Question 1: Search Algorithms for the 15-Puzzle

a) Draw up a table with four rows and five columns. Label the rows as UCS, IDS, A∗ and IDA∗, and the columns as start10, start20, start27, start35 and start43. Run each of the following algorithms on each of the 5 start states:

	Start10	Start20	Start27	Start35	Start43
UCS	*G = 10* N = 2565	Mem	Mem	Mem	Mem
IDS	*G = 10* N = 2407	*G = 20* N = 5297410	Time	Time	Time
A*	G = 10 N = 33	G = 20 N = 915	G = 27 N = 1873	Mem	Mem
IDA*	G = 10 N = 29	G = 20 N = 952	G = 27 N = 2237	G = 35 N = 215612	G = 43 N = 2884650

b) Briefly discuss the time and space efficiency of these four algorithms

Uniform Cost Search expands the node whose total cost from the start is the shortest. If the cost to reach the goal state is *c* and each step gets us *e* units closer to the goal, the time complexity is $O(b^{(c/e)+1})$, where *b* is the branching factor. This is because we need to reach a depth of *c/e + 1* to generate the cheapest solution and at every node we expand, we encounter all the nodes at the subsequent level.

All nodes encountered need to be stored, as they either form part of the solution, or part of the pool that we select the next node to expand. So, the space complexity is also $O(b^{(c/e)+1})$. For solutions where the depth of the cheapest solution is large, UCS can run out of memory – as can be seen in the table below, where the memory is exhausted for starting positions 20, 27, 35 and 43.

Iterative Deepening Search has time complexity $O(b^d)$. As IDS does a series of depth-limited searches until a solution is found, the time complexity follows from the fact that if the first solution is found at *d*, the root is expanded *d+1* times, the b nodes at depth 1 are expanded *d* times, the b^2 nodes at depth 2 are expanded *d-1* times and so on. It is clear from the results above that if we have a high branching factor, and depth of the cheapest solution is deep, the time complexity is very poor. This can be seen in the table above, where IDS does not find a solution within 5 minutes for starting positions 27, 35 and 43.

The space complexity of IDS is O(bd) or O(d) depending on whether a queue is used to store unexplored nodes instead of recursion.

A* seeks to minimize the value of f(n) = g(n) + h(n), so the total cost of n from the start and also the estimated cost from n to the goal, at every move. The time complexity in the worst case is $O(b^d)$ where d is the depth of the shortest-path solution that is found. As A* does not generate as many nodes as UCS and does not time out at the same starting positions as UCS, the heuristic must be very good, causing A* search to generate less nodes than UCS at every level.

A* keeps all nodes in memory, so the space complexity is poor - $O(b^d)$, causing it to run out of memory for the final 2 starting positions.

Iterative deepening A* search has worst case time complexity $O(b^d)$ as it is a variant of iterative deepening search. However, as it concentrates on exploring the most promising nodes based on f(n) = g(n) + h(n), it often does not have to go to the same depth everywhere in the search tree. This is likely

the reason why iterative deepening A* search has not run out of time for the last 3 starting positions that iterative deepening search ran out of time for.

The space complexity is only O(d) where d is the depth of the goal that IDA*S has found, as like in depth-first search algorithms, only the nodes that represent the branch of the tree being expanded are kept track of.

Question 2: Deceptive Starting States

a) Print state start49 and calculate its heuristic value. Do the same for start51 and include both results in your solutions

Start state 49 is the below (alternatively, it is represented as S = [4/1,2/2,1/2,1/4,1/3,3/3,3/2,4/4,2/4,3/4,3/1,2/3,2/1,1/1,4/3,4/2]). It has a heuristic value of 25.

Start state 51 is the below (alternatively, it is represented as S = [2/1,3/4,4/2,2/4,4/4,3/1,4/1,1/1,3/3,1/4,1/3,1/2,4/3,2/2,2/3,3/2]). It has a heuristic value of 43.

b) Use [idastar] to search, starting from start51, and report the number of nodes that are expanded.

The number of nodes expanded is 551168 using idastar search starting from start51.

c) When [idastar] is used to search from start49, the number of nodes expanded is 178880187. Briefly explain why the search from start49 expands so many more nodes than the search from start51, even though the true path length is very similar

Even if the difference between the two path lengths is rather insubstantial, this may cause an exponential difference in the number of nodes expanded because Iterative Deepening A* Search re-expands previously expanded nodes with values f(n) = g(n) + h(n) below the current threshold in addition to any new nodes needing to be expanded. Suppose for example, that the cheapest solution for start49 is at some depth ***d,*** that corresponds to the solution found when the IDA* threshold is ***t1***. Now suppose the path length of the solution for start51 is incrementally longer than the path length of the solution for start49 with at least one node with value f(n) = g(n) + h(n) that exceeds the threshold ***t1*** for start49. This means the solution for start51 cannot be found at the depth that corresponds to the IDA* threshold being ***t1***. So, we increase this threshold to the minimum cost of all values that exceed

this threshold. Suppose this value is ***t2***. In this case, IDA* for start51 will need to expand all the previously explored nodes for start49 in addition to any new nodes that have a value larger than ***t1*** and not exceeding ***t2***. In worst case, this could contribute to exponentially more nodes being needed to be expanded. This is quite trivial to see, as, suppose we have a node with a value smaller than ***t1***. Now imagine all b children of that node have values larger than ***t1*** and not exceeding ***t2*** needing to be expanded by IDA* search, and suppose that in turn that all b children of these b nodes have values larger than ***t1*** and not exceeding ***t2*** needing to be expanded by IDA* search and so on.

It is apparent from following this reasoning that IDA* search can expand exponentially more nodes for a path length that is only marginally longer, as can be observed in the difference between the number of nodes expanded for start49 and start51.

Question 3: Heuristic Path Search

(a) Run [greedy] for start49, start60 and start64, and record the values returned for G and N in the last row of your table (using the Manhattan Distance heuristic defined in puzzle15.pl).

	Start49		Start60		Start64	
IDA*	49	178880187	60	321252368	64	1209086782
1.2	51	988332	62	230861	66	431033
1.4	57	311704	82	3673	94	188917
Greedy	133	5237	166	1617	184	2174

(b) Now copy idastar.pl to a new file heuristic.pl and modify the code of this new file so that it uses an Iterative Deepening version of the Heuristic Path Search algorithm discussed in the Week 4 Tutorial Exercise, with w = 1.2 . In your submitted document, briefly show the section of code that was changed, and the replacement code.

The function depthlim(Path, Node, G, F_limit, Sol, G2) was modified. The line highlighted below was changed from F1 is G1 + H1 to the Iterative Deepening version of the Heuristic Path Search – F(n) = (2-w)g(n) + wh(n) where w = 1.2 This change can be seen in the code below:

```
depthlim(Path, Node, G, F_limit, Sol, G2) :-
        nb_getval(counter, N), N1 is N + 1,
        nb_setval(counter, N1),
        % write(Node),nl, % print nodes as they are expanded
        s(Node, Node1, C),
        not(member(Node1, Path)), % Prevent a cycle
         G1 is G + C,
        h(Node1, H1),
        F1 is (2-1.2)*G1 + (1.2)*H1,
        F1 =< F_limit,
        depthlim([Node|Path], Node1, G1, F_limit, Sol, G2).
```

(c) Run [heuristic] on start49, start60 and start64 and record the values of G and N in your table. Now modify your code so that the value of w is 1.4 instead of 1.2 ; in each case, run the algorithm on the same three start states and record the values of G and N in your table.

As above in the table in (a).

(d) Briefly discuss the tradeoff between speed and quality of solution for these four algorithms.

	Start49		Start60		Start64	
IDA*	49	178880187	60	321252368	64	1209086782
1.2	51	988332	62	230861	66	431033
1.4	57	311704	82	3673	94	188917
Greedy	133	5237	166	1617	184	2174

IDA* search guarantees an optimal solution if the heuristic is admissible. In this case the heuristic is admissible as we are using the Manhattan distance as our heuristic, so the solution that is produced by IDA* search is optimal. As can be seen in the table above, IDA* search produces solutions with significantly cheaper costs than Greedy and a marginally cheaper cost than the Heuristic search with w = 1.2 and w = 1.4.

However, as IDA* is a variant of Iterative Deepening Search, it does not have any 'state' and re-expands nodes already previously expanded. This means IDA* has a time complexity of $O(b^d)$, where *d* is the depth of the optimal solution, as the root is expanded d+1 times, the b nodes at level 1 are expanded d times, the b^2 nodes at level 2 are expanded d-1 times and so on. The number of nodes expanded for IDA* are significantly worse than the number of nodes expanded for all other search algorithms in the table.

Greedy has a worst-case time complexity of $O(b^m)$, where m is the maximum depth of the state space, which is worse than IDA*'s worst-case time complexity of $O(b^d)$ (unless the depth of the optimal solution is equal to the depth of the tree). However, as can be seen in the table above, Greedy expands substantially less nodes than all other search algorithms. We can infer from this that the heuristic being used is a good heuristic as it has managed to decrease the time complexity from the worst-case scenario quite substantially.

Greedy however, produces the least optimal solution out of all four search algorithms as the number of nodes in all of its solutions are greater than the number of nodes in the solutions generated by the other search algorithms. Again, however, as the difference between the cost of the optimal solution and the solution generated by Greedy is not that substantial (at least for small C*, where C* denotes the cost of the optimal solution), it seems that the Manhattan heuristic works quite well in this case for Greedy (though obviously for larger C*, the Greedy's performance would be much worse in comparison to IDA*).

The Heuristic searches with w=1.2 and w=1.4 are in between A* search (when w = 1) and Greedy search (when w=2). With w=1.2, the path obtained by the Heuristic search has a length that is very close to the optimal path length, and while the number of nodes expanded is large, it still represents a significant decrease from the number of nodes expanded in IDA* search. With w = 1.4, the number of nodes expanded falls dramatically compared to the number of nodes expanded with w=1.2. The path cost, however, also increases (this increase is most apparent when we consider larger C* values). These trends clearly slow that as we approach Greedy (w=2), the number of nodes expanded

decreases but path cost increases, and as we approach A* search (w=1), the number of nodes expanded exponentially increases, but the cost of the solution decreases.

Question 4: Maze Search Heuristics

(a) Give the name of another admissible heuristic which dominates the StraightLine-Distance heuristic, and write the formula for it in the format $h(x,y,x_G,y_G) = \ldots$

An admissible heuristic h(n) dominates another admissible heuristic g(n) if for all states h(n) >= g(n). For the 2-dimensional maze, the Manhattan/Taxicab distance heuristic dominates the Straight-Line Distance heuristic. As the Euclidean Straight-Line distance heuristic simply measures the distance between the current position and the Goal, this heuristic assumes we can walk through the obstacles in the maze. In contrast, the Manhattan/Taxicab distance assumes we are 'driving' along streets and not walking through the obstacles, so it clearly dominates the Straight-Line Distance heuristic.

The formula for the Manhattan distance is:

$h(x,y,x_G,y_G) = |x_G - x| + |y_G - y|$

(b)

(i) Assuming that the cost of a path is the total number of (horizontal, vertical or diagonal) moves to reach the goal, is the Straight-Line Distance heuristic still admissible? Explain why

The Straight-Line Distance heuristic is no longer admissible. This is because the heuristic will overestimate the cost of moving between two states that are diagonally accessible from each other. For example, if our starting position the square (0,0) and our goal is at the square (1,1), we only need 1 diagonal move to reach our goal from the starting position. The Straight-Line Distance heuristic however, would overestimate the cost of navigating between these two squares when the hypotenuse of the triangle formed is greater than 1 (i.e. when $sqrt((x_G-x_{start})^2 + (y_G-y_{start})^2 > 1$). In this case of the example above, the hypotenuse of the triangle formed is $sqrt((1-0)^2 + (1-0)^2) = sqrt(2)$, which is greater than 1. (Alternatively, to make this example more precise, we could consider the starting position to be in the centre of the square (0,0) and the goal to be the centre of the square (1,1), but the result is still the same – i.e. the Straight-Line Distance heuristic estimates the cost to be $sqrt((1.5-0.5)^2+(1.5-0.5)^2) = sqrt(2)$, but the real cost is 1)

As a result, the Straight-Line Distance heuristic does not always guarantee an underestimate of the true path cost, so it is no longer admissible.

(ii) Is your heuristic from part (a) still admissible? Explain why.

Clearly, as the Straight-Line Distance is no longer admissible, the Manhattan distance also can no longer be an admissible heuristic, as it dominates the straight-line heuristic.

Like the Straight-Line Distance heuristic, the Manhattan distance heuristic similarly overestimates the cost of moving between two states that are diagonally accessible. For example, if our starting position is the square (1,1) and our goal (or an intermediate state we need to get to in order to reach our goal) is (2,2), the Manhattan distance would be $|2-1|+|2-1| = 2$. This clearly overestimates the actual cost of reaching the goal, which is 1, as we only need 1 diagonal move to reach the goal from the starting state. Given that admissible heuristics cannot overestimate the cost of reaching the goal, the heuristic from (a) is also no longer admissible.

(iii) Try to devise the best admissible heuristic you can for this problem, and write a formula for it in the format: h(x, y, xG, yG) =

The best admissible heuristic I can think of is the following:

It is clear that the Manhattan distance and the Straight-Line Distance fail because they overestimate the true cost of moving between states that are diagonally accessible to one another.

We remove all the obstacles from the maze and try to construct a heuristic that never overestimates the true cost of moving between any two states. It is clear that the Straight-Line distance heuristic is still admissible for moving between states that are vertically or horizontally accessible to one another as in the heuristic's worst case, this vertical/horizontal distance corresponds to the number of moves needed.

To ensure the best admissible heuristic would never overestimate the true cost of moving between two states, we need a heuristic that at worst case, corresponds to the minimum number of moves of moving between two states (similar in a sense to the Straight-Line distance heuristic when it is applied for moving vertically or horizontally between states).

To accommodate the ability to move diagonally into the Straight Line heuristic, we can formulate a heuristic, h as follows: $h(x,y,x_G,y_G) = \max(|x_G-x|, |y_G-y|)$. The difference between the x coordinates $|x_G-x|$ corresponds to the number of horizontal moves we need to make to get between two states. Similarly, the difference between the y coordinates $|y_G-y|$ corresponds to the number of vertical/diagonal moves we need to make to get between two states. We take the max of the two in case our goal and starting state are in the same horizontal or vertical line (as one of $|x_G-x|$ or $|y_G-y|$ would then be 0, but the number of moves would not necessarily be zero – e.g. if we wanted to move between (1,3) and (1,5)).

This heuristic clearly corresponds to the minimum number of moves that we need to make in order to get between two states if we can move diagonally, vertically or horizontally and is thus admissible as it never underestimates the true cost.

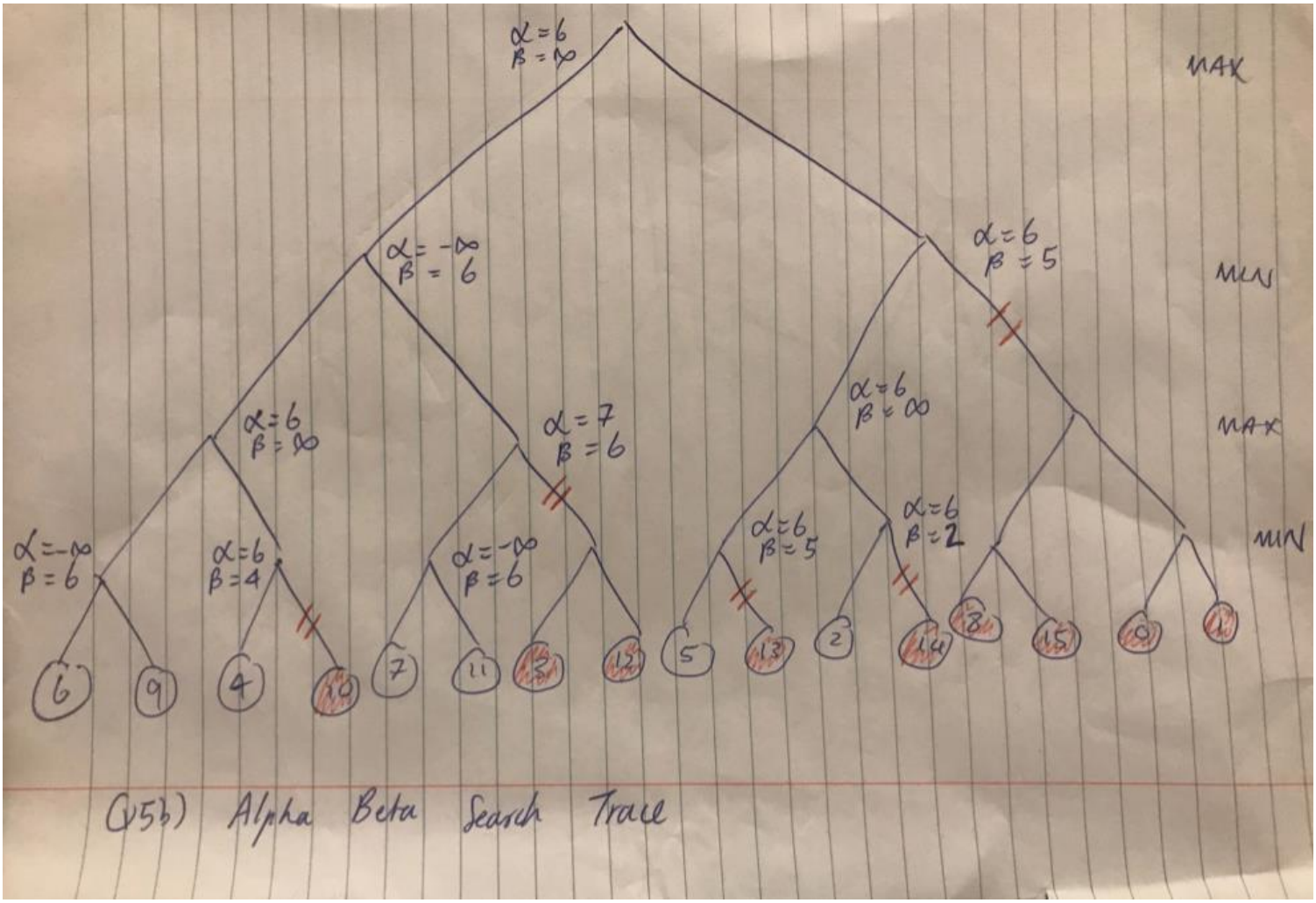
Question 5: Game Trees and Pruning

(a) Consider a game tree of depth 4, where each internal node has exactly two children (shown below). Fill in the leaves of this game tree with all of the values from 0 to 15, in such a way that the alpha-beta algorithm prunes as many nodes as possible. Hint: make sure that, at each branch of the tree, all the leaves in the left subtree are preferable to all the leaves in the right subtree (for the player whose turn it is to move).

An example of such an ordering (left to right) is as follows: 6 9 4 10 7 11 3 12 5 13 2 14 8 15 0 1. This allows for optimal pruning by the Alpha Beta search algorithm.

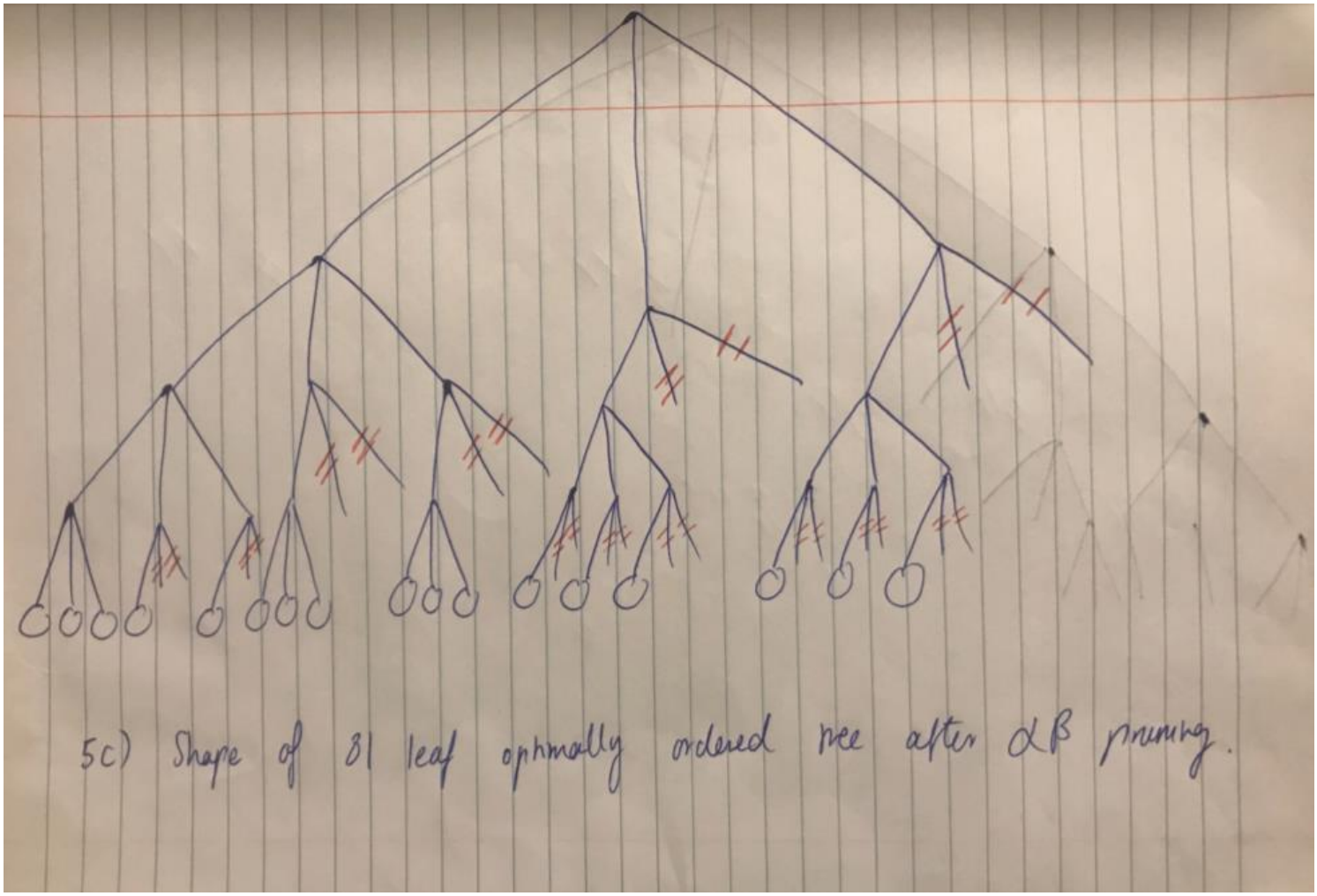
(b) Trace through the alpha-beta search algorithm on your tree, showing clearly which nodes are evaluated and which ones are pruned.

Please see diagram below. Intermediate values of alpha and beta and the values at nodes that are not the leaf nodes are not shown for clarity and readability of the diagram. A double dash in red indicates pruning occurs. Leaf nodes coloured in red indicate that the node has been pruned. All other nodes are evaluated.



(c) 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 the alpha-beta algorithm prunes as many nodes as possible. Draw the shape of the pruned tree. How many of the original 81 leaves will be evaluated? Hint: If you look closely at the pruned tree from part (b) you will see a pattern. Some nodes explore all of their children; other nodes explore only their leftmost child and prune the other children. The path down the extreme left side of the tree is called the line of best play or Principal Variation (PV). Nodes along this path are called PV-nodes. PV-nodes explore all of their children. If we follow a path starting from a PV-node but proceeding through non-PV nodes, we see an alternation between nodes which explore all of their children, and those which explore only one child. By reproducing this pattern for the tree in part (c), you should be able to draw the shape of the pruned tree (without actually assigning values to the leaves or tracing through the alpha-beta algorithm).

See the diagram below. Some vertices and terminal nodes in the tree are omitted for diagram readability. Terminal nodes are indicated by circles. Pruning is represented by a double dash in red.



(d) What is the time complexity of alpha-beta search, if the best move is always examined first (at every branch of the tree)? Explain why.

If we arrange our tree in the optimal arrangement for alpha-beta pruning, the time complexity will be $O(b^{d/2})$ where d is the depth of the tree.

The reasoning for this is as follows. Assume we have a tree with branching factor b (i.e. there are b successors at every node n of the tree). If the tree is arranged optimally for alpha-beta pruning, the effective branching factor is only $b^{1/2}$ instead of b. This is somewhat intuitive. For each of our moves, we only need the opponent's best move (which gives us our worst possible result) to refute this move. If the best of each player's moves is revealed as early as possible, this means we effectively cuts the branching factor in half, as we will have to evaluate b leaf node positions, then 1 leaf node position, then b leaf node positions, then 1 leaf node position etc. in an alternating pattern until we reach the depth of the tree. So, we have O(b*1*b*1..*b) (for odd depth) and O(b*1*b*1*..*1) (for even depth) or $O(b^{d/2})$ leaf nodes that are evaluated.