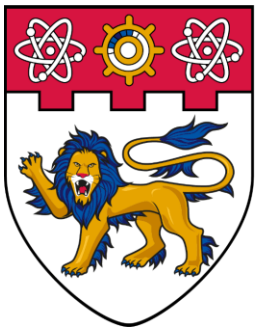

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ3005 ARTIFICIAL INTELLIGENCE

LAB EXERCISE 1

LAB REPORT

Shearman Chua Wei Jie (U1820058D)

LAB GROUP: TSP1

Question 1

For each of the following, give a graph that is a tree (there is at most one arc into any node), contains at most 15 nodes, and has at most two arcs out of any node.

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

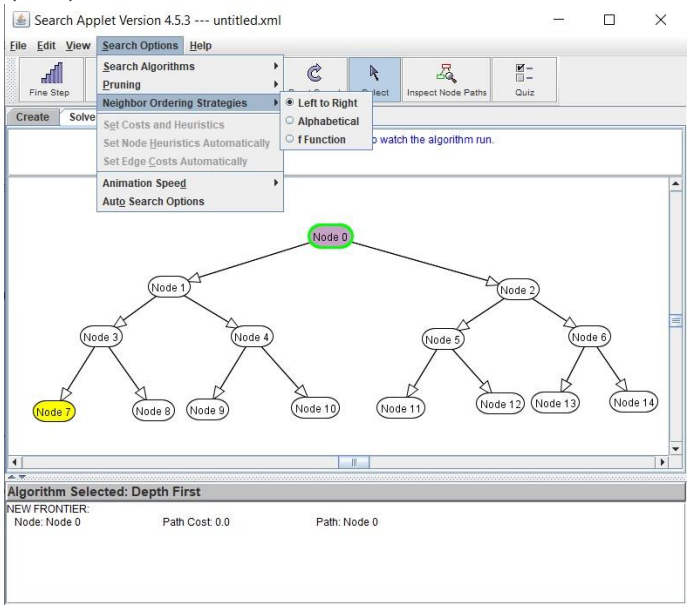

Fig.1

For this question, we have a graph where:
Start Node: Node 0
Goal Node: Node 7
Branching Factor: 2
For this graph, the edge cost for each edge is the same with cost 1.0 and each node has no heuristic function. The neighbour ordering for both breadth-first search (BFS) and depth-first search (DFS) is set so that for nodes at the same level, the left most node is always expanded first as can be seen in Fig. 1.

Depth-First Search (DFS)

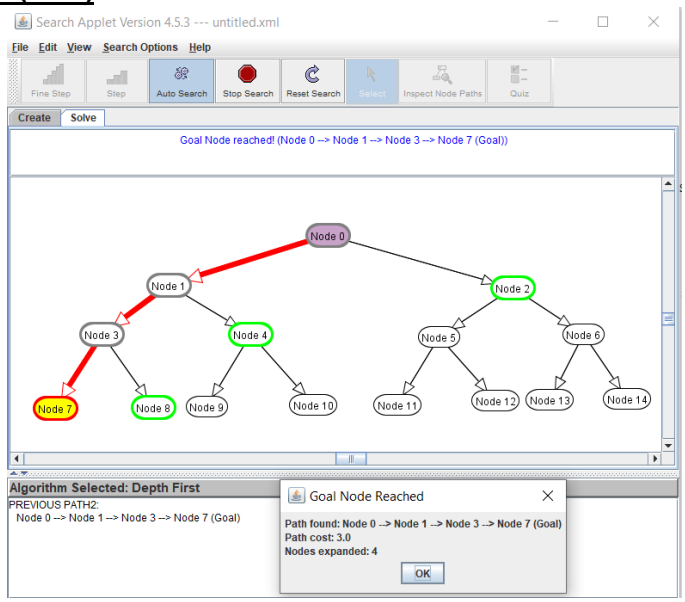

Fig.2

For this graph, the Goal node is located at the last level of the graph with it being the most left node of the last level. This causes the Depth-First Search (DFS) to perform very well as it will always search depth-first, expanding the left most node at each level which allows the Goal node to be searched with the least number of possible nodes expanded.

Breadth-First Search (BFS)

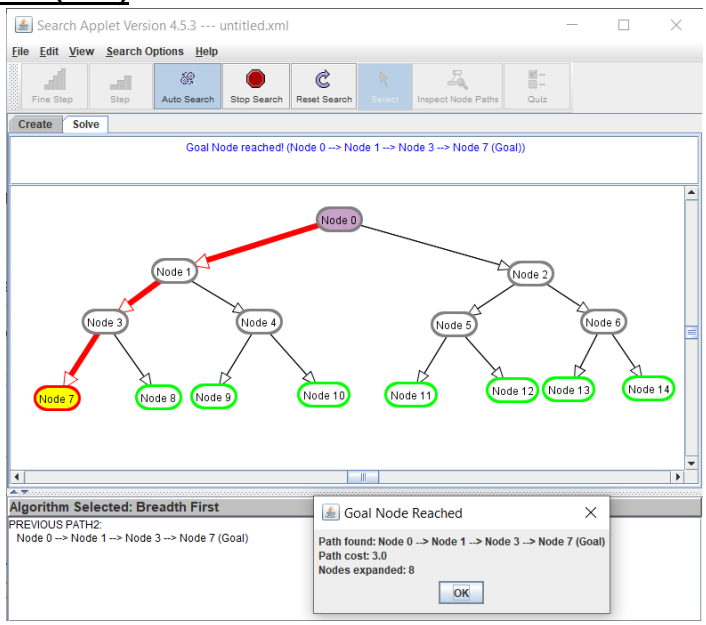

Fig.3

For this graph, the Goal node is located at the last level of the graph with it being the most left node of the last level. This causes the Breadth-First Search (BFS) algorithm to perform poorer than the Depth-First Search (DFS) algorithm, as when a Goal node is located at the maximum depth of the graph and located at the left-most position of that level, the algorithm will first search all the other nodes located at the higher levels, level by level, before reaching the goal node. In this case, the BFS algorithm must expand twice as many nodes as the DFS algorithm. In general, when the neighbour nodes are ordered from left to right, and the Goal node is towards the left side of the graph at maximum depth, the DFS algorithm performs better than the BFS algorithm.

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

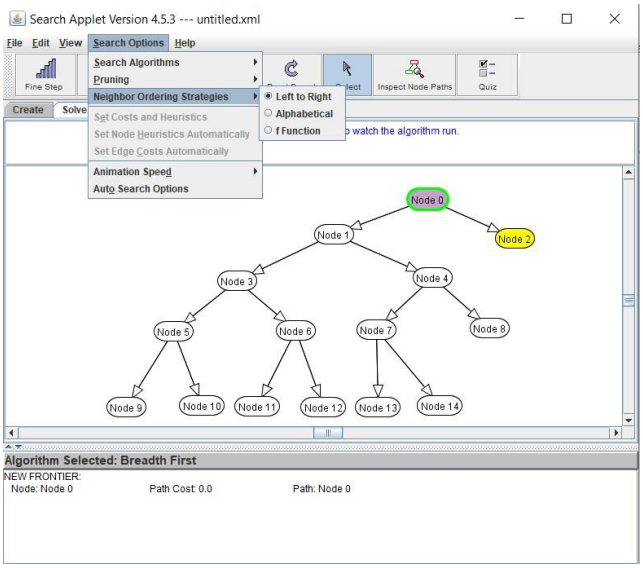

Fig.4

For this question, we have a graph where:
Start Node: Node 0
Goal Node: Node 2
For this graph, the edge cost for each edge is the same with cost 1.0 and each node has no heuristic function. The neighbour ordering for both breadth-first search (BFS) and depth-first search (DFS) is set so that for nodes at the same level, the left most node is always expanded first as can be seen in Fig. 4.

Breadth-First Search (BFS)

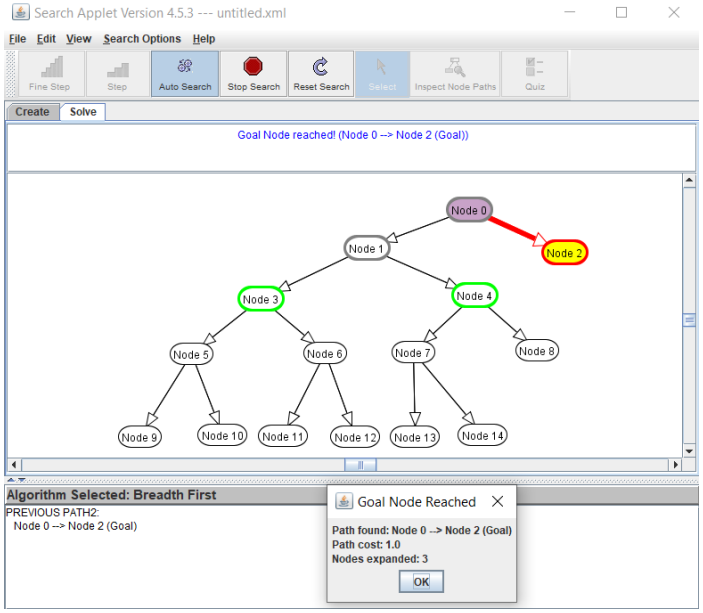

Fig.5

For this graph, the Goal node is the Right most node at the second level of the graph. This causes the Breadth-First Search (BFS) algorithm to perform well as the Goal node is found after just searching the level the Goal node is located at, with only 3 nodes expanded.

Depth-First Search (DFS)

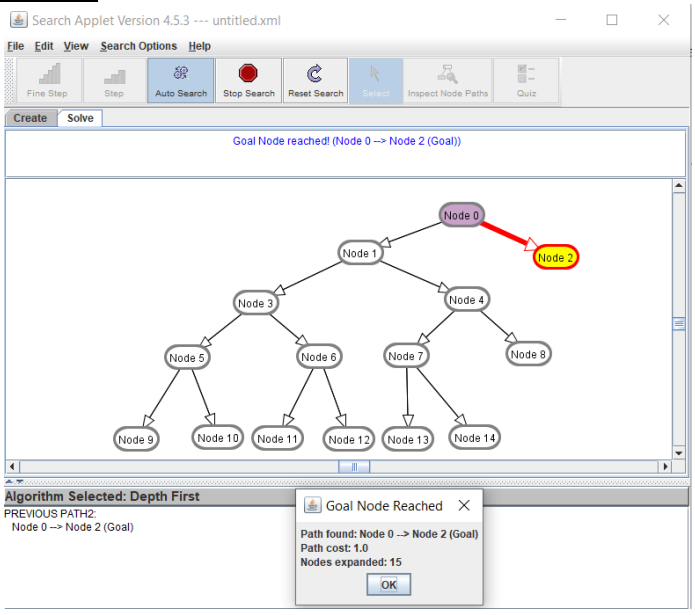

Fig.6

For this graph, the Goal node is the right most node at the second level of the graph. This causes the Death-First Search (DFS) algorithm to perform poorer than the Breadth-First Search (BFS) algorithm as DFS always searches until the maximum depth of the graph to the leaf node,

expanding the left-most node at each level before searching other nodes, which causes the Goal node to be expanded last as it is located at the right most position at the second level. In general, if the Goal node is located at the higher levels of the search graph and if the neighbours are ordered left to right, BFS will perform better than DFS if the node is located more towards the right side of the graph.

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

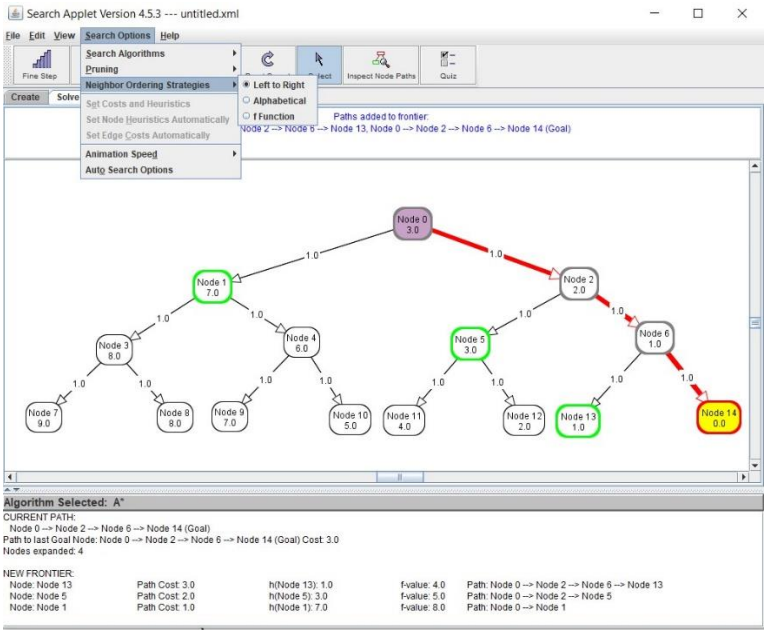

Fig.7

For this question, we have a graph where:
Start Node: Node 0
Goal Node: Node 14
Branching Factor: 2
For this graph, the edge cost for each edge is the same with cost 1.0 and the heuristic function of each node is written on the node itself. The neighbour ordering for both breadth-first search (BFS) and depth-first search (DFS) is set so that for nodes at the same level, the left most node is always expanded first as can be seen in Fig. 7.

A* Search

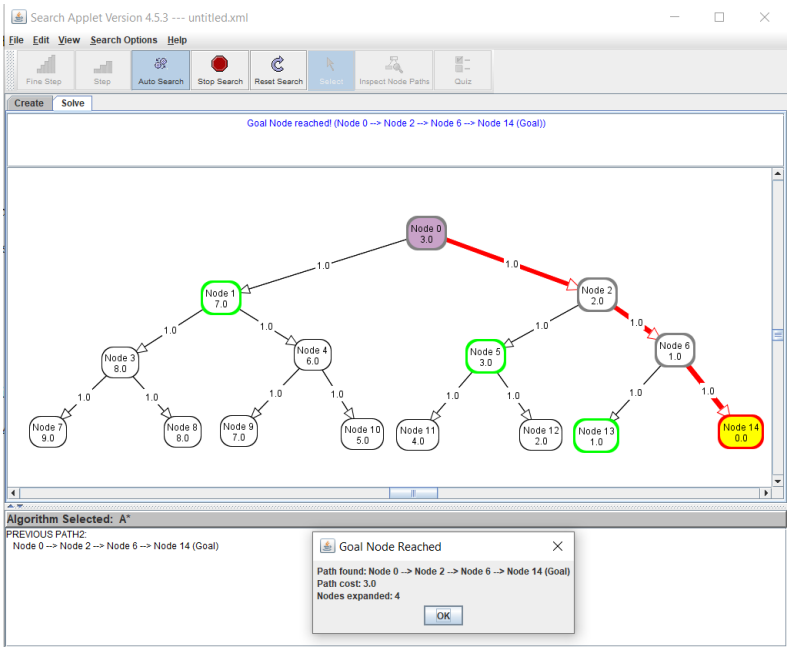

Fig.8

For this graph, the A* Search algorithm performed better than both the Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms. The A* algorithm expanded the least amount of nodes possible, with 4 nodes to find the Goal node of the graph. This is due to the fact that the A* Search algorithm expands nodes based on the Evaluation function: $f(n) = g(n) + h(n)$, where g(n) is the Path-cost function and h(n) is the estimation function from n to Goal and the A* Search algorithm expands the frontier node with the least f-cost at each iteration. The heuristic function, h(n), of the graph allowed the A* Search algorithm to expand nodes that are heuristically closer to the Goal node, ignoring the other nodes in the search process, allowing the A* Search to be more efficient than BFS and DFS.

Breadth-First Search

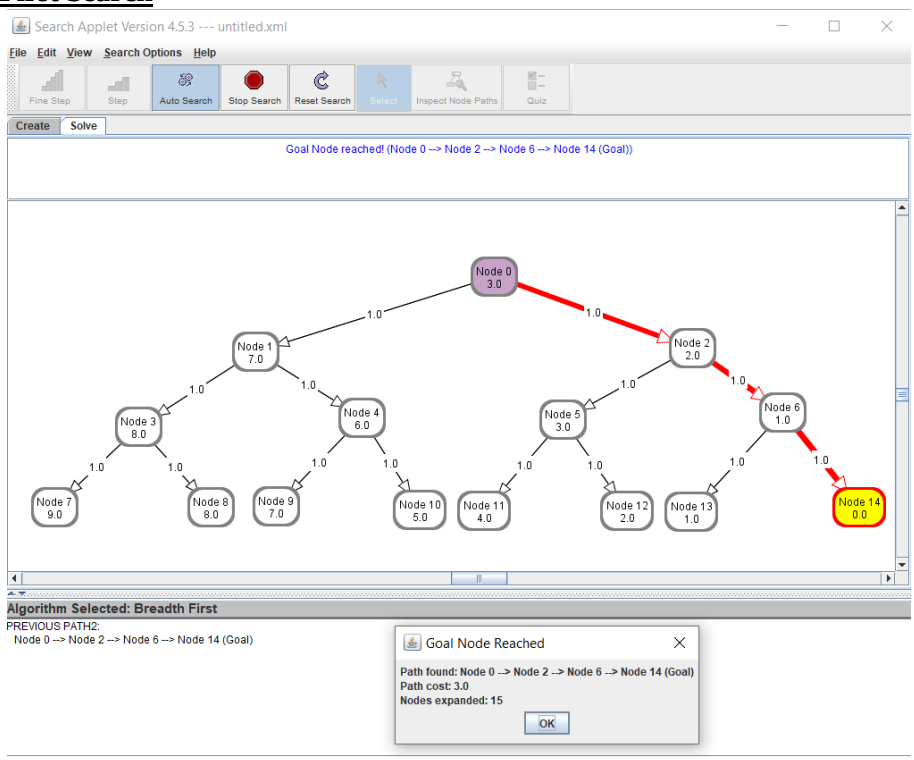

Fig.9

For this graph, as Breadth-First Search (BFS) does not take into account the heuristic costs or path costs of each node, and the algorithm expands nodes level by level until it finds the Goal node, the BFS algorithm performed poorly as the neighbour nodes for each level are ordered left to right and the Goal node is at the most bottom right of the graph, which causes the BFS to expand every node in the search graph before actually reaching the Goal node.

Depth-First Search

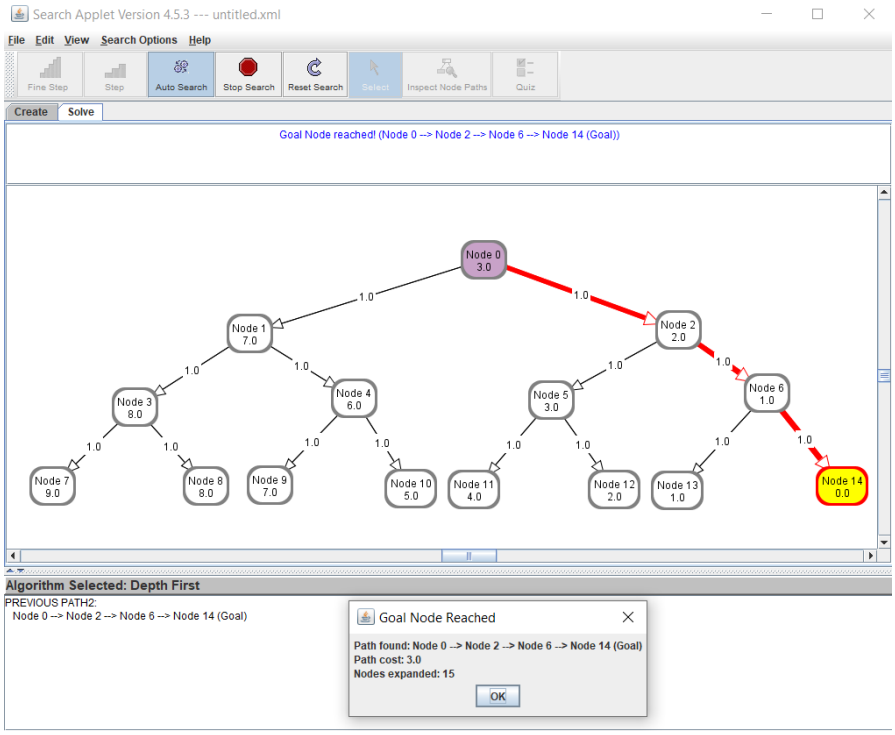

Fig.10

Similar to BFS, the Depth-First Search (DFS) algorithm does not take into account the heuristic costs and path costs of each node when searching the Search Graph. The DFS algorithm will expand nodes depth-first, expanding the left-most node at each level until it reaches maximum depth before searching the other nodes. For this Search Graph, because the Goal node is located at the bottom-right most location of the graph, the DFS algorithm will search though all depths and expand all the nodes in the graph before actually reaching the Goal node.

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

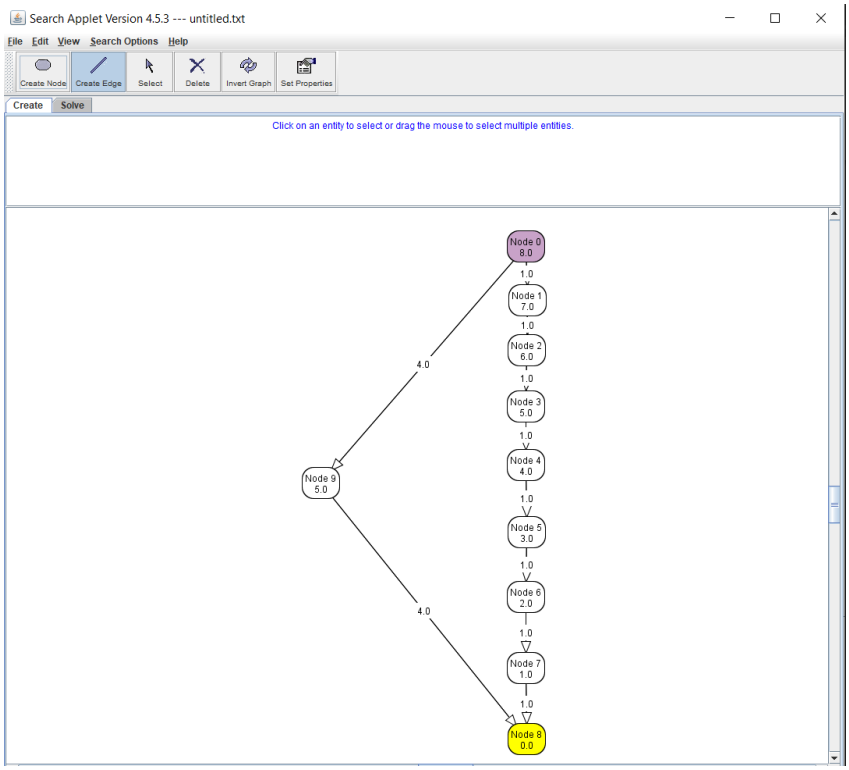

Fig.11

For this question, we have a graph where:
Start Node: Node 0
Goal Node: Node 8
For this graph, the edge cost for each edge is the same with cost 1.0 and the heuristic function of each node is written on the node itself. The neighbour ordering for both breadth-first search (BFS) and depth-first search (DFS) is set so that for nodes at the same level, the left most node is always expanded first as can be seen in Fig. 11.

A* Search

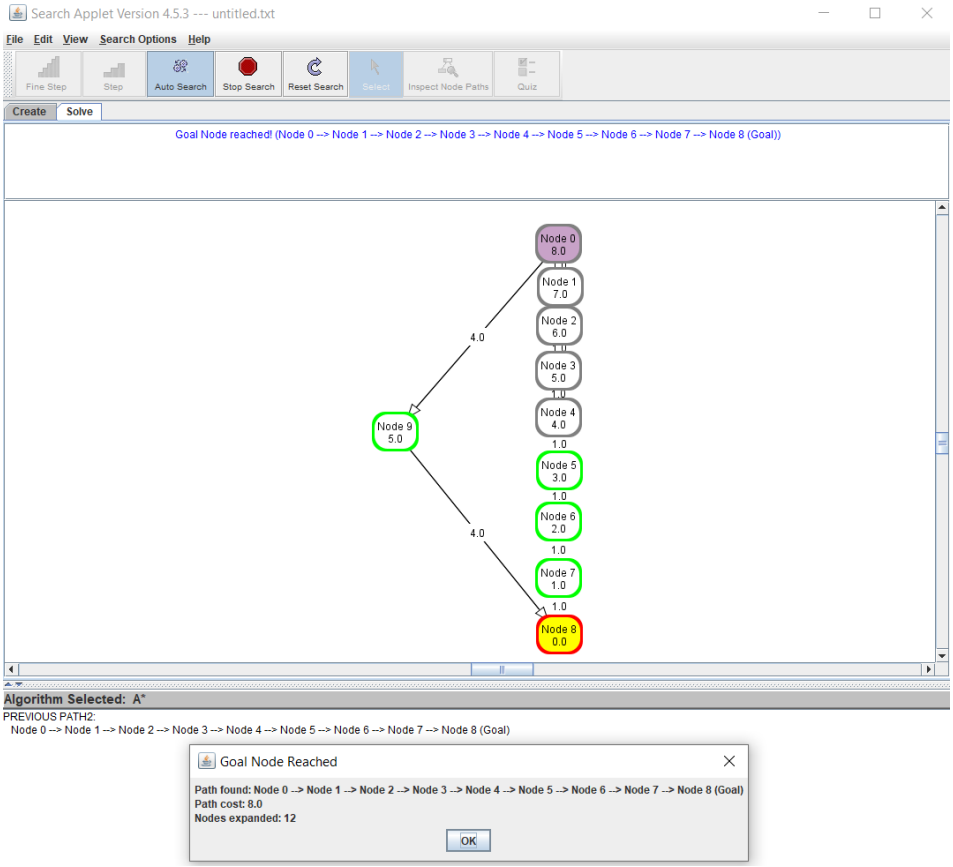

Fig.12

For this graph, the A* Search considers the heuristic costs of the nodes in the graph, where each heuristic cost is actually the exact path cost from the node to the Goal node. A* Search algorithm expands nodes based on the Evaluation function: $f(n) = g(n) + h(n)$, where g(n) is the Path-cost function and h(n) is the estimation function from n to Goal and the A* Search algorithm expands the frontier node with the least f-cost at each iteration. Due to the fact that the combined heuristic cost and path cost of top four nodes at the right side of the graph are smaller than the combined path and heuristic cost for node 9, the A* Star Search actually has to expands some of the other nodes in the graph before actually reaching the Goal node, but the algorithm finds the most optimal path to the Goal node in terms of path cost. This search graph goes to show that although the A* Search algorithm always finds the most optimal path to the goal, it may not be the most efficient algorithm all the time as it will have to check the path cost and heuristic cost of each frontier node at each iteration and expand the node with the least combined heuristic and path cost.

Breadth-First Search

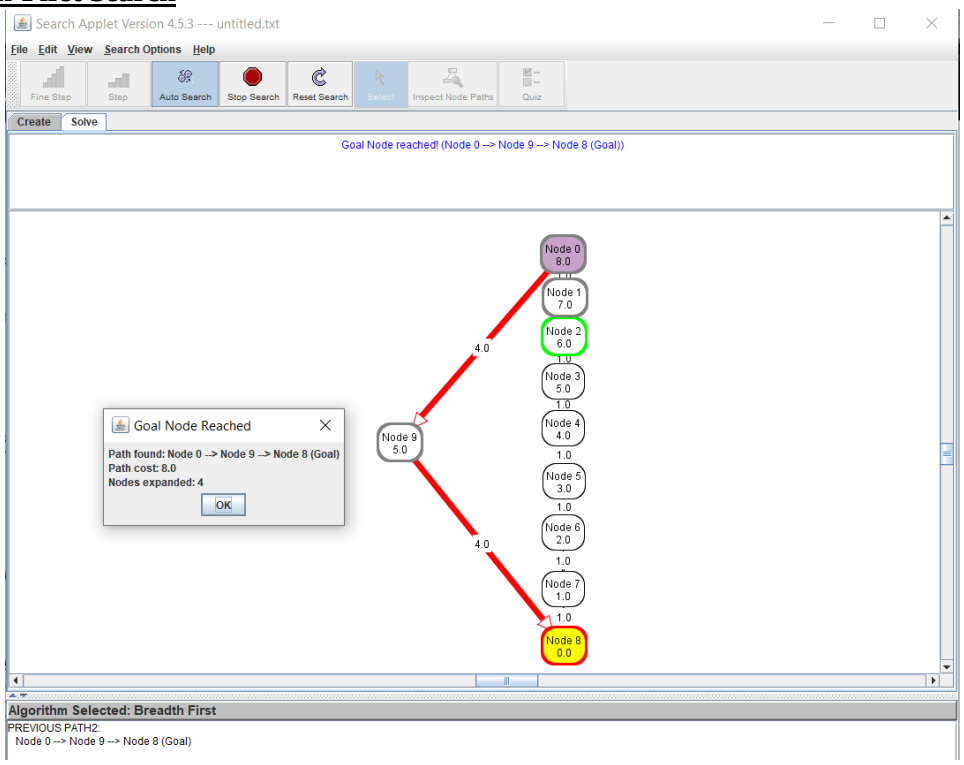

Fig. 13

For the Breadth-First Search (BFS) algorithm, the algorithm ignores the heuristic costs and path costs of the nodes in the Search Graph. It therefore only expanded along the breadth of the Search Graph and found the Goal node at the second level of the graph without having to expand other nodes, unlike the A* Search algorithm, therefore finding the optimal path more efficiently than the A* algorithm.

While A* uses a priority queue, BFS utilizes a queue. Usually, queues are much faster than priority queues (e.g. Dequeue () is O (1) vs O (log n)). The benefit of A* is that it normally expands far fewer nodes than BFS, but if that is not the case, BFS will be faster. That could occur if the heuristic used is poor, or if the graph is very sparse or small, or if the heuristic fails for a given graph.

Depth-First Search

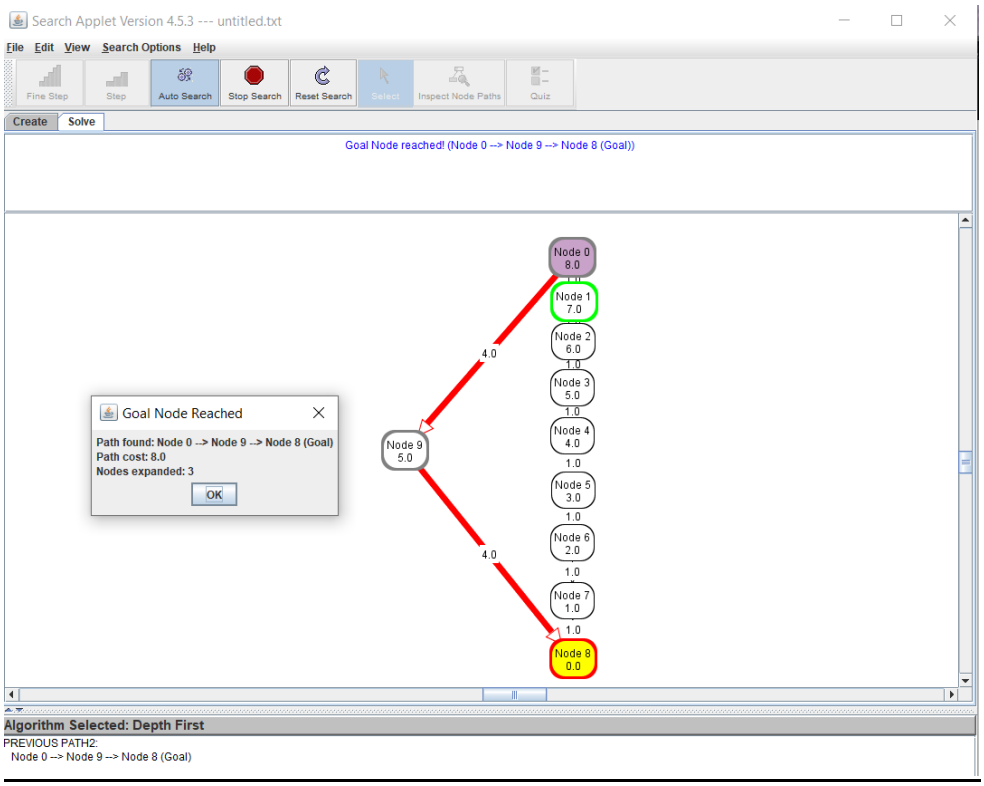

Fig. 13

The Depth-First Search (DFS) algorithm, like the Breadth-First Search (BFS) algorithm, does not take into account the heuristic costs and path costs of the nodes in the Search Graph. It just searches depth-first until the max possible depth to the leaf nodes before searching other nodes. Therefore, this graph is advantageous to the DFS algorithm as compared to both the BFS and A* Search algorithm as it expands the least amount of nodes whilst searching depth-first to reach the Goal node, with only 3 nodes expanded.

Question 2

For this question you are to think about the effect of heuristic accuracy on A* search. That is, you are to experiment with, and think about how close h(n) is to the actual distance from node n to a goal affects the efficiency and accuracy of A*. To get full marks you must at least invent one (plausible, nontrivial) conjecture and either prove it and show some empirical evidence for your answer or show that it is false.

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

For most circumstances, when you reduce h(n) when h(n) is already an underestimate, the answer is that more nodes will be expanded by the A* Search algorithm. However, it could be the case that the same number of nodes are expanded in certain search graphs, depending on the heuristic and path costs of the frontier nodes at each iteration. However, when tested on lots of examples with multiple paths to the node, this this is not always the case either. The reason is that a different optimal path may be found. One true statement is that any sub-optimal path explored with h_1 will be explored with h_2 if $h_2(n) <= h_1(n) <= \text{cost}(n, g)$, where cost (n, g) is the actual cost from n to g.

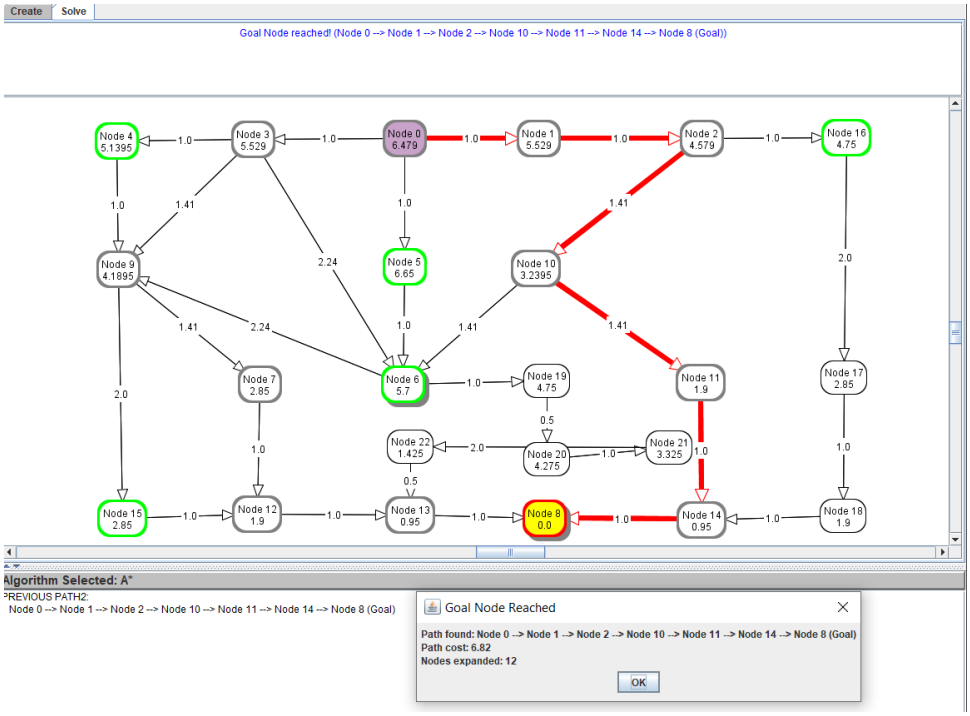

Fig. 14

In the figure above, this is the initial graph where h(n) for all nodes are already underestimates of the true path cost to the Goal node. The h(n) functions are the actual path cost of the nodes to the goal node (cost (n, g)) multiplied by a factor of 0.95 to give an initial baseline graph with underestimated h(n). The results are that 12 nodes are expanded, which is the same number of nodes expanded when the h(n) function is the exact distance from n to a goal. This result goes to show that even with a heuristic function that is an underestimate of the exact path cost, the same number of nodes are expanded in certain search graphs.

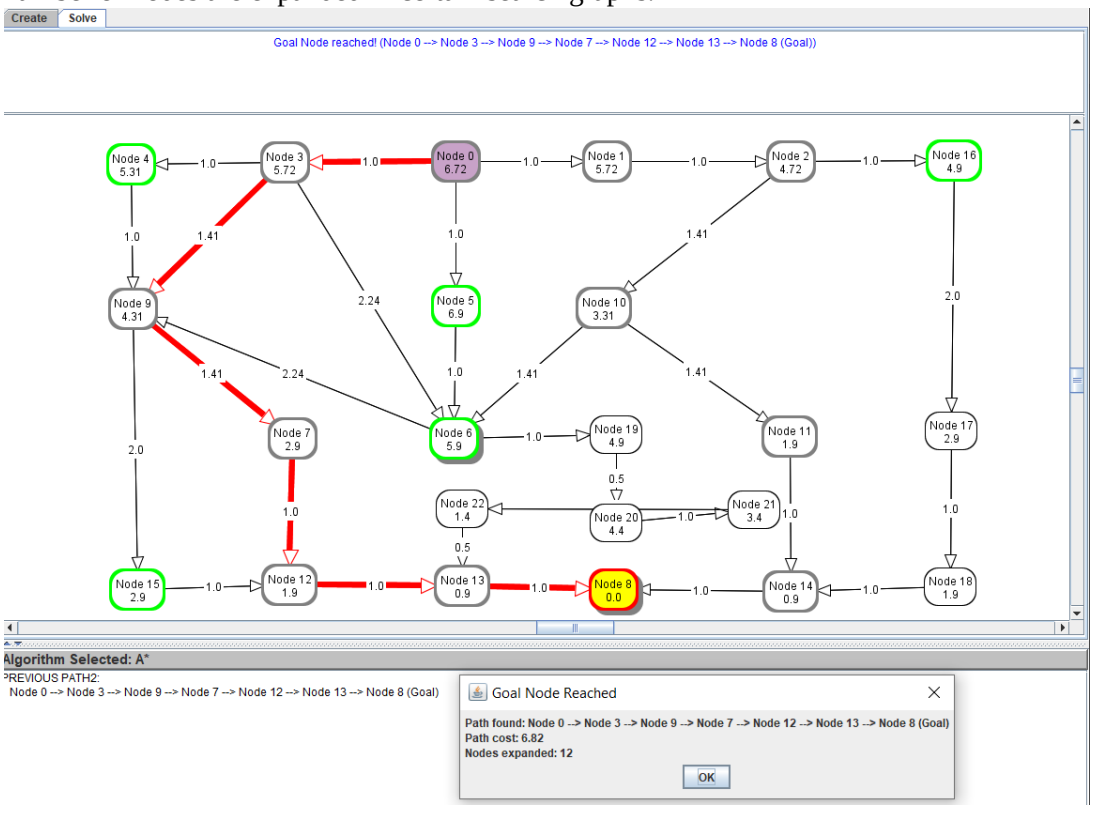

Fig. 15

In the figure above, this is the graph where h(n) for all nodes are further underestimates of the true path cost to the Goal node. The h(n) functions are the actual path cost of the nodes to the goal node (cost (n, g)) minus 0.1. The results are that 12 nodes are expanded. This result goes to show that the same number of nodes can be expanded in certain search graphs, when reducing h(n) when h(n) is already an underestimate, depending on the heuristic and path costs of the frontier nodes at each iteration. Notice that the A* search algorithm took a different optimal path from the previous figure, given 2 optimal paths in the search graph.

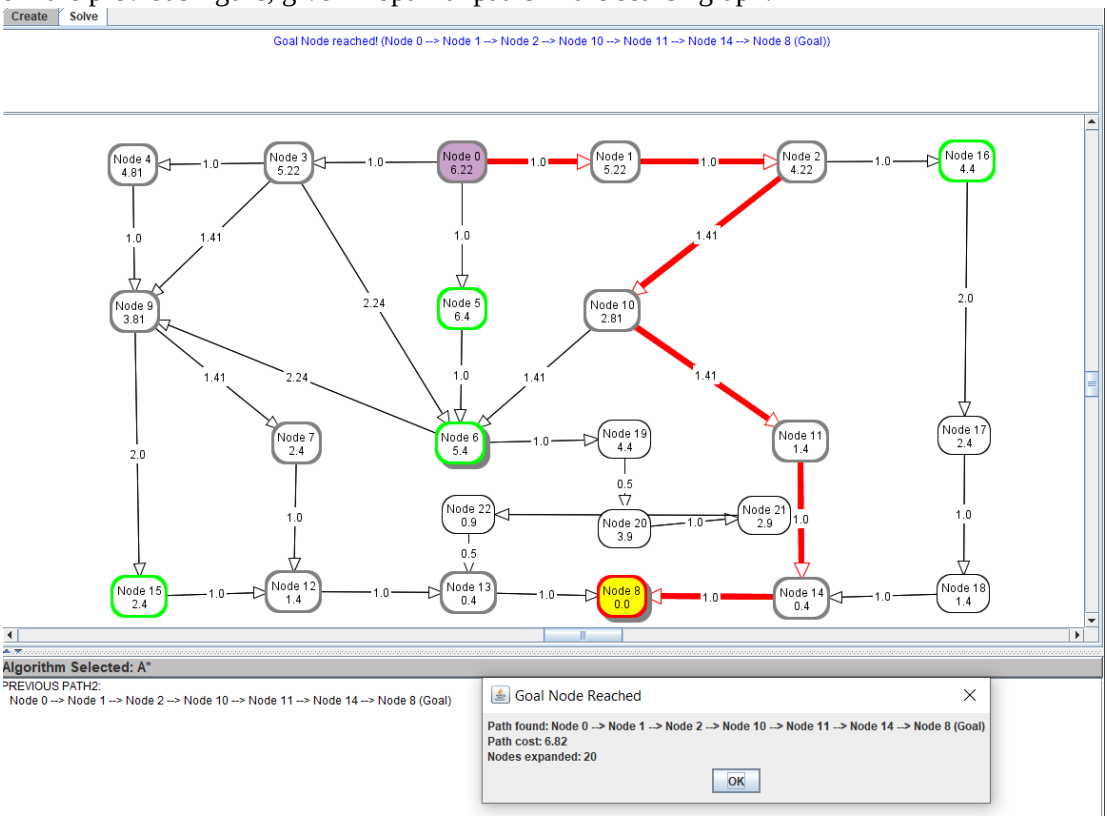

Fig.16

In the figure above, this is the graph where h(n) for all nodes are even further underestimates of the true path cost to the Goal node. The h(n) functions are the Euclidean distances of the nodes minus 0.6. The results are that 20 nodes are expanded. This result goes to show that when you reduce h(n) when h(n) is already an underestimate, more nodes may be expanded by the A* Search algorithm.

h(n) used	Nodes Expanded
h(n) = cost (n, g) (exact path cost)	12
h(n) = cost (n, g) * 0.95	12
h(n) = cost (n, g) – 0.1	12
h(n) = cost (n, g) – 0.2	12
h(n) = cost (n, g) – 0.3	12
h(n) = cost (n, g) – 0.4	12
h(n) = cost (n, g) – 0.5	12
h(n) = cost (n, g) – 0.6	20
h(n) = cost (n, g) – 0.7	20
h(n) = cost (n, g) – 0.8	20

The above table shows the various test results of varying the underestimate amount of the h(n) function. The amount of nodes expanded does not always increase when you increase the underestimate amount all the time.

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

Usually, when h(n) is the exact distance from n to a goal, the A* Search performs the best as it is able to find the optimal path, and at the same time will proceed to the goal without expanding any node off the optimal path.
When there are multiple optimal paths to the goal, if the frontier acts as a stack for nodes with equal f(n)-values (where f(n) = g(n) + h(n), given g(n) the Path-cost function), then it will proceed to the goal without expanding any node off a single optimal path.

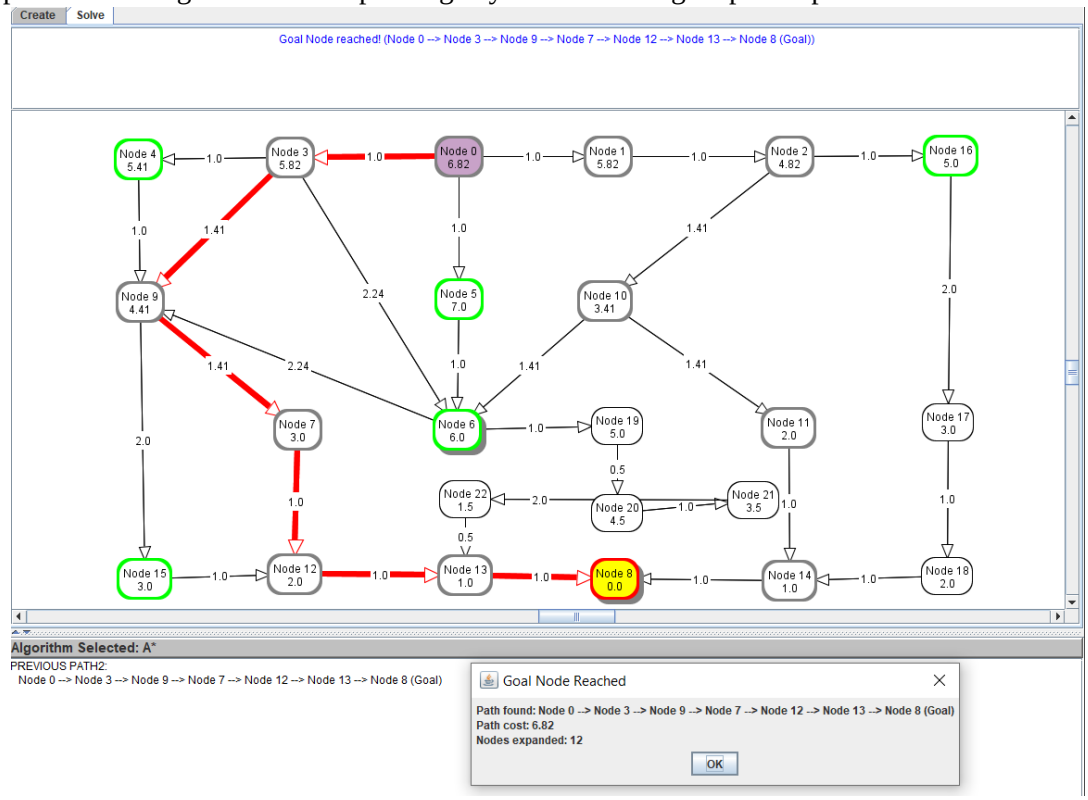

Fig. 17

As can be seen from the figure above, when h(n) is the actual distance from n to a goal (h(n) = cost(g, n)), the result is that the A* search algorithm only searches the nodes that are along the 2 optimal paths that are present in the search graph, without expanding any other nodes that are not on the 2 optimal paths present in the search graph, the expanded nodes are the nodes with bolded grey borders.

c) What happens if h(n) is not an underestimate? You can give an example to justify your answer.

When h(n) is the exact distance from n to a goal, there will be no issues as we are able to find the optimal path with the least path cost and expand only the nodes along the optimal paths in the search graph, giving the A* algorithm the best performance.
However, for A* Search to be able to find the optimal path all the time, the heuristic function h(n) must be admissible. In computer science, specifically in algorithms related to pathfinding, a heuristic function is said to be admissible if it never overestimates the cost of reaching the goal, i.e. the cost it estimates to reach the goal is not higher than the lowest possible cost from the current point in the path.
Therefore, if the heuristic function is an overestimate of the actual path cost, the A* Search may not always find the optimal solution.

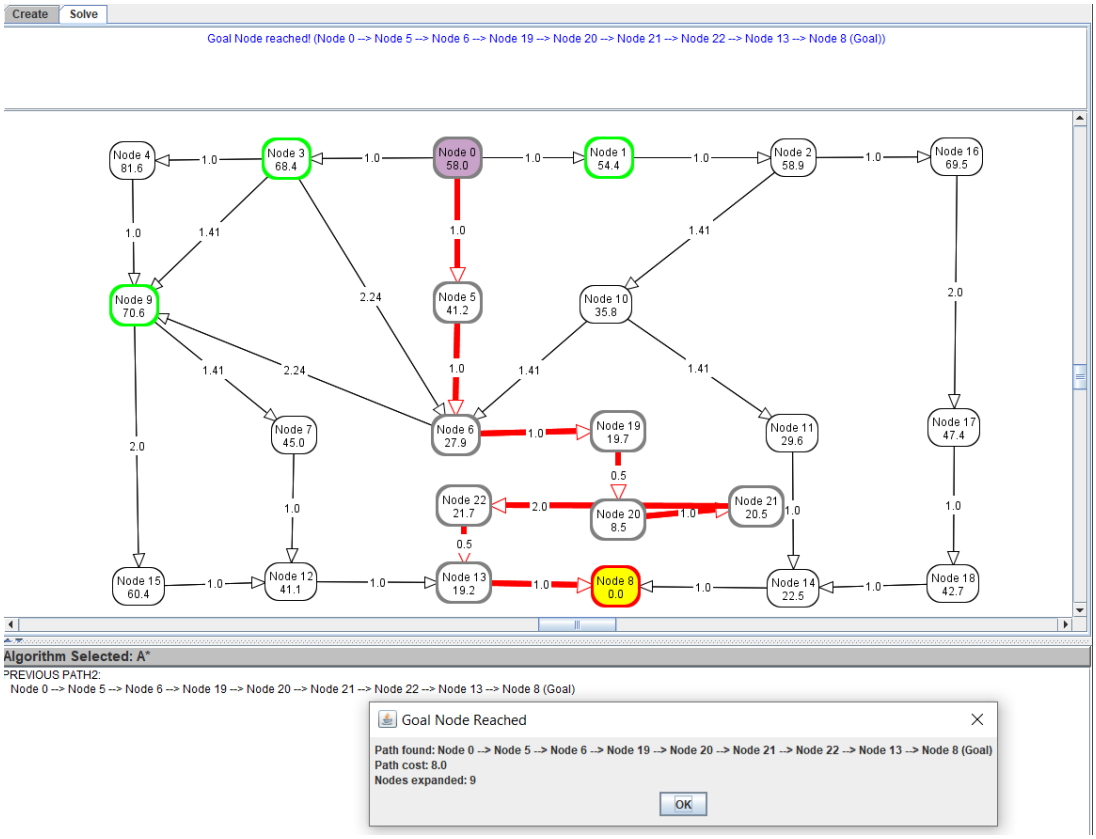

Fig. 18

In the figure above, with the same graph that is used for all the other experiments, the heuristic function is set using the "Set Node Heuristics Automatically" button in the applet. This gives us overestimated heuristic function h(n) for all the nodes in the graph. As can be seen from the figure, the path that was taken by the A* Search is not the optimal path with the least path cost. The optimal paths in this search graph will give a path cost of 6.82 as can be seen in the previous figures when the h(n) functions are either the exact path cost or underestimates of the actual cost. The path chosen by the A* search algorithm when we used an overestimated heuristic function h(n) for all the nodes in the graph has a path cost of 8.0 which is not the least path cost path in the search graph. Therefore, we can conclude that if the heuristic function is not admissible, the A* Search may not find the optimal path.

There are some general statements that are true:

If h(n) <= cost (n, g) +eps (for eps >= 0), the first path found will be at most eps from optimal.

If h(n) <= cost (n, g) ×gamma (for gamma >= 1), the first path found will be at most gamma times optimal.

Conjecture

h(n) used	Nodes Expanded	Optimal Path (Yes/no)
h(n) = cost (n, g) * 0.95	12	yes
h(n) = cost (n, g) – 0.1	12	yes
h(n) = cost (n, g) – 0.2	12	yes
h(n) = cost (n, g) – 0.3	12	yes
h(n) = cost (n, g) – 0.4	12	yes
h(n) = cost (n, g) – 0.5	12	yes
h(n) = cost (n, g) – 0.6	20	yes
h(n) = cost (n, g) – 0.7	20	yes
h(n) = cost (n, g) – 0.8	20	yes
h(n) = cost (n, g) (heuristic is actual path cost from n to goal)	12	yes
h(n) = cost (n, g) + 0.1	11	yes
h(n) = cost (n, g) + 0.2	10	yes
h(n) = cost (n, g) + 0.3	11	yes
h(n) = cost (n, g) + 1	11	yes
h(n) = cost (n, g) + 2	11	yes
h(n) = cost (n, g) + 3	11	yes
h(n) = cost (n, g) + 10	11	yes
h(n) = cost (n, g) + 20	11	yes
h(n) = cost (n, g) + 30	11	yes
h(n) = cost (n, g) + 100	11	yes
h(n) = cost (n, g) * 22.4	7	yes
Set h(n) using “Set Node Heuristics Automatically” button	9 (not optimal path)	No
$h(n) = 2\ cost\ (n, g)^2 + 13\ cost\ (n, g) + 0.4$	7	Yes
h(n) = Euclidean (n, g)	27	Yes
h(n) = Euclidean (n, g) + 1	25	Yes
h(n) = Euclidean (n, g) + 2	25	Yes
h(n) = Euclidean (n, g) + 3	25	Yes
h(n) = Euclidean (n, g) + 10	25	Yes
$h(n) = Euclidean\ (n, g)^2 + 0.3$	9	No
$h(n) = Euclidean\ (n, g)^2 + 1$	9	No
h(n) = Euclidean (n, g) * 2.5	10	No
h(n) = Euclidean (n, g) * 3.5	9	No

The conjecture that I have derived, after conducting experiments with different heuristic functions with the results as shown in the table above, is that for A* Search algorithm, even when the heuristic function h(n) is not admissible, as long as h(n) is an overestimate of the actual path cost of each node in the form of h(n) = cost (n, g) + X where X is any positive number, the A* Search algorithm will in a majority of cases, find the optimal path and will expand lesser nodes as compared to when the heuristic function is the exact path cost of n from Goal node, or an underestimate of the exact path cost of n from Goal node. However, when h(n) is an underestimate of the actual path cost of each node n from the Goal node, in the form of h(n) = cost (n, g) - X where X is any positive number and h(n) >0, the A* Search algorithm will always find the optimal path, but the number of nodes expanded can be greater than the number of nodes along the optimal paths present in the search graph. This is proven by the results in the table above.

The next conjecture is that the heuristic function of the nodes that will cause the A* Search to not be able to find the optimal path to the Goal node in all cases, is when the heuristic function are either the

multiples of the Euclidean distance of the nodes to the Goal node, or non-linear functions of the Euclidean distance of the nodes to the Goal node, where the Euclidean distance of the nodes in the search graph by taking the hypotenuse of the horizontal and vertical distances of the nodes to the Goal node. For example, in Fig.19 below, the Euclidean distance of node 22 is calculated by squaring the vertical distance of 0.5 and adding it to the squared horizontal distance of 1.0 and taking the square root of the results to get a hypotenuse of 1.12.

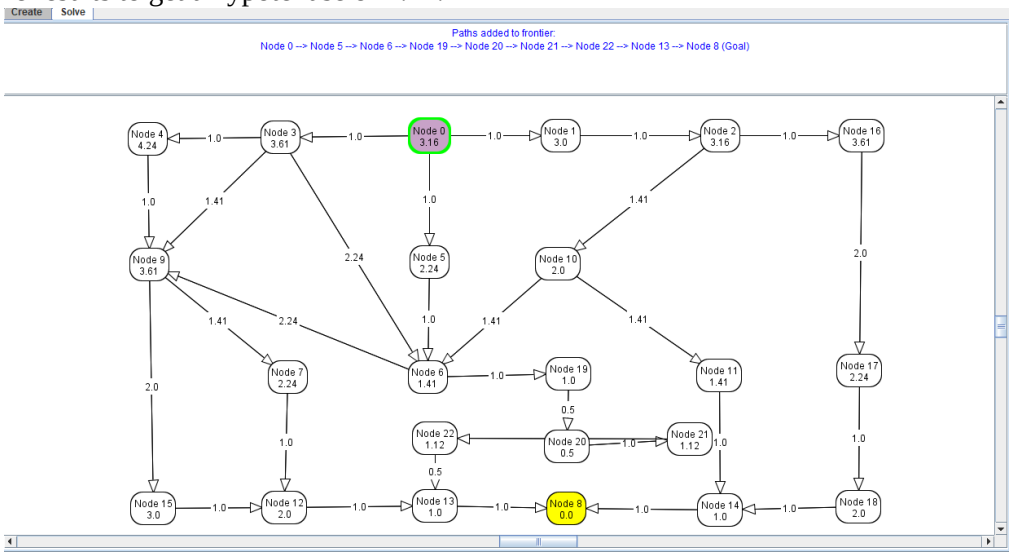

Fig.19

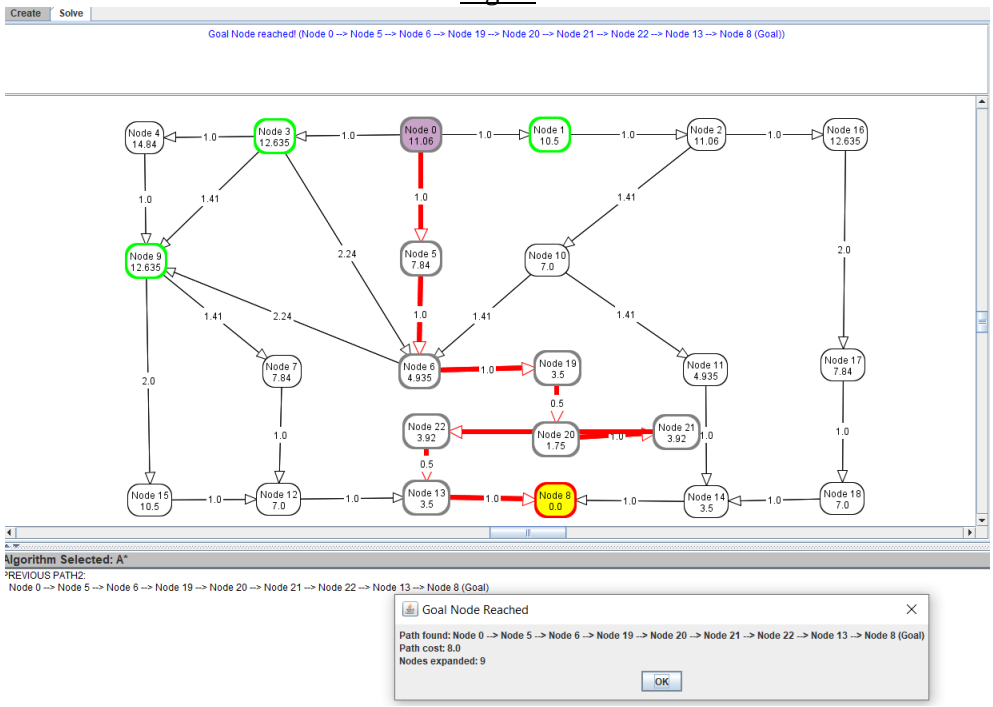

Fig.20

As can be seen from Fig.20 above, when the heuristic function of the nodes is h(n) = Euclidean (n, g) * 3.5, the A* Star search did not find the optimal path that should have the path cost of 6.82, instead it found a path with a higher actual path cost of 8.0. This is due to the fact that when the heuristic function of the nodes is a multiple of the Euclidean distance of the nodes to the Goal Node or a non-linear function of the Euclidean distance of the nodes to the Goal Node, the A* Search algorithm will search the nodes that are heuristically closer to the Goal node, for example nodes 19, 20, 21,22, 13, that may not be on the optimal path than searching the nodes that are on the optimal path but are

heuristically further to the Goal node. This is due to the fact that the A* Search algorithm expand frontier nodes that have a f(n) that has the least cost, where f(n) = g(n) + h(n), given g(n) the Path-cost function and h(n) the heuristic functions. As for this graph, the g(n) of the nodes are way smaller than the non-admissible, overestimated heuristic costs of the nodes, the A* Search is dictated by the h(n) functions of the nodes, therefore, the A* Search expanded the nodes that were heuristically closer to the Goal nodes that were not on the optimal path and gave a solution that was not optimal.

Conclusion

The heuristic function, h(n), can be used to control A* Search's behaviour.

At one extreme, if h(n) is 0, then only g(n) plays a role, and A* Search turns into Dijkstra's Algorithm, which is guaranteed to find a shortest path. If h(n) is always lower than (or equal to) the cost of moving from n to the goal (h(n) is admissible), then A* Search is guaranteed to find a shortest path. The lower h(n) is, the more node A* Search expands, making it slower. If h(n) is exactly equal to the cost of moving from n to the goal, then A* Search will only follow the best path and never expand anything else, making it very fast. Although you cannot make this happen in all cases, you can make it exact in some special cases. Given perfect information of the search graph, A* Search will behave perfectly. If h(n) is sometimes greater than the cost of moving from n to the goal (h(n) is not admissible), then A* Search is not guaranteed to find a shortest path, but it can run faster. At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role, and A* Search turns into Greedy Best-First Search.

Therefore, we have an interesting situation in that we can decide what we want to get out of A* Search. With 100% accurate estimates, we will find the most optimal solutions really quickly. If h(n) is too low, then we will continue to find the most optimal solutions, but it will slow down. If h(n) is too high, then we not be guaranteed to find the optimal solutions, but A* will run faster.

In a game, this property of A* can be very useful. For example, you may find that in some situations, you would rather have a "good" path than a "perfect" path. To shift the balance between g(n) and h(n), you can modify either one.