COMP3411: Assignment 1
Gabriela Roznawska
zID: 5662538
Term 1, 2025

PART A

Question 1: Path Search

1a. Draw up a table

Start State	BFS	IDS	Greedy	A*
start 1	41891	101934	136288	54
start 2	153265	224359	21	76
start 3	1060543	2893651	47719	93

1b. Briefly discuss the efficiency of these four search strategies.

Analysis of the table presenting the number of nodes expanded in the graph under each of the search strategies informs us about the amount of moves performed by each search strategy before getting to the desired answer. Naturally, in computer science we want to create solutions that can solve problems the fastest for the most, if not all, cases.

While each of the strategies yielded an answer, some did significantly better than others. A* Search is undefeated in its efficiency - finding double-digit solutions when the alternatives consider millions. It is clear that A* success is not case-based as the number of expanded nodes remains low in all three starting states. The algorithm's strategy of using both path cost from the start - g(n) - and heuristic - h(n) - allows it to find the optimal solution efficiently.

The repeatability of yielding optimal and efficient solution sets A* Search apart from Greedy search. While Greedy Search slightly outperforms A* in start 2, it does incomperably worse in the other two. That discrepancy in results just goes to show the nature of greedy approach - due to using only heuristic h(n) (distance to goal) it can sometimes be surprisingly optimal but cannot be reliable across different scenarios.

Another algorithm under our consideration is IDS - Iterative Deepening Search. While, on a positive note, we can expect it to return an optimal solution, it underperforms in all three cases in relation to all other search algorithms (other than in one very bad run for

Greedy Search) by having the largest number of expanded nodes. This marks it as the least efficient search method out of those considered.

Based on the expanded nodes, I assume that the start locations are progressively further from the goal, which would naturally explain the increasing number of expanded nodes from start 1 to start 3. While observed across all strategies, it is especially profound for BFS, which seems to be getting significantly less efficient the more nodes there are to consider.

It is interesting to look at these results and consider the internal structure of each of these search algorithms that explain the choices each of them makes that ultimately lead them to some better or worse efficiency.

In summary, I find A* Search to be the consistent best-performer for path search. Greedy Search seems to me as rather a gamble that may be worth a chance in certain cases, but not all. IDS is a rather underwhelming algorithm but I can image scenarios where it can find its purpose - when simplicity of implementation is more important than efficiency. Finally, BFS is a classic search algorithm that should be used with care - it may be worth reconsidering if we expect to have to traverse a large input.

Question 2: Heuristics Path Search for 15-puzzle

2a. Prove that Heuristic Path Search is optimal when $0 \leq w \leq 1$.

We want to prove that: Heuristic path search is optimal when $0 \leqslant w \leqslant 1$

It can be described using the function:

$$f_w(n) = (2-w)\, g(n) + w\, h(n)$$

where $h()$ is an admissable heuristic and w is a number between 0 and 2.

To prove the statement we will that minimizing $f(n) = (2-w)\, g(n) + w h(n)$ is the same as minimizing $f'(n) = g(n) + h'(n)$ for some function $h'(n)$ with a property that $h'(n) \leqslant h(n)$ for all n.

w can take values between 0 and 2. When plugged-in into the function $f_w(n)$, they generate functions of different heuristic search algorithms.

Optimal Cost search: $f_w(n) = (2-0)\, g(n) + 0 \cdot h(n)$
$w = 0$ $\quad f_0(n) = 2\, g(n)$

OCS prioritises expanding nodes that has the lowest cost in relation to starting node. Since it only pursues the path only if it is shorter than any other alternative, it ensures finding an optimal path. $f_w(n)$ is thus optimal for $w = 0$.

A* Search: $f_w(n) = (2-1)\, g(n) + 1 \cdot h(n)$
$w = 1$ $\quad = f_w(n) = g(n) + h(n)$

A* Search calculates its path using both $g(n)$ and $h(n)$. It is an optimal solution as long as we know that $h(n)$ is admissable: the heuristic is not an overestimation of the real distance between a node and goal. Here, we know $h(n)$ is admissable thus A* Search is optimal.

Greedy Search: $f_w(n) = (2-2)\, g(n) + 2 \cdot h(n)$
$w = 2$ $\quad f_w(n) = 2h(n)$

Despite $h(n)$ being admissable, greedy algorithm is not optimal as it can be easily misled even by non-overestimated $h(n)$.

The function $f'(n) = g(n) + h'(n)$ resembles the structure of A* Search as it includes both $g(n)$ and $h(n)$. We know $h(n)$ is optimal and that $h'(n) \leqslant h(n)$ for all n. Since function $h(n)$ is admissable when not overestimated, then any function equal or smaller than $h(n)$ is admissable as well. Thus $h'(n)$ is admissable.

To explore the relationship between function $h(n)$ and value of w, let's establish the $h'(n)$.

$f(n) = (2-w) \cdot g(n) + w \cdot h(n)$

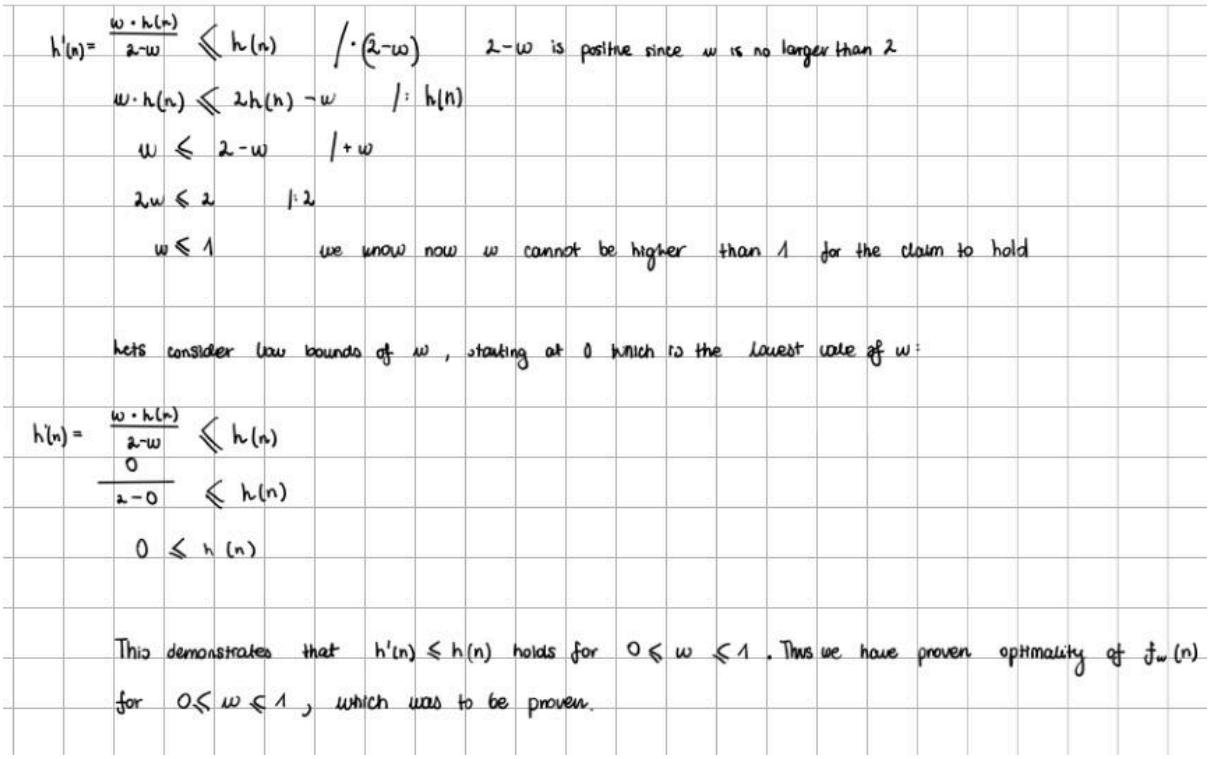
lets factor out $(w-2)$

$f(n) = (2-w) \cdot \left(g(n) + \frac{w}{2-w} \cdot h(n)\right)$

which brings us to $h'(n) = \frac{w}{2-w} h(n)$ and $f(n) = (2-w)\left(g(n) + h'(n)\right)$

Since $h'(n) = \frac{w}{2-w} h(n)$ then $\frac{w \cdot h(n)}{2-w} \leqslant h(n)$

let's verify the values of w for which $h'(n) \leqslant h(n)$ holds:



2b. Draw up a table in this format (the top row has been filled in for you):
Run the code on each of the three start states shown below, using Heuristic Path Search with w = 1.1, 1.2, 1.3 and 1.4 .

	start 4		start 5		start 6	
IDA* Search	48	1606468	52	3534563	54	76653772
HPS, w = 1.1	50	43086	52	92355	56	8695135
HPS, w = 1.2	50	6374	56	85553	58	1929384
HPS, w = 1.3	58	29105	66	74212	64	182673
HPS, w = 1.4	66	33995	70	14640	66	17781

2c. Briefly discuss the tradeoff between speed and quality of solution for Heuristic Path Search with different values of w.

The table includes information about the length of the path that was found as well as the number of nodes that were expanded. It shows the relationship between the quality of solution and the speed of computation with regards to different values of w.

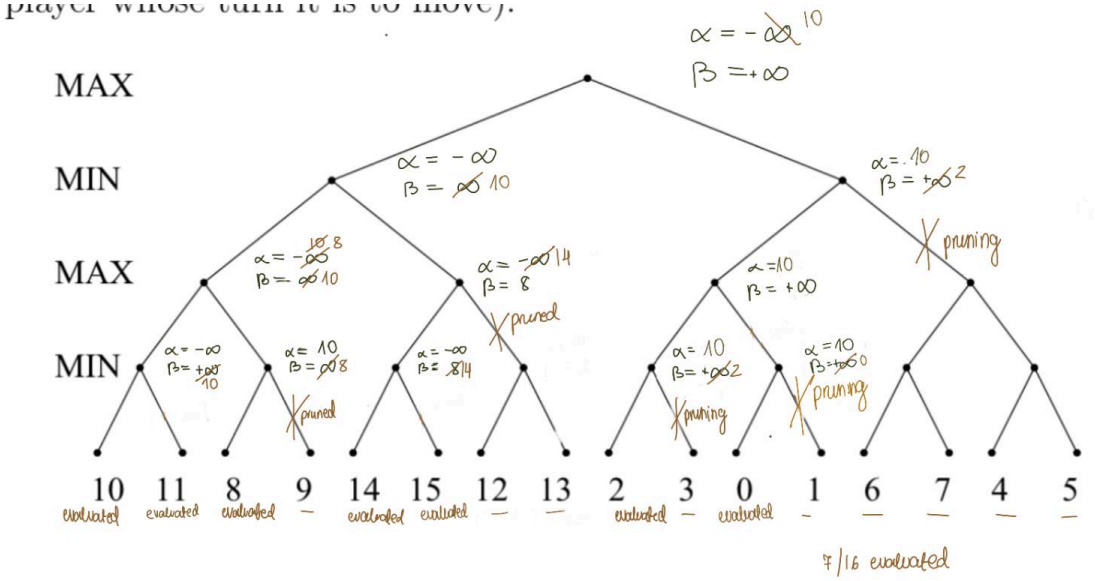
Analyzing the table suggests that in order to obtain a high quality solution - as short of a path as possible - more nodes have to be expanded, decreasing the speed of the algorithm. It is clear that when prioritizing the guaranteed shortest path, the algorithm will have to visit more nodes to evaluate the lowest-cost options - resulting in a higher runtime. On the other hand, an algorithm prioritizing heuristic-based paths is oriented towards expanding towards the goal node. it visits 'just enough' nodes, often missing those that ultimately lead to the shortest path, which does not yield the highest-quality path but is fast. Therefore the shorter the length of the path the higher the quality of search, while the lower the number of expanded nodes, the higher the speed.

It is crucial to notice the change in the length of the path and number of expanded nodes in relation to the value of w. Every decimal point that the value of w increases by, results in a significant drop in number of expanded nodes - increased speed - and increase in length of the path - the compromised quality of solution. This directly visualises how g(n) and h(n) in Heuristic Path Search can be adjusted to yield desired tradeoff between the quality and speed. The discrepancy between the results for different values of w and IDA* is truly impressive: in the start 5 dropping from 3534563 to 14640 between IDA* and w = 1.4. The reactivity of quality and speed to the values of g(n) and h(n) just emphasizes how essential it is to understand the use case of the algorithm in order to adjust it best to the scenario.

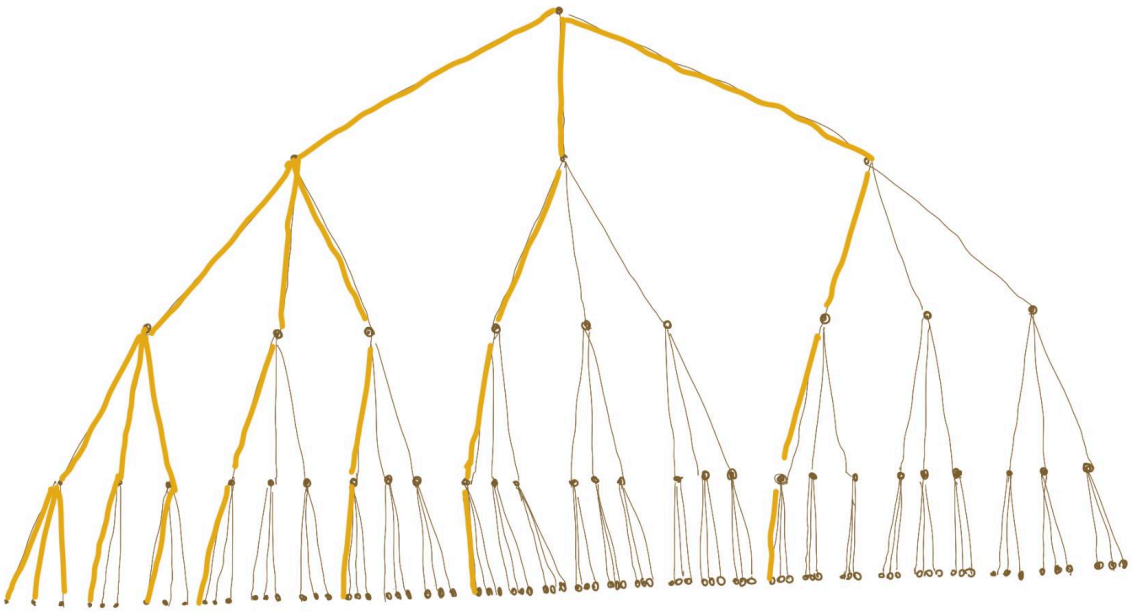
I believe that the decision over whether it is the speed or quality of solution that should be prioritized depends strictly on the problem that needs to be solved. If we are working with an agent that acts in real-time - we may want to ensure that it can provide solutions fast enough, even if they are not as high quality as we wish they were. However, in a stable environment where it is the quality we are oriented towards, we may need to be willing to sacrifice the time it takes for desired computations.

Question 3: Trees and pruning

3a. Trace through the alpha-beta search algorithm on this tree, showing the values of alpha and beta at each node as the algorithm progresses, and clearly indicate which of the original 16 leaves are evaluated



3b. **Now consider another game tree of depth 4, but where each internal node has exactly three children. Assume that the leaves have been assigned in such a way that alpha-beta search prunes as many nodes as possible. Draw the shape of the pruned tree. How many of the original 81 leaves will be evaluated?**



In this scenario 9 out of the original 81 leaves will be evaluated. The image shows the path to the nodes that are evaluated in yellow, while the pruned ones remain in brown. Such a result is achieved thanks to the nodes being arranged in a way facilitating the pruning.

3c.What is the time complexity of alpha-beta search, if the best move is always examined first. Explain why.

The time complexity of Alpha Beta Search is b^(d/2) for a best-case scenario and b^2 for the worst-case scenario, where: b to the power of d over 2. Where b is is the branching factor, and d is the depth. The optimal scenario for the alpha-beta search tree is for the first move to be examined first (thus located as the leftmost leaf). In analysing the time complexity of alpha-beta search it is worth looking at closely related Minimax search. Minimax evaluates all possible moves for some depth d, without pruning, and thus having time complexity of b^d for both best and worst case scenarios. This way it often evaluates nodes that are never actually going to influence the final decision - as they lead to less favorable outcomes than already evaluated alternatives. Alpha-Beta's strength lies in the ability to detect these nodes that do not need to be pursued, and pruning them off, so that no time is wasted on calculating non-relevant information.

The difference between best-case and worst-case scenario runtimes in alpha-beta search is the order in which the nodes are evaluated. If the values at leafs are progressively increasing (node2 = node1 + n; n > 0) when max is making the move (or decreasing for min making the move), then it will never be able to prune the next node, as it will be more favorable than the previous one. Therefore, in such cases alpha-beta search will evaluate every node in the tree before finding the path from the root, resulting in the same runtime as Minmax: b^n. Now, let's consider the contrary - the best case scenario.

In the best scenario, the values at leafs are progressively decreasing (node2 = node1 + n; n > 0) when max is making the move (or increasing for min making the move). In other words, we want the value of alpha to be the highest and value of beta to be the lowest from the start. This way we only need to evaluate one path at every level and can prune the remaining paths at the level. Therefore, for a alpha-beta tree with a branching factor b, we will evaluate:

for even b: O(b x 1 x b x 1 x b x 1 … x 1)
for odd b: O(b x 1 x b x 1 x b x 1 … x b)
(since we need to evaluate one more branching level).

Thus the two scenarios evaluate to: even b^(d/2) and odd: b^((d+1)/2). Thus if the best move is evaluated first at every branch of the tree, we get the best-case complexity.