Question 1: Search Algorithms for the 15-Puzzle (2 marks)

(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:

(i) [ucsdijkstra]

(ii) [ideepsearch]

(iii) [astar]

(iv) [idastar]

In each case, record in your table the number of nodes generated during the search. If the algorithm runs out of memory, just write "Mem" in your table. If the code runs for five minutes without producing out- put, terminate the process by typing Control-C and then "a", and write "Time" in your table.

	Start 10	Start 20	Start 27	Start 35	Start 43
UCS	2565	Mem	Mem	Mem	Mem
IDS	2407	5297410	Time	Time	Time
A*	33	915	1873	Mem	Mem
IDA*	29	952	2237	215612	2884650

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

From above, we can find that:

UCS (Uniform Cost Search) costs the most time and space. The time and space complexity are both about $O(b^{\lceil C^*/\varepsilon \rceil})$.

IDS (Iterative Deepening Search) generates the most nodes until Start 20 which has 5297410 expanded nodes and it is expensive. Thus, although its space complexity $\mathrm{O(bd)}$ is good, the time complexity $\mathrm{O}(b^d)$ is expensive resulting in 'Time-out' when it runs in complicated situation.

A* Search has good efficiency, in Start 10, Start 20 and Start 27, it performs good. Its time complexity is $\mathrm{O}(b^{\varepsilon d})$. And the $\varepsilon = \frac{(h^*-h)}{h^*}$. However, the time complexity can be up to $\mathrm{O}(b^d)$. And when in Start 35 and Start 43, it runs out of space. This is because its space complexity can be up to $\mathrm{O}(b^d)$, which may result in out of space.

IDA*(Iterative Deepening A* Search) is the best algorithm in those four algorithms. Although in Start 20 and Start 27, it expands more nodes than A*, in other states, its performance is better than all other algorithms, especially it is the only one algorithm which can search a result from Start 35 and Start 43. This is because its time complexity may be larger than A* Search $\mathrm{O}(b^{\varepsilon d})$, but its space complexity is just $\mathrm{O(bd)}$, which is extremely less than A* Search.

Question 2: Deceptive Starting States (1.5 marks)

Consider the two starting states start49 and start51.

(a) Print state start49 and calculate its heuristic value by typing start49(S), showpos(S), h(S,H). Do the same for start51 and copy the result you're your solutions.

Start49:

```
MBDC
LAKH
JFEI
 ONG
```

S = [4/1, 2/2, 1/2, 1/4, 1/3, 3/3, 3/2, 4/4, ... / ...|...],

H = 25.

Start51:

```
GKJI
 MNC
EOHA
FBLD
```

S = [2/1, 3/4, 4/2, 2/4, 4/4, 3/1, 4/1, 1/1, ... / ...|...],

H = 43.

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

Starting from start51, the number of nodes is 551168.

(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.

Although the true path length is similar, the IDA* uses a series of depth-first searches, but cuts off each search when the sum f() =g()+h() exceeds some pre-defined threshold. This means that the number of nodes in IDA* algorithm increases exponentially with the increase of solution paths. The reason may be that in start49, the breadth-first search after the depth-first search has traversed too many nodes.

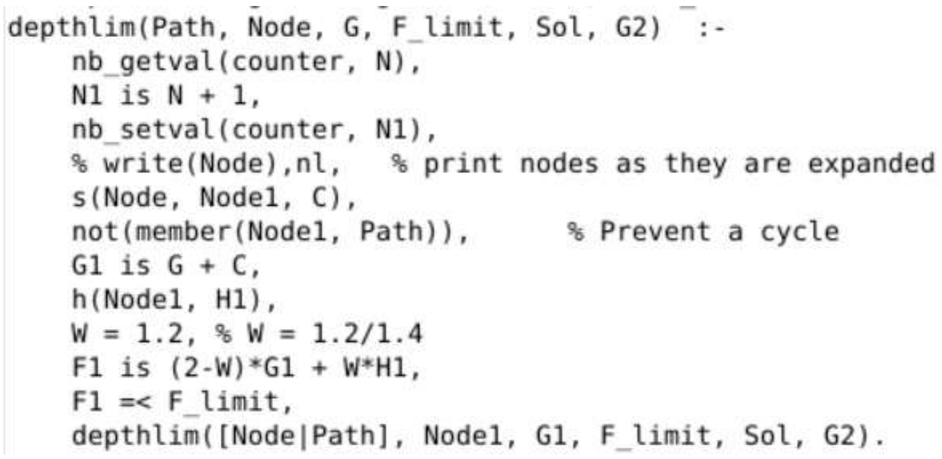
Question 3: Heuristic Path Search (2 marks)

In this question you will be exploring an Iterative Deepening version of the Heuristic Path Search algorithm discussed in the Week 4 Tutorial. Draw up a table in the following format:

	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**

(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).
As above.
(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 replacement code shows 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),
    W = 1.2, % W = 1.2/1.4
    F1 is (2-W)*G1 + W*H1,
    F1 =< F_limit,
    depthlim([Node|Path], Node1, G1, F_limit, Sol, G2).
```

The new F1 is based on the function:
f(n) = (2-w) * g(n) + w * h(n)
And W = 1.2. In (c), just change W =1.2 to W = 1.4.

(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.

(d) Briefly discuss the tradeoff between speed and quality of solution for these four algorithms.
Greedy Search is the most efficient algorithm and it finds the least cost in each step, but it cannot ensure its solution is the optimal.
IDA* is the least efficient algorithm, and it generates the most nodes. However, its solution may be optimal compared with other algorithms.
For other heuristic algorithms, the efficiency and quality may be different when the value of W changed. For example, when W increases, the algorithms will be more like **Greedy Search.** Thus, the speed will be increased. However, its quality of solution may become less optimal.

Question 4: Maze Search Heuristics (2.5 marks)
(a) Assume that at each time step, the agent can move one unit either up, down, left or right, to the center of an adjacent grid square.
One admissible heuristic for this problem is the Straight-Line-Distance heuristic:

$$\boldsymbol{h_{SLD}(x, y, x_G, y_G) = \sqrt{(x - x_G)^2 + (y - y_G)^2}}$$

However, this is not the best heuristic.

Give the name of another admissible heuristic which dominates the Straight-Line Distance heuristic, and write the formula for it in the format:

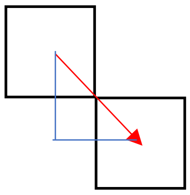
$$h(x, y, x_G, y_G) = |x - x_G| + |y - y_G|$$

The Manhattan-Distance dominates the Straight-Line Distance. Since its movement directions are just up, down, left or right, which means it cannot be diagonal, the Manhattan-Distance is the better heuristic. The Manhattan-Distance is calculating the sum of distance between two points in two axes. Therefore, the distance will be no less than the Straight-Line-Distance, but it is still a better choice in maze since it is more suitable when the movement directions just include up, down, left and right.

(b) Now assume that at each time step, the agent can take one step either up, down, left, right or diagonally. When it moves diagonally, it travels to the center of a diagonally neighboring grid square, but a diagonal step is still considered to have the same "cost" (i.e. one "move") as a horizontal or vertical step (like a King move in Chess).

(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.

No. Since a heuristic cannot be overestimated the cost than the lowest possible cost, it is not admissible. The Straight-Line-Distance always calculates the actual distance between the two points. However, from the question, we know when the movement direction is diagonal, the step is still considered to one cost. This means when the step moves diagonally like the red line below, it will be calculated one move, but in fact, in Straight-Line-Distance, it will be calculated as $\sqrt{1+1} = \sqrt{2}$. Thus, the Straight-Line-Distance overestimates the lowest cost, it should not be admissible.



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

No. It is the same as above. The Manhattan-Distance calculates the sum of distance of two points in two axes. As above, the Straight-Line-Distance calculates the actual distance between the two points and it calculates the diagonal movement as one move. However, in Manhattan-Distance, it will be calculated as 2 (Like above diagram, 1 in x-axis and 1 in y-axis.). Thus, the Manhattan-Distance also overestimates the lowest cost, it should not be admissible.

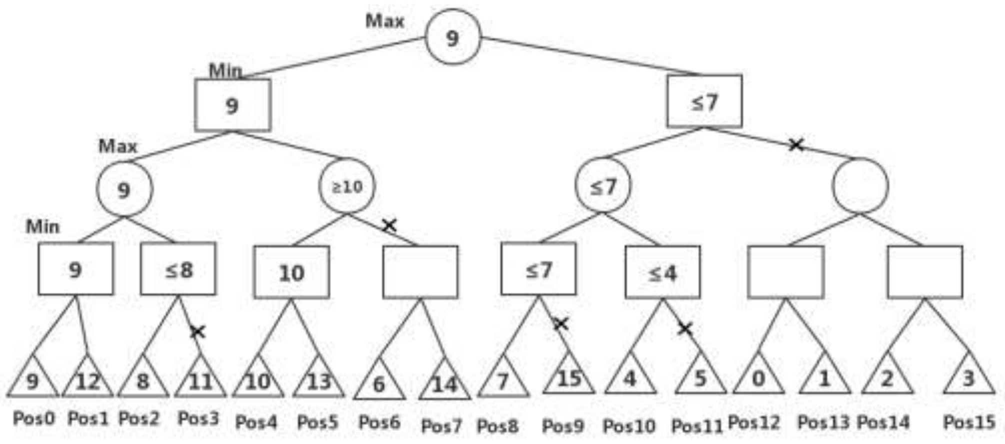
(iii) Try to devise the best admissible heuristic you can for this problem, and write a formula for it in the format:

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

The above formula can be admissible heuristic. It counts the maximum of coordinate differences (Either X or Y) when moving. If the agent moves diagonally, the distance will still be counted as one, and other directions are also counted as one. Thus, this heuristic is admissible.

Question 5: Game Trees and Pruning (4 marks)

(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.



This is the one of possible tree that the alpha-beta algorithm prunes as many nodes as possible.

As above, sign Pos0 to Pos15 follow the sequence from left to right. We can assume that Pos0 < Pos1, Pos2<Pos3, Pos4<Pos5. And using the alpha-beta algorithm, we can get:

When Pos0 ⩾ Pos2, cut Pos3;

When Pos4 ⩾ Pos0, cut Pos6 and Pos7;

When Pos0 ⩾ Pos8, cut Pos9;

When Pos0 ⩾ Pos10, cut Pos11;

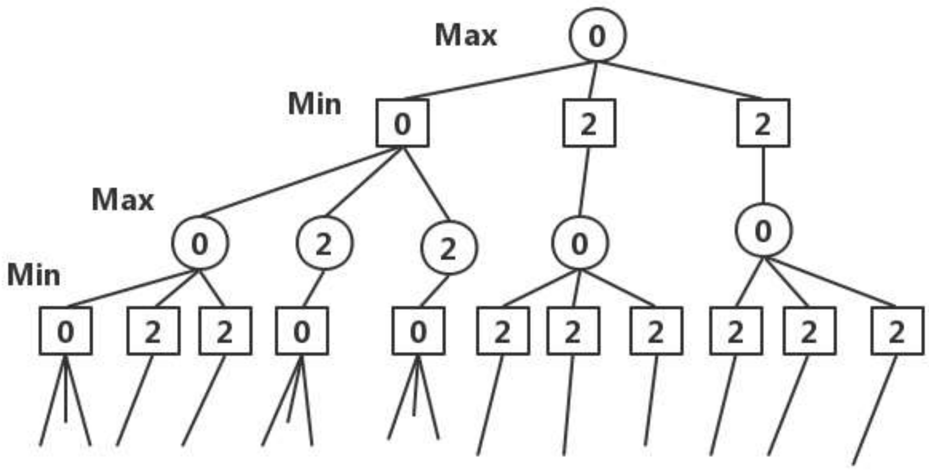
When Pos0 ⩾ Pos8 or Pos0 ⩾ Pos10, cut Pos12, Pos13, Pos14, Pos15.

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

As above, we can know that the Pos0, Pos1, Pos2, Pos4, Pos5, Pos8, Pos10 are evaluated, which is 7 in total. Other nodes are pruned. (Like the diagram shows, the cross means that it will be pruned.)

(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?

We can get the below diagram by using alpha-beta algorithm in this problem:



Every node has three children and the number means the number of pruned children nodes. For example, 0 means that all the children nodes are not pruned, and 2 means that there are 2 children nodes that are pruned at this node. In addition, all the pruned children nodes are not left children nodes because it can ensure the best time complexity. This means if the pruning happens, the nodes' left children node will be reserved and other children nodes will be cut.
Thus, we can find that it would be 17 original leaves will be evaluated.

(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.

The time complexity of alpha-beta search is $\mathrm{O}(b^{\frac{d}{2}})$.
With a branching factor of b, and a search depth of d plies, the maximum number of leaf node positions evaluated (when the move ordering is worst) is $\mathrm{O}(\mathrm{b} * \mathrm{b} * \mathrm{b} \ldots) = \mathrm{O}(b^{d})$ – the same as a simple minimax search. If the move ordering for the search is optimal (meaning the best moves are always searched first), the number of leaf node positions evaluated is about $\mathrm{O}(\mathrm{b} * 1 * \mathrm{b} * 1 \ldots \mathrm{b})$ for odd depth and $\mathrm{O}(\mathrm{b} * 1 * \mathrm{b} * 1 \ldots 1)$ for even depth. This is because all the first player's moves must be studied to find the best one, but for each, only the best second player's move is needed to refute all but the first (and best) first player move – alpha–beta ensures no other second player moves need be considered.

Thus, the time complexity of alpha-beta search is $\mathrm{O}(b^{\frac{d}{2}})$.