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

FIrstly, I would like to express my sincere gratitude to my advisor Dr. Pierre Chardaire for the patience, guidance, wisdom and brilliance he has provided throughout my final year. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly.

Contents

1 Introduction

In computer science a search algorithm is a series of steps that can be used to find a desired state or a path to a particular 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. The main difference between informed search and uninformed search is the use of heuristic functions that guide the search towards the goal while avoiding non-promising paths Felner (2015). Informed search algorithms such as A* Hart et al. (1968), IDA* Korf (1985) are analysed in this paper with various heuristics including Manhattan distance, linear conflict Hansson et al. (1985) and pattern databases (PDB) Culberson and Schaeffer (1996). A comparison between informed and uninformed search is made along with an in-depth analysis of different heuristics 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 8-puzzle, the 4×4 15-puzzle and the 5×5 24-puzzle. The 8-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. The number of possible solvable states for the 8-puzzle is $9!/2 = 181440$ so can be solved by means of brute-force search. However for the 15-puzzle with $16!/2 \approx 1.05 \times 10^{13}$ and 25-puzzle with $25!/2 \approx 7.76 \times 10^{24}$ a more sophisticated informed search is required.

The Towers of Hanoi puzzle consists of 3 pegs and n discs all of varying sizes. The discs are initially stacked on the leftmost 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 Preliminaries

4 Design

4.1 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. Algorithm 1 was used to create testing states for the sliding-tile puzzle, states with randomised permutations were constructed and then checked to see if they were solvable, once a total of 1000 solvable test cases were created the search algorithms were then experimented with.

5 Brute-Force Search

Brute-force search is a general problem-solving technique that consists of systematically enumerating all the possible states for a given solution and checking to see whether that given state 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.

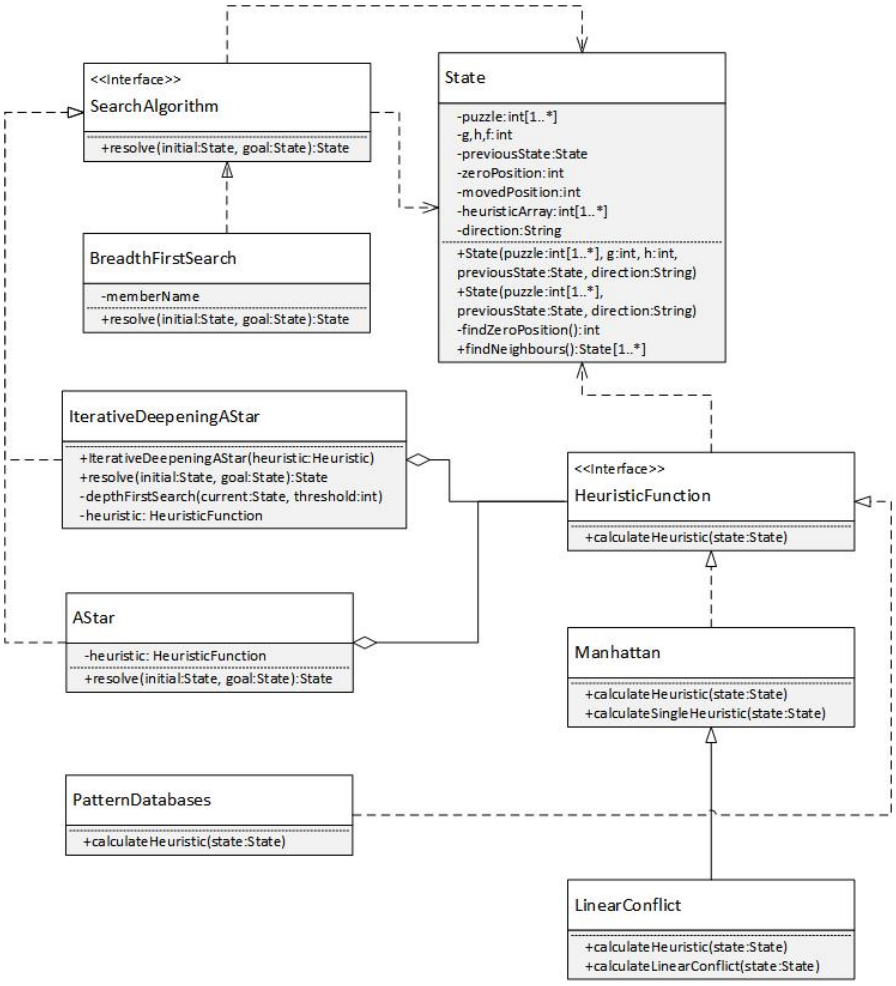


Figure 1: Class Diagram

5.1 Completeness and Optimality

Often in search the input may be an implicit representation of an infinite graph. Given these conditions, a search algorithm is characterised as being complete if it is guaranteed to find a goal state provided one exists. Breadth-first search is complete and when applied to infinite graphs it will eventually find the solution. Depth-first search is not complete and may get lost in parts of the graph that do not contain a goal state.

Algorithm 1 Is Current State Solvable

```
procedure ISSOLVABLE(state)
    puzzleLength ← state.size()
    gridWidth ← √puzzleLength
    blankRowEven ← true
    for i ← 1, puzzleLength do
        if state[i] = 0 then
            blankRowEven ← (i/gridWidth) mod 2 = 0
            continue
            for j ← i + 1, puzzleLength do
                if state[i] > state[j] and state[j] ≠ 0 then
                    parity ←!parity
    if gridWidth mod 2 = 0 and blankRowEven then
        return !parity
    return parity
```

5.2 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 space requirement of breadth-first search is its largest deficiency. The 8-tile has a search space of $9!/2 = 181,400$ states 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 $16!/2 = 10,461,394,944,000$ states and a maximum number of 80 moves to solve.

BFS begins by declaring an empty set s which stores all the states that have been

expanded. A queue q is then initialised which stores states to be expanded. The initial state is added to both q and s, q is then repeatedly looped through until a solution is found or q is empty meaning there are no possible solutions. The $currentState$ is dequeued from q, checked against the goal state and returned if a match. Otherwise, each possible move from $currentState$ is found and added to q to be expanded see Algorithm 2.

Algorithm 2 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 is not empty 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)
```

5.3 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 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.

DFS begins by declaring an empty set *visted* which stores all the states that stores expanded states. The state is checked against the goal and returned if a match. All possible moves from the current state are looped through and recursively call the function to repeat the process see Algorithm 3.

Algorithm 3 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)
```

Depth-first search is not the preferred algorithm for solving the sliding-tile 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.

5.4 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; searching all possibilities up to the specific depth and if it doesn't

find the goal state it increases the depth increases. DFID performs depth-first search to depth one, then starts over and executes a depth-first search to depth two and 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).

DFID begins by looping from 0 to the maximum depth of the problem domain. A function is then called with the current state and the specified depth which returns $found$ if the goal state has been discovered or *null* otherwise. The function depth-limited depth-first search *DLS* is called, the state is checked against the goal state provided the depth is 0 as the previous depths have already been checked. If the depth is greater than zero each possible move is found and used in a recursive call of *DLS* with a decremented depth see Algorithm 4.

Algorithm 4 Depth-First Iterative Deepening

```
1:  procedure DFID(state)
2:      for depth ← 0, ∞ do
3:          found ← DLS(state, depth)
4:          if found ≠ 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 ← DLS(neighbour, depth − 1)
12:             if found ≠ 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 1: 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

5.5 Evaluation Of Brute-Force Searches

Depth-first search often induces the possibility of traversing down the left-most path forever, it is also not guaranteed to find the solution and the likelihood that the optimal solution is found is low.

Breadth-first search is very 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.

6 Heuristic Search

A heuristic is a function that ranks possible moves at each branching step to decide which branch to follow. The goal of a heuristic is to produce a fast estimation of the cost from the current state to the desired state, the closer the estimation is to the actual cost the more accurate the heuristic function. In the context of the sliding-tile puzzle, to find best move from a set configuration the heuristic function is executed on each of the child states, the child state with the smallest heuristic value is chosen.

6.1 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 $0 \leq h(n) \leq h^*(n)$; admissible heuristic functions are optimistic meaning they are lower bounds on the actual cost.

6.2 Consistent Heuristics

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 $h(G) = 0$ (Figure 2).

6.3 Heuristic Evaluation Functions

6.3.1 Manhattan Distance

Manhattan distance is the classic heuristic function for the sliding-tile puzzles, for each tile in the puzzle the Manhattan distance counts the number of grid units between its current location and its goal location and summing these values for all tiles. Manhattan

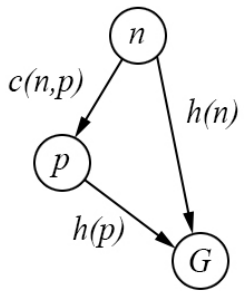


Figure 2: Consistent Heuristics Diagram

distance is a lower bound on actual solution length, because every tile must move at least its Manhattan distance, and each move only moves one tile. 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).

6.3.2 Linear Conflict Heuristic

Linear Conflict heuristic is an improvement to the Manhattan distance, it applies when two tiles are positioned in their desired row or column but are reversed relative to their goal positions, meaning a given tile must move out of the row or 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 or column and then back into its desired position. These two extra moves are not included in the Manhattan distance so can be added to the total without violating admissibility.

Figure 2 displays a visual example the linear conflict heuristic. The initial state shows the first row to be reversed meaning values 1,2 and 3 are positioned 3,2 and 1. The *max* (maximum value in the row or column) 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 meaning a linear conflict thus 2 is added to the total, value 1 is also not larger than 3 so another two is added to the

Algorithm 5 Manhattan Distance

```
1:  procedure MANDIST(state)                    ▷ The current puzzle configuration
2:      total ← 0
3:      puzzleLength ← state.size()
4:      dimensions ← √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
```

total heuristic value.

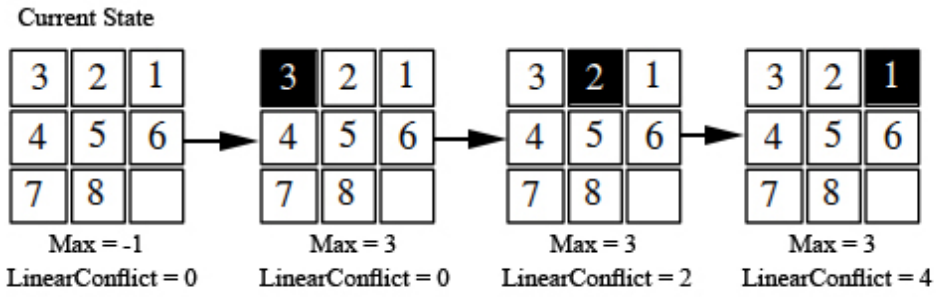


Figure 3: Calculating Linear Conflict

6.4 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 a heuristic function. 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.

A* begins by declaring two empty sets: *closedSet* stores states that have already been visited; *openSet* contains states to be visited. The initial state is added to the *openSet*, similar to BFS the *openSet* is iterated through until it finds a goal state or there are no more states to explore, implying no solutions. The most promising state is retrieved from the *openSet* (the state with the lowest F score) and assigned to *currentState*. This state is then removed from the *openSet* and added to the *closedSet*. The *currentState* is checked to see whether it matches the goal state and returned if so. From the *currentState* all the possible moves are found and denoted as *neighbours*, each *neighbour* is checked against the *closedSet* to make sure the state hasn't already been expanded. The *neighbour* is then checked against the *openSet* and if the *openSet* contains *neighbour* the G values are compared to check which version of the state has the best route from the starting position see Algorithm 6.

6.4.1 Manhattan Distance and Linear Conflict

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 6 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) ← g(state) + h(state)
4:  neighbours(state)   ▷ Expands possible moves from current state ordered by g + h
5:  procedure ASTAR(startState)
6:      closedSet ← empty set
7:      openSet ← empty set
8:      openSet.add(startState)
9:      while openSet is not empty do
10:         currentState ← state in openSet with lowest f value
11:         openSet.remove(currentState)
12:         closedSet.add(currentState)
13:         if currentState = goal then
14:             return currentState
15:         for neighbour in neighbours(currentState) do
16:             if neighbour not in closedSet then
17:                 if neighbour in openSet then
18:                     if g(currentState) < g(neighbour) then
19:                         g(neighbour) ← g(currentState)
20:                 else
21:                     openSet.add(neighbour)
```

Table 2: Manhattan Distance Vs. Linear Conflict - A*

	Manhattan Distance			Linear Conflict		
Min Moves	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.

7 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 had 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 3: 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.

7.1 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 plus 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 which have a lower 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 with lower memory requirements.

The IDAStar algorithm is very similar to to DFID; IDAStar iteratively deepens dependent on the F value rather than a fixed depth. The initial *bound* is assigned to the F score of state, which is also just the H value as G is inherently 0 as no moves have been made to get to this state. A depth-first limited search is then executed to the specified threshold. If the F value of the current state is greater than the *bound* this is returned and assigned to the new *bound*. If the H estimate of the state is 0, no more moves are required to get to the goal state thus the solution is found. Similar to A* the neighbour with the smallest F value is prioritised see Algorithm 7

Table 4: Iterative Deepening A* Results

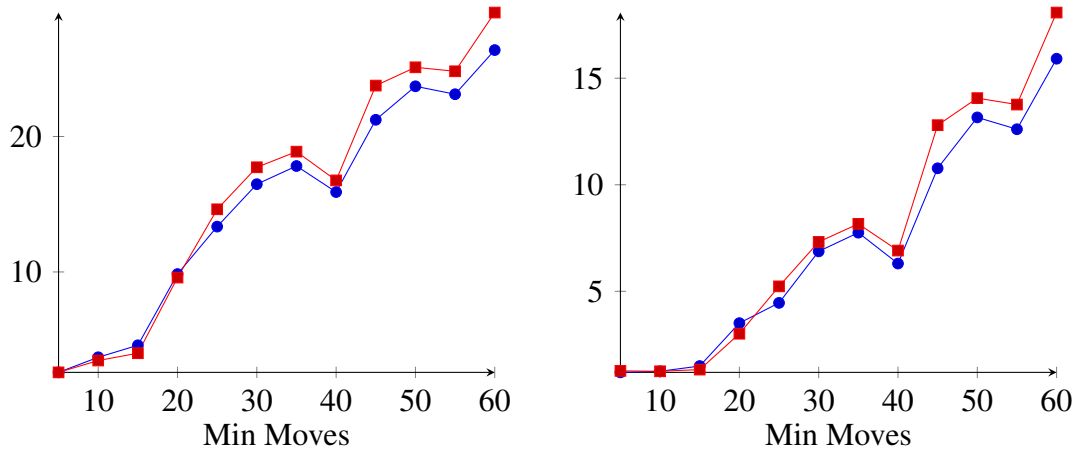
Algorithm 7 Iterative Deepening A Star

```
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 found do                  ▷ Loops until a solution is found
9:         bound ← DLS(state, bound)       ▷ Performs a bounded depth-first search

10: procedure DLS(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 found
15:     min ← ∞
16:     for neighbour in neighbours(state) do
17:         temp ← DLS(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 its 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.

8 Pattern Databases

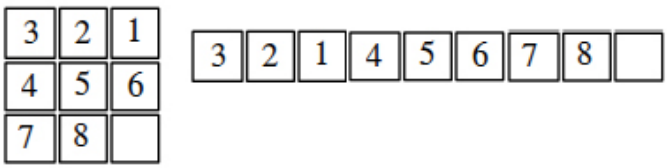


Figure 5: Puzzle Parity

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.

8.1 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 heuristic is estimated dependent on the current positions of the

fringe tiles and the blank, the remaining tiles are disregarded. These values are pre-computed 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.

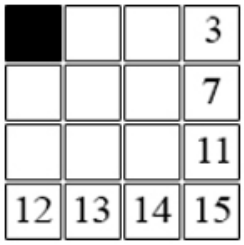


Figure 6: Fringe Database Pattern For Fifteen Tile

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.

8.1.1 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.

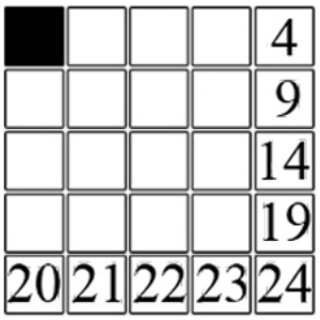


Figure 7: Fringe Database Pattern For The Twenty-five Tile

8.2 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.

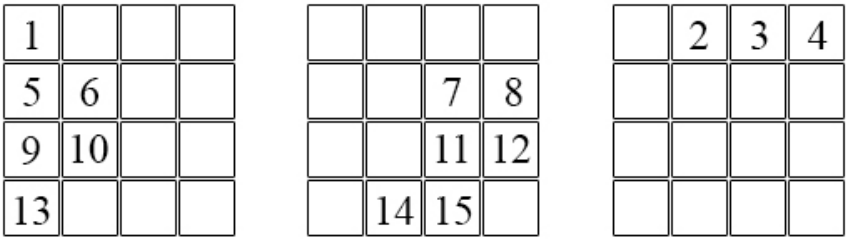


Figure 8: 6-6-3 Partioning

9 Program Architecture

9.1 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.

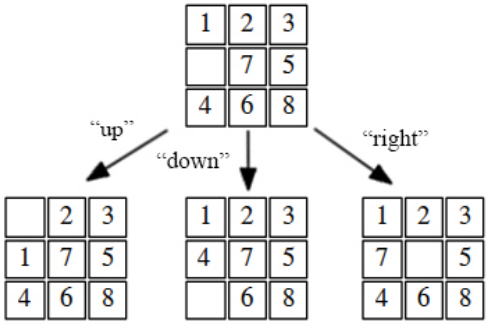


Figure 9: Finding possible moves

10 Results

10.1 BFS

Table 1 shows the results of the 8-puzzle using breadth-first search. The tests were performed on all $9!/2 = 181440$ states, an initial BFS was executed from the goal configuration and each state along with the number of moves required was stored. States were split up into testing lists dependent on the number of moves as shown by column 1 and 2. Tests were then executed using breadth-first search on each state, column 3 displays the average number of nodes expanded before finding the goal state, column 4 is the average time taken to solve a given state and column 5 shows the total time taken to solve all states in the specific testing list.

Table 5: Breadth-First Search Analysis: 8-puzzle

Min Moves	No. States	Avg Nodes Expanded	Avg. Time(ms)	Total Time(ms)
0-4	31	747.19±435.76	0.5483	18
5-9	389	4249.35±4277.99	1.0668	432.0
10-14	4347	23257.75±20745.99	5.7888	25164
15-19	33042	58446.06±35484.49	17.7823	587563
20-24	102326	114724.84±36410.64	41.0058	4195959
25-29	41082	162772.14±15654.31	60.0550	2467181
30+	223	180524.70±548.90	67.3542	15006

Table 6: Breadth-First Search Analysis: 8-puzzle (10000 Random States)

Average Moves	Avg Nodes Expanded	Avg. Time(ms)
21.7142±0.065	89201.4297	29.7373

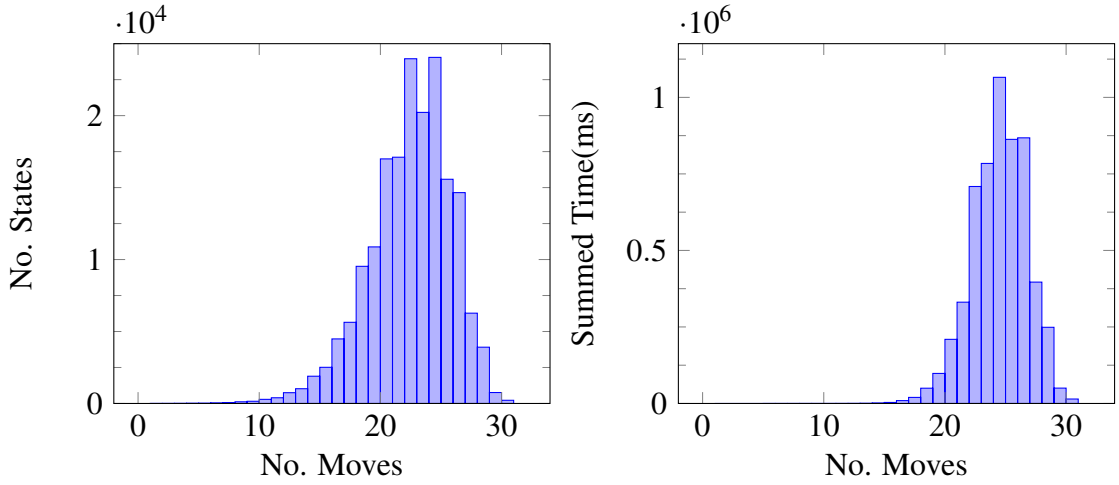


Figure 10: A representation of the number of states that require a specific amount of moves to be solved optimally in the 8-tile and the summed time taken for all these states to be solved

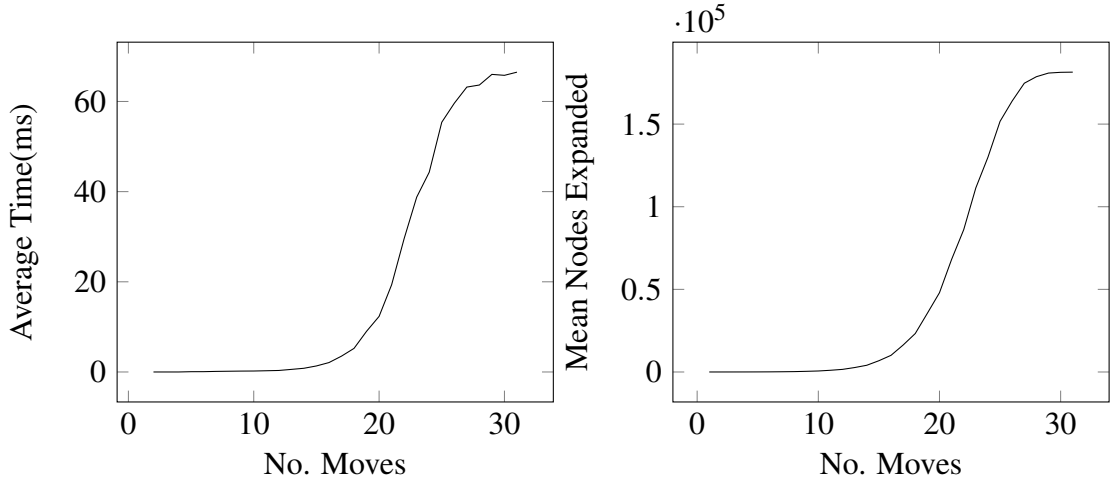


Figure 11: 8-tile analysis using breadth-first search to display how the minimum number of moves required to reach the goal state affects the time(ms) and the number of nodes expanded

References

Borrajo, D., Felner, A., Korf, R. E., Likhachev, M., Linares López, C., Ruml, W., and Sturtevant, N. R. (2014). The fifth annual symposium on combinatorial search. *AI Commun.*, 27(4):327–328.

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. and Schaeffer, J. (1996). Searching with pattern databases. *Advances in Artifical Intelligence*, pages 402–416.

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

Felner, A. (2015). Early work on optimization-based heuristics for the sliding tile puzzle. In *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*.

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.

Hansson, O., Meyer, A. E., and Yung, M. M. (1985). *Generating Admissible Heuristics by Criticizing Solutions to Relaxed Models*. Department of Computer Science, Columbia University.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109.

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.