's statement. All that is required to execute a brute-force is some legal operators, an initial state and an acknowledged goal state.

4.1 Breadth-First Search

Breadth-first search expands the nodes in a tree in the order of their given distance from the root, so it expands all the neighbouring nodes before going to the next level of the tree. The algorithm doesn't trawl to the deeper levels of the tree without first expanding the lower levels thus ensures the finding of the shortest path. The amount of time used by breadth-first search is linear to the number of nodes expanded, since each node can be generated in constant time, and is a function of the branching factor b and the solution depth d. Since the number of nodes at level d is b^d, the total number of nodes generated in the worst case is $O(b^d)$. (Korf, 1995).

The initial algorithm implemented to solve the 8-tile puzzle was the breadth-first search as it guarantees to find the solution in minimal steps. The space requirement of breadth-first search is it's largest deficiency. The 8-tile has a search space of $9!/2 = 181,400$ with a maximum number of 31 moves to solve. In terms of practicality, with larger problem states such as the 15-tile puzzle a breadth-first search will exhaust the available memory rather quickly with its $16!/2 = 10,461,394,944,000$ search space and a maximum number of 80 moves to solve.

Table 1 shows the results of the breadth-first search algorithm in the 8-tile puzzle. The algorithm was tested on states of varying complexity from a minimum of 5 moves to solve, to a minimum of 20. As established, breadth-first search finds the optimal solution to a goal state with the minimum number of moves. As the difficulty of the initial puzzle state increases the number of nodes expanded increases exponentially along with the time taken. This proves the previously stated theory that the number of nodes generated in the worst case is $O(b^d)$.

4.2 Depth-First Search

Depth-first search (DFS) addresses the limitations of breadth-first by always generating next a child of the deepest unexpanded node. Breadth-first search manages the list as a first-in first-out queue, whereas depth-first search treats the list as a last-in first-out stack. Depth-first search is implemented recursively, with the recursion stack taking the place of an explicit node stack. Given a starting state, depth-first search stores all the

Table 1: Breadth-First Search Analysis: 8-Tile

Min Moves	Moves	Nodes Expanded	Time(ms)
5	5	60	2.097152
10	10	619	10.48576
15	15	8159	448.79053
20	20	65638	34621.883

unvisited children of that state in a stack. It then removes the top state from the stack and adds all of the removed states children to the stack. Depth first search generates a long sequence of moves, only ever reconsidering a move when it reaches the end of a stack, this can become a serious problem given a graph of significant size and there's only one solution, as it may end up exploring the entire graph on a single DFS path only to find the solution after looking at each node. Worse, if the graph is infinite the search might not terminate.

Depth-first search is not the preferred algorithm for solving the sliding-tile puzzle especially with the goal of solving the puzzle in the minimum number of moves, it would be more preferred in an implementation which has goals in every path. The algorithm was implemented regardless to analyse whether the results match expectation. Table 2 displays the results of the depth-first search implementation, there is no real noticeable pattern in the data apart from they are consistently inconsistent. The results prove that the algorithm doesn't find the goal state in the minimum number of moves which further confirms that DFS should not be used in the sliding-tile problem.

4.3 Depth-First Search Iterative Deepening

Depth-First Iterative-Deepening (DFID) is an extension of depth-first search, it combines breadth-first search's completeness and depth-first search's space efficiency. DFID has a maximum depth, so it searches all possibilities up to a certain depth and if it doesn't find the goal state it increases the search depth. DFID performs depth-first search to depth one, then starts over and executes a depth-first search to depth two and

---

Algorithm 1 Breadth-First Search

1: **procedure** BFS(*state*)

2:     *s* ← *empty set*

3:     *q* ← *empty queue*

4:     *q.add*(*state*)

5:     *q.enqueue*(*state*)

6:     **while** *q.size*() > 0 **do**

7:         *currentState* ← *q.dequeue*()

8:         **if** *currentState* = *goal* **then**

9:             **return** *current*

10:         **for** *neighbour* **in** *neighbours*(*currentState*) **do**

11:             **if** *neighbour* is not in *s* **then**

12:                 *s.add*(*neighbour*)

13:                 *q.enqueue*(*neighbour*)

Algorithm 2 DFS

1: *visited* ← *emptySet*

2: **procedure** DFS(*state*)

3:     **if** *state* = *goal* **then**

4:         **return** state

5:     **for** *neighbour* **in** *neighbours*(*state*) **do**

6:         **if** *neighbour* **not in** *visited* **then**

7:             *DFS*(*neighbour*)

8:     *visited.add*(*state*)

Table 2: Depth-First Search Analysis: 8-Tile

Min Moves	Moves	Nodes Expanded	Time(ms)
5	57510	112377	352451.56
10	7817	8199	1021.31305
15	61050	94710	273506.38
20	21123	22420	11370.758

continues deeper and deeper until a solution is found. The complexity of DFID is only O(d) where d is the depth, this is because at a given point it is executing only a depth-first search and saving only a stack of nodes. DFID ensures that the shortest path to the goal state will be found as does breadth-first search. (Meissner and Brzykcy, 2011).

Algorithm 3 Iterative Deepening Depth-First Search

1: **procedure** IDDFS(state)
2:     **for** $depth \leftarrow 0, \infty$ **do**
3:         $found \leftarrow DLS(state, depth)$
4:         **if** $found \neq null$ **then**
5:             **return** $found$
6: **procedure** DLS(state, depth)
7:     **if** $depth = 0$ **and** $state = goal$ **then**
8:         **return** $found$
9:     **if** $depth > 0$ **then**
10:         **for** $neighbour$ **in** $neighbours(state)$ **do**
11:             $found \leftarrow DLS(neighbour, depth - 1)$
12:             **if** $found \neq null$ **then**
13:                 $return found$
14:     **return** $null$

Although DFID maintains the advantage of requiring a comparable supply of memory as depth-first search and doesn't get caught in infinite loops, DFID's time taken to find a

solution is vast and generally longer than both depth-first search and breadth-first search, this is because it has to expand the same states multiple times. Table 3 are the results for DFID with the depth preset to the minimum number of moves required to solve the puzzle state. It is clear that the number of nodes expanded is far greater in comparison with breadth-first search shown in Table 1.

Table 3: Depth-First Iterative Deepening: 8-Tile

Min Moves	Moves	Nodes Expanded	Time(ms)
5	5	56	0.9521
10	10	6392	16.777216
15	15	3350884	2902.4583

4.4 Evaluation Of Brute-Force Searches

Depth-first search often provokes the possibility of going down the left-most path forever, it is also not guaranteed to find the solution and the likelihood that the solution found is optimal is unlikely, so for the sliding-tile puzzle in general this is not useful.

Breadth-first search is relatively fast when considering the 8-tile puzzle as the search space is small, however it uses too much memory to be useful with larger problems such as the 15-tile puzzle.

Depth-first iterative deepening is beneficial as it avoids infinite cycling, it obtains the same result as BFS whilst saving memory, however it is very slow as it has to repeatedly expand nodes it has already visited. DFID is only really beneficial when the solution depth is known, else it would have to trawl through all possibilities up to the solution depth and then through all possibilities at the solution depth until it reaches the goal. It is unlikely for the sliding-tile puzzle that the user knows the solution depth, thus implies ample execution time to find the goal state.

5 Heuristic Search

In order to improve the search efficiency heuristic functions were used. A heuristic is a method that might not always find the best solution but it will guarantee to find a good solution in a reasonable time. The idea behind a heuristic function is that it sacrifices the completeness in order to increase efficiency. They are usually utilised in solutions in which may take infinite time or a very long time to compute. (Sokolova and van Beek, 2014).

5.1 Admissible Heuristics

A heuristic function is said to be admissible if it never overestimates the cost of reaching the goal, i.e the estimated cost to the goal from the current node is not higher than the lowest possible cost from the current node. The lowest possible cost $h^*(n)$ is the cost of the optimal path from n to a goal node G. The heuristic function $h(n)$ is admissible if $0 \leq h(n) \leq h^*(n)$, essentially meaning admissible heuristic functions are optimistic.

5.2 Consistent Heuristics

One crucial characteristic of the heuristic required for my analysis is that it must be consistent. A heuristic function is said to be consistent if the cost from the current node n to a successor node p, plus the estimated cost from the successor node to the goal state is less than or equal to the estimated cost from the current node to the goal node $h(n) \leq c(n, p) + h(p)$, where $c(n, p)$ is the cost of reaching node p from n and the heuristic value of the goal G is zero $h(G) = 0$.

Heuristic search allowed me to find faster solutions to the sliding-tile puzzle and even gave me the opportunity to increase the search space, so moving up to the 15-tile puzzle.

5.3 Heuristic Evaluation Functions

5.3.1 Manhattan Distance

A common heuristic function for the sliding-tile puzzle is called Manhattan distance. The Manhattan distance is the distance between two points in a grid, based on a strictly
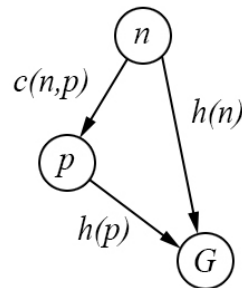
Figure 2: Consistent Heuristics Diagram

horizontal and/or vertical path and summing these values over all tiles. For a fixed goal state, a heuristic evaluation is a function of a node, $h(n)$ in which estimates the distance from the node to said goal state.

Heuristic evaluation functions estimate the actual cost with very little computational power, thus very compelling. The complexity of the Manhattan Distance is proportional to the number of tiles between the initial state and the goal state. Most heuristic functions are lower bounds on the actual cost deeming them admissible. The Manhattan distance is a lower bound for the number of moves required to solve an occurrence of the sliding-tile puzzle, since every tile must move at least as many times as its distance to its goal position. (Linnert et al., 2014).

5.3.2 Linear Conflict Heuristic

Linear Conflict is somewhat similar to Manhattan distance in regards to the sum of the displacement of each each tile, however it also recognises that given two tiles which are both correctly placed in their desired row/column but they are in the wrong order, one of these tiles will have to move out of the row/column in order to let the other pass. If there is a linear conflict then an extra two moves will be added to the total sum as the tile which must move, has to transition out of the row/column and then back into it's desired position.

Algorithm 4 Manhattan Distance

1: **procedure** MANDIST(*state*)   ▷ The current puzzle configuration

2:     *total* ← 0

3:     *puzzleLength* ← *state.size*()

4:     *dimensions* ← $\sqrt{puzzleLength}$

5:     **for** *i* ← 1, *puzzleLength* **do**   ▷ Loops through each tile of the puzzle

6:        *tileValue* ← *state*[*i*]

7:        *expectedRow* ← (*tileValue* − 1) ÷ *dimensions*

8:        *expectedCol* ← (*tileValue* − 1) mod *dimensions*

9:        *rowNum* ← *i* ÷ *dimensions*

10:       *rowNum* ← *i* mod *dimensions*

11:       *total* ← *total*+ | *expectedRow* − *rowNum* | + | *expectedCol* − *colNum* |

12:     **return** *total*   ▷ The heuristic is the total

```
private int linearVerticalConflict(int[] state) {
    int dimension = 3;
    int linearConflict = 0;
    int count =0;
    for (int row = 0; row < dimension; row++) {
        int max = -1;
        for (int column = 0; column < dimension; column++) {
            int cellValue = state[count];
            count++;
            if (cellValue != 0 && (cellValue - 1) /
             dimension == row) {
                if (cellValue > max) {
                    max = cellValue;
                } else {
                    linearConflict += 2;
                }
            }
        }
```

```
        }
    }
    return linearConflict;
}
```

Figure 2 displays a visual example of how the linear conflict is working. The initial state shows the first row to be reversed meaning values 1,2 and 3 are positioned 3,2 and 1. When looping through the values the *max* is compared to 3, as 3 is greater than -1 *max* is now 3. For value 2 it is not larger than the current *max* 3 so a linear conflict has happened thus 2 is added to the total, value 1 is also not larger than 3 thus another two is added to the total heuristic value.
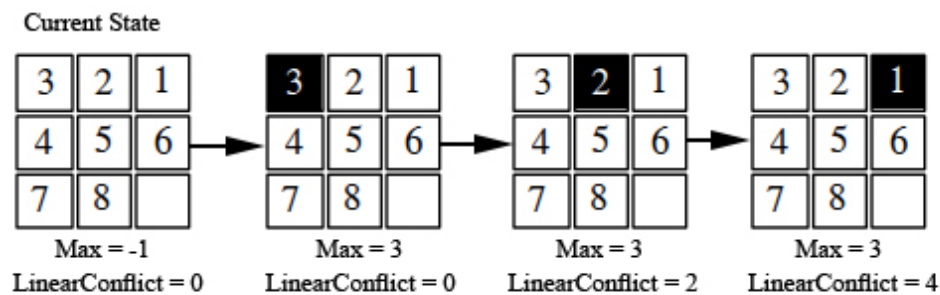
Figure 3: Calculating Linear Conflict

5.4 A* Algorithm

A* search is a combination of lowest-cost-first and best-first searches that considers both path cost and heuristic information in its selection of which path to expand. For each path on the frontier, A* uses an heuristic of the path cost from a start node to a goal node constrained to start along that path. This allows A* search to ensure the prioritisation of states in which are more likely to result in a low-cost goal state. The calculation of the heuristic value is dependent on the problem that is being solved. For example, for problems which aim to reach a location the Euclidean distance between the state and the goal is often used as the heuristic.

The A* algorithm uses $cost(p)$, the cost of the path found, as well as the heuristic function $h(p)$, the estimated path cost from the end of p to the goal. As A* uses admissible heuristic it will always find solutions in the order of their cost. (Brewka, 1996). A* uses breadth-first traversal thus uses large chunks of memory in comparison to applications that use depth-first traversal. The performance of A* is also heavily reliant on the accuracy of the heuristic.

The implementation for the A* algorithm works by initially declaring an empty *OPENSET* and an empty *CLOSEDSET* which are both *ArrayList*s of the *State* object. The commencing *State* is added to the *OPENSET*, the *OPENSET* is then looped through to find the *State* with the smallest *F* value, which is assigned to the *winner* variable. The *currentState* is then assigned to the State which contains that smallest *F* value, this is then checked against the *GOAL* state to check whether it is the solution. If it isn't a goal state it is removed from the *OPENSET* and added to the *CLOSEDSET* because we don't want to expand the same *State* again. The neighbours for the *currentState* are then found and looped through. If the neighbour is not in the *CLOSEDSET* it hasn't been expanded, the *G* value is assigned to the neighbour which is just the *currentStates*'s $G+1$, as in the sliding tile puzzle a tile can only move one position at a time. It then checks whether the neighbour is in the *OPENSET* as there's a chance we got to this neighbour more optimally than the previous path, if so it reassigns the values. The *F* value is then calculated for the neighbour by adding the *G* and the *H* value, the program then repeats the process.

5.4.1 Linear Conflict Or Manhattan Distance

My tests displayed in Table 4 show that A* search with the linear conflict heuristic improves the performance of A* with Manhattan distance for the 8-tile puzzle. On average A* with linear conflict would take around 1ms longer than A* with Manhattan distance, until the number of moves required to solve the puzzle were greater than 25, as this showed a substantial decline in time when comparing the linear conflict to the Manhattan heuristic. This implies that the performance of linear conflict exceeds the performance of Manhattan distance when the state space grows, so the more difficult the puzzle the greater the gap in performance between the two heuristics becomes.

Algorithm 5 A*

1: $g(state)$ ▷ The cost to reach the current state

2: $h(state)$ ▷ Estimated cost of the cheapest path from state to goal

3: $f(state) \leftarrow g(state) + h(state)$

4: $neighbours(state)$ ▷ Expands possible moves from current state ordered by g + h

5: **procedure** AStar($startState$)

6:      $closedSet \leftarrow empty\ set$

7:      $openSet \leftarrow empty\ set$

8:      $openSet.add(startState)$

9:      **while** $openSet$ **is not** $empty$ **do**

10:          $currentState \leftarrow null$

11:          **for** $state$ **in** $openSet$ **do**

12:             **if** $f(state) < f(currentState)$ **then**

13:                 $currentState \leftarrow state$

14:          $openSet.remove(currentState)$

15:          $closedSet.add(currentState)$

16:          **if** $currentState = goal$ **then**

17:             **return** $currentState$

18:          **for** $neighbour$ **in** $neighbours(currentState)$ **do**

19:             **if** $neighbour$ **not in** $closedSet$ **then**

20:                 **if** $neighbour$ **in** $openSet$ **then**

21:                     **if** $g(currentState) < g(neighbour)$ **then**

22:                         $g(neighbour) \leftarrow g(currentState)$

23:                 **else**

24:                     $openSet.add(neighbour)$

Table 4: Manhattan Distance Vs. Linear Conflict - A*

Min Moves	Manhattan Distance			Linear Conflict		
	Time(ms)	Mean Memory	States	Time(ms)	Mean Memory	States
5	0.3170	4.0267	6	1.3602	4.0267	6
10	0.4326	4.0267	11	1.3391	4.0267	11
15	0.5599	4.0267	16	1.4346	4.0267	16
20	0.7307	4.0267	21	1.6672	4.0267	21
25	4.4329	4.0267	141	4.2708	4.0267	103
30	532.9964	4.9195	5640	277.7063	5.3643	3752

As the state space grows exponentially according to the depth the growth is smaller for the A* with linear conflict, this is because the heuristic is a more accurate representation of the displacement of the tiles, thus the H value is always larger than or equal to the H value of the Manhattan Distance. Table 5 shows the difference in H value for 10 calculations of both algorithms and how the linear conflict is nearly always larger than the Manhattan distance.

Table 4 shows the exponential growth by the number of states the algorithm expanded. The average number of States for the Manhattan distance heuristic from 5 to 30 moves is 972.5, whereas linear conflict's average is 651.

6 Fifteen Tile Puzzle

I initially tested the A* Manhattan distance and the A* linear conflict on the fifteen tile puzzle, however could only get results for puzzle states which require a small number of moves to find the goal. The reasoning for the lack of results is the large amount of States that the *CLOSEDSET* has to store.

A* uses breadth-first traversal thus the number of states that are stored increases exponentially with depth. Finding an algorithm which would require less memory but still have the benefits of a heuristic search was necessary. This is similar to the previous scenario when considering how to reduce the memory used in breadth-first by taking

Table 5: Manhattan Distance Vs Linear Conflict Values

State	Manhattan Distance	Linear Conflict
1	20	22
2	20	26
3	20	24
4	22	24
5	21	23
6	21	23
7	21	23
8	21	23
9	22	22
10	20	22

depth-first search's memory used as inspiration.

The memory used by Depth-first iterative deepening is linear with respect to the depth. In theory DFID could be used to solve the 15-puzzle without running out of memory, however taking the results found in Table 3, the time taken to reach the goal state for any random instance of the fifteen tile puzzle would be unreasonable.

6.1 Iterative Deepening A*

Iterative-deepening-A* (IDA*) eradicates the memory complication of A* by using depth-first traversal, without sacrificing solution optimality at the cost of increasing the time taken. Each iteration of A* is a complete depth-first search that keeps track of the cost, $f(n) = g(n) = h(n)$, of each node generated. If an expanded node's cost generated exceeds a threshold for that iteration, its path is cut off and the search backtracks before continuing. IDA* ranks states the same way as A*: The expected cost of a state is the cost of reaching said state and the heuristic's estimate of the cost of reaching the nearest goal state. The maximum expected cost is assigned to the heuristic cost of the starting state, states in which have a lowest cost than the starting state are traversed and states

with a higher expected cost are discarded.

If IDA* has searched all states lower than the current maximum without finding the goal state then it increases the maximum to the lowest value of the discarded states and begins the search again. The goal state is admissible, the current maximum cost will never be higher than the lowest cost solution, thus IDA* with an admissible heuristic will always find the lowest cost solution. IDA* searches many nodes multiple times thus is slower than A*. Hence, A* should be used on smaller applications that aren't going to have memory issues.

The Java implementation of the IDA* algorithm above begins by initially retrieving the *H* value of the *start* node as it has already been set during object creation. The code then loops through until a solution is returned from the *depthFirstSearch*() method that isn't null, thus the goal node. The float *currentCostBound* is assigned to *nextCostBound* and used to perform the depth-first search to that threshold, the bound is increased each time.

The *depthFirstSearch*() initially checks whether the *current* State is equal to the goal node and if so returns it, else it will then further loop through it's neighbours. Whilst looping, the *G* value is set for the neighbours which is the number of moves taken to get there, so it is the parent's *G* value incremented by one. The heuristic value is then calculated, in this case the linear conflict heuristic and then the sum of the *G* and the *H* form the *F*, which is used to determine which of the moves is the best from the *current* State, and if the *F* value is lower than the *currentCostBound* then a recursive call on *depthFirstSearch* method is called with the neighbour as the argument.

Table 6: Iterative Deepening A* Results

---

Algorithm 6 Iterative Deepening A Star

1: *state* ▷ The current puzzle configuration

2: *g*(*state*) ▷ The cost to reach the current state

3: *h*(*state*) ▷ Estimated cost of the cheapest path from state to goal

4: *f*(*state*) ← *g*(*state*) + *h*(*state*)

5: *neighbours*(*state*) ▷ Expands possible moves from current state ordered by g + h
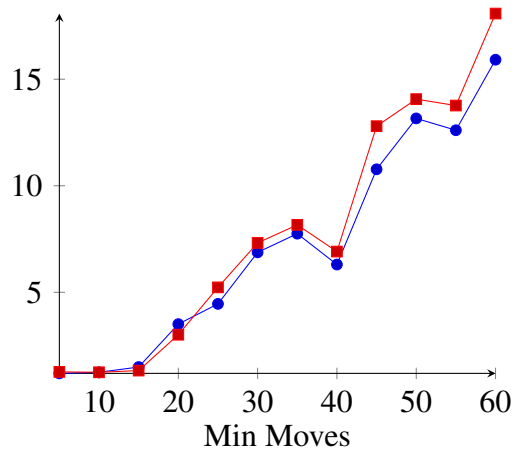
6: **procedure** IDASTAR(*state*)

7:     *bound* ← *f*(*state*)

8:     **while** **not** solved **do** ▷ Loops until a solution is found

9:         *bound* ← *DFS*(*state*, *bound*) ▷ Performs a bounded depth-first search

10: **procedure** DFS(*state*, *bound*)

11:     **if** *f*(*state*) > *bound* **then**

12:         **return** *f*(*state*)

13:     **if** *h*(*state*) = 0 **then** ▷ No more moves needed to reach goal state

14:         **return** *solved*

15:     *min* ← ∞

16:     **for** *neighbour* **in** *neighbours*(*state*) **do**

17:         *temp* ← *DFS*(*neighbour*, *bound*)

18:         **if** *temp* < *min* **then**

19:             *min* ← *temp*

20:     **return** *min* ▷ Returns the smallest of the neighbours

---

	Linear Conflict		Manhattan Distance	
Min Moves	Time(ms)	States Expanded	Time(ms)	States Expanded
5	2.2913	6	2.4060	6
10	2.3751	13	2.3739	11
15	2.8140	24	2.5081	16
20	11.3692	919	8.0932	768
25	21.9585	10420	37.5986	25272
30	117.3271	91471	159.5604	217239
35	215.4406	231508	287.0284	482893
40	78.972869	61141	120.4332	110903
45	1750.4441	2468006	7111.9494	14202684
50	9162.8202	13704525	17143.5055	36202294
55	6233.2757	9132143	13912.0609	29537720
60	61771.3520	87466771	277232.1371	596363837
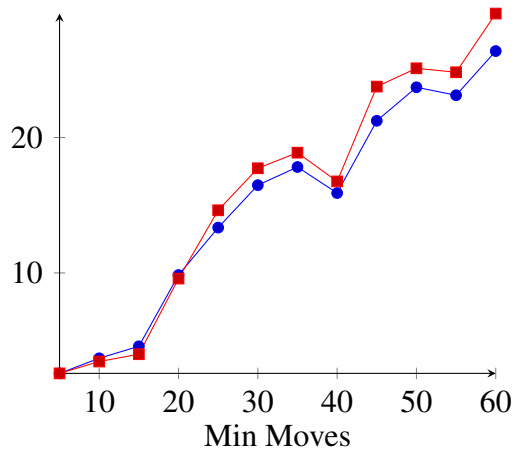
(a) Log Of The States Expanded   (b) Log Of The Time

Figure 4: Logarithmic Comparison Of IDA* Fifteen Tile

The time complexity of IDA* is measured by the number of nodes expanded, provided each node can be expanded and it's successors evaluated in constant time, the asymptotic time complexity of IDA* is the total nodes expanded. Otherwise it is the product of the total number of nodes and the time taken to expand each. Provided the heuristic is

consistent (see 4.2), IDA* must expand all nodes with the f value less than or equal to c which is the cost of an optimal solution $f(n) \leq c$. This is shown in my code by

```
if(value <= currentCostBound)
```

On the final iteration of IDA* the cost threshold is equal to c, the cost of an optimal solution.(Korf et al., 2001).

The results gathered show that in Table 6 and Figure 4 further compliment the findings of the contrast in efficiency of the two heuristics Manhattan distance and Linear conflict. With the number of states expanded increasing the time taken to find the goal state increases and thus the difference between the linear conflict and Manhattan Distance widens.

7 Pattern Databases

7.1 Solvability Of Tile Games

Problem domains may have constraints that can preclude parts of the search space. For example, although the fifteen puzzle has $16! \approx 10^{13}$ possible positions, only one half of them can be reached from the goal. (Culberson and Schaeffer, 1998).

To determine whether a sliding-tile puzzle is solvable we must calculate the sum of the inversions. An inversion is when a tile precedes another tile with a smaller number on it. The first step we need to take is to flatten the puzzle as shown in Figure 5, for my implementation the puzzle is already stored as an array. There are 3 rules we must adhere to: (Ryan, 2004)

- If the grid width is odd, then the number of inversions must be even.

- If the grid width is even, and the blank is on an even row counting from the bottom then the number of inversions must be odd.

- If the grid width is even, and the blank is on an odd row counting from the bottom then the number of inversions must be even.

Here is some code written to calculate the sum of the inversions in an array, from this we can determine whether it is solvable by following the guidelines above.
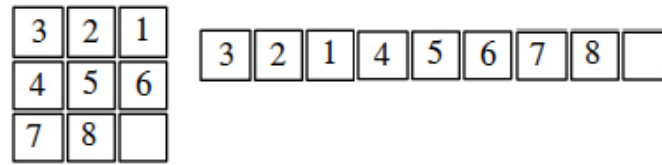
Figure 5: Puzzle Parity

Algorithm 7 Is Current State Solvable

 1: **procedure** ISSOLVABLE(state)

 2:     *puzzleLength ← state.size()*

 3:     *gridWidth ← √puzzleLength*

 4:     *blankRowEven ← true*

 5:     **for** *i ← 1, puzzleLength* **do**

 6:         **if** *state[i] = 0* **then**

 7:             *blankRowEven ← (i/gridWidth)* mod 2 = 0

 8:             **continue**

 9:             **for** *j ← i + 1, puzzleLength* **do**

10:                 **if** *state[i] > state[j]* **and** *state[j] ≠ 0* **then**

11:                     *parity ←!parity*

12:         **if** *gridWidth* mod 2 = 0 **and** *blankRowEven* **then**

13:             **return** !*parity*

14:     **return** *parity*

7.2 Non-Additive Pattern Databases

A Pattern Database stores a collection of solutions to sub-problems that much be achieved to solve the overall problem. They are admissible heuristic functions implemented as lookup tables that store the lengths of optimal solutions for sub-problem instances. (Felner et al., 2007).

The initial pattern database applied to the fifteen tile as shown in Figure 6 is the fringe pattern. The minimum number of moves required to position the fringe tiles to their goal, including required moves of other tiles is a lower bound on the number of moves needed to solve the entire puzzle. The heuristic is estimated dependent on the current

positions of the fringe tiles and the blank, the remaining tiles are disregarded. These values are precomputed and stored in a database and are called upon when a heuristic estimate is needed. The total number of possible permutations of fringe tiles and blank is $16!/(16-8)! = 518,918,400$.

To retrieve all possible permutations of the fringe tiles a single breadth-first search must be executed, commencing at the goal state. Tiles which are not fringe are all equivalent and in the database the positioning of the fringe tiles and the blank are stored along with the number of moves taken to get to that state.

Figure 6: Fringe Database Pattern For Fifteen Tile

Once the breadth-first search has finished and it has trawled through all $518,914,400$ states, using IDA* the heuristic function determines the positions of the fringe tiles and the blank which are used as an index to the row with the given configuration in the database, the value for that row is then retrieved and used as the heuristic value.
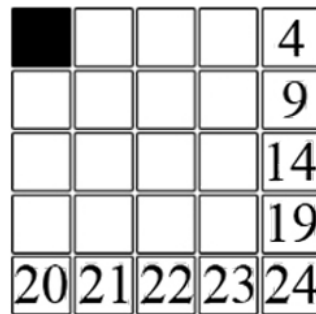
7.2.1 Non-Additive Pattern Database Limitations

If the tiles are divided into disjoint groups, the best way to combine them admissibly is to take the maximum of their values; non-additive pattern database values include all moves needed to solve the configuration including moves of other tile. (Felner et al., 2011).

Another major issue is space required to store all possible states. When considering the 15 tile: provided each row in the database is 1 byte and there are 519,914,400 rows that's a total of 519Mb, and the time taken to perform the breadth-first search is

extensive, Table1: Breadth-First Search Analysis shows that the time taken to expand just 65638 nodes was 34621.883ms.

Scalability is another issue regarding non-additive pattern databases. Take Figure 7 as an example, if we were to scale up to the twenty-five tile puzzle and performed the same calculations for taking the fringe: $25!/(25-10)! = 1.1861676e + 13$, provided each row in the database is 1 byte as previously stated, the summation is roughly 12 Terabytes worth of memory.

Figure 7: Fringe Database Pattern For The Twenty-five Tile

7.3 Statically-Partitioned Additive Database Heuristics

In order to develop a statically-partitioned additive database for the sliding tile puzzle, the tiles were partitioned into disjoint groups, meaning each tile is placed in a group and only appears in that group. The same method as explained in 6.2 for finding all the states was computed for each subset; a breadth-first search from the goal state with all possible configurations of each tile in the disjoint groups and the number of moves required to get to the goal state. For a particular state in the search, for each position of the tiles an index was computed for the corresponding row in the database, then retrieved the number of moves required to solve the tiles in that group and added this value to each value of the other disjoint groups. The summed total of these values will always be equal to or larger than the Manhattan distance of the state as it considers the interactions between tiles within a given subset.

I initially chose to implement the 6-6-3 partitioning as hypothesised it will be the best compromise between speed and memory in comparison with other possible solutions, Figure 8 shows the partitioning. The first two images display the fifteen tile with the subsets comprising of 6 tiles, $2 \times (16!/(16-6)!) = 11531520$ states to be stored, the final subset requires $16!/(16-3)! = 3360$ states stored that's a total of 11534880 rows in the database.

Figure 8: 6-6-3 Partioning

References

Brewka, G. (1996). *Artificial intelligence - a modern approach* by stuart russell and peter norvig, prentice hall. series in artificial intelligence, englewood cliffs, NJ. *Knowledge Eng. Review*, 11(1):78–79.

Culberson, J. C. and Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, 14(3):318–334.

Felner, A., Hanan, S., and Korf, R. E. (2011). Additive pattern database heuristics. *CoRR*, abs/1107.0050.

Felner, A., Korf, R. E., Meshulam, R., and Holte, R. C. (2007). Compressed pattern databases. *J. Artif. Intell. Res. (JAIR)*, 30:213–247.

Korf, R. E. (1995). Heuristic evaluation functions in artificial intelligence search algorithms. *Minds and Machines*, 5(4):489–498.

Korf, R. E., Reid, M., and Edelkamp, S. (2001). Time complexity of iterative-deepening-a*. *Artif. Intell.*, 129(1-2):199–218.

Linnert, B., Schneider, J., and Burchard, L. (2014). Mapping algorithms optimizing the overall manhattan distance for pre-occupied cluster computers in sla-based grid environments. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014, Chicago, IL, USA, May 26-29, 2014*, pages 132–140.

Meissner, A. and Brzykcy, G. (2011). Reasoning with the depth-first iterative deepening strategy in the dlog system. In Meersman, R., Dillon, T. S., and Herrero, P., editors, *On the Move to Meaningful Internet Systems: OTM 2011 Workshops - Confederated International Workshops and Posters: EI2N+NSF ICE, ICSP+INBAST, ISDE, ORM, OTMA, SWWS+MONET+SeDeS, and VADER 2011, Hersonissos, Crete, Greece, October 17-21, 2011. Proceedings*, volume 7046 of *Lecture Notes in Computer Science*, pages 504–513. Springer.

Ryan, M. (2004). Solvability Of Tile Gamesthe university of birmingham school of computer science.

Sokolova, M. and van Beek, P., editors (2014). *Advances in Artificial Intelligence - 27th Canadian Conference on Artificial Intelligence, Canadian AI 2014, Montréal, QC, Canada, May 6-9, 2014. Proceedings*, volume 8436 of *Lecture Notes in Computer Science*. Springer.