COMP9814

Assignment 2

Yisong Jiang

z5123920

Question 1:

(a)

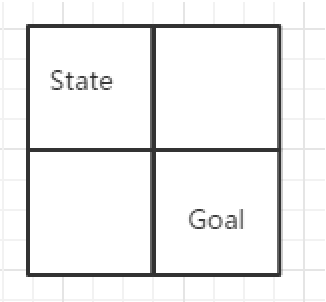
$h_{XY}(x, y, x_G, y_G) = |x_G - x| + |y_G - y|$

It is also the Manhattan Distance Heuristic of the maze. It dominates the Straight-Line-Distance heuristic because $h_{XY} \geq h_{SLD}$ all the time. The average number of nodes expanded will always be fewer.

(b)

(i)

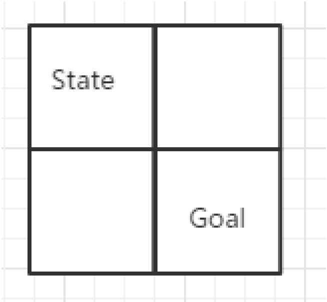
Straight-Line-Distance heuristic is not admissible.



The actual cost from State to Goal is 1 since agent can move diagonally. But the h_{SLD} of this state is $\sqrt{2} > 1$. So h_{SLD} may overestimate the cost needed. Then h_{SLD} is not admissible.

(ii)

h_{XY} is not admissible.



The actual cost from State to Goal is 1 since agent can move diagonally. But the h_{XY} of this state is 2. So h_{XY} may overestimate the cost needed. Then it is not admissible.

(iii)

$h_{min}(x, y, x_G, y_G) = min(|x_G - x|, |y_G - y|)$

This heuristic function is the value of minimize distance of x direction or y direction. Each move will get to a node with less distance (in x or y) to the goal. Estimated moving cost never over the actual cost. Hence, the function is admissible.

Question 2:

	start10	start12	start20	start30	start40
UCS	2565	Mem	Mem	Mem	Mem
IDS	2407	13812	5297410	Time	Time
A*	33	26	915	Mem	Mem
IDA*(Man)	29	21	952	17297	112571
IDA*(Mis)(solution1)	35	87	Time	Time	Time
IDA*(Mis)(solution2)	35	87	4345	2105465	Time

Solution1:

Changed code:

```
totdist([], [], 0).

totdist([Tile|Tiles], [Position|Positions], D) :-
    mandist(Tile, Position, D1),
    totdist(Tiles, Positions, D2),
    D1 > 0,
    D is 1 + D2.

totdist([Tile|Tiles], [Position|Positions], D) :-
    mandist(Tile, Position, D1),
    totdist(Tiles, Positions, D2),
    D1 =:= 0,
    D is D2.
```

Explain:

I changed the predicate ***totdist*** to the image above. Usually, we need to create a new predicate to check if one tile is on its position. However, we can just take advantage of the ***mandist*** to help us do the task. Because if one tile has a Manhattan distance (>0) to its position, then it is a misplaced tile and we need to count it. If a tile has no Manhattan distance (=0) to its position, then it is a good tile and we don't need to count it. The solution is slow and can only calculate start10 and start12.

Solution2:

Changed code:

```
misplace(X/Y,X1/Y1,0):-
    X1 =:= X,
    Y1 =:= Y.
misplace(X/Y,X1/Y1,1):-
    X1 \= X,
    Y1 =:= Y.
misplace(X/Y,X1/Y1,1):-
    X1 =:= X,
    Y1 \= Y.
misplace(X/Y,X1/Y1,1):-
    X1 \= X,
    Y1 \= Y.
totdist([], [], 0).

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

Explain:

The first solution is tricky and seems reasonable, but it is very slow since more calculates are needed. The solution2 is a traditional one. I create a new predicate ***misplace*** to judge if one tile is at its position. It will return 1 if the tile is misplaced and return 0 if it is a good one. The total number of misplace tail is the sum of return number. This solution is much faster and it can calculate position start20 and start30 in 5 minutes.

(c)

Efficiency:

The UCS will need much more space than other algorithms, total number of states generated are also very big.

IDA will generate lots of states too but need less space than UCS. Its running speed is very slow.

A* will generate less states and run very fast, but the space it needs is also a bit large.

IDA*(Man) need less space than A*, a bit slow but total performance is best.

IDA*(Mis) generates more states than IDA*(Man), and it is much slower than IDA*(Man) too.

Question 3:

	start50		start60		start64	
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

Changed code (an instance of 1.8, same as others except the coefficient):

```
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 0.2*G1 + 1.8*H1,
    F1 =< F_limit,
    depthlim([Node|Path], Node1, G1, F_limit, Sol, G2).
```

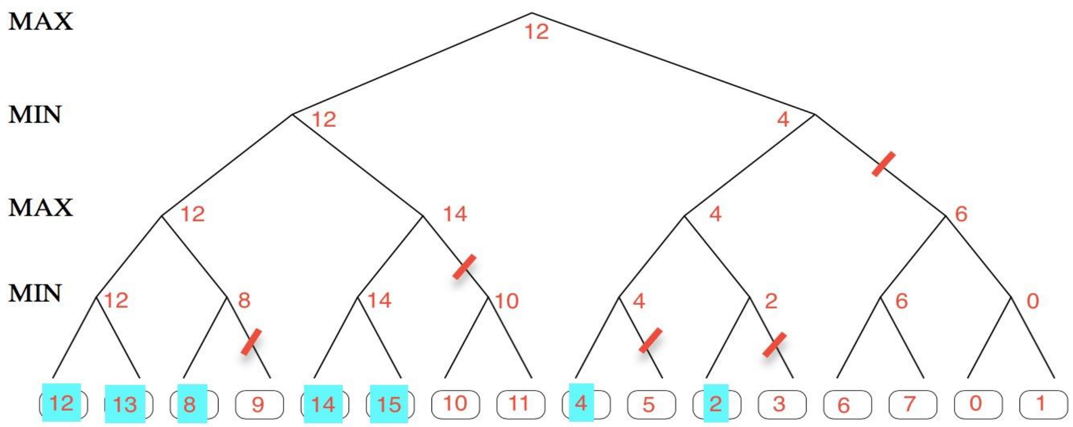
Explain: I changed the code of predicate ***depthlim***. The original code is "$F1\ is\ G1 + H1$", which represents $F(n) = G(n) + H(n)$. Now, we get a value of ω, and let "$F1\ is\ (2-\omega) * G1 + \omega * H1$". Then the $F(n) = (2-\omega) * G(n) + \omega * H(n)$, which is the result we want.

Efficiency:

IDA* is actually the situation when $\omega = 1$ and Greedy is the situation when $\omega = 2$.

In general cases, as $\omega\ grows\ from\ 1\ to\ 2$, the search algorithm runs faster, but the solution path become longer. Hence the speed is faster but quality is worse as ω increase.

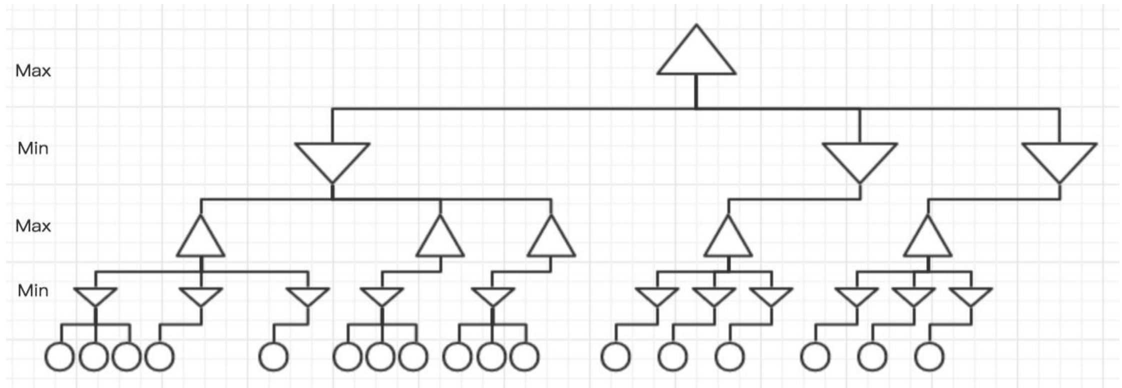
Question 4:



(a)&(b)

My game tree is designed as the image above. The nodes which will be pruned by alpha-beta algorithm are marked with red lines. 7 of 16 leaves are evaluated (marked as blue in the image).

(c)



The shape of game tree is shown above. 17 of 81 leaves are evaluated.

(d)

Time complexity of alpha-beta search:

Assume the branching factor is b and the tree depth is d.

The order in which the states are examined will determine the time needed. If we need to examine all the node (the worst case), then we need $O(b^d)$. If the best move is always examined first (at every branch of the tree), the time complexity become $O\ (b * 1 * b\ ...)(little\ different\ when\ d\ is\ odd\ or\ even, but\ result\ is\ same\ in\ big\ O)$, because when the first player's round to move, he just need to consider the best one move of second player. Hence the time complexity is $O(b^{d/2})$.