Dipartimento di elettronica e informazione
Politecnico di Milano

Artificial Intelligence 2010-11

6. Informed search strategies

As we have already said, an *informed search strategy* is a search strategy that, besides accessing the *description of a problem* like all search strategies, can exploit additional information to optimise the process of finding a solution (not to optimise the solution itself!)

6.1 The A* strategy

As we already know, UC is an optimal search strategy that can applied to problems when action costs are not all identical. A* is a variant of UC, in which every node n in frontier is ordered according not to the cost of a path from the root to n, but according to a more complex measure, that gives an estimation of the cost of a complete optimal solution from the root to a goal, passing through node n.

More precisely, let n be any node in a search tree. We define:

$g(n)$ = cost of an *optimal* path from the root to n
$h(n)$ = cost of an *optimal* path from n to a goal node
$f(n) = g(n) + h(n)$

Therefore, $f(n)$ is the cost of an optimal solution going from the root to a goal, and subject to the further constraint of passing through node n.

If we apply a strategy like UC (under the hypotheses we have examined in Section 5.4), when we expand node n we already know the value of $g(n)$. In general, however, we do not know the value of $h(n)$, as this depends on a part of the tree that has not been built yet. Suppose, however, that we are able to compute an *estimation* $\tilde{h}(n)$ of $h(n)$; then, we can compute an estimation $\tilde{f}(n)$ of $f(n)$, that is:

$$\tilde{f}(n) = g(n) + \tilde{h}(n).$$

The A* search strategies is exactly like UC, with the only exception that it keeps the frontier ordered according to non-decreasing values of $\tilde{f}(n)$, instead of ordered according to non-decreasing values of $g(n)$. Note that when $\tilde{h}(n)$ is identically equal to 0, A* reduces to UC, which in turn reduces to BF if all action costs are identically equal to 1. Therefore, A* can be viewed as a generalisation of UC, which in turn is a generalisation of BF.

Function $\tilde{h}(n)$ is called a *heuristic function*[1]; we shall comment on it in Section 6.2.

*Optimality of A**

A* can be proved to be optimal under suitable conditions. First, we have to assume the optimality conditions of UC:

- every node of the search tree has a finite number of successors;
- there is a positive lower bound ε to the costs of actions; that is, for every action a and state s:
  $cost(a,s) \geq \varepsilon$, with $\varepsilon > 0$.

Then further conditions must be assumed. In the case of treeSearch, it is sufficient to assume that the heuristic function $\tilde{h}$ is *admissible*, that is,

$$\tilde{h}(n) \leq h(n).$$

[1] The term "heuristic" comes from the Greek verb *heuriskein*, "to discover." A *heuristic function*, in general, is a function that helps to discover a solution.

In other words, $\tilde{h}(n)$ must provide a *lower bound* on the actual value of $h(n)$. With graphSearch, however, this assumption is not sufficient to guarantee optimality; we must also assume that the heuristic function $\tilde{h}$ is *consistent* (or *monotone*), that is:

- for every pair (n,m) of nodes in the tree, such that node m is reached from node n by executing action a, then:

  $\tilde{h}(n) \leq cost(a,n) + \tilde{h}(m)$

- for every goal node g,

  $\tilde{h}(g) = 0$.

This conditions guarantee that function $\tilde{f}$ is monotone (i.e., does not decrease) when one goes from n to m by executing action a:

$$\begin{aligned}\tilde{f}(n) &= g(n) + \tilde{h}(n)\\ &\leq g(n) + cost(a,n) + \tilde{h}(m)\\ &= g(m) + \tilde{h}(m)\\ &= \tilde{f}(m)\end{aligned}$$

Note that consistency implies admissibility: therefore, every heuristic function that makes A* graphSearch optimal, also makes A* treeSearch optimal. To see that this is the case, suppose that from a generic node n_1 to a goal node g can be optimally reached by executing actions a_1, a_2, ..., a_k (and thus passing through nodes n_2, n_3, ..., n_k, g). Then consistency guarantees that:

$$\begin{aligned}\tilde{h}(n_1) &\leq cost(a_1,n_1) + \tilde{h}(n_2)\\ &\leq cost(a_1,n_1) + cost(a_2,n_2) + \tilde{h}(n_3)\\ &\leq ...\\ &\leq cost(a_1,n_1) + cost(a_2,n_2) + ... + cost(a_k,n_k) + \tilde{h}(g)\\ &= cost(a_1,n_1) + cost(a_2,n_2) + ... + cost(a_k,n_k) + 0\\ &= h(n_1)\end{aligned}$$

Finally, note that the heuristic function

$\tilde{h}(n) = 0$ for all n

is consistent (adopting this heuristic function turns A* into UC).

*The effect of A**

If we adopt an admissible (or consistent) heuristic function, then A* search is optimal. This is good news, but in fact UC is already optimal, and this means that *A* will return the same solution returned by UC* (if any): so, what is the advantage of using A*?

It can be proved that, to find a solution, A* generates fewer nodes than UC. In other words, it acts by lowering the average branching factor of the search tree, from the 'real' branching factor b to an 'apparent' branching factor $b^* \leq b$. This results into lower memory and time complexities.

The apparent branching factor is lower the closer $\tilde{h}(n)$ gets to $h(n)$. In other words, a better estimation of the cost from n to a goal produces a more efficient A* search.

6.2 Heuristic functions

Where does the $\tilde{h}$ function come from? Typically, $\tilde{h}$ is defined 'externally' and then fed into the function implementing A* as additional information.

Defining a good heuristic function requires understanding certain structural properties of the state space, which in turn derive from concrete properties of the environment that is modelled by the state space. For example, the following heuristic functions $\tilde{h}_1$ and $\tilde{h}_2$ are commonly used for the 8-puzzle:

- the heuristic function

  $\tilde{h}_1(n)$ = number of misplaced tiles in the state of n

  which captures the idea that every misplaced tile will have to be moved at least once to reach its goal position;

- the heuristic function

  $\tilde{h}_2(n)$ = sum of Manhattan distances of misplaced tiles to correct locations in the state of n

  which captures the idea that every misplaced tile will have to be moved at least k times to reach its goal position, if k is the Manhattan distance of the tile from its goal position.

For every heuristic function it is important to prove consistency (if the graph search version of A* is used) or at least admissibility (if the tree search version of A* is used). We now do this for the heuristic function $\tilde{h}_1$ (the proof for $\tilde{h}_2$ is left to the reader). Let node m be reachable from node n by executing action a. There are two possibilities: (i), action a does not bring any misplaced tile to its goal position; (ii), action a brings exactly one misplaced tile to its goal position. Now:

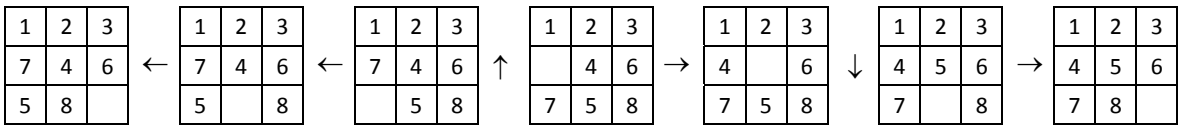
in case (i): $\tilde{h}(n) = \tilde{h}(m)$
in case (ii): $\tilde{h}(n) = \tilde{h}(m) + 1$

As $cost(a,n) = 1$, in both cases we have:

$$\tilde{h}(n) \leq cost(a,n) + \tilde{h}(m)$$

6.3 An 8-puzzle problem

Here is an 8-puzzle problem that admits of an optimal (i.e., shortest) solution of length 6:



Branching factor

The complete search tree (*with* state repetitions) down to level 6 has $N = 585$ nodes, so distributed:

level	0	1	2	3	4	5	6
nodes	1	2	6	16	48	128	384

Remembering that $N = (b^{d+1}–1)/(b–1)$, the empirical value of b (*with* state repetitions) for this problem is $b = 2.68$.

The complete search tree (*without* state repetitions) down to level 6 has $N = 91$ nodes, so distributed:

level	0	1	2	3	4	5	6
nodes	1	2	4	8	16	20	40

The empirical value of b (*without* state repetitions) for this problem is therefore $b = 1.87$.

BF graph search

Ordering the actions as $\rightarrow \uparrow \leftarrow \downarrow$, BF graph search generates 88 of the 91 nodes (see the tree at pages 5 and 6).

A* graph search with $\tilde{h}_1$ heuristic function

A* graph search with the consistent heuristic function

$\tilde{h}_1(n)$ = number of misplaced tiles in the state of n

generates 20 nodes, so distributed (see the tree at page 7):

level	0	1	2	3	4	5	6
nodes	1	2	4	7	2	3	1

with an empirical value of $b_1^* = 1.35$.

A* graph search with $\tilde{h}_2$ heuristic function

A* graph search with the consistent heuristic function

$\tilde{h}_2(n)$ = sum of Manhattan distances of misplaced tiles to correct locations in the state of n

generates 14 nodes, so distributed (see the tree at page 7):

level	0	1	2	3	4	5	6
nodes	1	2	4	1	2	3	1

with an empirical value of $b_2^* = 1.23$.

A* graph search with exact *h*

A* graph search with the consistent heuristic function

$h(n)$ = exact distance from n to the goal

generates 12 nodes, so distributed (see the tree at page 8):

level	0	1	2	3	4	5	6
nodes	1	2	2	1	2	3	1

with an empirical value of $b^* = 1.18$.

The total number of nodes is approximately equal to $b \cdot 6$ (with the empirical value of b computed before):

$$1.87 \cdot 6 = 11.22 \approx 12$$

In fact, when there is only one optimal solution and the exact distance $h(n)$ is known, no 'useless' part of the search tree is generated, and therefore the total number of generated nodes is $b \cdot d$ (linear in d).

As we already know, if the

$\tilde{h}_0(n) = 0$ (for every n)

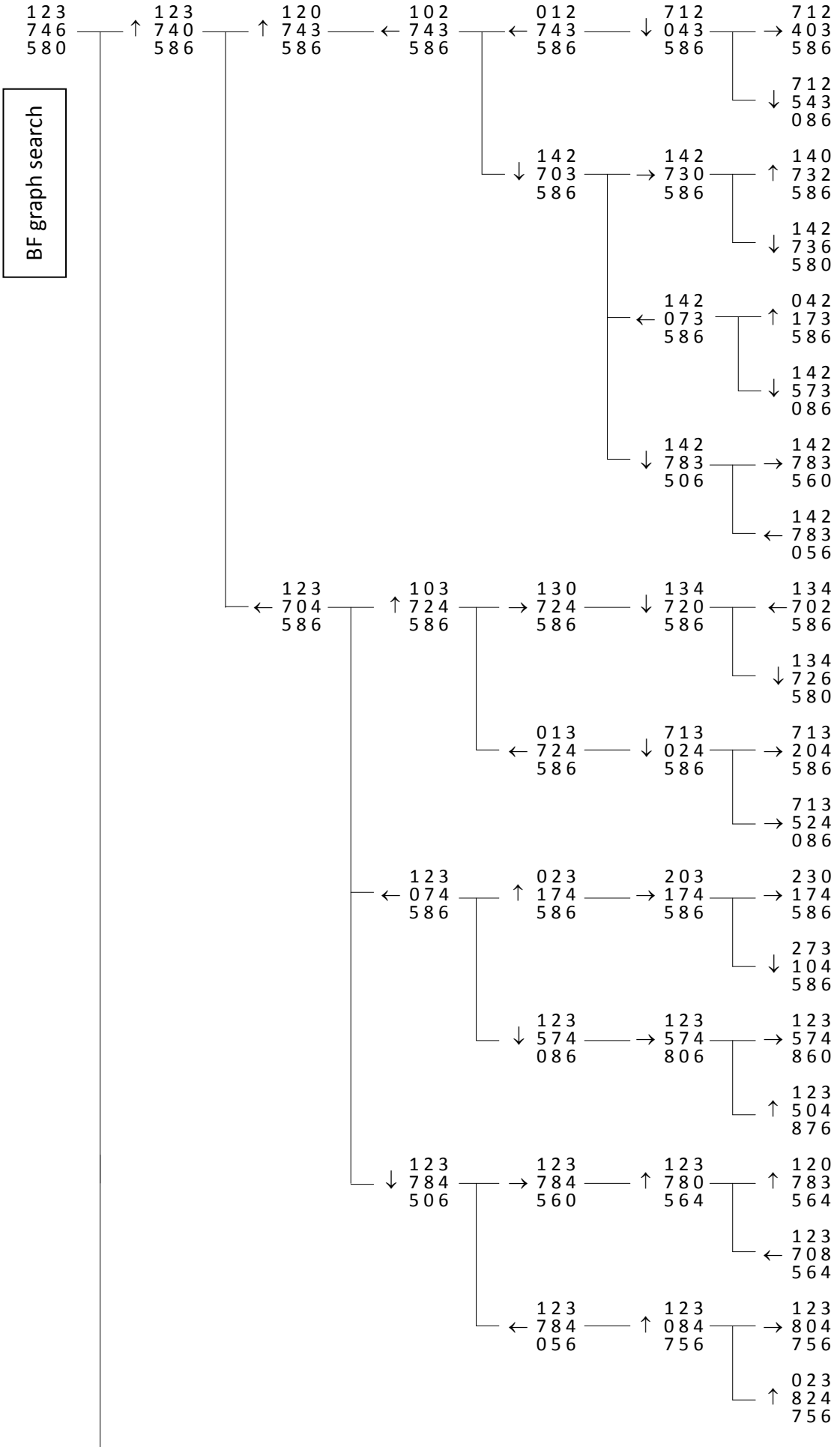
A* search reduces to UC search, which with equal action costs reduces to BF search. For every node n, we have

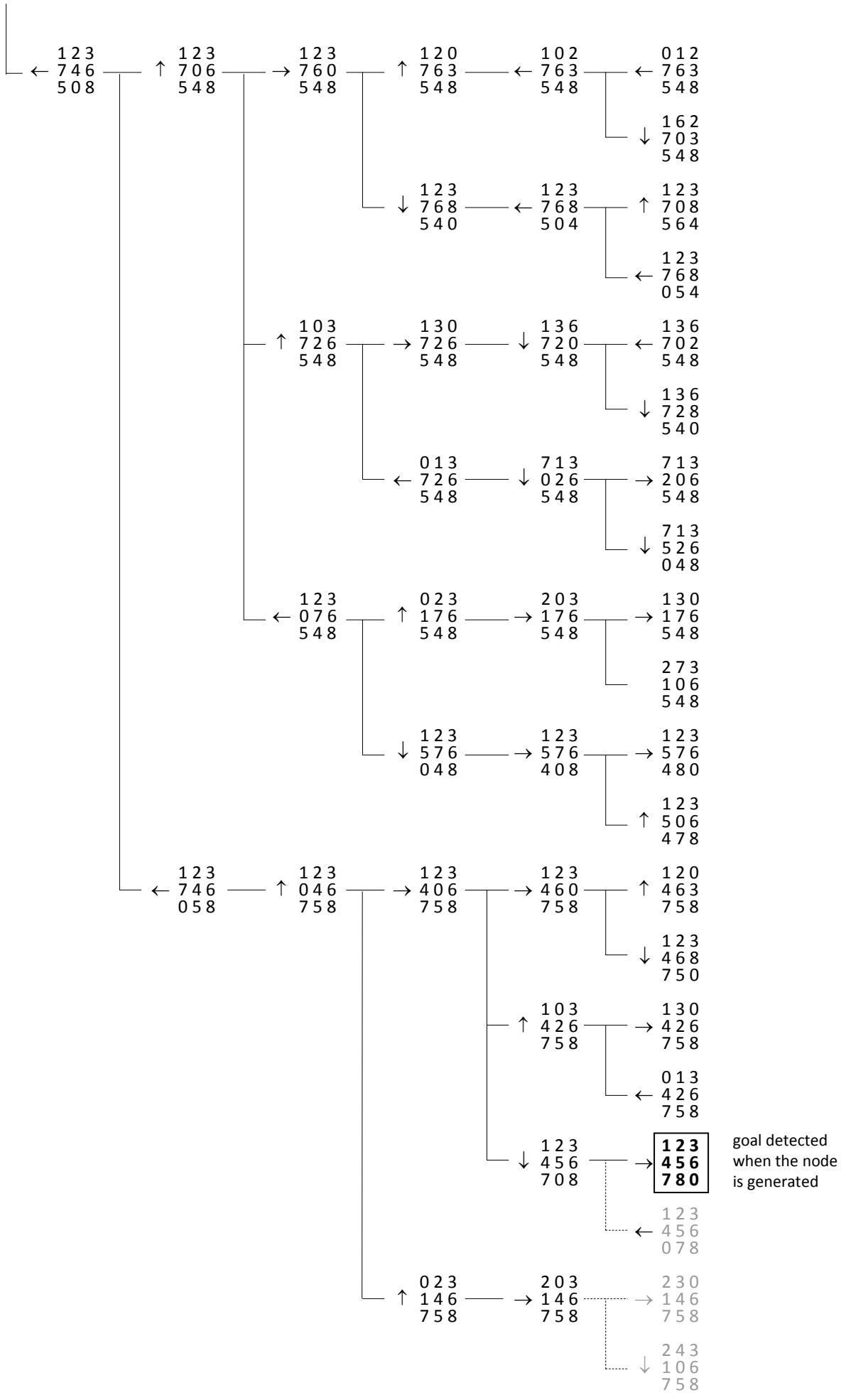
$$\tilde{h}_0(n) < \tilde{h}_1(n) \leq \tilde{h}_2(n) \leq h(n)$$

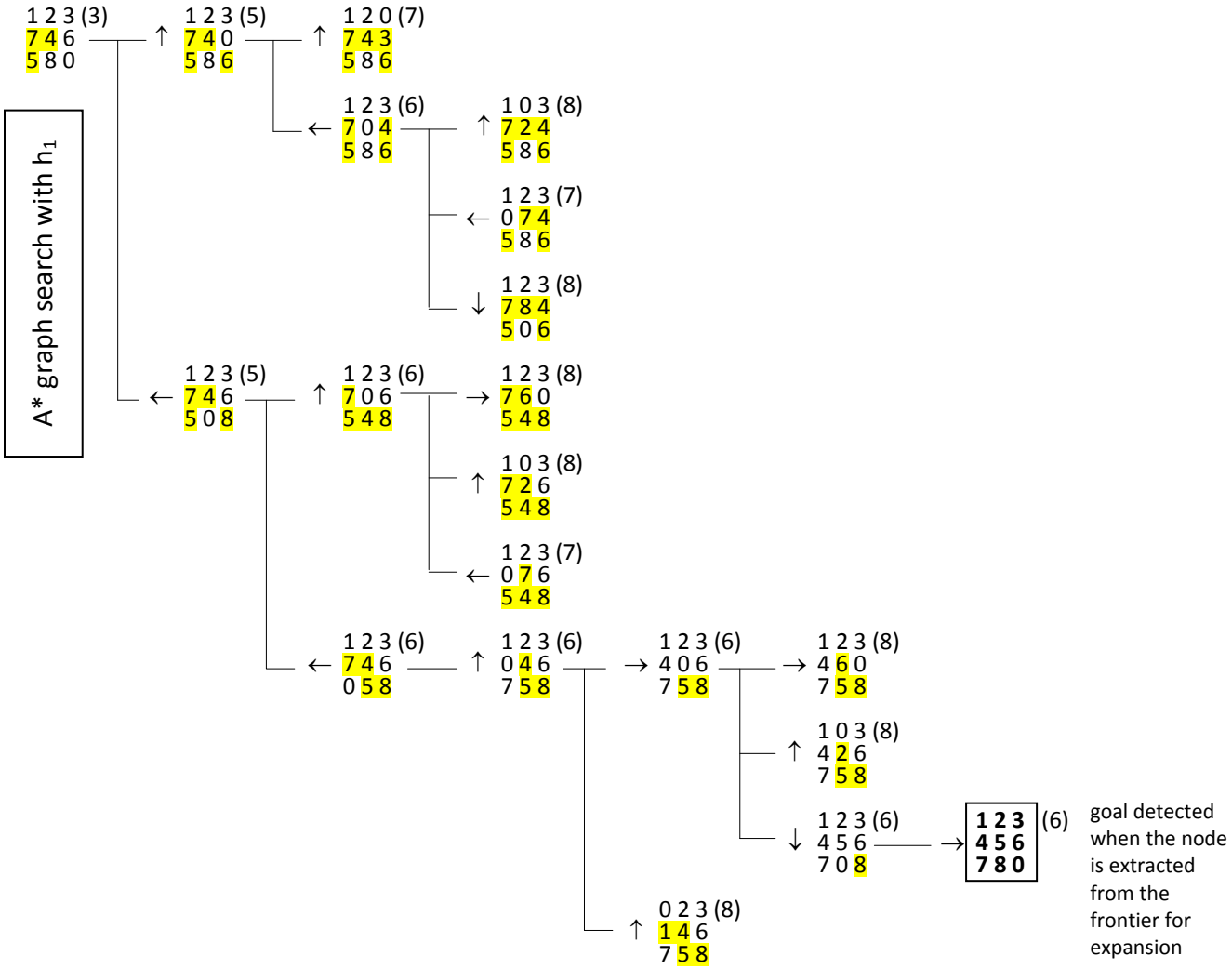
At the same time, we have:

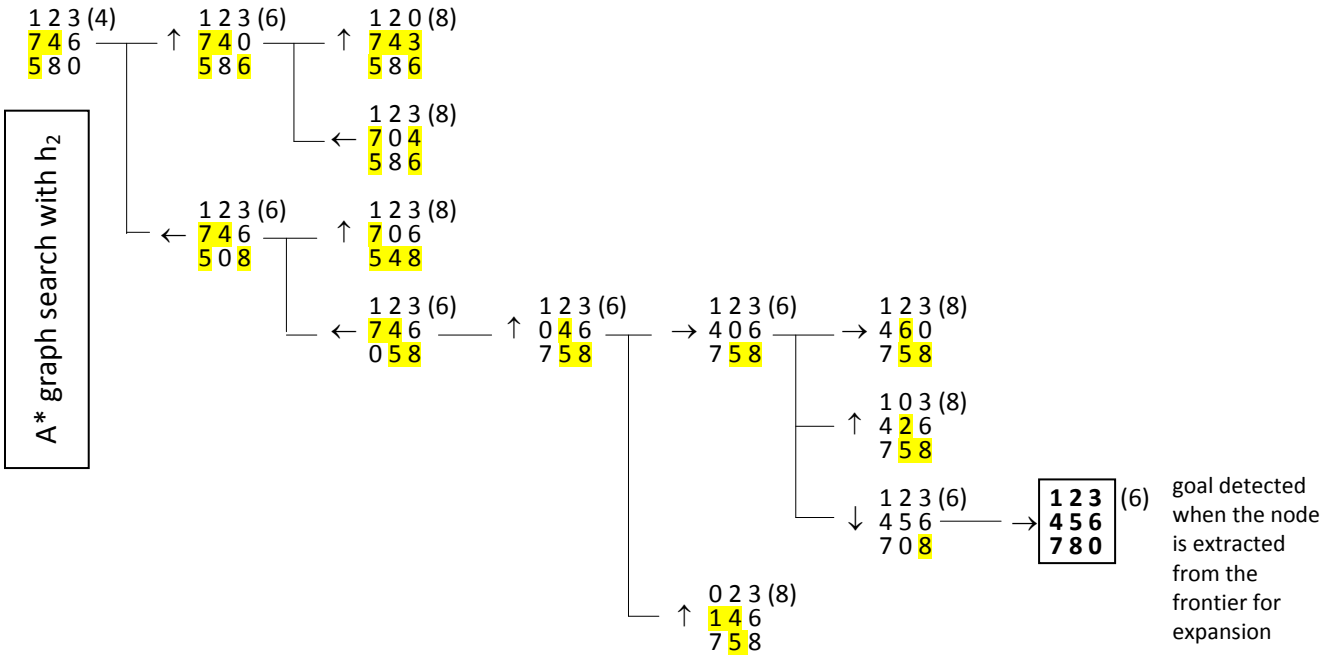
$$b_0 > b_1^* > b_2^* > b^*$$

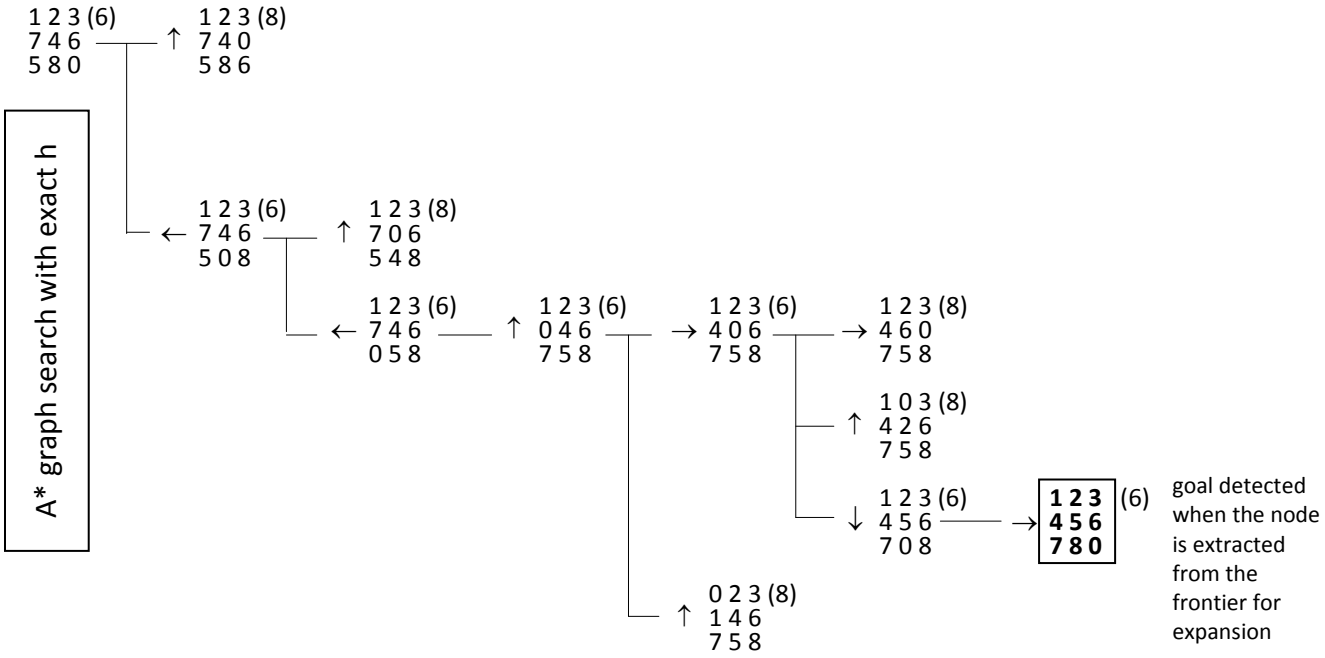
In general, the closed a heuristic function gets to the real distance of a node from the goal, the smaller is the branching factor (and therefore the number of generated nodes).


BF graph search
1 2 3
7 4 6
5 8 0
↑ 1 2 3
7 4 0
5 8 6
↑ 1 2 0
7 4 3
5 8 6
← 1 0 2
7 4 3
5 8 6
← 0 1 2
7 4 3
5 8 6
↓ 7 1 2
0 4 3
5 8 6
→ 7 1 2
4 0 3
5 8 6
↓ 7 1 2
5 4 3
0 8 6
↓ 1 4 2
7 0 3
5 8 6
→ 1 4 2
7 3 0
5 8 6
↑ 1 4 0
7 3 2
5 8 6
↓ 1 4 2
7 3 6
5 8 0
← 1 4 2
0 7 3
5 8 6
↑ 0 4 2
1 7 3
5 8 6
↓ 1 4 2
5 7 3
0 8 6
↓ 1 4 2
7 8 3
5 0 6
→ 1 4 2
7 8 3
5 6 0
← 1 4 2
7 8 3
0 5 6
← 1 2 3
7 0 4
5 8 6
↑ 1 0 3
7 2 4
5 8 6
→ 1 3 0
7 2 4
5 8 6
↓ 1 3 4
7 2 0
5 8 6
← 1 3 4
7 0 2
5 8 6
↓ 1 3 4
7 2 6
5 8 0
← 0 1 3
7 2 4
5 8 6
↓ 7 1 3
0 2 4
5 8 6
→ 7 1 3
2 0 4
5 8 6
→ 7 1 3
5 2 4
0 8 6
← 1 2 3
0 7 4
5 8 6
↑ 0 2 3
1 7 4
5 8 6
→ 2 0 3
1 7 4
5 8 6
→ 2 3 0
1 7 4
5 8 6
↓ 2 7 3
1 0 4
5 8 6
↓ 1 2 3
5 7 4
0 8 6
→ 1 2 3
5 7 4
8 0 6
→ 1 2 3
5 7 4
8 6 0
↑ 1 2 3
5 0 4
8 7 6
↓ 1 2 3
7 8 4
5 0 6
→ 1 2 3
7 8 4
5 6 0
↑ 1 2 3
7 8 0
5 6 4
↑ 1 2 0
7 8 3
5 6 4
← 1 2 3
7 0 8
5 6 4
← 1 2 3
7 8 4
0 5 6
↑ 1 2 3
0 8 4
7 5 6
→ 1 2 3
8 0 4
7 5 6
↑ 0 2 3
8 2 4
7 5 6


goal detected
when the node
is generated


A* graph search with h1
goal detected when the node is extracted from the frontier for expansion


A* graph search with h2
goal detected when the node is extracted from the frontier for expansion


A* graph search with exact h
1 2 3 (6)
7 4 6
5 8 0
1 2 3 (8)
7 4 0
5 8 6
1 2 3 (6)
7 4 6
5 0 8
1 2 3 (8)
7 0 6
5 4 8
1 2 3 (6)
7 4 6
0 5 8
1 2 3 (6)
0 4 6
7 5 8
1 2 3 (6)
4 0 6
7 5 8
1 2 3 (8)
4 6 0
7 5 8
1 0 3 (8)
4 2 6
7 5 8
1 2 3 (6)
4 5 6
7 0 8
1 2 3
4 5 6
7 8 0
(6)
goal detected when the node is extracted from the frontier for expansion
0 2 3 (8)
1 4 6
7 5 8