**Informed Search Strategies**

Informed search uses domain specific hints about the location of the goal to optimize its search. The hint comes in the form of a heuristic function h(n):

h(n): is the estimated cost of the cheapest path from the current state, located a node n, to a goal state.

1. **Greedy Best First Search**

Greedy best first search chooses to expand the node with the lowest estimated cost, on the grounds that it is likely to lead to a solution quickly. Therefore, the evaluation function is the equivalent to the heuristic function. Typically, the heuristic function cannot be computed by the problem description itself (ACTIONS and RESULT functions). In the case of navigating in the Romania problem would be the miles of reaching a resulting state. Instead, the heuristic function uses a shortcut to give an estimated value with the motivation that it will be like the actual distance.

The base algorithm is not cost optimal as it doesn’t return the optimal path without extra care. Instead, the algorithm tries to get as close to the goal state as possible for each iteration. This is where the greedy nature of the algorithm and consequently its name comes from. Greedy-best first search is complete in finite state spaces and not in infinite state spaces.

1. **A\* Search**

The most common informed search algorithm is A\* search (A-star search) which uses an evaluation function, f(n) that combines the current cost with an updated estimated cost starting from the current node to the goal node. Therefore:

f(n) = g(n) + h(n): where g(n) is the path cost from the initial state to the current state.

f(n): is the estimated cost of the best path that continues from the current node n to the goal node.

A star search is complete but the property of the heuristic determines if the search is also cost-optimal. Two important properties are admissibility and consistency. An admissible heuristic generates the lowest estimated cost every time. In other words, admissibility suggest that the cost generated by the heuristic will either be less than or equal to the actual cost. Therefore, an evaluation function with an admissible heuristic is cost optimal.

Consistency: a heuristic is consistent if for every node n the following applies:

h(n) <= c (n, a, n’) + h(n’) = action\_cost (n, a, n’) + heuristic(n’)

This property, consistency, is in the form of the triangle inequality which suggest that the side of a triangle, a, can’t be greater than the sum of the other two sides. In the case of the property sum of the sides c (n, a, n’) and h (n’) cannot be greater than h(n). In other words, the sum of actual cost of doing action a in n and the estimated cost, heuristic, from the successive node to the optimal path cannot be greater than estimated cost of the direct parent node. When this property is true for all nodes in the state space then the heuristic is consistent. Every consistent heuristic is admissible, however not every admissible heuristic is consistent making consistency a stronger property.

Consistent heuristic will always choose nodes on the optimal path. Consequently, consistency eliminates the need to update the path-cost of reached nodes with lower cost. Inconsistent heuristics, suffer from the possibility of reaching the same state from different paths. If each successive path that reaches the same state is lower than the previous, then the same node will be entered into the frontier each time. Therefore, it could help to take greater care to prevent entering the same node more than once and instead update the successors with the new cost. This requires that parents hold references to children as well as their own parents.

Although, admissible heuristics are not cost optimal, there are two cases where it will always find the optimal path.

1. The first case suggests that for every node along a path, if at least one of its successors is admissible, then it will always be found.
2. The second case suggest that if the difference between the second-best path, C2 and the optimal cost, C\*, C2 - C\*, is greater than an overestimated cost generated by h(n) than A\* is guaranteed to be cost optimal.
3. **Search Contours**

Search contours are a good way of visualizing how an evaluation function will cause a search to behave. For A start search the bands for each increasing f-cost will become narrower focusing in on the optimal-cost path. Uniform-cost-search will produce a pattern of concentric circles with no bias towards the goal.

As a path is extended the g cost is monotonic because the path cost is always increasing. The path cost is determined by the actions executed, which the act of executing an action is always positive, from the initial state to the current state.

It is not clear whether the evaluation function f(n) = g(n) + h(n) is also monotonic. As we go from n to n’ the evaluation function goes from **f(n) = g(n) + h(n)** **to f(n’) = g(n) + c (n, a, n’) + h(n’).** The second evaluation says that the estimated path cost of n’ is equal to the actual cost of the parent node, n, plus the cost of going from n to n’ and lastly plus the estimated cost of going from n’ to the goal state. The equality is consequence of the fact that the previous path-cost must be less than or equal to because the cost of doing something is always positive.

**g(n) + h(n)** **<= g(n) + c (n, a, n’) + h(n’)**

**g(n) – g(n) + h(n) <= c (n, a, n’) + h(n’)**

**h(n) <= c (n, a, n’) + h(n’)**

**the heuristic must be consistent for f to monotonically increase.**

If C\* is the optimal path, then the following hold true for A star search:

1. A star search will expand all nodes reachable from the initial state if each node on the path has f(n) < C\*. We say that these nodes are surely expanded. In other words, it is going to search for the path with the least cost.
2. There exists a chance that A star search will expand nodes along the same goal contour until it has selected the goal node. This is because there is a chance that several nodes will produce the same **f(n) = g(n) + h(n). This will happen when c (n, a, n’) is equal to h(n). In other words, h(n’) should decrease with each step and the cost will always be positive. Therefore the**
3. A star search will never choose to expand a node whose f(n) > C\*.

A star search with a consistent heuristic is optimally efficient. The grounds for this are that it must check all nodes whose current f(n) cost is than the optimal cost because there is a chance that it is apart of the optimal path.

There is also the case where multiple nodes with f(n) = C\* exist. There is not way to tell whether the node chosen will be the goal node and consequently we do not consider this when evaluating the optimality of the algorithm.

An efficient property of A star search is that it will prune away nodes that are not necessary for find the optimal solution given the evaluation function. This is because a path with a cost of 10 and another with a cost of 8 will never be considered of the solution path is 7. This idea that an intelligent agent need not consider irrelevant possibility before examining them is important for many areas of AI.

Even though A star search is cost optimal, optimally efficient, and complete it is not the end of the story. In the case where there a multiple optimal solutions A star will visit all these solutions before finishing.

1. Satisficing Search: Inadmissible heuristics and Weighted A\*

Note: An admissible heuristic is one that never overestimates the cost to reach the goal.

Because A\* visits all surely expanded nodes it tends to visit a lot of nodes in general. However, if we are willing to accept suboptimal solutions using an inadmissible heuristic then there is possibility that the A\* will be more accurate, consequently visiting less nodes when finding a solution. The compromise is that we must accept a suboptimal solution in favor of efficiency.

For straight line distance there is an idea called the detour index that tries to account for the curvature of roads. Because there is not road that is straight forever. Typically, this index is between 1.2 and 1.6 for most cities. Therefore, an evaluation that would produce 10 originally would now generate 13 if the index was 1.3.

This idea can be extrapolated to for all searches involving a star by applying a weight to the heuristic function. This means that we weight the heuristic value more heavily. Therefor if f(n) = g(n) + h(n) for A star search than for weighted A star search we have:

f(n) = g(n) + W\*h(n)

where 1 < W < infinity

There is still a chance to find the optimal solution using weighted A star. However, if the optimal path ever strays outside of the search contour than it will never find the optimal solution. In general, A star will find a solution that cost anywhere between C\* and WxC\* but in practice typically the solution leans more towards C\*.

The evaluation function for A star search says that it will evaluation a given node, n, by the actual cost from the initial node to n plus the expected cost from node n to the goal node. In other words, it is always trying to give us a rough estimate of how much each node will cost and chose the nodes where most optimal first and where f(n) is less than the optimal path cost. In uniform cost, the path cost for each action is the same everywhere, and therefore ignore the heuristic function altogether. Like A star search greedy best first search contours in favor of the goal state. That means with each new expanded node the goal node is increasingly become more focus with respects to the search contours. It does this by selecting nodes purely on a heuristic. Weighted A star tries to reduce the number of nodes visited by overestimating some nodes out of the search counter. The idea is that we compromise finding the optimal solution for a good-enough-solution in favor of performance.

The idea of good enough solutions allows for more search designs such as bounded and unbounded searches. In bounded searches we guarantee that the solution path may not be optimal if it strays out of the search contour but will be less than some constant factor C of the optimal cost. Unbounded search will find a solution at any cost if it can be found quickly.

1. Memory Bounded Search

The main problem with A star search is its memory consumption. Memory is split between the frontier and reached states. A node in the frontier is obviously stored in the frontier so we know what to expand next as well as the reached states, so we know if we have been here before.

There are few notable memory bounded searches such as beam search which is typically implemented in two ways. Both ways discard nodes that do not fit within a specific range. One way is to keep a certain number of nodes with the best f score and therefore discarding the rest. Another implementation is keeping all nodes to fit within a specified range of f and Ᵹ, where f is the best f score and Ᵹ is a predetermined number representing the lower bounds.

Iterative Deepening A Star search operates in the same fashion as regular Iterative Deepening thus eliminating the need for a reached state persistence and instead traversing paths with each increase of the counter bound. In terms of IDA\* the cutoff is the current most optimal f score. When no solution is found the next most optimal f score is chosen and the search starts over from the initial state.

Recursive best first search resembles a recursive depth first search. However, in infinite state spaces depth first search can go on forever. RBFS prevents this keeping track of the f\_limit variable which represents the current most second optimal path. If the current optimal f score exceeds f\_limit, then the algorithm unwinds itself to the f\_limit node and then starts traversing in that direction.

If the second-best path becomes the best path Recursive Best First Search will unwind to the new optimal node to follow its path. These changes correspond to an iteration of IDA\*. This could consequently require the re-expansion of several forgotten nodes.

RBFS is optimal if the heuristic function does not overestimate. RBFS uses linear space where the max is equal to the depth of the deepest optimal node. The time complexity is harder to explain because it is dependent on h(n) and how often the best path changes. IDA\* and RBFS both suffer from not using enough space. Further, RBFS forgets most of the work it has already done whenever the second-best path becomes the optimal path so even if we utilize more space there is nothing, we can do with it. This can become cost if the optimal path becomes the second-best path for every alternating calls.

In order to optimize for performance by increasing the usage of space we can calculate the amount of space we have. Two searches that do this are **Memory-Bounded A Star** and **Simplified Memory-Bounded A Star.** From a high-level

Simplified Memory-Bounded A\* proceeds down the best path until memory has run out in which case the node with the highest f score is discarded and then backs up the forgotten node to its parent.