Heuristic Search

Prerequisite

- Recursive Descent

Main Concept

The main goal of *heuristic search* is to use an *estimate* of the ``goodness'' of all states to improve the search for a solution.

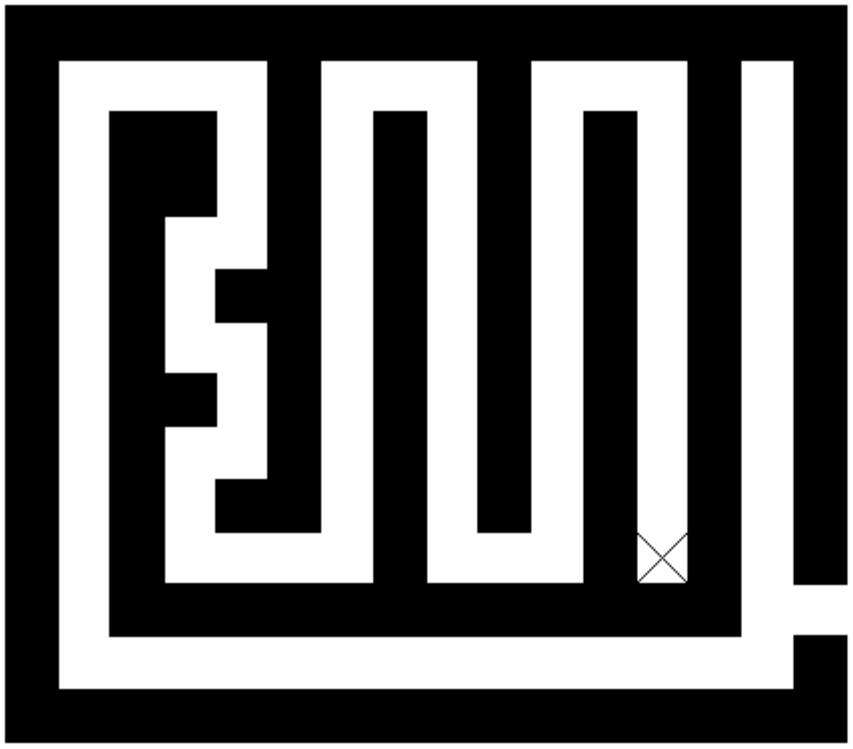
Generally, this estimate of ``goodness'' is expressed as a function of the state, and such functions are called *heuristic functions*. Examples of potential heuristic functions are as follows:

- When looking for an exit in a maze, the euclidean distance to the exit.
- When trying to win a game of checkers, the number of checkers you have minus the number of checkers your opponent has.
- When trying to win freecell, the number of cards already home plus 5 times the number of freecells that are empty.

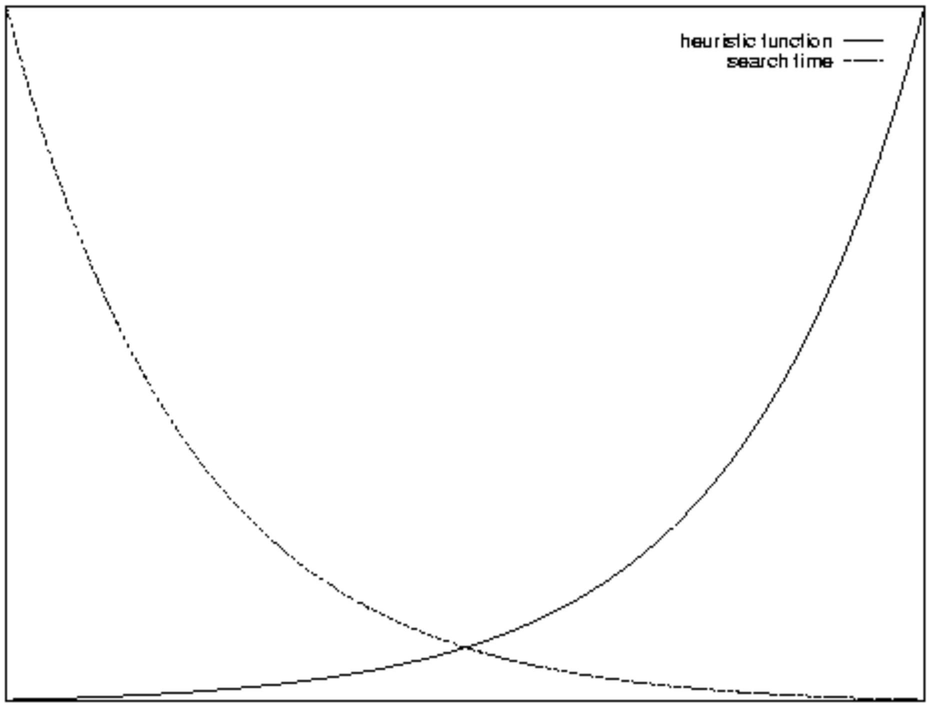
Designing Heuristic Functions

Intuitively, the better a heuristic function is, the better (faster) a search will be. The question is: how does one evaluate how good a heuristic function is?

A heuristic function is evaluated in terms of how well it estimates the cost of a state. For example, for the case of a maze, how well does it estimate the distance to the exit? Euclidean distance can be very bad, as even for simple mazes, it can be far off from the same solution:



In general, the better the heuristic function, the faster the search will go. In general, the search time and accuracy of the heuristic function behave in the following manner.



It's important to notice that even a stupid heuristic function can significantly improve the search (if it's used properly of course).

Another concept is important when looking for a heuristic function, *admissibility*. A heuristic function is called *admissible* if it *underestimates* the cost of a state and is nonnegative for all states. For example, the euclidean distance for the maze is an under-estimate, as the example above clearly shows.

Idea #1: Heuristic Pruning

The easiest and most common use for heuristic functions is to prune the search space. Assume the problem is to find the solution with the minimum total cost. With an admissible heuristic function, if the cost of the current solution thus far is A, and the heuristic function returns B, then the best possible solution which is a child of the current solution is A+B. If the a solution has been found with cost C, where C < A+B, there is no reason to continue searching for a solution from this state.

This is simple to code and debug (assuming one starts with a working, but slow, program, which is how this should be used) and can yield *immense* returns in terms of run-time. It is especially helpful with depth-first search with iterative deepening.

Idea #2: Best-First Search

The way to think of best-search is as greedy depth-first search.

Instead of expanding the children in an arbitrary order, a best-first search expands them in order of their ``goodness,'' as defined by the heuristic function. Unlike greedy search, which *only* tries the most promising path, best-first search tries the most promising path first, but later tries less promising ones. When combined with the pruning described above, this can yield very good results.

Idea #3: A* Search

The *A* Search* is akin to the greedy breadth-first search.

Breadth-first search always expands the node with minimum cost. A* search, on the other hand, expands the node which looks the most promising (that is, the cost of reaching that state plus the heuristic value of that state is minimum).

The states are kept in a priority queue, with their priority the sum of their cost plus the heuristic evaluation. At each step, the algorithm removes the lowest priority item and places all of its children into the queue with the appropriate priority.

With an admissible heuristic function, the first end state that A* finds is guaranteed to be optimal.

Example Problems

Knight Cover [Traditional]

Place as few knights as possible on an *n x n* chess board so that every square is attacked. A knight is not considered to attack the square on which it sits.

Analysis: One possible heuristic function is the total number of squares attacked divided by eight (the number of squares a knight can attack). This can be used for any of the forms, although A* has problems, as the queue becomes large.

8-puzzle [Traditional]

Given a 3 x 3 grid of numbers, with one square free, you are allowed to move any number adjacent to the blank square from its current location to the blank square.

```
5 _ 4     5 1 4
```

```
6 1 8    6 _ 8
7 3 2    7 3 2
```

What is the minimum number of moves required to get back to the following state (you are guaranteed to be able to):

```
1 2 3
4 5 6
7 8 _
```

Heuristic function: The `taxi-cab' distance between each number and the location it's supposed to be in.

Analysis: Because the state space is fairly small (only 362,880), A* will work well, if you ensure no state is ever in the queue twice, and no visited state is ever put into the queue.