Assignment 2 - Heuristics and Search

COMP9414

April 26, 2017

1)

a)

The straight line distance is the shortest path between two points, so the heuristic is clearly admissible. Since we can move in horizontal and vertical directions but not diagonal an admissible heuristic that dominates the straight line distance is the Manhattan or city-block distance, shown below in equation 1.

$$h(x, y, x_G, y_G) = |x - x_G| + |y - y_G| \tag{1}$$

b)

(i)

The straight line distance is not admissible since the cost of traversing a longer distance, i.e through the diagonal, is the same as an vertical or horizontal traversal. Consider the following example: We are in (1,1) and the goal is in (2,2), the exact cost is 1 since we can move over the diagonal with unit cost. But $h_{L_2}(1, 1, 2, 2) = \sqrt{2}$ which is larger than 1, hence the heuristic is no longer admissible.

(ii)

Since the Manhattan distance dominates the straight line distance it's trivially no longer admissible either. Consider the same example as above: $h_{L_1}(1, 1, 2, 2) = 2$ which is greater than both the true cost and the straight line distance.

(iii)

To find the best possible heuristic let's consider at boundary or extreme-case: that the maze is empty in the sense there are no obstacles, this reflects an ideal case which will give us the minimum cost. Since we can move diagonally with the same cost as horizontally or vertically the minimum cost for moving between two set of points is the cost in the dimension the points differ the most in distance. Hence a lower bound on the cost is the Chebyshev distance (equation 2), this is also be definition the best heuristic for the problem since we have not relaxed it but only considered the optimal-cost case.

$$h(x, y, x_G, y_G) = max\{|x - x_G|, |y - y_G|\} \tag{2}$$

2)

a),b)

Algorithm	start10	start12	start20	start30	start40
UCS	G=10, N=2565	MEM	MEM	MEM	MEM
IDS	G=10, N=2407	G=12, N=13812	G=20, N=5297410	TIME	TIME
A*	G=10, N=33	G=12, N=26	G=20, N=915	MEM	MEM
IDA* (Man)	G=10, N=29	G=12, N=21	G=20, N=952	G=30, N=17297	G=40, N=112571
IDA* (Mis)	G=10, N=35	G=12, N=87	G=20, N=4345	G=30, N=2105465	TIME

Table 1: Table of length of solution path (G) and total number of states expanded during the search (N) for different algorithms and starting states (startX indicates that the optimal solution path is X long). Lack of memory is denoted MEM and if the algorithm takes to long to finish the search it is denoted with TIME.

The Prolog code to run the heuristic with misplaced tiles instead of Manhattan distance uses the predicate in Listing 1 and replace the call to Manhattan distance predicate with a call to the misplaced predicate as illustrated in Listing 2.

```
% misplaced(+Tile,+Position,-D). Takes a tile and a goal position and
   bounds D to 1 if the tile is  misplaced and 0 otherwise.

% If same, it is not misplaced -> D = 0
misplaced(T,T,0).

% If not same, it is misplaced -> D = 1
misplaced(T1,T2,D):-
  T1 \== T2,
  D is 1.
```

Listing 1: Misplaced predicate.

```
totdist([Tile|Tiles], [Position|Positions], D) :-
    misplaced(Tile, Position, D1), %Instead of mandist(Tile, Position, D1)
    totdist(Tiles, Positions, D2),
    D is D1 + D2.
```

Listing 2: The change in totdist.

c)

As expected all the algorithms finds the optimal path (if there is enough memory and time) since they're all complete and optimal, given admissible heuristics and positive step cost. As expected Uniform cost search, even if it's a memory efficient one, runs into problem when the depth of the solution increases. When it does find the optimal solution it does it with bad time complexity. Iterative Deepening Search searches for the solution throughout the entire state space without any information (an un-informative search) and hence runs out of time. When it does find solutions it's with bad time-complexity. The A* algorithm combines the the pros of Greedy search and Uniform Cost search and hence overcomes a part of the problem with memory and time. But because A* keeps all nodes in memory it runs out even if it searches intelligently. Hence we turn Iterative deepening search which cuts off when the cost becomes lager than a certain threshold. We use two different heuristics and see that the Manhattan distance is far better than the misplaced tiles heuristic. This is since the Manhattan distance dominates the misplaced tiles heuristic and hence will expand fewer nodes, i.e better time-complexity.

3)

a),b),c)

Using a weighted version of the normal objective function, $f(n) = (2 - w)g(n) + wh(n)$, yields the table below.

	start50		start60		start64	
IDA*	50	1462512	60	321252368	64	1209086782
1.2	52	191438	62	230861	66	431033
1.4	66	116174	82	3673	94	188917
1.6	100	34647	148	55626	162	235852
1.8	236	6942	314	8816	344	2529
Greedy	164	5447	166	1617	184	2174

Table 2: Comparison of different algorithms in term of length of solution path, total number of states expanded during the search and length of optimal solution path.

The code changed is displayed below in Listing 3.

```
depthlim(Path, Node, G, F_limit, Sol, G2)  :-
    nb_getval(counter, N),
    N1 is N + 1,
```

```
nb_setval(counter, N1),
write(Node),nl,
s(Node, Node1, C),
not(member(Node1, Path)),
G1 is G + C,
h(Node1, H1),
F1 is (2-w)*G1 + w*H1, % Instead of F1 is G1 + H1
F1 =< F_limit,
depthlim([Node|Path], Node1, G1, F_limit, Sol, G2).
```

Listing 3: The change in idastar.

d)

First we note that optimal solution through iterative deepening A* search has an awful time-complexity whereas a suboptimal solution through greedy search takes literally no time at all in comparison. By weighting the objective function as described above we can move in the space between greedy search and IDA*, in fact w=1 means we are doing IDA* and w=2 Greedy Search. The reason we are not finding optimal solutions is that since w $>$ 1 the heuristic is not admissible any more. Hence as we are increasing the value of w we are exploring less and less nodes and finding a less optimal solution. The trade off between optimality and time complexity is clearly non-linear and as we approach to higher w values and greedy search we are far from the optimal solution. Similarly as we approach lower w values and IDA* we come closer to the optimal path length the time-complexity increases drastically.

4)

a)

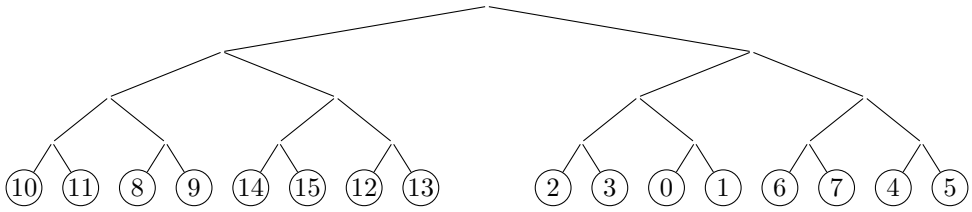

Figure 1: Leaf value assignment such that alpha-beta algorithm prunes as many nodes as possible.

b)

We start by traversing all the way down to 10 since $\alpha < \beta$. The min node above is first assigned 10 and has to check 11 as well since $\alpha = -\infty < 10$. $10 < 11$ and the min node is assigned 10. Going upwards the max node and its α is assigned 10 since it's greater than minus infinity. Since $10 < \beta = \infty$ we have to go down the right child and reach 8. 8 is assigned to the min node parent and its β. But now since the value of the min node is going to be less than or equal to 8 we can prune ($\alpha = 10 > 8 = \beta$), the above max node is not going to pick 8 or less when it can have 10.

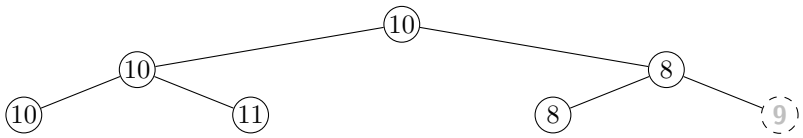

Figure 2: Step 1.

Next lets consider the right sub-tree of the left sub-tree. The beta value is updated to 10 from the previous sub-tree and we start to traverse down. Since $\beta = 10 > -\infty = \alpha$ we traverse al the way down to 14. We do

not update β and can not prune, so we check the next child which is 15. β is still not updated and the min node is assigned the value of 14. The max node is than assigned 14 since it's better then $-\infty$. Now the value of $\alpha = 14 > 10 = \beta$ so we prune. Now the min node can chose between 10 and 14 which means it chooses 10.

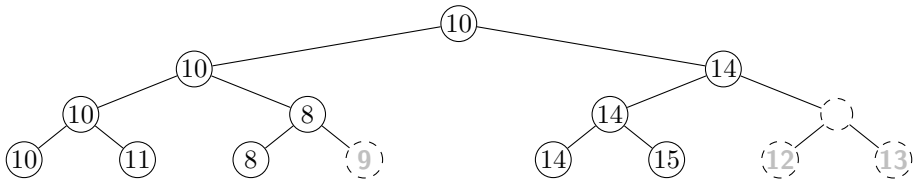

Figure 3: Step 2.

The 10 becomes the new α for the root and we start traversing the right sub-tree. We reach the leaf 2 and updates the min nodes value and β. Now because $\beta = 2 < 10 = \alpha$ we can prune. The two is assigned to the parent max node but α is not updated. Since we can not prune we traverse down to the 0. We update the min nodes value and β. Now because $\beta = 0 < 10 = \alpha$ we can prune. The parent max node is updated. The parent min node and its β is updated to 2. Since $2 < 10$ we can prune and assign the root to 10.

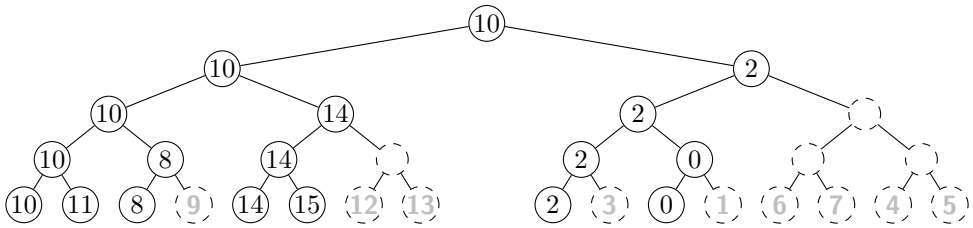

Figure 4: Final tree.

I.e of the 16 leaves only 7 is evaluated.

c)

For max nodes we want to visit the best child first so we don't have to waste time exploring worse scenarios. For min nodes we want to visit, from max point of view, the worst child first. Pruning causes all the rest of the children to be pruned away at every other level of the tree.

With reference to Figure 5 assume that we first traverse down to the leftmost deepest leaf and find the best possible value (X) for the root max node, and the best possible value for the parent min node. Hence after exploring the other two leaves we assign this value to the parent min node and next to the parent max node. Now since the the max node has the best possible value we only have to check two leaves in the two different sub-trees and prune the rest since the values from there is going to be $<$ X. Next we go down the middle sub-tree of the leftmost sub-tree and by the same reasoning we can prune after just visiting the three first leaves. The same applies for the right sub-tree of the leftmost sub-tree.

The root and the α is now assigned the value of X. As we move down the middle sub-tree and reach the leftmost leaf, this leaf will be less then X so we can prune. As the value is less than X we do not update the max node above. Instead we move down the next sub-tree which leads to the same pruning. The max node picks the highest value of these ($<$ X) and that value is assigned to the min root of the middle sub-tree. Now since the β is less than $\alpha = X$ we can prune. The same reasoning applies to the right sub-tree. The final tree is displayed below with the pruned branches dashed.

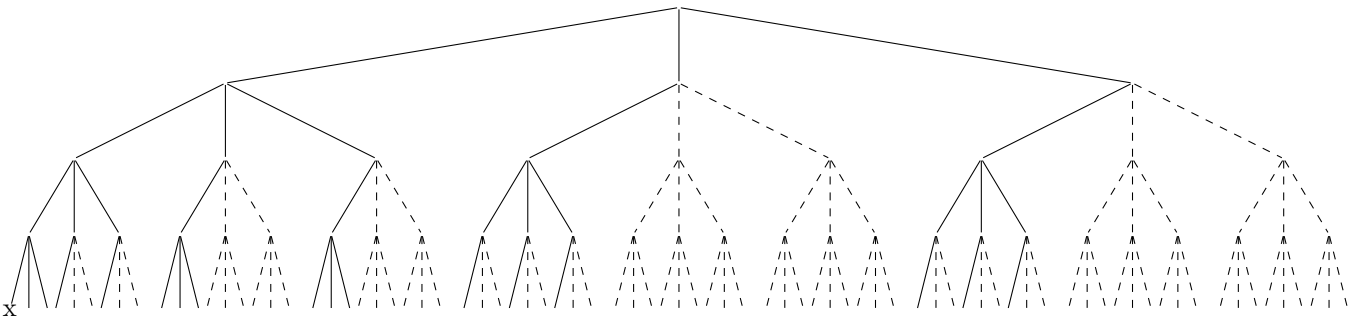

Figure 5: Final tree.

I.e of the 81 leaves only 17 is evaluated.

d)

From the previous two examples we notice that when we perform optimal pruning we only have to expand every node at every other depth (with the exception of the first sub-tree). In our case this means that we only have to expand all children when we're at an max node and only one when we're at a min node. To generalize: All of the first players children has to be expanded to find the best one. For each of these only the best second players children has to be expanded to disprove the best (and first) first player value.

Hence for a tree with constant branching factor b and depth d the time complexity is $O(b*1*b*1*b*1....) = O(b^{d/2})$, regardless if d is even or odd.