Informed search algorithms

Chapter 3, Section 3.6

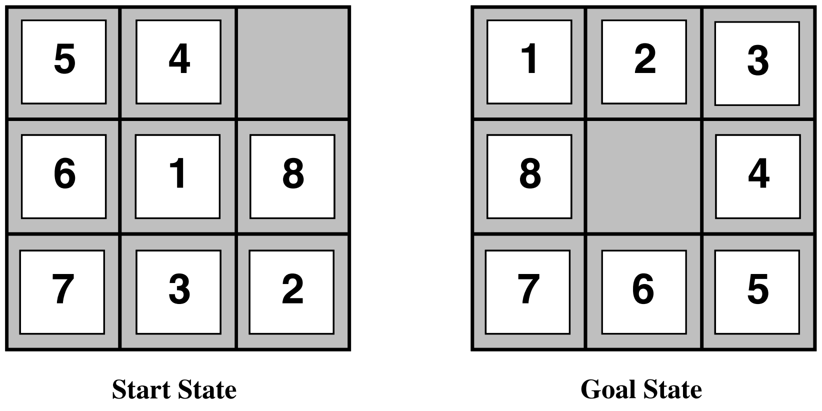
Admissible Heuristics

So far we have seen one heuristic - how do we devise new ones?

Also how important is it to find a good heuristic?

Heuristics - What for?

E.g., for the 8-puzzle:



What is the branching factor?

8-puzzle

$\diamondsuit$ b is about 3

$\diamondsuit$ Average solution takes about 20 steps

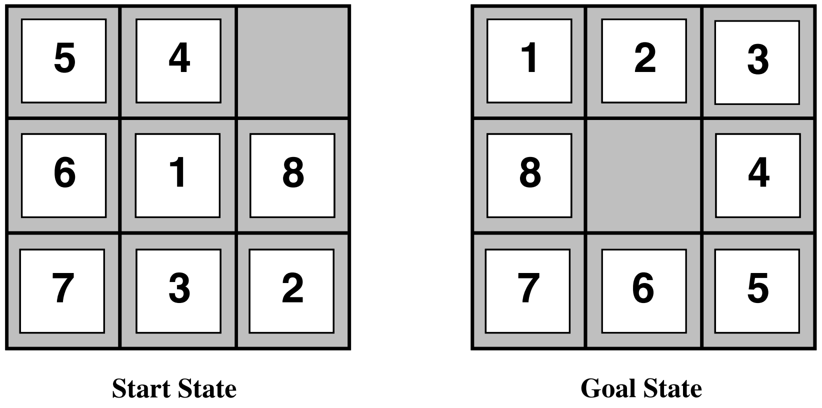
$\Rightarrow$ Exhaustive search algorithms would look at $3^{20} \approx 10^{10}$ nodes

$\diamondsuit$ But there are only 9! different states ($\approx 350,000$)

Hence, a good heuristic can reduce the search space drastically.

Admissible heuristics

E.g., for the 8-puzzle:

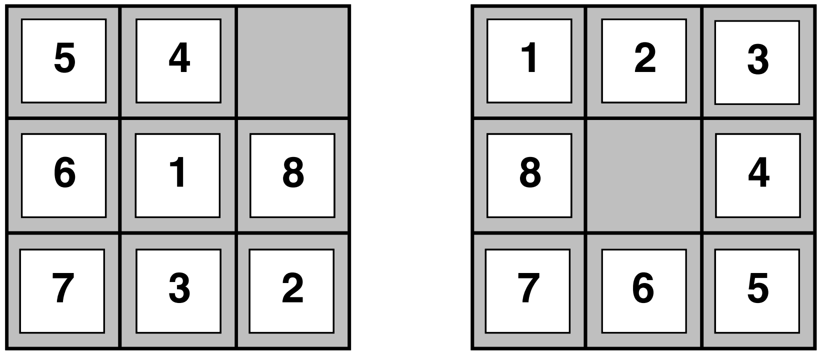


$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)

$h_1(S) = ??$
$h_2(S) = ??$

Admissible heuristics

E.g., for the 8-puzzle:



Start State Goal State

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)

$h_1(S) =$?? 7
$h_2(S) =$?? $4+2+2+2+2+3+3 = 18$ (tile5 would need to move 4 steps, tiles 3 and 2 need to move 3 steps each)

Which heuristic is better

Remember:

♢ We want admissable heuristics which always underestimate the remaining distance to the goal.

♢ We dont want the trivially admissable heuristic which says the remaining distance to the goal is 0 (or the like)

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible)
then h_2 *dominates* h_1 and is better for search

Dominance

Typical search costs:

$d = 14$ IDS = 3,473,941 nodes
A$^*(h_1)$ = 539 nodes
A$^*(h_2)$ = 113 nodes

$d = 24$ IDS $\approx$ 54,000,000,000 nodes
A$^*(h_1)$ = 39,135 nodes
A$^*(h_2)$ = 1,641 nodes

Note how much improvement there is between A* and IDS, but also between A* with a good and average heuristic (factor of 10-30).

Effective Branching Factor

◇ How many nodes are examined by a heuristic depends on the depth of the solution and the maximum depth of that particular problem, ordering of the branches etc...

◇ One good quantitative measure to compare two heuristics is to look at their **effective branching factors**: the branching factor of a *uniform tree* of depth d in order to contain n nodes

Effective Branching Factor

$\diamondsuit$ Effective Branching Factor b can be found by solving for b in the following formula, assuming a uniform tree with N nodes (visited nodes) and d depth (depth of solution).

Remember: $N = 1 + b + b^2 + ... + b^d = (b^{(d+1)} - 1)/(b - 1)$

$\diamondsuit$ The idea is to judge how the heuristics is able to **focus** the search (can they zoom in to the goal or are they almost blindly searching?).I.e. find the effective branching factor for the 3 algorithms below:

$d = 14$ IDS $=$ 3,473,941 nodes
A$^*(h_1)$ $=$ 539 nodes
A$^*(h_2)$ $=$ 113 nodes

Dominance and Effective Branching Factor

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

♢ N is found by averaging over 100 instances of the 8-puzzle problem, with solution lengths of 2,4,...20, since the search will be slightly different for a given starting state.

How to find a problem with known d?: Come back d steps from the solution...

Generating Admissable Heuristics

◇ A version of a problem with fewer constraints or fewer restrictions on the actions is called a *relaxed* version of the problem.

◇ Admissable heuristics can be generated using relaxed versions of a problem.

Generating Admissable Heuristics

Relaxed problems (hence heuristics) can be automatically generated if the problem can be defined using a formal language.

The relaxed form of the operators of the 8-puzzle can be written as:

A tile can move from square A to B, if A is adjacent to B and B is blank.

◇ A tile can move from square A to B, if ....

◇ A tile can move from square A to B, if ....

◇ A tile can move from square A to B, if ....

Generating Admissable Heuristics

Relaxed problems (hence heuristics) can be automatically generated if the problem can be defined using a formal language.

The relaxed form of the operators of the 8-puzzle can be written as:

A tile can move from square A to B, if A is adjacent to B and B is blank.

- ♢ A tile can move from square A to B, if A is adjacent to B
- ♢ A tile can move from square A to B, if B is blank
- ♢ A tile can move from square A to B

Which one corresponds to h1 or h2?

Relaxed problems

If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then $h_1(n)$ gives the shortest solution (A tile can move from square A to B)

If the rules are relaxed so that a tile can move to any adjacent square *even if occupied*, then $h_2(n)$ gives the shortest solution (A tile can move from square A to B, if A is adjacent to B)

What about: "A tile can move from square A to B, if B is blank"? (very close to h1).

Relaxed problems

♢ The cost of an optimal solution to a relaxed problem can be used as an admissable heuristic for the original problem! Why?

Proof:The optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem - hence the generated heuristic is admissable.

♢ Finding the cost of the optimal solution to the relaxed problem should be done without searching!

Generating Admissable Heuristics

ABSOLVER program can automatically generate heuristics using the ”relaxed problem” method (Prieditis 1993)

It generated the best heuristic thus far, for 8-puzzle and the first useful heuristic to the Rubik’s cube.

Generating Admissable Heuristics

Another way to generate heuristics is to use solutions to a **sub-problem** of the given problem.

e.g. only put the first 4 tiles into their correct positions.

It turns out that this can be substantially better than the relaxed problem solutions.

Heuristics Issues

- ♢ Can we combine heuristics to find a better one?
- ♢ What about the computational cost of the heuristic?
- ♢ Learning heuristics?

Heuristics Issues

♢ Can we combine heuristics to find a better one?

Use $h(n) = max(h1(n), ..., hm(n))$

If the component heuristics are admissable, h will also be admissable.

♢ What about the computational cost of the heuristic?!

It is clear that one should not do a breadth-first-search to compute a heuristic value!

Heuristics Issues

◊ Learning heuristics

Another solution to devising heuristics is to learn from experience. Experience here means solving lots of 8-puzzles, for instance.

"Each optimal solution to an 8-puzzle problem provides examples from which h(n) can be learned: Each example consists of a state from the solution path and the actual cost of the solution from that point. From these examples, a learning algorithm can be used to construct a function h(n) that can (with luck) predict solution costs for other states that arise during search. "

♢ Inductive learning methods work best when supplied with **features** of a state that are relevant to predicting the states value, rather than with just the raw state description.

♢ Number of misplaced tiles in 8-puzzle or number of pieces left, the area you control on the board etc in chess.

♢ The resulting algorithm may not be admissable or consistent.

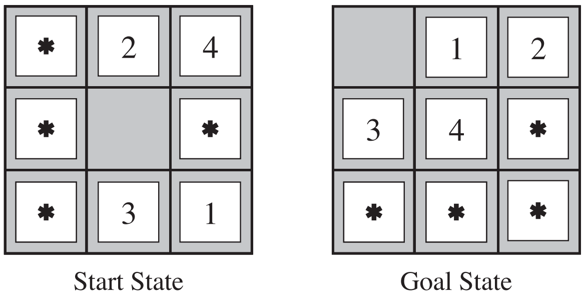


Figure 3.30 A subproblem of the 8-puzzle instance given in Figure 3.28. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

This composite heuristic uses whichever function is most accurate on the node in question. Because the component heuristics are admissible, h is admissible; it is also easy to prove that h is consistent. Furthermore, h dominates all of its component heuristics.

3.6.3 Generating admissible heuristics from subproblems: Pattern databases

SUBPROBLEM

Admissible heuristics can also be derived from the solution cost of a **subproblem** of a given problem. For example, Figure 3.30 shows a subproblem of the 8-puzzle instance in Figure 3.28. The subproblem involves getting tiles 1, 2, 3, 4 into their correct positions. Clearly, the cost of the optimal solution of this subproblem is a lower bound on the cost of the complete problem. It turns out to be more accurate than Manhattan distance in some cases.

PATTERN DATABASE

The idea behind **pattern databases** is to store these exact solution costs for every possible subproblem instance—in our example, every possible configuration of the four tiles and the blank. (The locations of the other four tiles are irrelevant for the purposes of solving the subproblem, but moves of those tiles do count toward the cost.) Then we compute an admissible heuristic h_{DB} for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database. The database itself is constructed by searching back[13] from the goal and recording the cost of each new pattern encountered; the expense of this search is amortized over many subsequent problem instances.

The choice of 1-2-3-4 is fairly arbitrary; we could also construct databases for 5-6-7-8, for 2-4-6-8, and so on. Each database yields an admissible heuristic, and these heuristics can be combined, as explained earlier, by taking the maximum value. A combined heuristic of this kind is much more accurate than the Manhattan distance; the number of nodes generated when solving random 15-puzzles can be reduced by a factor of 1000.

One might wonder whether the heuristics obtained from the 1-2-3-4 database and the 5-6-7-8 could be *added*, since the two subproblems seem not to overlap. Would this still give an admissible heuristic? The answer is no, because the solutions of the 1-2-3-4 subproblem and the 5-6-7-8 subproblem for a given state will almost certainly share some moves—it is

[13] By working backward from the goal, the exact solution cost of every instance encountered is immediately available. This is an example of **dynamic programming**, which we discuss further in Chapter 17.

unlikely that 1-2-3-4 can be moved into place without touching 5-6-7-8, and vice versa. But what if we don't count those moves? That is, we record not the total cost of solving the 1-2-3-4 subproblem, but just the number of moves involving 1-2-3-4. Then it is easy to see that the sum of the two costs is still a lower bound on the cost of solving the entire problem. This is the idea behind **disjoint pattern databases**. With such databases, it is possible to solve random 15-puzzles in a few milliseconds—the number of nodes generated is reduced by a factor of 10,000 compared with the use of Manhattan distance. For 24-puzzles, a speedup of roughly a factor of a million can be obtained.

DISJOINT PATTERN DATABASES

Disjoint pattern databases work for sliding-tile puzzles because the problem can be divided up in such a way that each move affects only one subproblem—because only one tile is moved at a time. For a problem such as Rubik's Cube, this kind of subdivision is difficult because each move affects 8 or 9 of the 26 cubies. More general ways of defining additive, admissible heuristics have been proposed that do apply to Rubik's cube (Yang *et al.*, 2008), but they have not yielded a heuristic better than the best nonadditive heuristic for the problem.

3.6.4 Learning heuristics from experience

A heuristic function $h(n)$ is supposed to estimate the cost of a solution beginning from the state at node n. How could an agent construct such a function? One solution was given in the preceding sections—namely, to devise relaxed problems for which an optimal solution can be found easily. Another solution is to learn from experience. "Experience" here means solving lots of 8-puzzles, for instance. Each optimal solution to an 8-puzzle problem provides examples from which $h(n)$ can be learned. Each example consists of a state from the solution path and the actual cost of the solution from that point. From these examples, a learning algorithm can be used to construct a function $h(n)$ that can (with luck) predict solution costs for other states that arise during search. Techniques for doing just this using neural nets, decision trees, and other methods are demonstrated in Chapter 18. (The reinforcement learning methods described in Chapter 21 are also applicable.)

FEATURE

Inductive learning methods work best when supplied with **features** of a state that are relevant to predicting the state's value, rather than with just the raw state description. For example, the feature "number of misplaced tiles" might be helpful in predicting the actual distance of a state from the goal. Let's call this feature $x_1(n)$. We could take 100 randomly generated 8-puzzle configurations and gather statistics on their actual solution costs. We might find that when $x_1(n)$ is 5, the average solution cost is around 14, and so on. Given these data, the value of x_1 can be used to predict $h(n)$. Of course, we can use several features. A second feature $x_2(n)$ might be "number of pairs of adjacent tiles that are not adjacent in the goal state." How should $x_1(n)$ and $x_2(n)$ be combined to predict $h(n)$? A common approach is to use a linear combination:

$$h(n) = c_1 x_1(n) + c_2 x_2(n) \,.$$

The constants c_1 and c_2 are adjusted to give the best fit to the actual data on solution costs. One expects both c_1 and c_2 to be positive because misplaced tiles and incorrect adjacent pairs make the problem harder to solve. Notice that this heuristic does satisfy the condition that $h(n) = 0$ for goal states, but it is not necessarily admissible or consistent.