```
function IDA*(problem) returns a solution sequence
   inputs: problem, a problem
   local variables: f-limit, the current f-COST limit
                    root, a node

   root ← MAKE-NODE(INITIAL-STATE[problem])
   f-limit ← f-COST(root)
   loop do
      solution, f-limit ← DFS-CONTOUR(root, f-limit)
      if solution is non-null then return solution
      if f-limit = ∞ then return failure; end
---------------------------------------------------------------
function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f-COST limit
   inputs: node, a node
           f-limit, the current f-COST limit
   local variables: next-f, the f-COST limit for the next contour, initially ∞

   if f-COST[node] > f-limit then return null, f-COST[node]
   if GOAL-TEST[problem](STATE[node]) then return node, f-limit
   for each node s in SUCCESSORS(node) do
      solution, new-f ← DFS-CONTOUR(s, f-limit)
      if solution is non-null then return solution, f-limit
      next-f ← MIN(next-f, new-f); end
   return null, next-f
```

Figure 4.10 The IDA* (Iterative Deepening A*) search algorithm.

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.

SMA*

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.