```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                      // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                      // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F, frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

Figure 3.14 Bidirectional best-first search keeps two frontiers and two tables of reached states. When a path in one frontier reaches a state that was also reached in the other half of the search, the two paths are joined (by the function JOIN-NODES) to form a solution. The first solution we get is not guaranteed to be the best; the function TERMINATED determines when to stop looking for new solutions.

For this to work, we need to keep track of two frontiers and two tables of reached states, and we need to be able to reason backwards: if state s' is a successor of s in the forward direction, then we need to know that s is a successor of s' in the backward direction. We have a solution when the two frontiers collide.[9]

There are many different versions of bidirectional search, just as there are many different unidirectional search algorithms. In this section, we describe bidirectional best-first search. Although there are two separate frontiers, the node to be expanded next is always one with a minimum value of the evaluation function, across either frontier. When the evaluation

[9] In our implementation, the *reached* data structure supports a query asking whether a given state is a member, and the frontier data structure (a priority queue) does not, so we check for a collision using *reached*; but conceptually we are asking if the two frontiers have met up. The implementation can be extended to handle multiple goal states by loading the node for each goal state into the backwards frontier and backwards reached table.