Department of Electronics, Information, and Bioengineering
Politecnico di Milano

Artificial Intelligence 2019-20

Marco Colombetti

3. State Space Search

3.1 The basic concepts

In the definition of a State Space Problem (SSP; see the textbook, Chapter 3), it is useful to distinguish two different components: a *state space*, and a *state space problem*. The state space can be regarded as a highly idealised representation of an agent's environment, and constitutes a framework in which many different SSPs can be specified. A state space **S** is constituted by the 5-tuple

$$\mathbf{S} = \langle S, A, Actions, Result, Cost \rangle$$

where:

- S is a nonempty, countable set of possible *states*
- A is a nonempty, finite set of possible *actions*
- *Actions* is a function assigning to every state $s \in S$ the set $Action(s) \subseteq A$ of all actions that the agent can execute in s
- *Result* is the function assigning to every action $a \in Action(s)$ and every state $s \in S$ the state $Result(a,s) = s' \in S$ resulting from the execution of action a in s
- *Cost* is the function that assigns to every action $a \in Action(s)$ and every state $s \in S$ the cost $Cost(a,s) = c$ of executing action a in s (a positive real number).

A state space problem can be viewed as a triple

$$\mathbf{P} = \langle \mathbf{S}, s_0, G \rangle$$

where:

- $\mathbf{S} = \langle S, A, Actions, Result, Cost \rangle$ is a state space
- $s_0 \in S$ is the initial state, and
- $G \subseteq S$ is the goal set.

The concepts of a solution, total cost of a solution, and optimal solution are now defined within a previously specified state space problem $\mathbf{P} = \langle \mathbf{S}, s_0, G \rangle$:

- the action sequence $\mathbf{a} = \langle a_1, ..., a_n \rangle$ is a *solution* if its execution transforms the initial state into a goal state
- $TotalCost(s_0,\mathbf{a})$, the *total cost* of the action sequence **a** starting from state s_0, is the sum total of the costs of all actions in the sequence
- **a*** is an *optimal solution* if it is a solution and no other solution exists with a lower total cost.

3.2 Search

The state space as a graph

A state space **S** = ⟨*S*, *A*, *Actions*, *Result*, *Cost*⟩ can be viewed as a graph. Suppose that *s*, *s'* ∈ *S*, and *a* ∈ *A* is an action such that: *a* ∈ *Action*(*s*), *s'* = *Result*(*s*,*a*), and *c* = *Cost*(*s*,*a*). This can be represented as a labelled edge of a directed graph as follows:

$$\begin{tikzcd} \boxed{s} \arrow[r, "a{,}c"] & \boxed{s'} \end{tikzcd}$$

For example, the state space of the 8-puzzle (textbook, Section 3.2.1) consists of 9! = 362,880 states, which form two disjoint subgraphs of equal size (181,440 states each). For example, states

$$\begin{matrix}1&2&3\\4&5&6\\7&8&0\end{matrix} \qquad \text{and} \qquad \begin{matrix}1&2&3\\4&5&6\\8&7&0\end{matrix}$$

are in two different partitions, and therefore cannot be reached from each other by performing the allowed actions (→, ↑, ←, ↓).

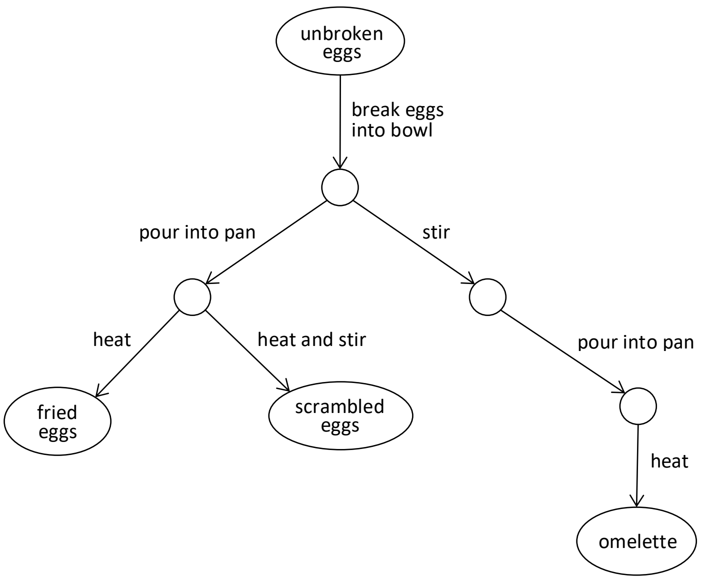
Searching for a solution may be regarded as looking for a path in the state space graph (SSG), from the initial state to any state contained in the goal set. However, the SSG is typically very large (or even infinite), and we do not want (or even cannot) represent it explicitly as a data structure in memory.

Different types of trees

In graph theory, a *tree* is usually defined as an undirected graph that has exactly one path connecting any pair of nodes. In dealing with state space search, however, we are interested in directed trees. We can distinguish between two types of directed trees:

- *out-trees*: an out-tree is a directed graph with a distinguished node n_0, called the *root*, from which every other node *n* can be reached by exactly one directed path from n_0 to *n*
- *in-trees*: an in-tree is a directed graph with a distinguished node n_0, also called the *root*, which can be reached from every other node *n* by exactly one directed path from *n* to n_0

The *search trees* built by search processes are in-trees, because nodes point to their parents. In general, the state space graph is not a tree: for example, neither the simplified road map of Romania (textbook, Fig. 3.2) nor the state space of the 8-puzzle are trees. A state space can occasionally be a tree, which means that in general a state can be reached from another state by at most one sequence of actions; here is an example (warning: this is a depiction of a state space, not of a search tree, and this is why it is not an in-tree!):



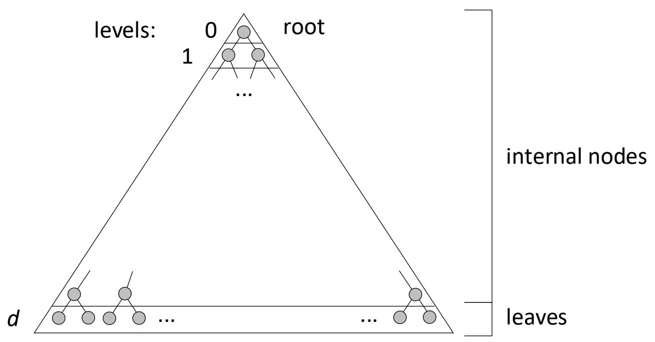
Tree and graph search

A search process looks for a solution of a state space problem by building a search tree **T**. Basically we can distinguish between tree-search and graph-search processes. It is important to bear in mind that both types of processes generate a search *tree*, and not a search *graph*! The difference between the two is that tree search is suitable for searching state spaces that are themselves trees, and graph search is suitable for searching state spaces that are themselves graphs.

The size of trees

Trees have some counterintuitive properties, due to their exponential growth when moving from the root to the lower levels. Let us start with the two factors determining the number of nodes of a tree: its *depth* and its *branching factor*.

The depth d of a tree is the number of distinct levels in the tree, starting from level 0 for the root:



If a tree has at least one infinite branch, then its depth is infinite, and the tree is also infinite (in the sense that it has an infinite number of nodes). Moreover, if we assume that a tree is *finitely branching* (i.e., every node has a finite number of successors), then the tree is infinite only if it has at least one infinite branch (this is a corollary of *König's Lemma* of graph theory).

We shall call a tree *regular* if every internal node (i.e., every node that is not a leaf) has exactly b successors, for some natural number $b > 1$.[1] Number b is called the *branching factor* of the tree. Moreover, we shall call a tree *complete* if all its leaves are at the same level (see the figure above). Levels are counted from 0 (the root's level), and the maximum level is called the *depth* of the tree (of course, a tree with an infinite branch does not have a finite depth).

If a tree is both regular and complete and has depth d, then every level $k \le d$ contains exactly b^k nodes and therefore:

- the total number of nodes (i.e., internal nodes + leaves) is

$$totalNodesl(b,d) = \sum_{k=0}^{k=d} b^k = \frac{b^{d+1}-1}{b-1}$$

- the total number of internal nodes is

$$internalNodes(b,d) = \sum_{k=0}^{k=d-1} b^k = \frac{b^d-1}{b-1}$$

- the total number of leaves is

$$leaves(b,d) = b^d = (b-1)\cdot internalNodes(b,d) + 1$$

[1] The value $b = 0$ would not make sense, because by definition an internal node has at least one successor. For $b = 1$ the tree would degenerate into a linear list.

The last formula, neglecting the "+ 1" term, shows that expanding one more level of a tree costs $b-1$ times as much as expanding *all previous levels*. Thus, although this may be counterintuitive, for $b>2$ the number of leaves is approximately equal to the number of all internal nodes. With $b = 5$ and $d = 10$ we have:

$$\begin{aligned} totalNodes(5,10) &= 12{,}207{,}031 \\ internalNodes(5,10) &= 2{,}441{,}406 \\ leaves(5,10) &= 9{,}765{,}625 \end{aligned}$$

This implies that when dealing with trees, even generating one level less will make a big difference.

In general, in a tree the number of successors varies from node to node. One may still try to use the formulas derived for uniform, complete trees, interpreting b as the *effective branching factor* of a tree of depth d, defined as the positive real number $b > 1$ such that the total number of nodes of the tree is equal to

$$\frac{b^{d+1}-1}{b-1}$$

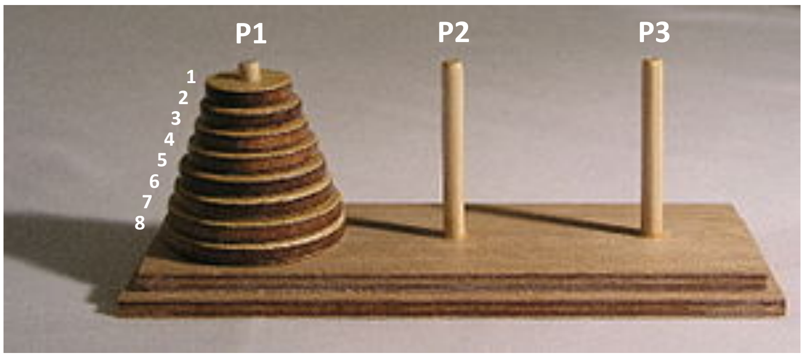
The concept of effective branching factor turns out to be important to understand the complexity of certain search strategies, like A*.

3.3 The GraphSearch algorithms at page 77

What is stored in the explored set is a set of *states*, not a set of *nodes*. In fact the algorithm avoids adding to the frontier the nodes *whose state* is part of another node that has already been generated. Note that the version of the algorithm for breadth first search does not contain this error.

3.4 Another example of SSP: the Tower of Hanoi

The Tower of Hanoi is a puzzle that we shall now use as an example of state space representation. The puzzle consists of $N \geq 1$ disks (numbered from 1 to N) that can slide in three pegs (P1, P2, P3), like in the figure below ($N = 8$; modified from https://en.wikipedia.org/wiki/Tower_of_Hanoi). The disk have different sizes, with the diameter of disk n always greater than the diameter of disk m when $n > m$.



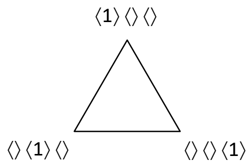
An action consists in picking up one disk from the top of a stack and sliding it in another peg. In no case, however, may a larger disk be placed above a smaller one. A typical problem is defined as follows:

- initial state: all disks in the same peg (say P1)
- goal: all disks in another peg (say P3)

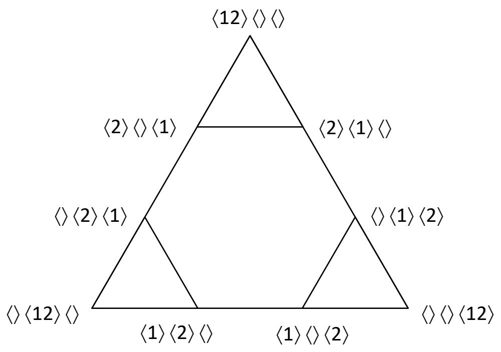
We shall now use this puzzle as an exercise in the definition of a state space ⟨S, A, *Actions*, *Result*, *Cost*⟩. (In fact we shall define a family **ToH**(N) of state spaces, parametric in the value of N.)

- S = three totally ordered, possibly empty sequences D_k (k = 1, 2, 3), such that their element sets are mutually disjoint and their union is the set {1, 2, ..., N}. Every sequence D_k represents the disks on peg Pk, from top to bottom.
- A = {move(P1,P2), move(P2,P1), move(P1,P3), move(P3,P1), move(P2,P3), move(P3,P2)}. (Note that according to the State Space approach the elements of A should be understood as *six different actions*, not as one parametric action with six different combinations of arguments.)
- *Actions*(s) = action move(Pk,Ph) can be executed in s if, and only if: (i), D_k is nonempty; and (ii), either D_h is empty or the first element of D_k is smaller than the first element of D_h.
- *Result*(s,move(Pk,Ph)) = the first element of D_k is removed and inserted at the beginning of D_h.
- *Cost*(s,move(Pk,Ph)) = 1.

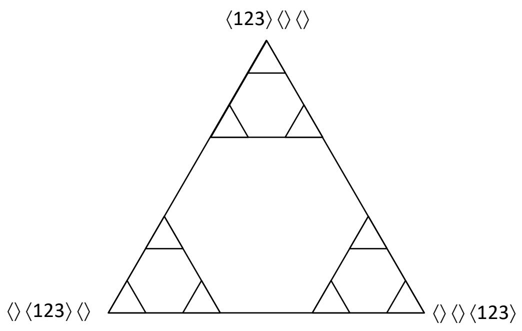
The graph of the **ToH**(1) is depicted below. Every node of the graph is labelled with the representation of a state, and every edge represents two actions (inverse of each other):

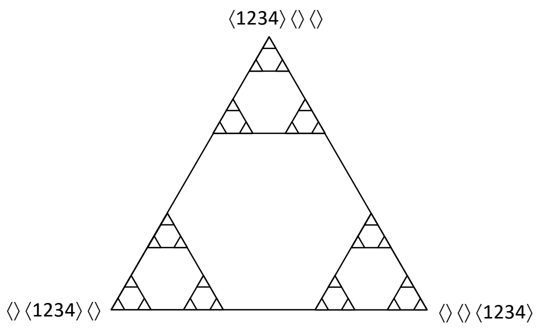


The graph of **ToH**(2) is as follows:



Analogously, these are the shapes of **ToH**(3) and **ToH**(4) (the representations of the intermediate states are not shown):





As it appears from these diagrams, the graph of **ToH**(N) has the structure of a *Sierpinski triangle* (https://en.wikipedia.org/wiki/Sierpinski_triangle). Now consider the *canonical problem* in **ToH**(N), that is, the problem with

$s_0 = \langle\langle 1,\ldots,N\rangle, \langle\ \rangle, \langle\ \rangle\rangle$ $G = \{\langle\langle\ \rangle, \langle\ \rangle, \langle 1,\ldots,N\rangle\rangle\}$

Then it is easy to prove that the optimal solution has length $2^N - 1$. The solution (which consists of the right side of the largest triangle in the above diagrams) can be found using the graph search algorithm (because all actions are invertible, and therefore tree search would generate lots of repeated states), for example with an ID strategy (to keep the memory complexity linear).

The Tower of Hanoi, however, is different from other puzzles (like for example the 8-puzzle) in that it admits of a general representation of the solution in closed form, that is, as a function that can directly compute the solution *without performing any search*. This function can be defined as follows. Let s be a data structure representing a state, and move(x,y,s) a function that returns the representation of the state obtained by moving one disk from peg x to peg y starting from state s. Then the following recursive function will solve the problem of moving N disks from x to y, with z as intermediate peg:

```
solve(N,x,y,z,s) {
    if (N == 1) return move(x,y,s);
    s1 = solve(N–1,x,z,y,s);
    s2 = move(x,y,s1);
    s3 = solve(N–1,z,y,x,s2);
    return s3;
}
```

Here is another form, which exploits a conditional statement and composition of functions:

```
solve(N,x,y,z,s) {
    return (N == 1)  ?  move(x,y,s)  :  solve(N–1,z,y,x,move(x,y,solve(N–1,x,z,y,s)));
}
```

To solve the canonical problem for $N = 4$ one would now compute:

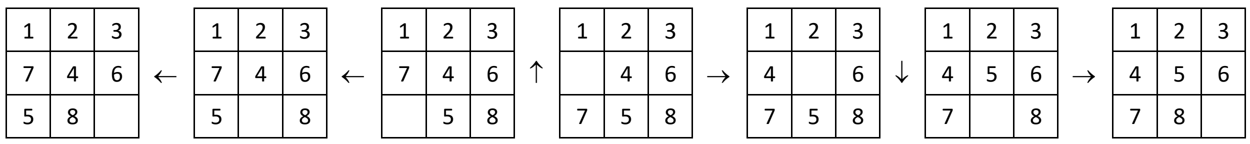
```
solve( 4, P1, P3, P2, ⟨⟨1,2,3,4⟩,⟨ ⟩,⟨ ⟩⟩ )
```

(Note that function solve does not return a representation of the solution: as an exercise, the reader may want to modify the function so that it outputs a sequence of actions that represents a solution.)

Of course, when there exists a closed-form representation of the solution, applying a search method is useless; therefore, nobody would seriously consider State Space search to deal with the Tower of Hanoi. On the other hand, there are plenty of cases (like the 8-puzzle) for which no closed-form representation of a solution is available; in such cases, search cannot be avoided.

3.5 A* search: an example

We shall now discuss an example, based on an 8-puzzle problem that admits of an optimal (i.e., shortest) solution of length 6:



All the computations can be checked on the search trees shown at the end of this document.

BF tree search

The complete tree-search tree (*with* state repetitions) down to level 6 contains 585 nodes, so distributed:

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

To compute the effective branching factor b (*with* state repetitions) for this problem we exploit the formula

$$totalNodesl(b,d) = \sum_{k=0}^{k=d} b^k = \frac{b^{d+1}-1}{b-1}$$

Now, in our case $d = 6$ and $totalNodes(b,d) = 585$. From this we can compute $b = 2.68$.

BF graph search (see search tree ①)

The complete graph-search tree (*without* state repetitions) down to level 6 contains 91 nodes, so distributed:

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

The empirical value of the effective branching factor (*without* state repetitions) for this problem is therefore $b_0 = 1.87$. Ordering the actions as $\rightarrow \uparrow \leftarrow \downarrow$, BF graph search generates 88 of the 91 nodes.

A graph search with h_1 heuristic function (see search tree ②)*

A* graph search with the consistent heuristic function

$h_1(n)$ = number of misplaced tiles in the state of n

generates 21 nodes, so distributed (see the tree at page 8):

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

with effective branching factor $b_1 = 1.36$.

A graph search with h_2 heuristic function (see search tree ③)*

A* graph search with the consistent heuristic function

$h_2(n)$ = sum of the Manhattan distances of all misplaced tiles to their correct locations in the state of n

generates 15 nodes, so distributed:

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

with effective branching factor $b_2 = 1.25$.

A graph search with exact distance h (see search tree ④)*

A* graph search with the consistent heuristic function

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

generates 13 nodes, so distributed:

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

with effective branching factor $b = 1.20$. The total number of nodes is approximately equal to $b_0 \cdot 6$ (with the empirical value of b_0 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 approximately $b \cdot d$ (linear in d).

As we already know, if

$h_0(n) = 0$ (for every n)

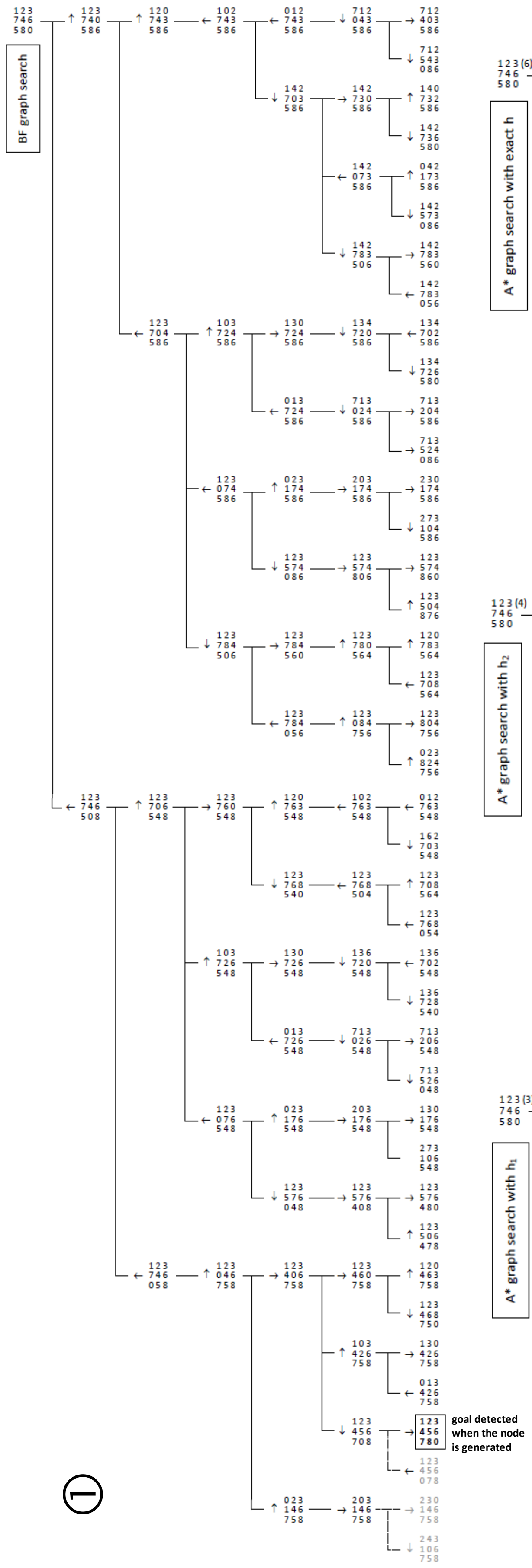
then A* search reduces to UC search, which with equal action costs reduces to BF search. For every node n, we have

$$h_0(n) < h_1(n) \leq h_2(n) \leq h(n)$$

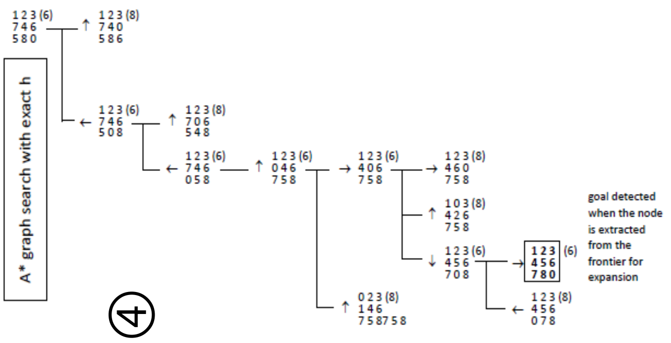
At the same time, we have:

$$b_0 > b_1 > b_2 > b$$

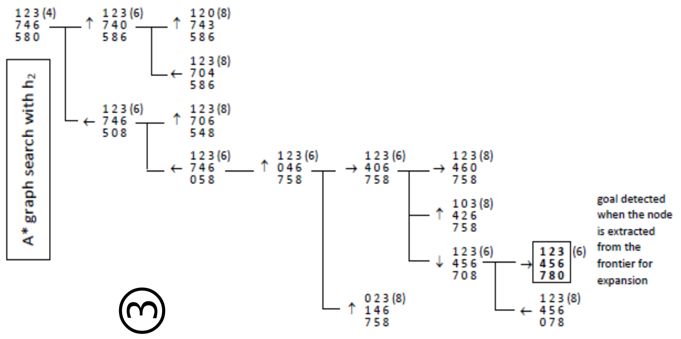
In general, the closer a heuristic function gets to the real distance of a node from the goal, the smaller is the effective branching factor (and therefore the number of generated nodes).


BF graph search
goal detected when the node is generated

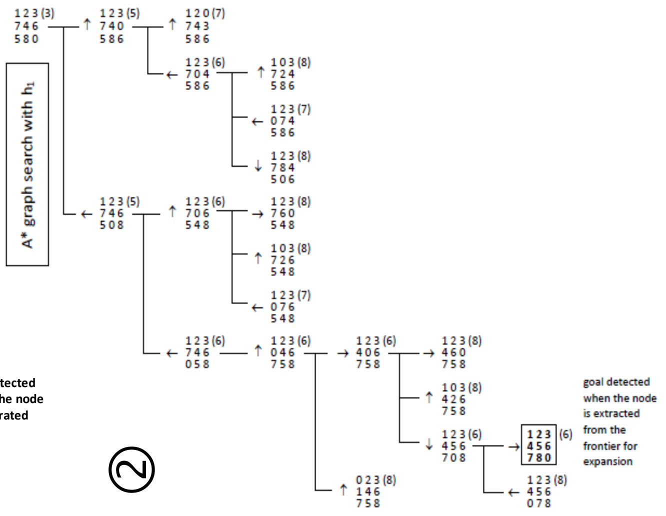
1


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

4


A* graph search with h_2
goal detected when the node is extracted from the frontier for expansion

3


A* graph search with h_1
goal detected when the node is extracted from the frontier for expansion

2