# Artificial Intelligence/Search/Heuristic search/Best-first search

## Contents



## Preamble

This article is about best-first search in its general form. There are a number of specific algorithms that follow the basic form of best-first search but use more sophisticated evaluation functions. A* is a popular variant of the best-first search. This page does not discuss A* or other variants specifically, though the information contained herein is relevant to those search algorithms.

## Definition

Best-first search in its most general form is a simple heuristic search algorithm. “Heuristic” here refers to a general problem-solving rule or set of rules that do not guarantee the best solution or even any solution, but serves as a useful guide for problem-solving. Best-first search is a graph-based search algorithm (Dechter and Pearl, 1985), meaning that the search space can be represented as a series of nodes connected by paths.

## How It Works

The name “best-first” refers to the method of exploring the node with the best “score” first. An evaluation function is used to assign a score to each candidate node. The algorithm maintains two lists, one containing a list of candidates yet to explore (OPEN),

and one containing a list of visited nodes (CLOSED). Since all unvisited successor nodes of every visited node are included in the OPEN list, the algorithm is not restricted to only exploring successor nodes of the most recently visited node. In other words, the algorithm always chooses the best of all unvisited nodes that have been graphed, rather than being restricted to only a small subset, such as immediate neighbours. Other search strategies, such as depth-first and breadth-first, have this restriction. The advantage of this strategy is that if the algorithm reaches a dead-end node, it will continue to try other nodes (Pearl, 1984).

# Algorithm

Best-first search in its most basic form consists of the following algorithm (adapted from Pearl, 1984):

The first step is to define the OPEN list with a single node, the starting node. The second step is to check whether or not OPEN is empty. If it is empty, then the algorithm returns failure and exits. The third step is to remove the node with the best score, *n*, from OPEN and place it in CLOSED. The fourth step "expands" the node *n*, where expansion is the identification of successor nodes of *n*. The fifth step then checks each of the successor nodes to see whether or not one of them is the goal node. If any successor is the goal node, the algorithm returns success and the solution, which consists of a path traced backwards from the goal to the start node. Otherwise, the algorithm proceeds to the sixth step. For every successor node, the algorithm applies the evaluation function, *f*, to it, then checks to see if the node has been in either OPEN or CLOSED. If the node has not been in either, it gets added to OPEN. Finally, the seventh step establishes a looping structure by sending the algorithm back to the second step. This loop will only be broken if the algorithm returns success in step five or failure in step two.

The algorithm is represented here in pseudo-code:

1. Define a list, OPEN, consisting solely of a single node, the start node, *s*.

2. IF the list is empty, return failure.

3. Remove from the list the node *n* with the best score (the node where *f* is the minimum), and move it to a list, CLOSED.

4. Expand node *n*.

5. IF any successor to *n* is the goal node, return success and the solution (by tracing the path from the goal node to *s*).

6. FOR each successor node:
    a) apply the evaluation function, *f*, to the node.

b) IF the node has not been in either list, add it to OPEN.

7.looping structure by sending the algorithm back to the second step.

Pearl (2012?) adds a third step to the FOR loop that is designed to prevent re-expansion of nodes that have already been visited. This step has been omitted above because it is not common to all best-first search algorithms.

# Evaluation Function

The particular evaluation function used to determine the score of a node is not precisely defined in the above algorithm, because the actual function used is up to the determination of the programmer, and may vary depending on the particularities of the search space. While the evaluation function can determine to a large extent the effectiveness and efficiency of the search (Pearl, 1984), for the purposes of understanding the search algorithm we need not be concerned with the particularities of the function.

# Applications

Best-first search and its more advanced variants have been used in such applications as games and web crawlers. In a web crawler, each web page is treated as a node, and all the hyperlinks on the page are treated as unvisited successor nodes. A crawler that uses best-first search generally uses an evaluation function that assigns priority to links based on how closely the contents of their parent page resemble the search query (Menczer, Pant, Ruiz, and Srinivasan, 2001). In games, best-first search may be used as a path-finding algorithm for game characters. For example, it could be used by an enemy agent to find the location of the player in the game world. Some games divide up the terrain into “tiles” which can either be blocked or unblocked. In such cases, the search algorithm treats each tile as a node, with the neighbouring unblocked tiles being successor nodes, and the goal node being the destination tile (Koenig, 2004).

# References


- Dechter, R. & Pearl, J. (1985). Generalized Best-First Search Strategies and the Optimality of A*. Journal of the Association for Computing Machinery, 32, 505-536.
- Menczer, F., Pant, G., Ruiz, M.E. & Srinivasan, P. (2001). Evaluating Topic-Driven Web Crawlers. Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval (pp. 241-249). New York: Association for Computing Machinery.
- Koenig, S. (2004). A Comparison of Fast Search Methods for Real-Time Situated Agents. Proceedings of the Third International Joint Conference on Autonomous

Agents and Multiagent Systems - Volume 2 (pp. 862-871). Washington, DC: IEEE Computer Society.
- Pearl, J. (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Reading, MA: Addison-Wesley.