COMP9414 Assignment 2

Student Code: z5100764
Student Name: Chunnan Sheng

Question 1

(a)

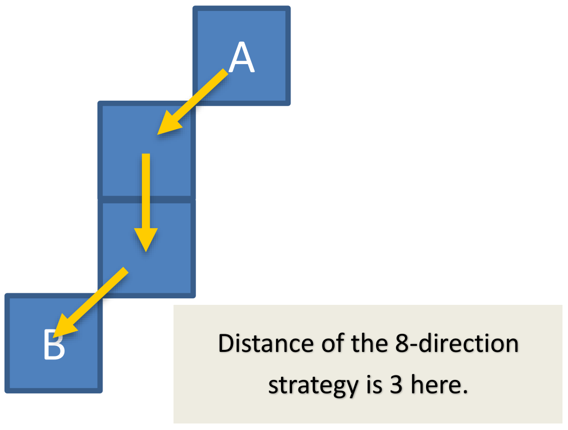
The Manhattan Distance Heuristic which is still admissible dominates the SLD Heuristic.
The Manhattan Distance can be described in the following way:

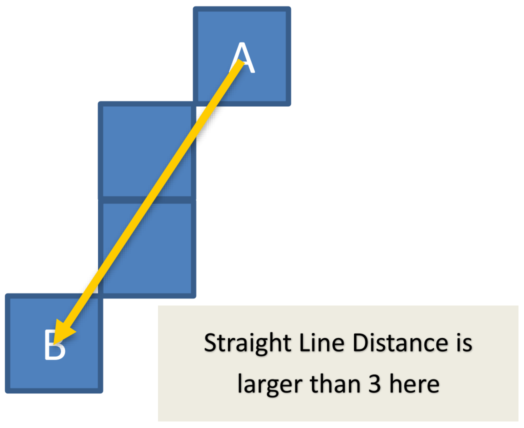
$$h(x, y, x_G, y_G) = |x - x_G| + |y - y_G|$$

(b)

(i)

The Straight Line Distance Heuristic is not admissible in this circumstance.
The distance of the 8-direction strategy is shorter than the Straight Line Distance, which means SLD over estimates the real distance.
The following illustration is an example of comparison of these two strategies on behalf of distance from A to B.



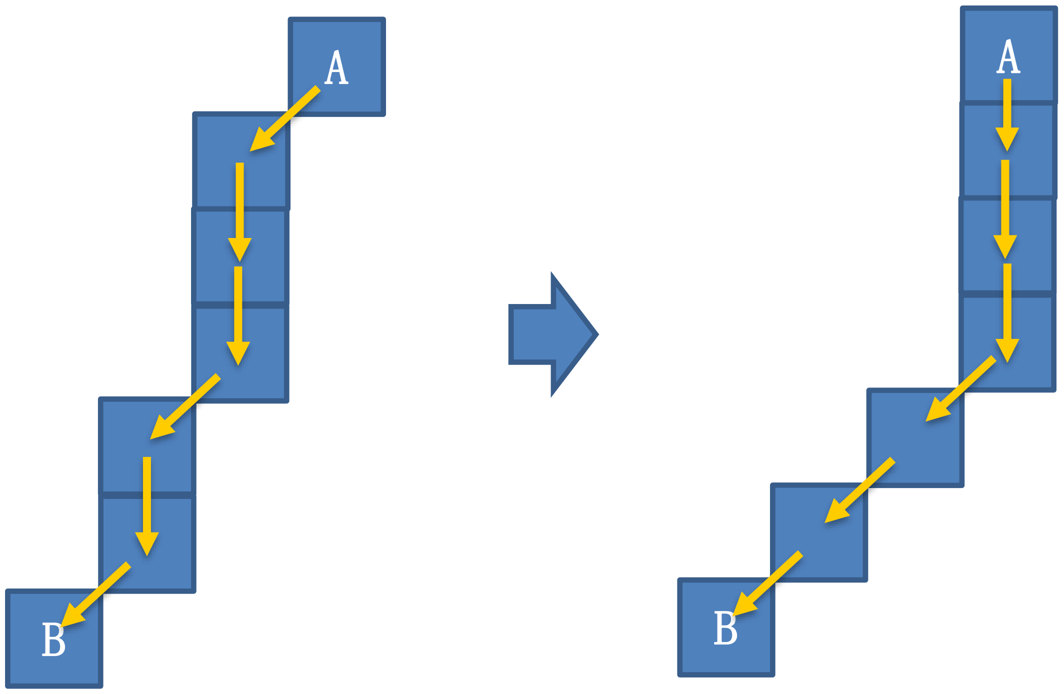


(ii)

The heuristic of part (a) is not admissible either because it is over estimation too.

(iii)

The distance of the new moving strategy can be divided into 2 parts: the vertical or horizontal distance and the 45° diagonal distance.



According to the illustration above, calculation of the new distance will be:

$$h(x, y, x_G, y_G) = \begin{cases} |x - x_G| & if \;\; |x - x_G| > |y - y_G| \\ |y - y_G| & if \;\; |x - x_G| \leq |y - y_G| \end{cases}$$

Question 2

(a)

	Start10	Start12	Start20	Start30	Start40
Uniform Cost Search	2565	Mem	Mem	Mem	Mem
Iterative Deepening Search	2407	13812	5297410	Time	Time
A*	33	26	915	Mem	Mem
Iterative Deepening A* (Man)	29	21	952	17297	112571

(b)

	Start10	Start12	Start20	Start30	Start40
Iterative Deepening A* (Mis)	175	1365032	Time	Time	Time

Two parts of puzzle15.pl should be changed.
Change part 1: The swap operation between *Empty* and *Tile* can be done provided that they are adjacent.

```
% Swap is allowed if and only if Empty and Tile are adjacent
swap(Empty, Tile, [Tile|Ts], [Empty|Ts]) :-
    adjacent(Empty, Tile).

swap(Empty, Tile, [T1|Ts], [T1|Ts1]) :-
    swap(Empty, Tile, Ts, Ts1).

misplace(X/Y, X1/Y1, 0) :-     % If positions are the same, then misplacement is 0
    X is X1,
    Y is Y1.

misplace(X/Y, X1/Y1, 1) :-     % Misplacement is 1 for any position differences
    X =\= X1;
    Y =\= Y1.

adjacent(X/Y, X1/Y1) :-       % Adjacent situation 1 (the same X coordinate)
    dif(X, X1, 0),
    dif(Y, Y1, 1).

adjacent(X/Y, X1/Y1) :-      % Adjacent situation 2 (the same Y coordinate)
    dif(X, X1, 1),
    dif(Y, Y1, 0).

% Original dif functions are preserved

dif(A, B, D) :-              % D is |A-B|
    D is A-B, D >= 0, !.

dif(A, B, D) :-              % D is |A-B|
    D is B-A.
```

Change part 2: totdist should be calculated by *misplace*.

```
totdist([Tile|Tiles], [Position|Positions], D) :-
    misplace(Tile, Position, D1),
    totdist(Tiles, Positions, D2),
    D is D1 + D2.
```

(c)

The Uniform Cost Search is the least memorial efficient since its space complexity is $O(b^{\lceil C^*/\varepsilon \rceil})$ which grows exponentially.

The Iterative Deepening Search's space complexity is only $O(bk)$ that makes it the most memorial efficient. However, it is time consuming and even sometimes not guaranteed to be complete.

The heuristic estimation has made A* more time efficient, but the memory is still an issue.

The Iterative Deepening Search can solve the memory issue of A*. On the other hand, A* can also help eliminate time issues of the Iterative Deepening Search.

The Manhattan Distance seems to be more practical to be applied to heuristics since it is more dominant than the Misplacement Distance. The Misplacement Distance is always ONE for each tile in the 15-Puzzle despite how long the distance is between the tile's current position and its goal position, which could make the algorithm much less efficient because it cannot estimate the cost precisely.

Question 3

(a)

It seems the Greedy Heuristic does find out solutions to the goals quickly, but they are NOT optimal (value of G is usually larger than actual depth of the goal).

	Start50		Start60		Start64	
	G	N	G	N	G	N
Iterative Deepening A*	50	1462512	60	321252368	64	1209086782
Greedy	164	5447	166	1617	184	2174

(b)

The modified part of the source code is in RED colour.

```
% Otherwise, use Prolog backtracking to explore all successors
% of the current node, in the order returned by s.
% Keep searching until goal is found, or F_limit is exceeded.
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 0.8 * G1 + 1.2 * H1,
    F1 =< F_limit,
    depthlim([Node|Path], Node1, G1, F_limit, Sol, G2).
```

The solutions are not optimal (52 > 50, 62 > 60 and 66 > 64) but they are relatively good.

	Start50		Start60		Start64	
	G	N	G	N	G	N
1.2	52	191438	62	230861	66	431033

(c)

	Start50		Start60		Start64	
	G	N	G	N	G	N
IDA*	50	1462512	60	321252368	64	1209086782
1.2	52	191438	62	230861	66	431033
1.4	66	116342	82	4432	94	190278
1.6	100	33504	148	55626	162	235848
1.8	240	35557	314	8814	344	2209
Greedy	164	5447	166	1617	184	2174

(d)

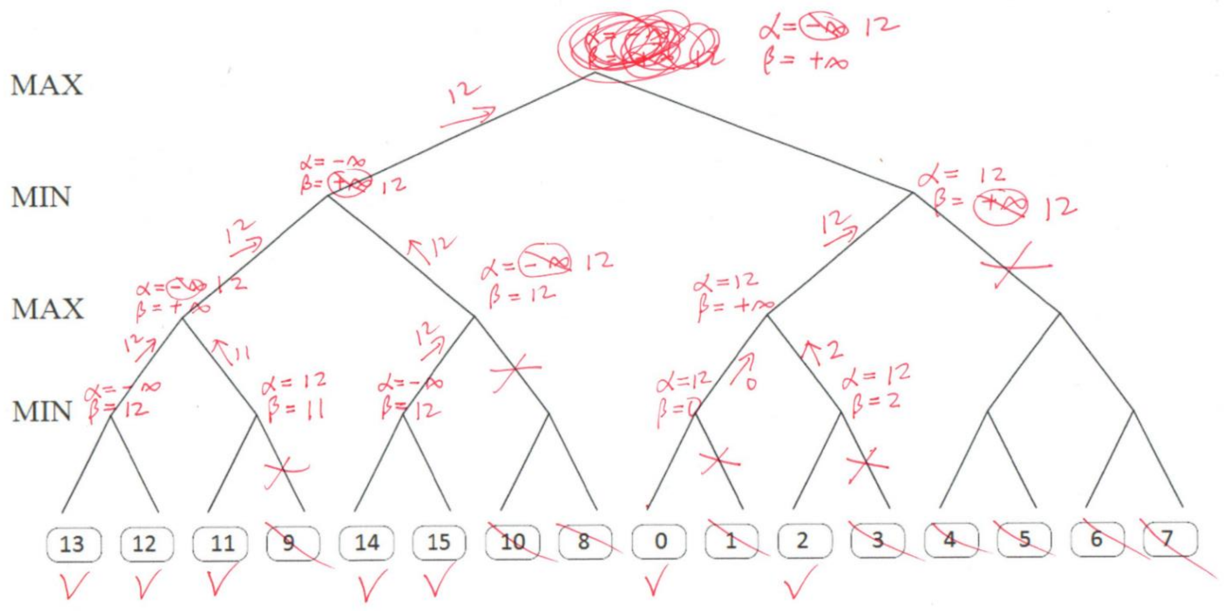
Generally speaking, more weights for the $f(n)$ may speedup the solution, but could also make the solution less optimal. It is also possible that large ω value like 1.8 may even lead to less accurate solution than Greedy Search because it is closer to a double Greedy Search that is $2f(n)$.

Question 4

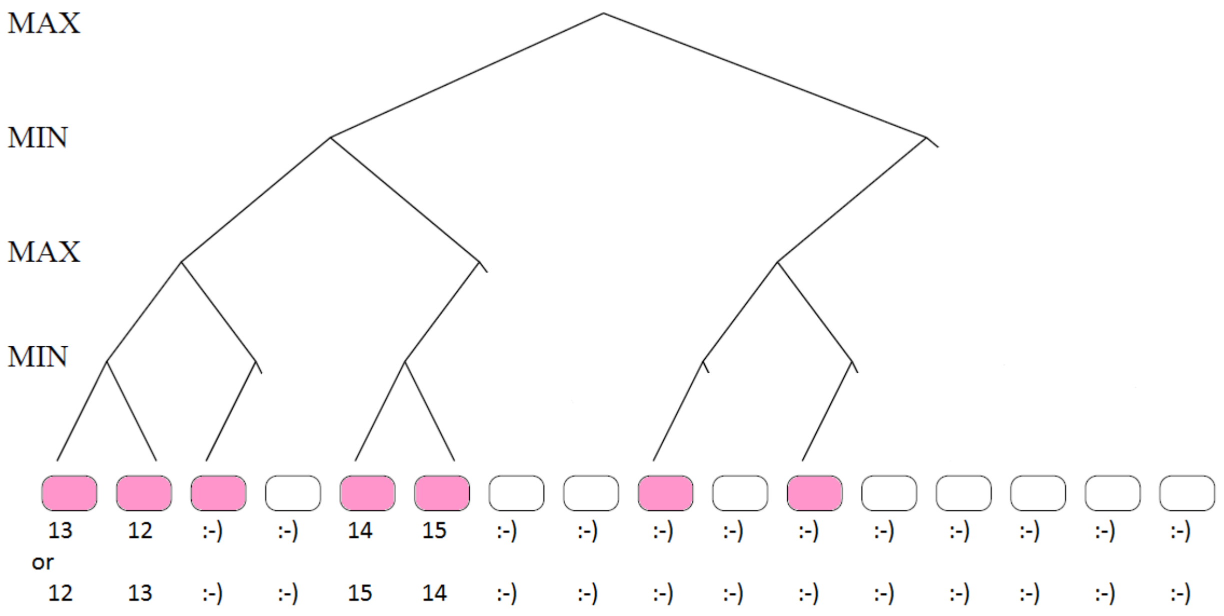
(a)

There are multiple answers to this question. If the 1st leaf and the 2nd leaf are 12 and 13, or 13 and 12, together with the 5th and 6th leaves which are 14 and 15, or 15 and 14, respectively, the alpha-beta search could be optimized despite values of other leaves.

Pruning by tracing through the tree with example values:



Pruned tree and possible answers:



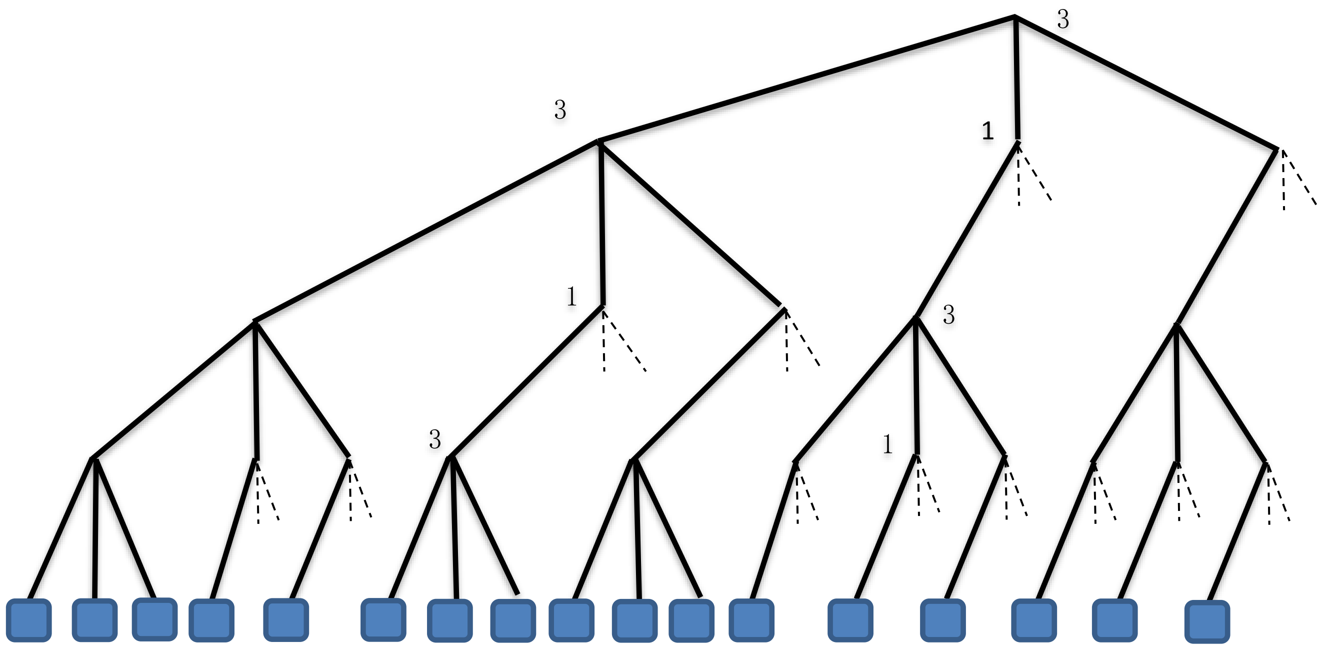
(b)

7 if the original 16 leaves are evaluated.

(c)

The pruned tree of depth 4 where each internal node has three children will only evaluate **17** of the original 81 leaves, provided that the leaves are organised in a way that the alpha-beta algorithm prunes the maximum number of nodes.

The following diagram shows how this pruned tree looks.



(d)

Except the extreme left side of the tree, all internal nodes in this tree follow an alternation rule of n children, 1 child, n children, … etc. So, the amount of leaves can be estimated as

$$n \times 1 \times n \times 1 \times \ldots$$

which is about $n^{\frac{d}{2}}$ where all leaves of the unpruned tree are amounted to n^d. Therefore, the time complexity of alpha-beta search in its best cases is $O(n^{\frac{d}{2}})$ where n is the branching factor while d is the tree depth.