11
Heuristic Network Search Algorithm

Many AI applications require a search for a best (or shortest) path through a network. Applications for network searches include routing problems (such as aircraft or sales route planning and terrain traversal for robotic vehicles).

Three popular search strategies for finding a short path through a network are: depth-first, breadth-first, and heuristically directed breadth-first. Depth-first searches are very inefficient, in general, because the first available path is searched exhaustively, even if it turns out to be a very long path. The advantages of depth-first searches are the ease of implementation and small data storage requirements. Breadth-first searches are much better since the search space tends to “fan out” from the starting node until the goal node is located. Breadth-first searches have a much larger run-time memory requirement than depth-first searches. The heuristically directed breadth-first search attempts to search possible paths in the order of paths that are heuristically judged to have the highest probability of being the best path. This directed search causes the search space to “fan out” preferentially towards the goal node. This directed search is very simple to implement since we have an accurate cost function for ordering the possible paths for each node being searched: the cost of any possible path is taken to be the Cartesian distance from the first node on a possible path to the goal node. This chapter will present a good technique for performing an efficient heuristic breadth-first network search, the A* search (Hart, Nilsson, and Raphael, 1968).

11.1 Theory of the A* Search

We will discuss the theory of A* search in the framework of a practical application: calculating the shortest path between cities that are connected by roads. The data that the A* algorithm uses is a set of nodes (the cities in this example), a set of edges or paths between nodes (the roads), a starting node (or city) and a goal node (the city we want to reach).

The computational steps used in the in the A* algorithm are simple to describe. At the starting node, we build a list of all nodes that are directly connected to the starting node by a path. We sort this list of nodes in order of proximity to the goal node (list the cities closest to the target city first). We then start a breadth-first search of all nodes in this list; at each node we search from, we sort the list of connected nodes by proximity to the goal node. The search terminates as soon as we search a node directly connected to the goal node.

This last step of sorting possible path segments by proximity to the goal node, in combination with hashing (storing computed information with an index key to find
it quickly) the calculated distance between directly or indirectly connected nodes makes a single search through the network much faster. The hashing of precomputed distances greatly speeds up subsequent searches through the network. Our example makes use of both of these tricks.

11.2 A* Network Search Program

The function $A^*\text{search}$ is called with the names of the starting and goal nodes for the search. Function $A^*\text{search}$ uses the local list $\text{possible-paths}$ to cache the Cartesian distances between nodes in the network as they are calculated. The helper function $a^*-\text{helper}$ performs a breadth-first search by starting with the first possible path (this list is kept sorted so that the possible paths with an ending node closest to the goal node are kept first on the list) and fanning the search out preferentially towards the goal node. Note that this search stops when the first path segment on the possible path list contains the goal node.

The function $\text{init-network}$ precalculates the lengths of all path segments in the network. The function $\text{init-path-list}$ puts calculated path lists as a property value on each node. By caching precalculated path costs for each node, the example program avoids recalculating any path sequence. For example, if after calculating the path between nodes $n1$ and $n11$ in Figure 11.1, function $A^*\text{search}$ is then asked to calculate the best path between nodes $n2$ and $n11$, no real search is necessary since the path sequence from $n2$ to $n11$ will already be cached at node $n2$.

![Figure 11.1. Sample Network with 11 Nodes and 12 Edges](image-url)