Heuristic Search

Prerequisite

- Recursive Descent

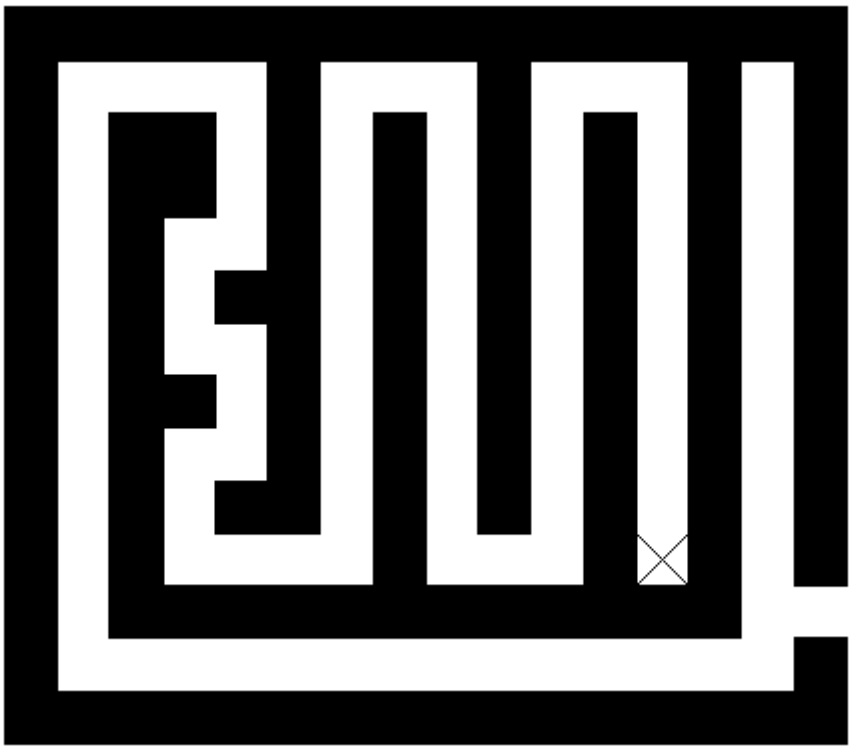
Main Concept

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:

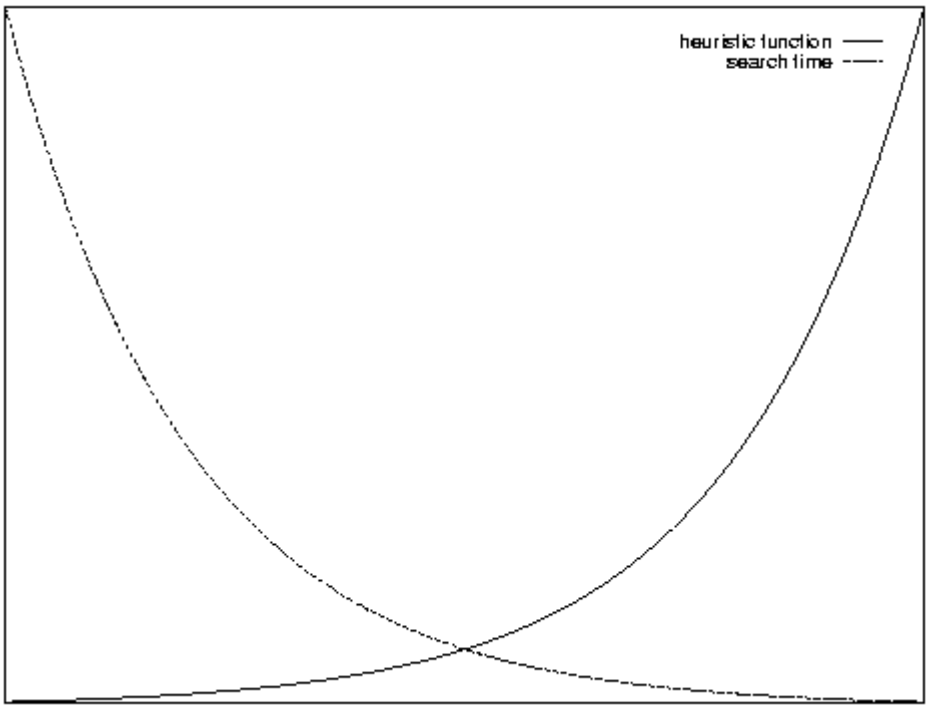
- When looking for an exit in a maze, the euclidean distance to the exit.
- When trying to win a game of checkers, the number of checkers you have minus the number of checkers your opponent has.
- When trying to win freecell, the number of cards already home plus 5 times the number of freecells that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

Heuristic Search

启发式搜索

译 By SuperBrother

知识准备

递归思想

主要思想

启发式搜索的主要思想是通过评价一个状态有"多好"来改进对于解的搜索.

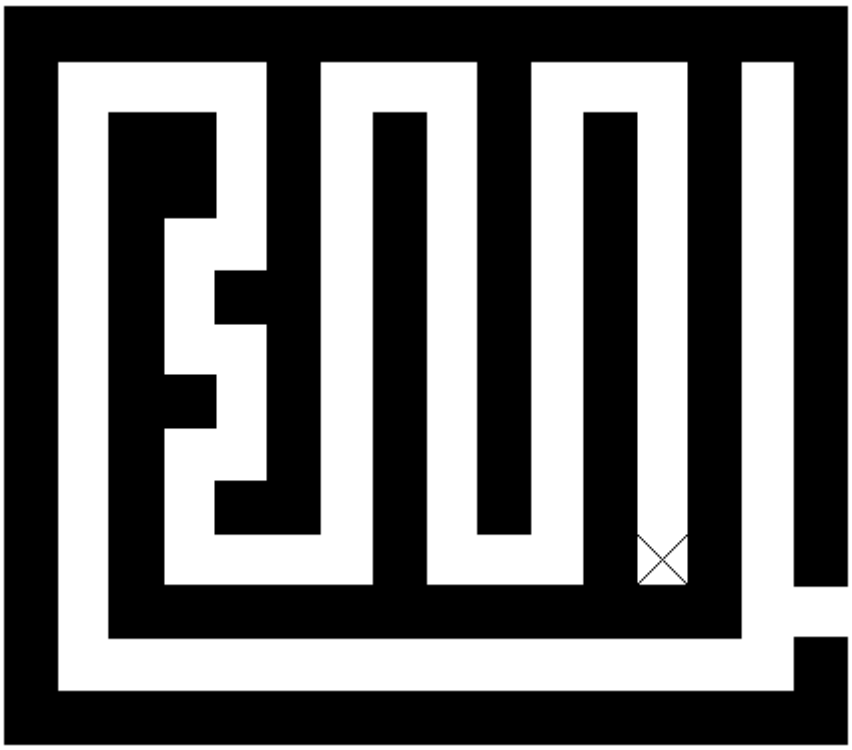
通常,我们用关于状态的一个函数来表示一个状态有多好.这种函数被称为估价函数.下面有估价函数的几个例子:

- 在迷宫中当前点至出口的欧氏距离;
- 在跳棋中你的跳棋数与对手的跳棋数之差;
- 空当接龙中放入空当的纸牌数加上空当数的5倍

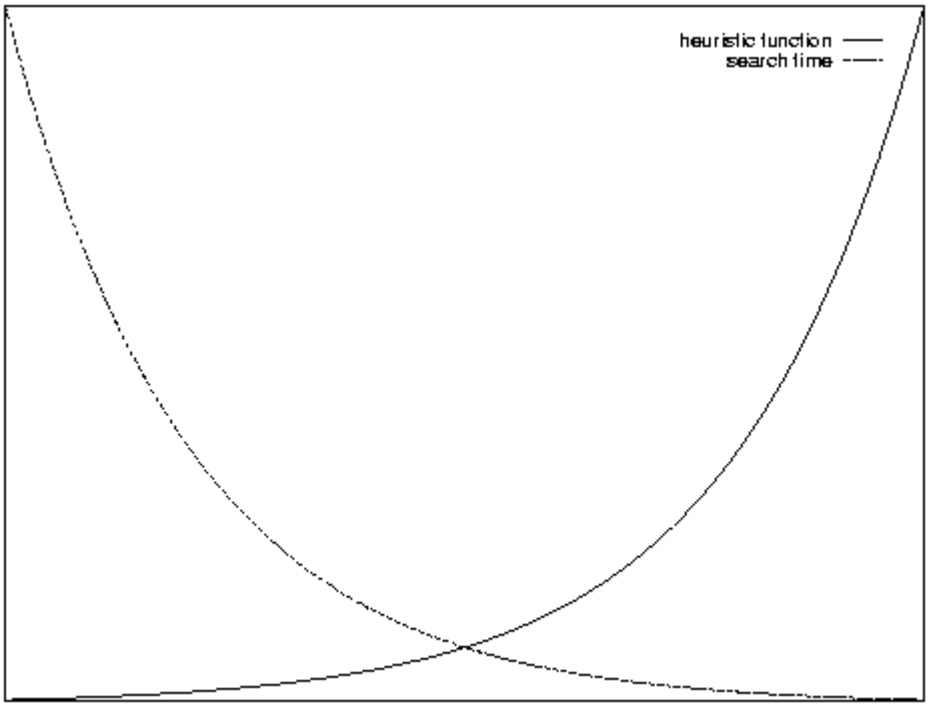
设计估价函数

直观上,强的估价函数一般使搜索更快.问题在于,如何评判一个估价函数的优劣?

评价估价函数的优劣,在于这个函数能够多好地表达状态的代价.例如,在迷宫中,如何估计到出口的距离?欧式距离很不好,甚至对于一些简单的迷宫,相同的解也会离的很远:



通常,估价函数越优,搜索越快,搜索时间和估价函数准确性有如下关系:



需要注意的是即使是很差的估价函数也能加快搜索速度(如果用的正确的话)

另外很重要的一点是设计估价函数时,考虑它的可行性.说一个估价函数有可行性,就是说这个函数低估了状态的代价,并且对于所有状态,是非负的.例如,在迷宫中欧式距离是不可行的,理由如上图所示.

方法#1:启发式剪枝

估价函数最简单最普通的用法是进行剪枝.假设有一个求最小代价的一个搜索,使用一个可行的估价函数.如果搜到当前状态时代价为 A,这个状态的估价函数是 B,那么从这个状态开始搜所能得到的最小代价是 A+B.如果当前最优解是 C 满足 C<A+B,那么就不需要从这个状态开始搜索.

把这个剪枝加入到一个朴素的搜索中,编程简单,易于调试,但可以带来时间效率的巨大提升.

方法#2:最佳优先搜索

最佳搜索可以看成贪心的深度优先搜索.

与一般搜索随意扩展后继节点不同,最优优先搜索按照估价函数所给的他们的"好坏"的顺序扩展节点.与只扩展最优节点的贪心不同,最佳优先搜索先扩展较"优"的后继,然后再扩展较"差"的后继.与上面的启发式剪枝组合,它可以很好地提升效率.

方法#3:A*搜索

A*搜索和"贪心"的广度优先搜索相似.

广度优先搜索总是扩展已达到的代价最小节点.但是,A*搜索,扩展最"有可能"的节点(就是说,到达1节点的代价和这个节点的估价函数之和最小)

状态被保存在优先队列中,优先级是状态的代价与估价函数之和.每一步,算法移除优先级最小的节点,将它的后继加入优先队列中.

如果估价函数可行,A*搜索到的第一个结束状态可以保证是最优的.

例题

骑士覆盖[经典问题]

在 n*n 的棋盘中放入尽可能少的骑士,使得所有的格子都能被攻击到.骑士所在的格子不会被自己攻击到.

算法:一个可行的估价函数是骑士攻击到的格子总数除以8(8即一个骑士能攻击的格子)(原句:the number of squares a knight can attack).这个可以被任意方式使用,虽然 A*由于队列的原因,空间需求量大.

8-方块[经典问题]

一个3*3的棋盘,一个格子为空,空格临近的数可以被移动到空格中.

5 _ 4---->5 1 4

6 1 8---->6 _ 8

7 3 2---->7 3 2

达到下面的状态所需最小步数是多少?(保证有解)

1 2 3

4 5 6

7 8 _

估价函数:起始状态到结束状态中所有数的位置的曼哈顿距离之和.

算法:状态空间很小(只有362,880),A*就很好,不过要保证每个状态只能被访问一次.