COMP3411 - 24T2 - Assignment 2

Mohammad Mayaz Rakib (z5361151)

April 8, 2024

Question 1

(a)

The following is a table, as specified by the task, comparing the efficiency of various search algorithms on various starting states of the 15-puzzle.

15-Puzzle Search Strategies				
Starting State	**BFS**	**IDS**	**Greedy**	**A***
Start 1 (2634-5178-AB0C-9DEF)	Expanded: 10978 Length: 12	Expanded: 25121 Length: 12	Expanded: 59182 Length: 12	Expanded: 30 Length: 12
Start 2 (1034-728B-5D6A-E9FC)	Expanded: 344890 Length: 17	Expanded: 349380 Length: 17	Expanded: 19 Length: 17	Expanded: 35 Length: 17
Start 3 (5247-61C0-9A83-DEBF)	Expanded: 641252 Length: 18	Expanded: 1209934 Length: 18	Expanded: 59196 Length: 22	Expanded: 133 Length: 18

(b)

The above experiment explores the efficiency of various search algorithms on three starting states of the 15-puzzle, with the first state being the simplest to solve (length 12 solution), the second state being moderately difficult (length 17 solution), and the third state being the most difficult (length 18 solution). We thus use the length of generated solution to crudely measure the correctness of the algorithm, or how accurate the output is to the actual value. We measure efficiency crudely via the number of nodes expanded, which is the subset of the search space the algorithm evaluates during the searching process. It is arguable that, given a search space, an algorithm that has to evaluate a lower amount of nodes in the search space in order to generate a solution is thus more efficient in traversing that search space for the given search problem.

Firstly, breadth-first-search (BFS) is a search algorithm that searches a tree-like data structure by exploring all nodes at each depth level before exploring further. It is a complete algorithm, since it explores all possible options, but is quite slow and memory-intensive. This is observable since the number of nodes expanded are high comparative to the other search algorithms, with the exception of IDS, for all three starting states.

Secondly, iterative deep search (IDS) is a search algorithm that incorporates both breadth and depth first search, performing depth-limited searches iteratively with increasing depth limits. It is a complete algorithm, but much like BFS, is also rather slow and memory-intensive. Although this algorithm solves the issue of exploring nodes too deep before finding a solution, it may still suffer from redundant node expansions due to repeatedly exploring nodes at shallower levels. Much like BFS, this algorithm's inefficiency is identifiable with its expanded nodes counts, scoring similar to that of BFS. In starting state 3, the expanded nodes count is much higher than BFS, exemplifying how the redundant node expansions associated with this algorithm may actually slow rather than accelerate the search for certain problem instances.

Thirdly, greedy search is a search algorithm that explores a node if it seems to be the most promising in the moment. This makes greedy search a heuristic search algorithm, as it is not a complete algorithm since it does not guarantee correct results. However, due its heuristic nature, it is fast and memory-efficient, and definitely faster and more memory-efficient than the more exhaustive BFS or IDS algorithms. Due to its heuristic nature, the greedy algorithm's efficiency depends highly on the starting state of the problem. In this case, for starting state 1, it performed worse than BFS and IDS in terms of nodes expanded. For starting state 3, it performed better than BFS and IDS, although not getting the correct answer given the higher path length, thus indicating the algorithm's lack of correctness. It is surprising that for starting state 2, greedy search outperformed even the highly efficient A* algorithm in terms of expanded nodes, exemplifying how truly random and instance-dependent the efficiency of the algorithm is.

Fourthly, A* is a search algorithm that incorporates both BFS and greedy search techniques, creating an algorithm that is both correct and fast. It may not be very memory-efficient, but that is not a problem for most modern-day computers with large random-access-memory reserves. It has hundreds, if not thousands, of times lower expanded nodes usage than any other algorithm, and always gets the minimal path length for all three starting states. It is without a doubt the most efficient algorithm out of the four.

Question 2

(a)

'Heuristic Path Search' (HPS) is a search algorithm that utilises the following objective function,

$$f_w(n) = (2-w)g(n) + wh(n)$$

where $0 \leq w \leq 2$, $g(n)$ is the actual cost from the initial state to node n, and $h(n)$ is the heuristic cost from node n to the goal state. It is assumed that the search algorithm is equivalent to a uniform cost search when $w = 0$, an A* search when $w = 1$, or a greedy search when $w = 2$. It is also assumed that the algorithm is complete for all three values of w. We aim to show that the heuristic path search algorithm is optimal when $0 \leq w \leq 1$.

First, let us analyse the objective function definitions at both $w = 0$ and $w = 1$. When $w = 0$, the objective function is,

$$f_0(n) = 2g(n)$$

which eliminates the heuristic cost altogether, only doubling the actual cost. When $w = 1$, the objective function is,

$$f_1(n) = g(n) + h(n)$$

which is simply the sum of both the actual cost and heuristic cost. Note that $f_1(n)$ is the exact same objective function as the A* algorithm. We can define a function $h'(n)$, where, if $w = 0$, then $h'(n) = g(n)$ since there is no heuristic cost and only an extra actual cost in $f_0(n)$, and if $w = 1$, then $h'(n) = h(n)$ since the heuristic cost is considered as is in $f_1(n)$. Thus, the range of $h'(n)$ is $[g(n), h(n)]$ for all n, which implies that $h'(n)$ will always be smaller or equal to $h(n)$, or $h'(n) \leq h(n)$ for all n.

Given the definition of the function $h'(n)$, we can rewrite the original heuristic function as,

$$f'_w(n) = g(n) + h'(n)$$

where if $w = 0$, then $f'_0(n) = g(n) + g(n) = 2g(n)$, and if $w = 1$, then $f'_1(n) = g(n) + h(n)$, both of which are equivalent to $f_0(n)$ and $f_1(n)$ respectively. Since both $f_w(n)$ and $f'_w(n)$ are equivalent for $0 \leq w \leq 1$, then minimising $f_w(n)$ is equivalent to minimising $f'_w(n)$. Thus, proving the optimality of $f'_w(n)$ also proves the optimality of $f_w(n)$.

When $w = 0$, $f'_w(n) = 2g(n)$, which only considers the actual cost without considering the heuristic cost. Since a uniform cost search algorithm only considers the actual cost when making a decision, the objective function $f'_w(n)$ acts as an objective function for a uniform cost search, thus making heuristic path search equivalent to a uniform cost search. Since we know that uniform cost search algorithms are optimal, this makes our heuristic path search optimal for $w = 0$.

When $w = 1$, $f'_w(n) = g(n)+h(n)$, which considers the actual and heuristic costs as is. Since an A* search algorithm has the exact same objective function, this makes heuristic path search equivalent to an A* search. Not only this, but since A* search algorithms are always optimal when the heuristic cost function $h(n)$ is admissible, then we know, since it is assumed $h(n)$ is admissible here, that the heuristic path search algorithm is optimal just as an A* search algorithm would be optimal in this circumstance. This makes our heuristic path search optimal for $w = 1$.

Thus, $f'_w(n)$ is optimal for all w where $0 \leq w \leq 1$. This implies $f_w(n)$ is optimal for all $0 \leq w \leq 1$. Thus, our heuristic path search algorithm is optimal for all w where $0 \leq w \leq 1$.

(b)

The following is a table, as specified by the task, comparing the efficiency of multiple varieties of the 'Heuristic Path Search' (HPS) algorithm, including the special case where HPS is equivalent to IDA* search.

Heuristic Path Search Strategies			
Strategy	**Start 4**	**Start 5**	**Start 6**
IDA* Search	Expanded: 545120 Length: 45	Expanded: 4178819 Length: 50	Expanded: 169367641 Length: 56
HPS, w = 1.1	Expanded: 523052 Length: 47	Expanded: 857115 Length: 54	Expanded: 13770561 Length: 58
HPS, w = 1.2	Expanded: 29761 Length: 47	Expanded: 64522 Length: 56	Expanded: 265672 Length: 60
HPS, w = 1.3	Expanded: 968 Length: 55	Expanded: 5781 Length: 62	Expanded: 9066 Length: 68
HPS, w = 1.4	Expanded: 9876 Length: 65	Expanded: 561430 Length: 70	Expanded: 37869 Length: 80

(c)

The above experiment measures the efficiency of the 'Heuristic Path Search' (HPS) algorithm on various starting states of the 15-puzzle. Once again, we use the length of the generated solution to crudely measure the correctness of the algorithm's output. We also use the number of expanded nodes, representing the subset of the search space the algorithm evaluates during the searching process, to crudely measure the algorithm's efficiency. Once again, starting positions 4, 5 and 6 vary in difficulty to complete/complexity of solution, from easiest, somewhere in between, and hardest, respectively.

We know, from analysing the HPS objective function, that the algorithm is equivalent to an A* search at $w = 1$ and a greedy search at $w = 2$. Thus, any values between $1 \leq w \leq 2$ are somewhere in between that of an A* search and a greedy search. We use the statistical results of the IDA* search algorithm, which is otherwise equivalent to the HPS algorithm at $w = 1$, as a baseline to analyse the correctness and efficiency of the HPS algorithm at other w values.

A valid assumption would be that, given the completeness of the IDA* search algorithm, as the HPS algorithm's w value moves away from A* search towards greedy search, the accuracy of generated outputs would decrease. Another valid assumption is that, since greedy search is a heuristic that does Not guarantee completeness, there is a possibility that its efficiency will increase as the HPS algorithm's w value moves towards greedy search. Both of these assumptions can be observed in the statistical data in the table.

Firstly, for all starting positions, as the values go from a perfect A* search to HPS with higher values of w, the length of the solutions starts getting larger. The more correct the solution, the lower its length should be, since solutions with higher length (and thus cost) are not as optimal as solutions with lower length (and thus cost). The increasing length values indicate a decrease in the correctness of the generated solutions. For starting positon 4, the length values go from 45, to 47 and 55, all the way to 65, ultimately a much longer solution than the initial 45. Same with starting positions 5 and 6, going from 50 to 70, and 56 to 80, respectively.

Secondly, for all starting positions, number of expanded nodes fluctuates as the HPS algorithm attains higher values of w. It decreases as the algorithm goes from the perfect A* statistics, to $w = 1.1$, to $w = 1.2$, reaching a minimum at $w = 1.3.3$. It seems that the fastest execution of all algorithms was the HPS algorithm at $w = 1.3$, producing extraordinarily low numbers of expanded nodes for all starting positions. This indicates the variability in greedy search efficiency, and how sometimes greedy search, even while producing a less correct output, can be much faster than algorithms such as A* search. This variability is even more strongly illustrated when the jump from $w = 1.3$ to $w = 1.4$, indicating an even more heuristic and greedy search (closer to perfect greedy search at $w = 2$) saw a sudden drop in efficiency, with number of expanded nodes jumping back up. There also seems to be much variability among the different starting positions for HPS at $w = 1.4$, with expanded nodes being as low as 9876 for starting position 4, to as high as 561430 for starting position 5. This inter-column variability in the table results illustrate how the efficiency of greedy search is highly dependent on the initial states of its inputs.

Question 3

(a)

Let $M(n, 0)$ be the minimum number of discrete time steps required to travel from 0 to n with no initial velocity, or in other words, $k = 0$. The following is a list of all optimal sequences of actions and their associated $M(n, 0)$ values, where $1 \leq n \leq 21$.

$n = 1$	$[+1, -1]$	$M(1, 0) = 2$
$n = 2$	$[+1, 0, -1]$	$M(2, 0) = 3$
$n = 3$	$[+1, 0, 0, -1]$	$M(3, 0) = 4$
$n = 4$	$[+1, +1, -1, -1]$	$M(4, 0) = 4$
$n = 5$	$[+1, +1, -1, 0, -1]$	$M(5, 0) = 5$
$n = 6$	$[+1, +1, 0, -1, -1]$	$M(6, 0) = 5$
$n = 7$	$[+1, +1, 0, -1, 0, -1]$	$M(7, 0) = 6$
$n = 8$	$[+1, +1, 0, 0, -1, -1]$	$M(8, 0) = 6$
$n = 9$	$[+1, +1, +1, -1, -1, -1]$	$M(9, 0) = 6$
$n = 10$	$[+1, +1, +1, -1, -1, 0, -1]$	$M(10, 0) = 7$
$n = 11$	$[+1, +1, +1, -1, 0, -1, -1]$	$M(11, 0) = 7$
$n = 12$	$[+1, +1, +1, 0, -1, -1, -1]$	$M(12, 0) = 7$
$n = 13$	$[+1, +1, +1, 0, -1, -1, 0, -1]$	$M(13, 0) = 8$
$n = 14$	$[+1, +1, +1, 0, -1, 0, -1, -1]$	$M(14, 0) = 8$
$n = 15$	$[+1, +1, +1, 0, 0, -1, -1, -1]$	$M(15, 0) = 8$
$n = 16$	$[+1, +1, +1, +1, -1, -1, -1, -1]$	$M(16, 0) = 8$
$n = 17$	$[+1, +1, +1, +1, -1, -1, -1, 0, -1]$	$M(17, 0) = 9$
$n = 18$	$[+1, +1, +1, +1, -1, -1, 0, -1, -1]$	$M(18, 0) = 9$
$n = 19$	$[+1, +1, +1, +1, -1, 0, -1, -1, -1]$	$M(19, 0) = 9$
$n = 20$	$[+1, +1, +1, +1, 0, -1, -1, -1, -1]$	$M(20, 0) = 9$
$n = 21$	$[+1, +1, +1, +1, 0, -1, -1, -1, 0, -1]$	$M(21, 0) = 10$

(b)

We must prove $M(n, 0) = \lceil 2\sqrt{n} \rceil$. To do this, we refer to observations made in the subsection (a).

One of the observations we make is that when s is a perfect square s^2, the number of discrete time steps required is $2s$. This is because, for a perfect square, the agent must accelerate and decelerate by 1 an equal number of times. Examples of this include $n = 4$, $n = 9$ and $n = 16$.

Another observation we make is that when there are two consecutive perfect squares s^2 and $(s + 1)^2$, the number of time steps required is $2s$ and $2s + 2$ respectively. In between two consecutive perfect squares, the number is $2s + 1$

if it comes before $2s + s$, but is $2s + 2$ after it. An example of this is $n = 4$, where $s = 2$, and that the minimum discrete time steps jumps from 5 to 6 at from $n = 7$ onwards, where the previous value $n = 6$ is just $n = 2^2 + 2$.

Thus, we will consider the pattern relative to each perfect square. We can refer to each number as its position j after a given perfect square s, such that each number is indexed as $s^2 + j$. Not only that, but we can go one step further, indexing before or after the point at which the minimum discrete time steps changes, which is at every $s^2 + s$ and is thus indexed as $s^2 + s + j$.

Since s is defined to be a perfect square, we can write s as $\sqrt{n}$. We then redefine everything in terms of $\sqrt{n}$. We state that, for every indexed position j after every perfect square s^2 such that $n = s^2 + j$, the value is $2\sqrt{n} + 1$, but after $n = s^2 + s + j$, it becomes $2\sqrt{n} + 2$ until the next consecutive perfect square $n = (s+1)^2$. From observation, we see that this relationship can be simplified to $\lceil 2s \rceil$, where it is $2s+2$ for values larger than s^2+s and $2s+1$ for values smaller than $s^2 + s$, both being clamped to the ceiling of the $2s$. Since the relationship can be simplified to $\lceil 2s \rceil$, we deduce it can also be simplified to $\lceil 2\sqrt{n} \rceil$.

(c)

We must prove that $M(n, k) = \lceil 2\sqrt{n + \frac{1}{2}k(k + 1)} \rceil - k$. We do this by considering the path of the agent as part of a larger path.

If $k = 0$, then our agent has an initial velocity of 0, and is thus stopped. This means we require an equal amount of acceleration and deceleration moves in order to slow it to a stop by the end. However, if $k > 0$, then our agent has an initial velocity that is more than 0, and is thus not stopped. If our agent is already moving at the start, it means that we will need more deceleration moves than acceleration moves in order to slow it to a stop by the end.

If our agent is already moving at the starting positon, we can imagine a path where the agent started moving before reaching the starting position. If the agent is moving at $k = 1$, it means it has a velocity of 1, meaning that one could imagine it moving from a stop from 1 position before the start. If the agent is moving at $k = 2$, it means it has a velocity of 2, meaning that one could imagine it moving from a stop from 3 positions before the start. Thus, the path of an agent already moving at it starting position is a subset of the path of an agent that started in a stopped position sometime before the start.

For this reason, we add to the path distance n an additional number of positions that maybe required to accelerate to the given initial velocity, where $k = 1$ would require 1 extra position, $k = 2$ would require 3 extra positions, and so forth. This acceleration can be represented by the number sequence $1+2+3+...+k$, which we know from mathematics is $\frac{1}{2}k(k + 1)$. Thus, we can rewrite the minimum discrete time steps equation from (b) as $M(n, k) = 2\sqrt{n + \frac{1}{2}k(k + 1)} \rceil$.

Note that this is only possible if $n > \frac{1}{2}k(k + 1)$. This is because, if n were any less than that amount, there wouldn't be enough imaginary space for it to have accelerated to its initial velocity in the first place, thus making the imaginary path extension method invalid.

Since the additional path extension is imaginary, we must remove any additions to our minimum discrete time steps count. Thus, after writing the formula in terms of a larger path, we then remove any such additions by subtracting the initial velocity from the time steps count. This is because acceleration can only occur 1 velocity value at a time, and each velocity value requires it own time step, totalling to k time steps, meaning each time step removed also totals to k time steps. Thus, we deduce our final expression as $M(n,k) = \lceil 2\sqrt{n + \frac{1}{2}k(k+1)} \rceil - k$.

(d)

Unfortunately, I am unable to come up with an answer to this question.

(e)

The admissible heuristic for the original two-dimensional game would require one-dimensional calculations defined above to be performed on each axis in the plane.

The initial velocities for the row and column directions would be $k = u$ and $k = v$ respectively. The distances to the goal points are calculated as distances to the coordinate of the goal in either the row or column axis as $r_G - r$ and $c_G - c$ respectively. Thus, $M(r_G - r, u)$ and $M(r_C - c, v)$ gives us the minimum discrete time steps required to traverse each plane axis.

Thus, we can deduce the admissible heuristic to be,

$$h(r, c, u, v, r_G, c_G) = max(M(r_G - r, u), M(r_C - c, v))$$

Question 4

(a)

The best way to maximise the number of nodes pruned by a minimax algorithm with alpha-beta pruning where each node has two children is to halve the tree every time, similar to a binary search, and minimising any backtracking. Generally, for any tree where each node has n children, we would divide the search space by $\frac{1}{n}$, giving us a $O(log_n(m))$ time complexity, where m is the height of the tree.

To do this, we must ensure that at each branch of the tree, all the leaves in the left subtree are preferable to all the leaves in the right subtree. If we had values $[0, n]$, then obviously, these preferable values would be the larger half

$[\,]\lceil\frac{n}{2}\rceil, n]$. This applies for every step of the tree, such that the preferable values are $[\lceil\frac{n}{4}\rceil, n]$, $[\lceil\frac{n}{8}\rceil, n]$, and so forth.

Trivially, one might believe it is correct that the leaf nodes of the game tree, from left to right, is the the range $[0, 15]$ in reverse order, or in other words, represented by the set $\{15, 14, ..., 1, 0\}$. However, because of the alternating nature of a minimax tree, the order itself has to be manipulated such that, for a MIN node, the lowest value is to the left, and for a MAX node, the highest value is to the left.

Lets start from the bottom left of the tree. The node above it is a MIN node, meaning that, of the two highest values 14 and 15, the correct order is $\{14, 15\}$ rather than $\{15, 14\}$ since it is more optimal for the 14 to be chosen and found to be the minimum. The same with the two leaf nodes to its right, $\{12, 13\}$ rather than $\{13, 12\}$, and then the same with two leaf nodes to its right, $\{10, 11\}$ rather than $\{11, 10\}$, and so forth. If we start going a level up to the MAX nodes, each MAX node once gain picks the left-most value, such as 14 over 12, 10 over 8, and so forth. This goes on, alternatingly, for all levels until the root MAX node at the top level.

Thus, the optimal sequence of leaf node values are the range $[0, 15]$ in reverse order but with every pair as (x_{i+1}, x_i) swapped as (x_i, x_{i+1}) where $i \in [0, 15]$ and i is even. This is written as,

$$\{14, 15, 12, 13, 10, 11, 8, 9, 6, 7, 4, 5, 2, 3, 0, 1\}$$

(b)

It is a bit difficult to perform a visual trace using LaTeX, but I will try to describe it as best as possible in words.

For the sake of clarity, we will number each lowest-level subtree, consisting of its leaf node value pair (x_i, x_{i+1}) and its corresponding root MIN node, as subtree i from left-to-right, such that the $\{14, 15\}$ subtree is subtree 1, the $\{12, 13\}$ subtree is subtree 2, and so forth, all the way to the $\{0, 1\}$ subtree is subtree 8. Any subtrees larger than these subtrees are referring to by a combination of the two constituent subtree numbers conjoined by a '-' symbol, such that the subtree names for the MAX nodes one level higher are subtree 1-2, subtree 3-4, and so forth.

Firstly, for the left-most corner of the tree, we have subtree 1, which has leaf nodes $\{14, 15\}$. Both of these values are searched, as $\alpha = -\infty$ and $\beta = \infty$, which are its starting worst-case values. This finds the initial non-worst-case value for β, which is $\beta = 14$ Next he subtree next to it, subtree 2, including $\{12, 13\}$, isexplored. However, this time, only the left-most side of the subtree is explored and the branch leading to 13 is pruned.

Similarly, subtree 3-4 is explore as well, but only the left-most outer edge of the subtree once again. Finally, only the left-most outer edge subtree 5-6 is explored.

Thus, the leaf node values that are evaluated by the search, in order, are as follows,

$$\{14, 15, 12, 10, 6\}$$

(c)

We can generalise the pattern discovered in part (b) to a game tree where each node has three children instead of two.We see a pattern in the subset of the leaf node values that are evaluated in a game tree with two children per node, and thus extrapolate it to a game tree with three children per node.

Essentially, from left-to-right, if we index the leaf node values with $i \in [1, n]$, then the leaf node values that are evaluated are $i = 1$, $i = 2$, $i = 3$, $i = 5$, $i = 8$, and so forth. These are the Fibonacci numbers, where $n_i = n_{i-1} + n_{i-2}$.

Unfortunately, I can not seem to figure out what the generalisation is for trees where each node has three children. My guess is that it would some variation of the Fibonacci sequence above.

(d)

The time complexity of the alpha-beta pruning, in the worst case, is $O(b^d)$, where b is the branching factor, or the number of children per node, and d is the depth of the tree.

However, if we choose the best move first, then we effectively halve the search space at each level, or in other words, explore a subtree that is essentially $\frac{d}{2}$ in depth.

Thus, the time complexity of the minimax algorithm with alpha-beta pruning, when the best move is picked first, is $O(b^{d/2})$.