SMA* search

IDA*'s difficulties in certain problem spaces can be traced to using *too little* memory. Between iterations, it retains only a single number, the current f-cost limit. Because it cannot remember its history, IDA* is doomed to repeat it. This is doubly true in state spaces that are graphs rather than trees (see Section 3.6). IDA* can be modified to check the current path for repeated states, but is unable to avoid repeated states generated by alternative paths.

In this section, we describe the **SMA*** (Simplified Memory-Bounded A*) algorithm, which can make use of all available memory to carry out the search. Using more memory can only improve search efficiency—one could always ignore the additional space, but usually it is better to remember a node than to have to regenerate it when needed. SMA* has the following properties:

- It will utilize whatever memory is made available to it.
- It avoids repeated states as far as its memory allows.
- It is complete if the available memory is sufficient to store the *shallowest* solution path.
- It is optimal if enough memory is available to store the shallowest optimal solution path. Otherwise, it returns the best solution that can be reached with the available memory.
- When enough memory is available for the entire search tree, the search is optimally efficient.

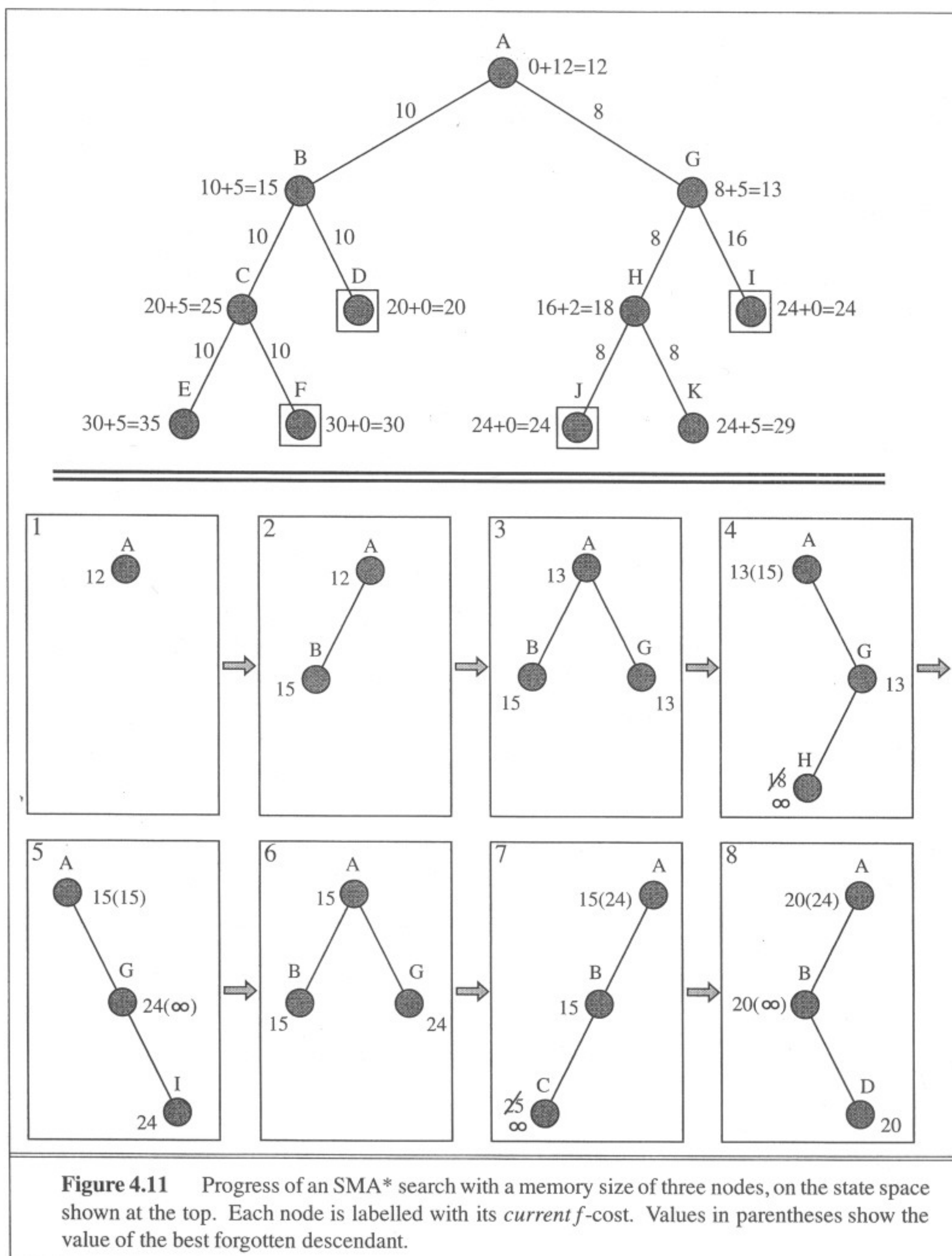

Figure 4.11 Progress of an SMA* search with a memory size of three nodes, on the state space shown at the top. Each node is labelled with its *current* f-cost. Values in parentheses show the value of the best forgotten descendant.

The one unresolved question is whether SMA* is always optimally efficient among all algorithms given the same heuristic information and the same memory allocation.

The design of SMA* is simple, at least in overview. When it needs to generate a successor but has no memory left, it will need to make space on the queue. To do this, it drops a node from the queue. Nodes that are dropped from the queue in this way are called **forgotten nodes**. It prefers to drop nodes that are unpromising—that is, nodes with high f-cost. To avoid reexploring subtrees that it has dropped from memory, it retains in the ancestor nodes information about the quality of the best path in the forgotten subtree. In this way, it *only* regenerates the subtree when *all other paths* have been shown to look worse than the path it has forgotten. Another way of saying this is that if all the descendants of a node n are forgotten, then we will not know which way to go from n, but we will still have an idea of how worthwhile it is to go anywhere from n.

SMA* is best explained by an example, which is illustrated in Figure 4.11. The top of the figure shows the search space. Each node is labelled with $g + h = f$ values, and the goal nodes (D, F, I, J) are shown in squares. The aim is to find the lowest-cost goal node with enough memory for only *three* nodes. The stages of the search are shown in order, left to right, with each stage numbered according to the explanation that follows. Each node is labelled with its current f-cost, which is continuously maintained to reflect the least f-cost of any of its descendants.[4] Values in parentheses show the value of the best forgotten descendant. The algorithm proceeds as follows:

1. At each stage, one successor is added to the deepest lowest-f-cost node that has some successors not currently in the tree. The left child B is added to the root A.
2. Now $f(A)$ is still 12, so we add the right child G (f = 13). Now that we have seen all the children of A, we can update its f-cost to the minimum of its children, that is, 13. The memory is now full.
3. G is now designated for expansion, but we must first drop a node to make room. We drop the shallowest highest-f-cost leaf, that is, B. When we have done this, we note that A's best forgotten descendant has f = 15, as shown in parentheses. We then add H, with $f(H) = 18$. Unfortunately, H is not a goal node, but the path to H uses up all the available memory. Hence, there is no way to find a solution through H, so we set $f(H) = \infty$.
4. G is expanded again. We drop H, and add I, with $f(I) = 24$. Now we have seen both successors of G, with values of ∞ and 24, so $f(G)$ becomes 24. $f(A)$ becomes 15, the minimum of 15 (forgotten successor value) and 24. Notice that I is a goal node, but it might not be the best solution because A's f-cost is only 15.
5. A is once again the most promising node, so B is generated for the second time. We have found that the path through G was not so great after all.
6. C, the first successor of B, is a nongoal node at the maximum depth, so $f(C) = \infty$.
7. To look at the second successor, D, we first drop C. Then $f(D) = 20$, and this value is inherited by B and A.
8. Now the deepest, lowest-f-cost node is D. D is therefore selected, and because it is a goal node, the search terminates.

---

[4] Values computed in this way are called **backed-up values**. Because $f(n)$ is supposed to be an estimate of the least-cost solution path through n, and a solution path through n is bound to go through one of n's descendants, backing up the least f-cost among the descendants is a sound policy.

In this case, there is enough memory for the shallowest optimal solution path. If J had had a cost of 19 instead of 24, however, SMA* still would not have been able to find it because the solution path contains four nodes. In this case, SMA* would have returned D, which would be the best reachable solution. It is a simple matter to have the algorithm signal that the solution found may not be optimal.

A rough sketch of SMA* is shown in Figure 4.12. In the actual program, some gymnastics are necessary to deal with the fact that nodes sometimes end up with some successors in memory and some forgotten. When we need to check for repeated nodes, things get even more complicated. SMA* is the most complicated search algorithm we have seen yet.

```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  local variables: Queue, a queue of nodes ordered by f-cost

  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
     if Queue is empty then return failure
     n ← deepest least-f-cost node in Queue
     if GOAL-TEST(n) then return success
     s ← NEXT-SUCCESSOR(n)
     if s is not a goal and is at maximum depth then
        f(s) ← ∞
     else
        f(s) ← MAX(f(n), g(s)+h(s))
     if all of n's successors have been generated then
        update n's f-cost and those of its ancestors if necessary
     if SUCCESSORS(n) all in memory then remove n from Queue
     if memory is full then
        delete shallowest, highest-f-cost node in Queue
        remove it from its parent's successor list
        insert its parent on Queue if necessary
     insert s on Queue
  end
```

Figure 4.12 Sketch of the SMA* algorithm. Note that numerous details have been omitted in the interests of clarity.

Given a reasonable amount of memory, SMA* can solve significantly more difficult problems than A* without incurring significant overhead in terms of extra nodes generated. It performs well on problems with highly connected state spaces and real-valued heuristics, on which IDA* has difficulty. On very hard problems, however, it will often be the case that SMA* is forced to continually switch back and forth between a set of candidate solution paths. Then the extra time required for repeated regeneration of the same nodes means that problems that would be practically solvable by A*, given unlimited memory, become intractable for SMA*. That is to say, memory limitations can make a problem intractable from the point of view of computation time. Although there is no theory to explain the trade-off between time and memory, it seems that this is an inescapable problem. The only way out is to drop the optimality requirement.