

**CZ3005 Artificial Intelligence**

Lab 1 Report

By:

Zhang Yuhan

U1823060F

yzhang152@e.ntu.edu.sg

Group SSP2

19 February 2020

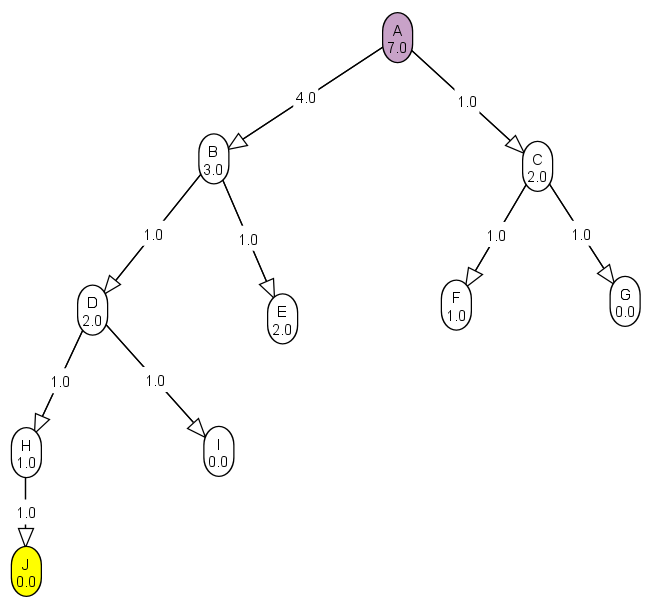
Q1a) Give a graph where depth-first search (DFS) is much more efficient (expands fewer nodes) than breadth-first search (BFS).

Figure 1a: Graph where DFS is more efficient than BFS

Both depth first search (DFS) and breadth first search (BFS) are algorithms that systematically traverses a tree in order to search for the goal node. A search is considered complete when the goal node is found, and the path taken from the start node to the goal node is returned. The DFS algorithm begins searching from the start node and expands as far as possible down one branch of the tree before backtracking and going another. Because of this, DFS will generally perform well when the goal node is located deep in the tree. On the other hand, the BFS algorithm will expand all nodes of its current depth before going to a lower level. Therefore, BFS generally performs better when the goal node is located shallow within the tree.

The efficiency of the different search algorithms for graph in Figure 1d are as follows:

DFS:

Path found: A (Start) -> B -> D -> H -> J (Goal)

Path costs: 7.0

Nodes expanded: 5

BFS: A (Start) -> B -> D -> H -> J (Goal)

Path costs: 7.0

Nodes expanded: 10

As seen in Figure 1a, the goal node K is located deep within the tree. The DFS algorithm is able find the goal node during its first expansion along the tree, while the BFS algorithm needs to expand all nodes on every level before finding the goal node. The nodes expanded for BFS far exceeds that of DFS for this particular situation. Therefore, DFS is much more efficient than BFS for this example.

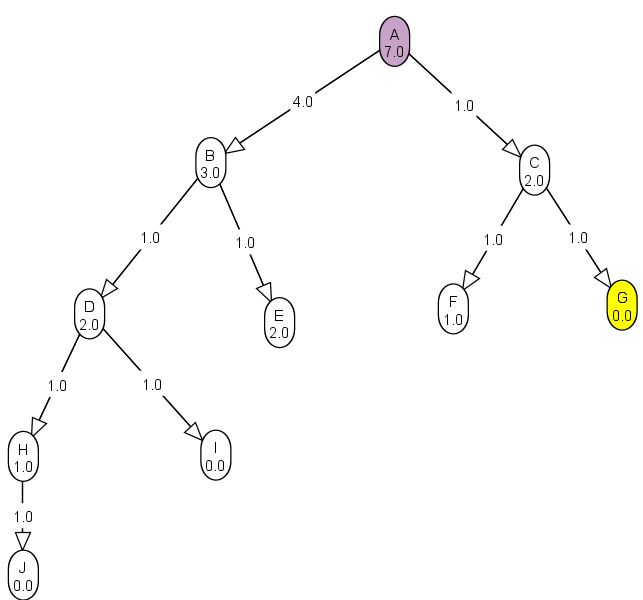
Q1b) Give a graph where BFS is much better than DFS.

Figure 1b: Graph where BFS is more efficient than DFS

The efficiency of the different search algorithms for graph in Figure 1d are as follows:

DFS:

Path found: A (Start) -> C -> G (End)

Path costs: 2.0

Nodes expanded: 10

BFS:

Path found: A (Start) -> C -> G (End)

Path costs: 2.0

Nodes expanded: 7

As seen in Figure 1b, the goal node D is located at a very shallow depth within the tree and along the right-most branch. Since the DFS algorithm used in AISpace expands each branch from left to right, the branch with the goal node the last branch the algorithm expands. On the other hand, BFS is able to find the goal node very efficiently since its located very shallow in the tree. The nodes expanded for DFS exceeds that of BFS. Therefore, BFS is more efficient than DFS in this example.

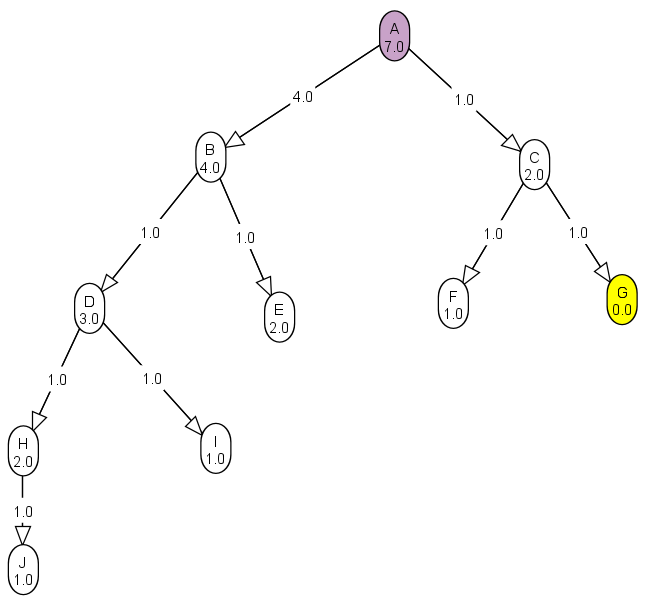
Q1c) Give a graph where A\* search is more efficient than either DFS or BFS.

Figure 1c: A\* search

The A\* search algorithm takes into account both the edge costs and heuristic costs of each node in order to decide which node in the frontier to expand first. It aims to expand the node that minimizes the result of the function: f(n) = g(n) + h(n), where g(n) is the actual cost of the path from the start node to n, and h(n) is the heuristic estimate of the cheapest path from n to the goal node. The A\* algorithm is always complete (guaranteed to return a solution given that one exists) and optimal given that its heuristics are admissible (returned solution must be optimal) and monotonic (heuristics must never overestimate edge costs).

The efficiency of the different search algorithms for graph in Figure 1d are as follows:

DFS:

Path found: A (Start) -> C -> G (Goal)

Path costs: 2.0

Nodes expanded: 10

BFS:

Path found: A (Start) -> C -> G (Goal)

Path costs: 2.0

Nodes expanded: 7

A\*:

Path found: A (Start) -> C -> G (Goal)

Path costs: 2.0

Nodes expanded: 3

As can be seen in Figure 1c, the edge costs and heuristic estimates of each node is not constant across all nodes. Since the A\* algorithm takes into account both path costs and heuristics, it decides to expand the node with the least path costs + heuristic estimations. This allows A\* to expand the C node first and find the goal node G immediately after, avoiding the expansion of unnecessary nodes. Therefore, the A\* algorithm is much more efficient than both DFS and BFS in this example.

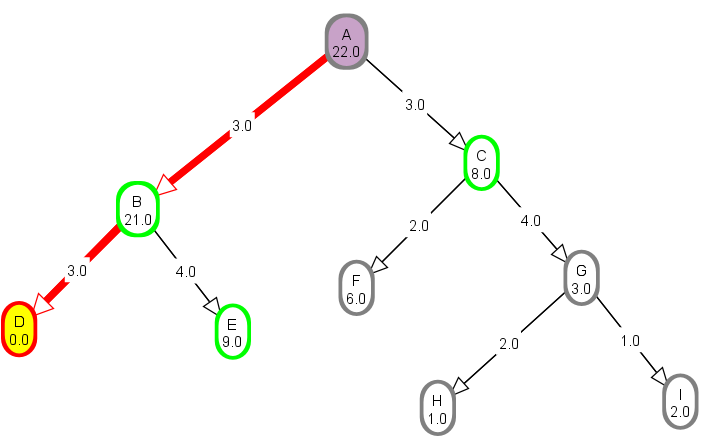
Q1d) Give a graph where DFS and BFS are both more efficient than A\* search.

Figure 1d: DFS and BFS are more efficient than A\*

The efficiency of the different search algorithms for graph in Figure 1d are as follows:

DFS:

Path found: A (Start) -> B -> D (End)

Path costs: 6.0

Nodes expanded: 3

BFS:

Path found: A (Start) -> B -> D (End)

Path costs: 6.0

Nodes expanded: 4

A\*:

Path found: A (Start) -> B -> D (End)

Path costs: 6.0

Nodes expanded: 8

It is clear from the statistics above that both BFS and DFS are more efficient at finding the goal node than A\* for the tree in Figure 1d. This is because the A\* algorithm will always expand the node with the lowest f(n) in the frontier, and the heuristics of the nodes leading to the goal node are heavily overestimated.

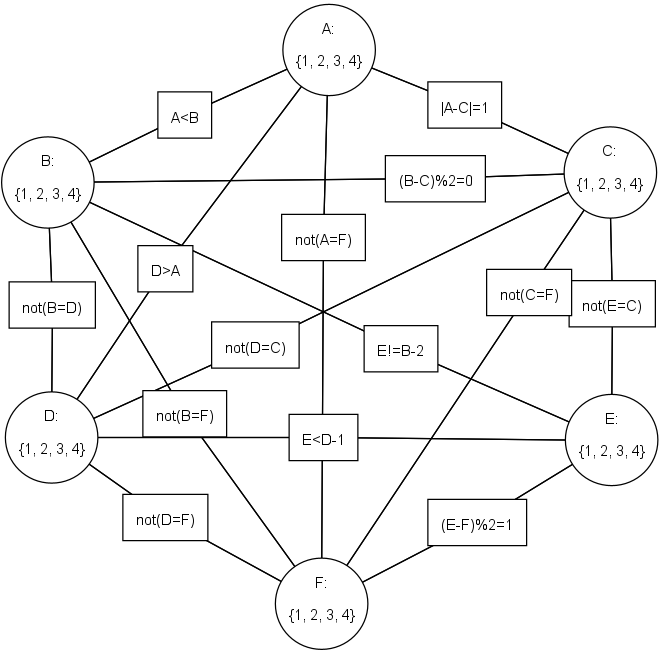
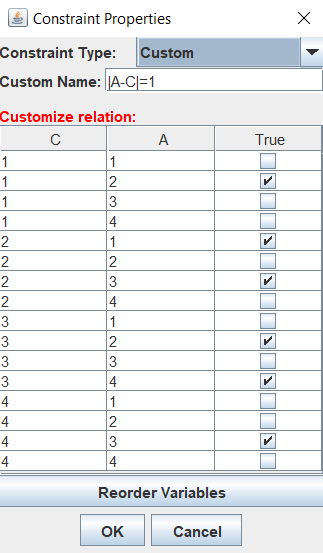
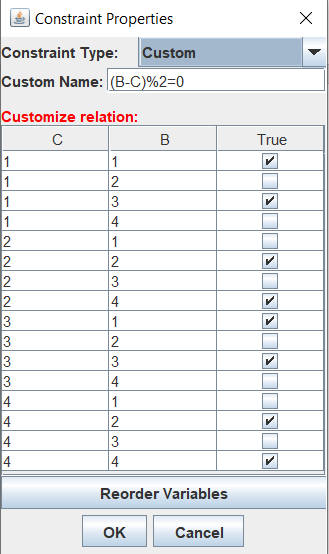
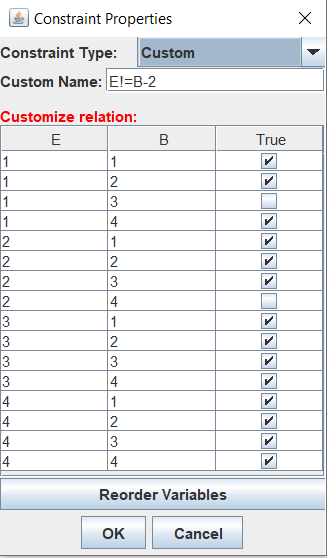
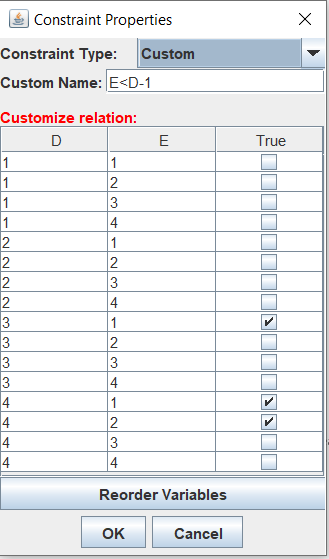
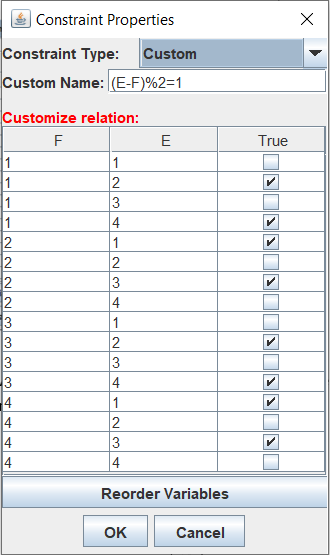
Q2a) Draw this graph in AIspace.org as a CSP problem (constraint graph).

Figure 2a: Constraint Graph

Truth tables for constraints:



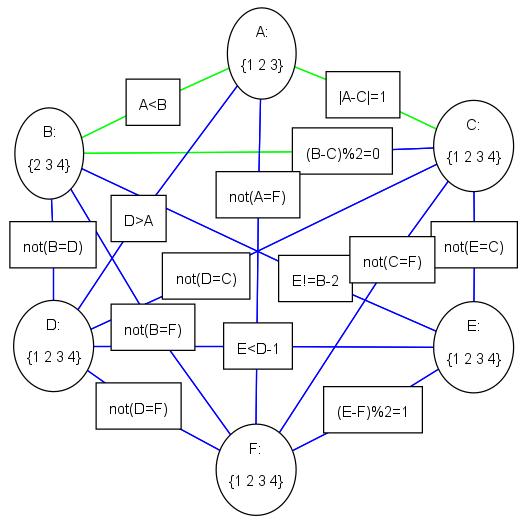
Q2b) For the first 5 instances of arc consistency, show which elements of a domain are deleted at each step, and which arc is responsible for removing the element.

Figure 2b: Constraint Graph after 5 instances of arc consistency

Table 1: Checks, results, and actions made at each instance of arc consistency for the first 5 instances.

|  |  |  |
| --- | --- | --- |
| Instance | Checks/Results | Actions |
| 1 | Selected arc: ( B, A<B ) to check.  Arc ( B, A<B ) is inconsistent. | 1 removed from the domain of B because of arc ( B, A<B )  New domain of B = {2,3,4} |
| 2 | Selected arc: ( A, A<B ) to check.  Arc ( A, A<B ) is inconsistent. | 4 removed from the domain of A because of arc ( A, A<B )  New domain of A = {1,2,3} |
| 3 | Selected arc: ( A, |A-C|=1 ) to check.  Arc ( A, |A-C|=1 ) is consistent. | No changes are made to domain of A. |
| 4 | Selected arc: ( C, |A-C|=1 ) to check.  Arc ( C, |A-C|=1 ) is consistent. | No changes are made to domain of C. |
| 5 | Selected arc: ( B, (B-C)%2=0 ) to check.  Arc ( B, (B-C)%2=0 ) is consistent. | No changes are made to domain of D. |

Domains of variables after the first 5 instances:

A: {1,2,3}

B: {2,3,4}

C: {1,2,3,4}

D: {1,2,3,4}

E: {1,2,3,4}

F: {1,2,3,4}

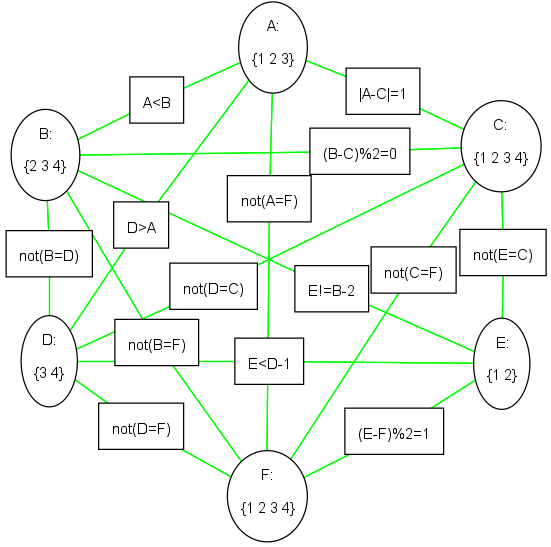
Q2c) Show explicitly the constraint graph after arc consistency has stopped.

Figure 2c: Constraint graph after arc consistency has stopped

Domains of variables after arc consistency has stopped:

A: {1,2,3}

B: {2,3,4}

C: {1,2,3,4}

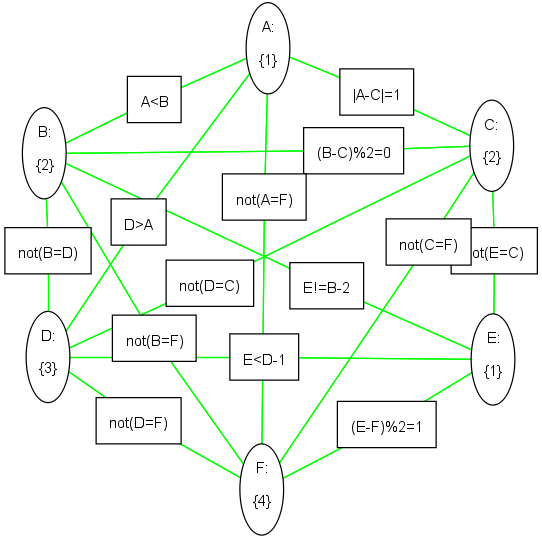
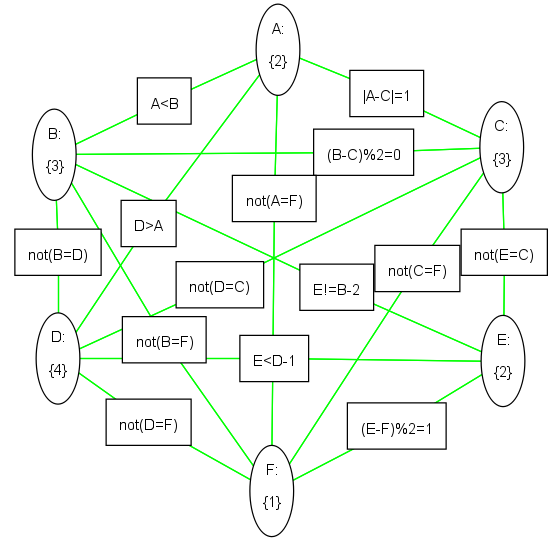
D: {3,4}

E: {1,2}

F: {1,2,3,4}

Q2d) Show how splitting domains can be used to solve this problem. Draw the tree of splits and show the solutions.

Domain splitting is a way of breaking down a CSP into smaller problems in order to simplify each problem. The domain with the smallest size is usually split first since it generally leads to less branches in the tree of splits. Subsequently, arc consistency will run again to search for solutions to the sub problems of the original CSP. A solution is found when there is only one element in the domain of each variable, in which case the algorithm will backtrack to find solutions of other subsets. If there are one or more variables with more than one element, a solution has not yet been found and domain splitting will be called again for the sub problem.



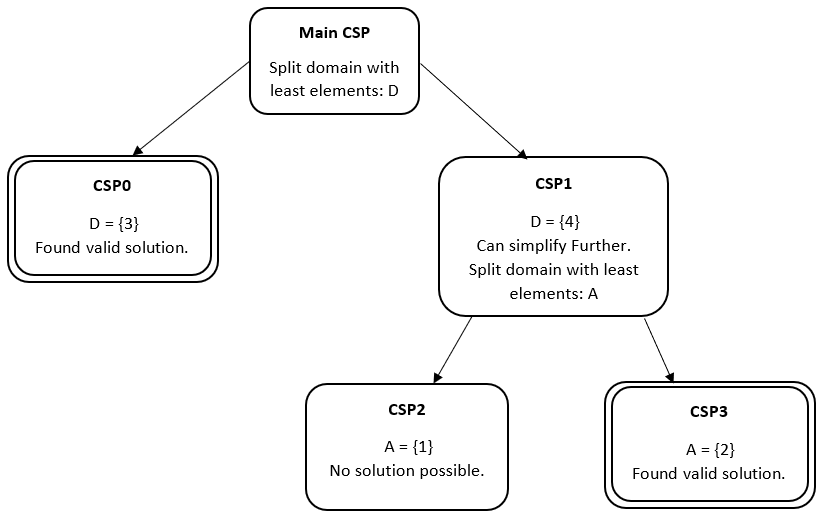
Figure 2d1: Solutions of CSP

Figure 2d2: Tree graph of domain splitting

Domain Splitting History:

D in {3}

Solution found: A = 1, B = 2, C = 2, D = 3, E = 1, F = 4

D in {4}

A in {1}

Cannot split variable A

A in {2}

Solution found: A = 2, B = 3, C = 3, D = 4, E = 2, F = 1

In order to solve this problem, the output from part 2c was taken. Then, the variable with the least number of elements (not including those with only 1 element) is chosen to have its domain split in half. In this case, variable D was chosen and split into {3} and {4}. Subsequently, arc consistency was run again and found the first solution when D was {3}. In order to test if other branches also lead to a solution, the algorithm backtracked and ran the arc consistency for when D was {4}. However, this did not lead to an immediate solution. Therefore, A was chosen for domain splitting again. In the case where D was {4} and A was {1}, a solution was not found since some domains returned to be empty. Finally, the algorithm backtracked again and found the second solution when D was {4} and A was {2}. This procedure can be seen in the domain splitting history section above.

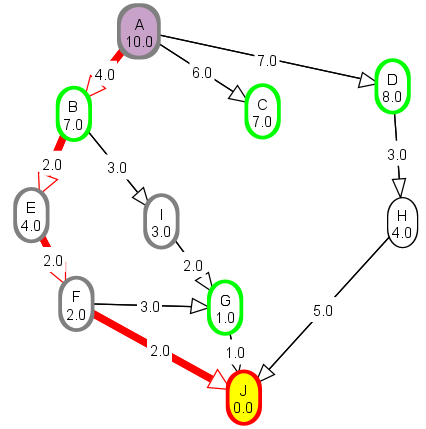
Q3a) What is the effect of reducing h(n) when h(n) is already an underestimate?

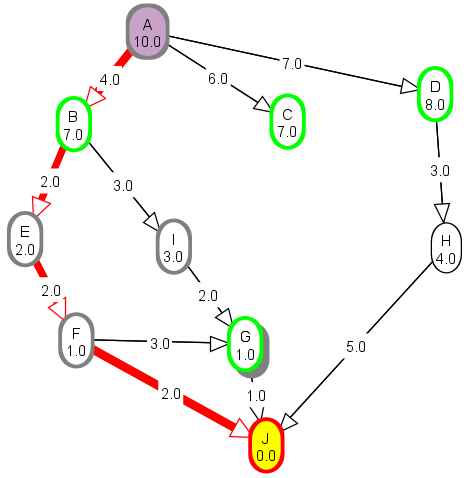
Figure 3a1: Graph for A\* search

Reducing h(n) when its already an underestimate may have different effects on the A\* search algorithm depending on where the node being reduced is. The graph in Figure 3a serves as a base graph for this problem, and the optimal path of this graph found by A\* was:

Path: A (Start) -> B -> E -> F -> J (Goal)

Path costs: 10.0

Nodes expanded: 7

Figure 3a2: h(n) reduced for variables on the original optimal path

If the h(n) of a node on the optimal path was reduce, the optimal path would not change. However, in some cases, it may lead to less nodes expanded before the path is found and thus more efficiency. This can be observed in Figure 3a2 when node E is reduced to 2 and F is reduced to 1:

Path: A (Start) -> B -> E -> F -> J (Goal)

Path costs: 10.0

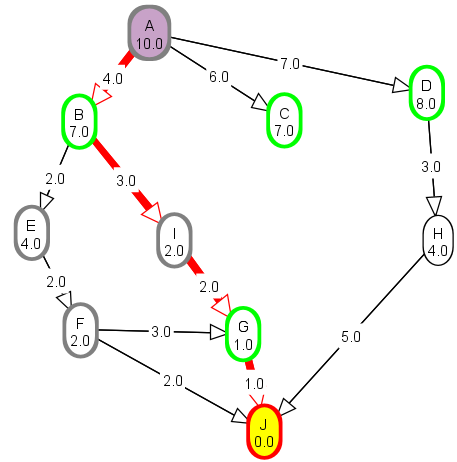
Nodes expanded: 6

Figure 3a3: h(n) reduced for variables not on original optimal path

If h(n) of a node not on the original optimal path was reduced, it may lead to the discovery of a new optimal path that is either more efficient than or just as efficient as the previous. This can be observed in Figure 3a3 where node I was reduced from 3 to 2. A new optimal path was subsequently found by the algorithm.

Path: A (Start) -> B -> I -> G -> J (Goal)

Path costs: 10.0

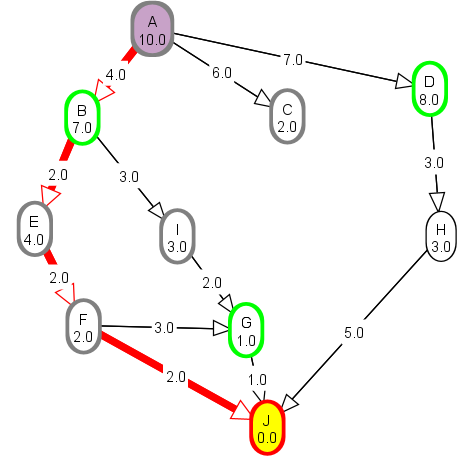
Nodes expanded: 7

Figure 3a4: h(n) reduced for variables not on original optimal path

Another situation may occur if variables not on the original optimal path is reduced where the optimal path is not affected at all. For example, in Figure 3a4, node C was reduced from 7 to 6, but it does not have a path that lead to the goal node, thus it has no way of effecting the optimal path. However, it is able to make the algorithm slightly less efficient since the A\* algorithm will expand it first due to its low f(n) value. On the other hand, the node H has a path that leads to the goal node. However, its large path costs make its path less efficient than the optimal path regardless of its h(n).

Path: A (Start) -> B -> I -> G -> J (Goal)

Path costs: 10.0

Nodes expanded: 8

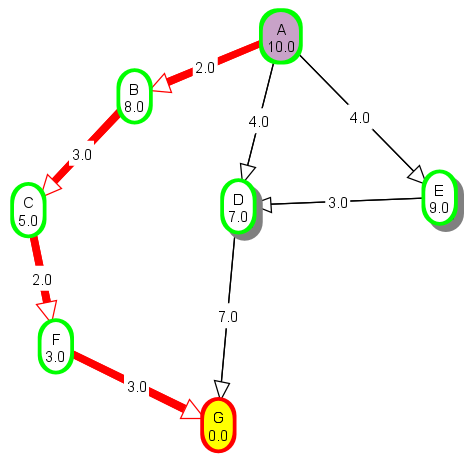
Q3b) How does A\* perform when h(n) is the exact distance from n to a goal?

Figure 3b: A\* search on graph where h(n) of all nodes are exact distance from n to goal

If the h(n) of all nodes are exactly the distance from n to goal, the A\* algorithm will only expand the nodes on the optimal path. This is because the nature of the algorithm finds and expands the node which minimized its f(n). Since f(n) = h(n) + g(n), g(n) is already known from expanding previous nodes and h(n) is the exact distance left to reach the goal (never underestimates or overestimates), the node with the smallest f(n) value will definitely be on the optimal path. Thus, the A\* algorithm will never expand any node that is not on the optimal path. For example, in the graph in Figure 3b, the optimal path is:

A (Start) -> B -> C -> F -> G (goal), where the f(n) for each node is always 10.

For another path on the same graph:

A (Start) -> D -> G (goal), the f(n) for each node is 11, thus none of its nodes were expanded.

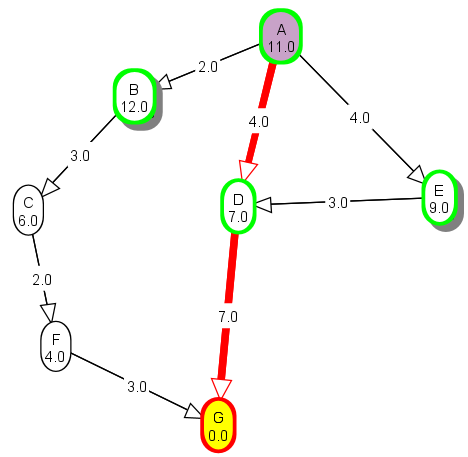
Q3c) What happens if h(n) is not an underestimate?

Figure 3c: Graph where h(n) overestimates for some nodes

One of the conditions of the optimality of the A\* search algorithm is that the heuristics must be admissible. This means that h(n) must never exceed the actual path cost from n to goal. If the admissibility of A\* is violated, it will still return a valid path from start to goal given that one exists. However, there is no guarantee than the path return will be the most optimal path (path with least total edge costs).

For example, on the graph in Figure 3c, nodes B, C, and F are slightly overestimated. Because of this, the A\* algorithm return a different path as it attempts to find the optimal path:

Return path: A (Start) -> D -> G (Goal)

Total path cost: 11.0

As can be observed from the graph with the same edge costs in Figure 3b, the path return by graph in Figure 3c is not the optimal path. The total path cost was found to be 11 when the actual optimal path had a path cost of 10. Thus, the A\* algorithm when there are overestimated nodes is complete but not necessarily optimal.

Q4)

Question 1: 2 Hours

Question 2: 3 hours

Question 3: 2 hours