Part2

Abanob Tawfik
z5075490

March 2020

1 Question 1

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, start12, start20, start30 and start40. 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. Note that you will need to re-start prolog each time you switch to a different search.

Solution:

Table 1 the number of nodes expanded to reach goal state with each search method

	UCS	IDS	A*	IDA*
Start10	2565	2407	33	29
Start12	MEM	13812	26	21
Start20	MEM	5297410	915	952
Start30	MEM	TIME	MEM	17297
Start40	MEM	TIME	MEM	112571

b) **b) Briefly discuss the efficiency of these four algorithms (including both time and memory usage).**

Solution:
The UCS search was a variation of Dijkstra which is an uninformed search, in terms of time it was the slowest, and ran out of memory on every single starting position except 10. This is because each path being expanded is added to the queue and we store all paths branching increasing size exponentially.

- Time: O($b^{[C*/\epsilon]}$) (note $[C*/\epsilon]$ = d when all transitions have same cost. d is depth, $C*$ is the cost of the optimal solution and ϵ is the minimum cost of transition)
- Space: O($b^{[C*/\epsilon]}$) (note $[C*/\epsilon]$ = d when all transitions have same cost. d is depth, $C*$ is the cost of the optimal solution and ϵ is the minimum cost of transition)

The IDS search would run around the same speed as UCS, but it would never run out of memory. It was memory efficient and the only drawback was time taken for larger depths, in the cases of TIME it will reach depth 20 in the first 30 seconds, and then spend over 5 minutes getting to depth 23, and with enough time it would have found the solution. However note this search is the slowest search overall in comparison to the rest

- Time: O(b^d)
- Space: O($b * d$)

The A* search was an improved version of UCS, it was quick (second fastest with this data) however it was drawn back by memory usage, A* would weed out inefficient paths meaning it could go deeper than UCS and avoid storing useless paths in memory, however it suffers from the same drawback of running out of memory because each path being expanded is still added to the queue and we store all paths branching increasing size exponentially. Due to the informed nature of the search it would not expand down inefficient paths however and it is the fastest search in theory amongst the four.

- Time: depends on the heuristic O(b^d) is worst case when heuristic = 0
- Space: O(b^d)

The IDA* search was the fastest search and also the only search able to complete Start40 under 5 minutes (it only took 30 seconds). The search was linear in memory usage, none of the cases would cause out of memory issues, it was a combination of A* speed and IDS memory efficiency. Note assuming infinite memory, this search is slower than A* and only in the data provided it happened to appear faster for the start20 case.

- Time: depends on the heuristic O(b^d) is worst case when heuristic = 0
- Space: O($b * d$)

In summary, UCS and IDS are both uninformed meaning their time efficiency was slower than the informed searches and should only be used if there is no domain specific knowledge. In the case of UCS, it suffered memory issues. IDS suffered no memory issues but at larger depths it was

extremely time inefficient and would timeout. A* and IDA* are both informed searches meaning they are marginally more time efficient and should always be used over the uninformed searches (with an admissible heuristic of course). A* suffers the same memory issues as UCS since it is a breadth first search that utilizes a heuristic. IDA* didn't suffer any memory issues and was the fastest search from the data used. IDA* can be slower than A* at larger depths and take a long time to find a solution, however it does not suffer from memory problems.

2 Question 2

a) **(a) Run [greedy] forstart50, 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).**

Solution:

Table 2 Path length and nodes expanded for greedy search

	Start50		Start60		Start64	
IDA*	50	14642512	60	321252368	64	1209086782
w = 1.2	52	191438	62	230861	66	431033
w = 1.4	66	116342	82	4432	94	190278
w = 1.6	100	33504	148	55626	162	235848
Greedy	**164**	**5447**	**166**	**1617**	**184**	**2174**

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 Weak 2 Tutorial Exercise, with w = 1.2 . In your submitted document, briefly show the section of code that was changed, and the replacement code.**

Solution:
using the evaluation function, $f(n) = (2 - w) \cdot g(n) + w \cdot h(n)$, where $0 \leq w \leq 2$, with w = 1.2 gives us the following function to use for measuring cost $f(n) = (0.8) \cdot g(n) + 2 \cdot h(n)$. The only modifications made to the code was changing the line that computed the evaluation function shown in Figure 1.

```
% 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,          % ONLY CHANGED LINE OF CODE
    F1 =< F_limit,
    depthlim([Node|Path], Node1, G1, F_limit, Sol, G2).
```

Figure 1: The only line of code changed to create heuristic.pl

The full code for heuristic.pl is shown in Figure 2.

```
% Iterative Deepening A-Star Search

% COMP3411/9414/9814 Artificial Intelligence, UNSW, Alan Blair

% solve(Start, Solution, G, N)
% Solution is a path (in reverse order) from Node to a goal state.
% G is the length of the path, N is the number of nodes expanded.
solve(Start, Solution, G, N)  :-
    nb_setval(counter,0),
    idastar(Start, 0, Solution, G),
    nb_getval(counter,N).

% Perform a series of depth-limited searches with increasing F_limit.
idastar(Start, F_limit, Solution, G) :-
    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,          % ONLY CHANGED LINE OF CODE
    F1 =< F_limit,
    depthlim([Node|Path], Node1, G1, F_limit, Sol, G2).
```

Figure 2: full program for heuristic.pl

c) **Run [heuristic] on start50, 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, 1.6 ; in each case, run the algorithm on the same three start states and record the values of G and N in your table.**

Solution:

Table 3 Path length and nodes expanded for heuristic path search

	Start50		Start60		Start64	
IDA*	50	14642512	60	321252368	64	1209086782
w = 1.2	**52**	**191438**	**62**	**230861**	**66**	**431033**
w = 1.4	**66**	**116342**	**82**	**4432**	**94**	**190278**
w = 1.6	**100**	**33504**	**148**	**55626**	**162**	**235848**
Greedy	164	5447	166	1617	184	2174

d) **Briefly discuss the tradeoff between speed and quality of solution for these five algorithms.**

Solution:

What can be observed from IDA* which was the slowest to compute, expanding the most nodes, is that it always found the optimal solution and didn't over-shoot the shortest path. From the three algorithms involving the heuristic path search weighting, what can be observed is that as the weighting is increased the length of the path had increased, however the number of nodes expanded was dramatically decreased. As the weighting increased, the evaluation cost would use less of the actual cost of the path and rely more on the heuristic cost of the path giving us a quicker solution however the quality of the solution would be worse. The greedy search was by far the fastest in terms of expanding the least number of nodes, however the solution given was the lowest in quality, having a path length of 164 for start50. There is a direct correlation between speed of the algorithm and quality of the solution, the faster the solution was (expanding less nodes), the worse the quality of the solution was (longer paths), note the change in speed does not have to do with the algorithm but the evaluation cost of paths. It can be noted that when w is closer to 1, we get a vanilla IDA* search, and as it approaches 2 the search becomes greedier, using less of the actual cost of the path, at w = 1.8 the quality and time of the solution in fact is worse than greedy, with G = 240 and N = 35557. In summary
Ranking Speed

1. Greedy
2. Heuristic Path Search with w = 1.6
3. Heuristic Path Search with w = 1.4
4. Heuristic Path Search with w = 1.2

5. IDA*

Ranking Quality of Search

1. IDA*
2. Heuristic Path Search with w = 1.2
3. Heuristic Path Search with w = 1.4
4. Heuristic Path Search with w = 1.6
5. Greedy

The graphs in Figure 3, 4 and 5 display the algorithms in terms of quality vs speed, and as can be noticed, the higher the quality (smaller path length) the more time was taken to perform the search (more nodes expanded). There were a few anomalies such as start60 and start64 using the heuristic path search algorithm with w = 1.4 and w = 1.6, where the length of the path is longer (lower quality solution) and more time was taken to complete the search (more nodes expanded). Generally lower quality solutions are faster to generate than more optimal solutions as the heuristic is no longer admissible after w = 1, we rely less on the actual cost of the path and over-estimate with the heuristic cost giving us worse quality solutions. however these solutions reach goal state much quicker.

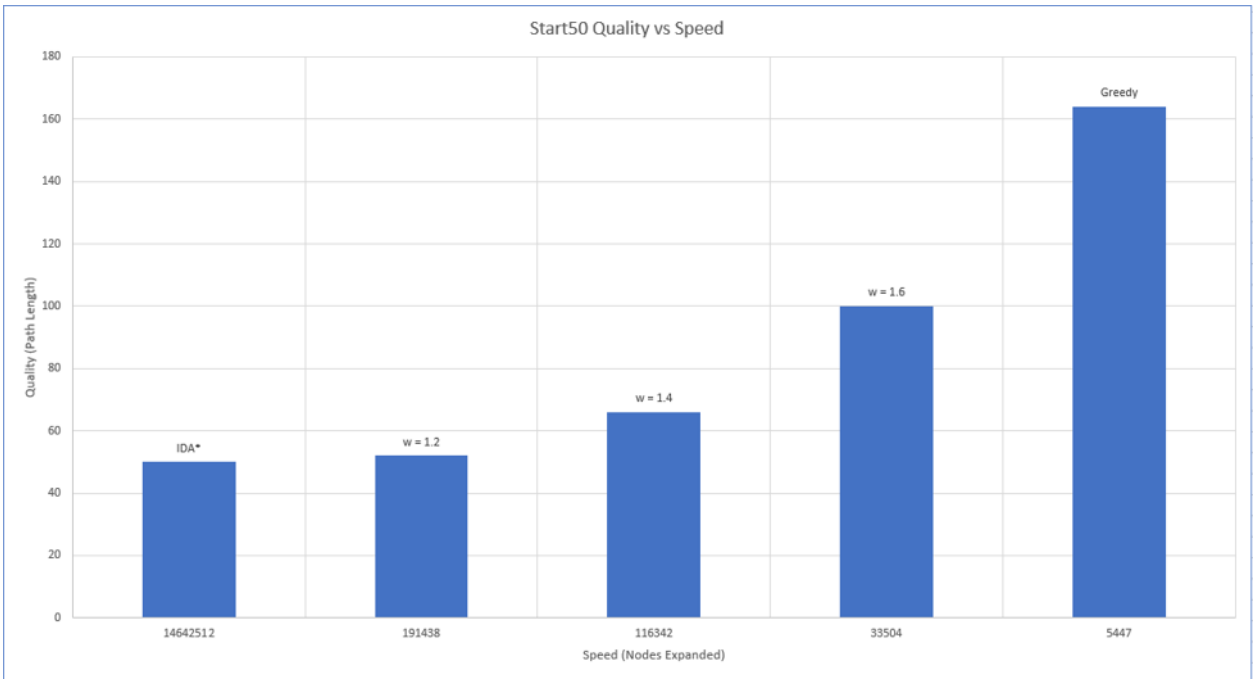

Figure 3: Quality vs Time graph for start50

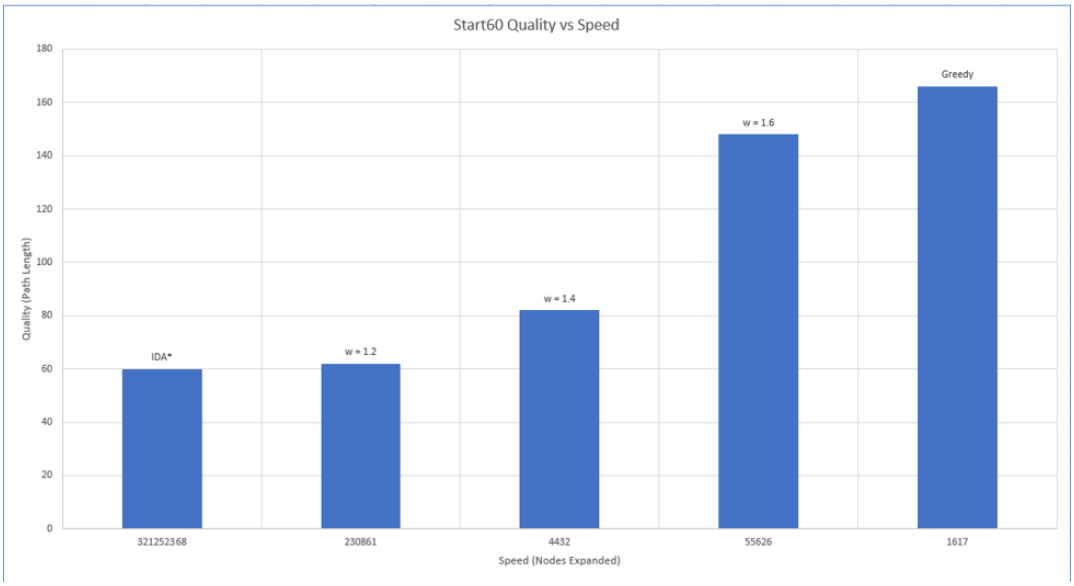

Figure 4: Quality vs Time graph for start60

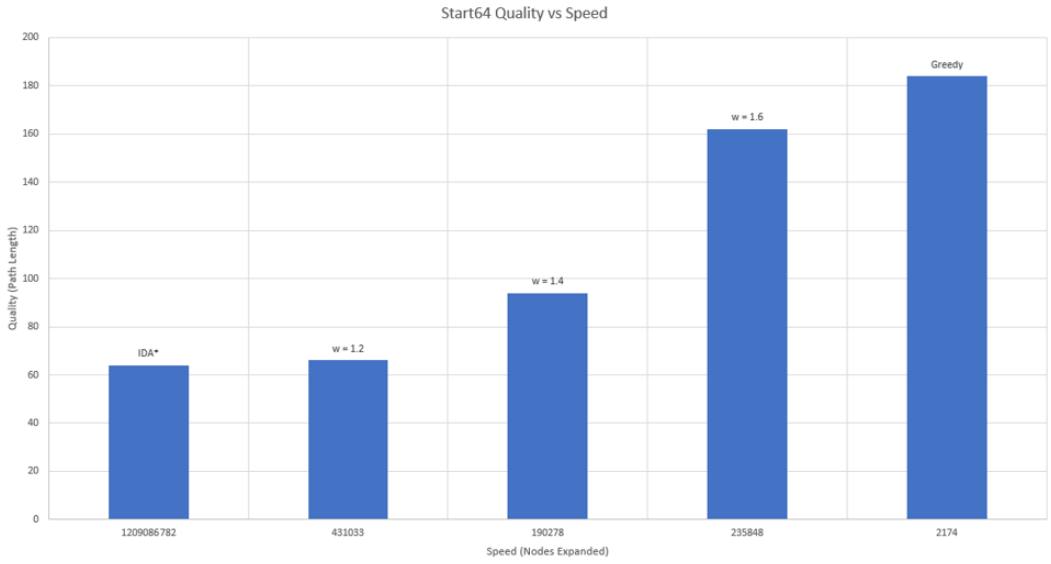

Figure 5: Quality vs Time graph for start64