Assignment 1 - Prolog and Search; Part 2 (T1, 2020)
Completed by Dheeraj Satya Sushant Viswanadham (z5204820)
Note: Have used the lecture notes for gathering information regarding time and space complexities of given algorithms.
Started: 07/03/2020 | Last edited: 15/03/2020

Question 1: Search Algorithms for the 15-Puzzle

A)

	start10	start12	start20	start30	start40
UCS	2565	Mem	Mem	Mem	Mem
IDS	2407	13812	5297410	Time	Time
A*	33	26	915	Mem	Mem
IDA*	29	21	952	17297	112571

Note: "Mem" refers to the algorithm running out of memory
Note: "Time" refers to the algorithm taking more than 5 mins to compute

B)
As seen from the computations shown in the above table:

1) UCS can be seen to be the least efficient of the four as it runs out of memory for start12 to start40 positions. UCS minimises the cost of path from start to node n, and is optimal and complete, however is not

Cost of a path is the sum of the costs of its arcs:

$$cost(\langle n_0, \cdots, n_k \rangle) = \sum_{i=1}^{k} cost(\langle n_{i-1}, \cdots, n_i \rangle)$$

Time complexity: Worst case, $O(b^{\lceil C^*/\epsilon \rceil})$ where C^* = cost of the optimal solution and assume every transition costs at least ϵ

Space complexity: $O(b^{\lceil C^*/\epsilon \rceil})$, $b^{\lceil C^*/\epsilon \rceil} = b^d$ if all step costs are equal

2) IDS can be seen to be slightly better than UCS, however, it is still not great as it takes too long to compute the calculations once the starting positions get bigger. It attempts to combine the benefits of depth-first (low memory) and breadth-first (optimal and complete).

Time: $O(b^d)$

Space? $O(bd)$

$$(d+1)b^0 + db^1 + (d-1)b^2 + \cdots + 2 \cdot b^{d-1} + 1 \cdot b^d = O(b^d)$$

3) A* can be seen to be slightly better as it did not take too long to compute the calculations, however, it still has its own memory limitations. It uses both the cost of the generated path and an estimate to the goal to order the nodes on the frontier, and combines UCS and greedy search to accomplish this where UCS minimises the cost of path whilst greedy search minimises the estimate to the goal from 'n'. However, it maintains a priority queue which can get quite big, hence introducing memory limitations.

A* Search $f(n) = g(n) + h(n)$ (cost from start to n plus estimated cost to goal)

Time complexity: O(b^ėd) Space complexity: O(b^d)

4) IDA* can be seen to be the most efficient both in terms of memory usage and time taken to compute the differing positions. This is due to it requiring lower memory usage than the A* algorithm where both perform depth-first search but differ in the fact that IDA* stops the search when it reaches its current threshold [from the sum of -> **f(n) = g(n) + h(n)**].

Time complexity: O(b^ėd) Space complexity: O(bd)

Question 2: Heuristic Path Search for 15-Puzzle

(A) + (C)

	start50		start60		start64	
IDA*	50	14642512	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
Greedy	164	5447	166	1617	184	2174

B) The heuristic path algorithm is a best-first search algorithm in which the objective function is:

$$f(n) = (2 - w) \cdot g(n) + w \cdot h(n), \text{ where } 0 \leq w \leq 2$$

Hence, we should adjust the code as shown below where w = 1.2, meaning F(1) = (2 - 1.2) * G1 + 1.2 * H1 => 0.8 * G1 + 1.2 * H1

Before:

```
idastar(Start, F_limit, Solution, G) :-
    write(F_limit),nl,
    F_limit1 is F_limit + 2,  % suitable for puzzles with parity
    idastar(Start, F_limit1, Solution, G).

% depthlim(Path, Node, Solution)
% Use depth first search (restricted to nodes with F <= F_limit)
% to find a solution which extends Path, through Node.

% If the next node to be expanded is a goal node, add it to
% the current path and return this path, as well as G.
depthlim(Path, Node, G, _F_limit, [Node|Path], G)  :-
    goal(Node).

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

After:

```
    depthlim([], Start, 0, F_limit, Solution, G).

idastar(Start, F_limit, Solution, G) :-
    write(F_limit),nl,
    F_limit1 is F_limit + 2,  % suitable for puzzles with parity
    idastar(Start, F_limit1, Solution, G).

% depthlim(Path, Node, Solution)
% Use depth first search (restricted to nodes with F <= F_limit)
% to find a solution which extends Path, through Node.

% If the next node to be expanded is a goal node, add it to
% the current path and return this path, as well as G.
depthlim(Path, Node, G, _F_limit, [Node|Path], G)  :-
    goal(Node).

% 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,     % Formula: (2 - w) * G1 + w * H1
    F1 =< F_limit,
    depthlim([Node|Path], Node1, G1, F_limit, Sol, G2).
```

D) Basically, as seen from the above table in (A), it is essentailly a tradeoff between speed and quality. If w = 1, then that means the quality of the solution will be better as the path length is shorter meaning we will get a more optimal path, whilst if we incrementally increase w by 0.2 from 1 to 2, this means that the quality of the solution is being sacrificed for speed i.e. as w approaches 2, we have: f(n) = (2 - 2) * g(n) + 2 * h(n) = 2 * h(n) where the greedy algorithm is f(n) = h(n) in its simplest form, whereas if we have w = 1, then we have the IDA* algorithm: f(n) = (2 - 1) * g(n) + 1 * h(n) = g(n) + h(n) in its simplest form.