function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with node as the only element
explored ← an empty set

loop do
  if EMPTY?(frontier) then return failure
  node ← POP(frontier) /* chooses the lowest-cost node in frontier */
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  add node.STATE to explored
  for each action in problem.ACTIONS(node.STATE) do
    child ← CHILD-NODE(problem, node, action)
    if child.STATE is not in explored or frontier then
      frontier ← INSERT(child, frontier)
    else if child.STATE is in frontier with higher PATH-COST then
      replace that frontier node with child

Figure 3.13 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure ??, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for frontier needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff

return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff

if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
else if limit = 0 then return cutoff
else
  cutoff_occurred? ← false
  for each action in problem.ACTIONS(node.STATE) do
    child ← CHILD-NODE(problem, node, action)
    result ← RECURSIVE-DLS(child, problem, limit - 1)
    if result = cutoff then cutoff_occurred? ← true
    else if result ≠ failure then return result
  if cutoff_occurred? then return cutoff else return failure

Figure 3.16 A recursive implementation of depth-limited tree search.