Depth-First and Breadth-First Search

In addition to specifying a search direction (data-driven or goal-driven), a search algorithm must determine the order in which states are examined in the tree or the graph. This section considers two possibilities for the order in which the nodes of the graph are considered: *depth-first* and *breadth-first* search.

Consider the graph represented in Figure 3.15. States are labeled (A, B, C ....) so that they can be referred to in the discussion that follows. In depth-first search, when a state is examined all of its children and their descendants are examined before any of its siblings. Depth-first search goes deeper into the search space whenever this is possible. Only when no further descendants of a state can be found are its siblings considered. Depth-first search examines the states in the graph of Figure 3.15 in the order A, B, E, K, S, L, T, F, M, C, G, N, H, 0, P, U, D, I, Q, J, R. The backtrack algorithm of Section 3.2.2 implemented depth-first search.

Breadth-first search, in contrast, explores the space in a level-by-level fashion. Only when there are no more states to be explored at a given level does the algorithm move on to the next level. A breadth-first search of the graph of Figure 3.15 considers the states in the order A, B, C, D, E, F, G, H, I, J, K, L, M, N, 0, P, Q, R, S, T, U.

We implement breadth-first search using lists, open and closed, to keep track of progress through the state space. open, like NSL in backtrack, lists states that have been generated but whose children have not been examined. The order in which states are removed from open determines the order of the search. closed records states already examined. closed is the union of the DE and SL lists of the backtrack algorithm.

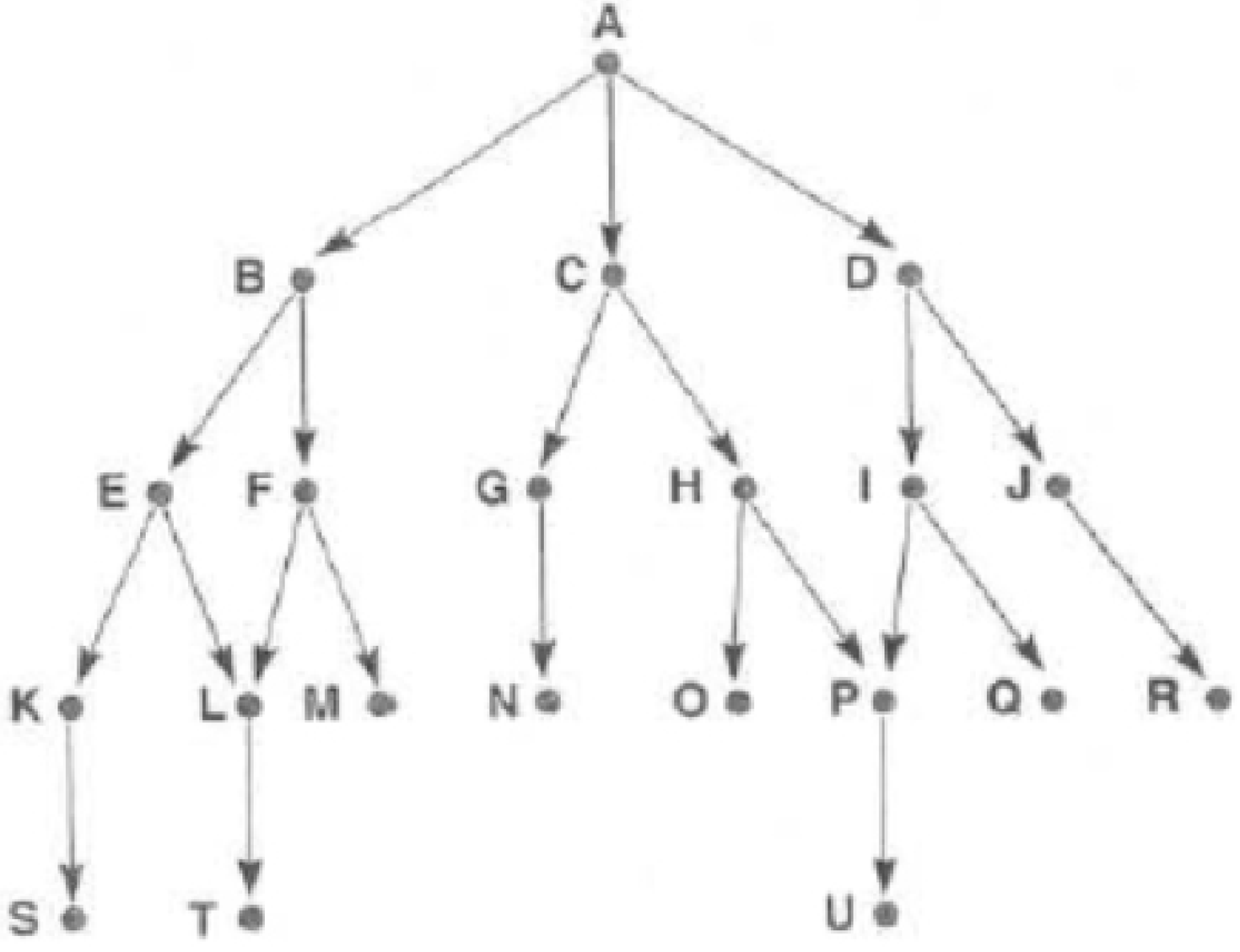

Figure 3.15 Graph for breadth- and depth-first search examples.

```
function breadth_first_search;

begin
   open := [Start];                          % initialize
   closed := [ ];
   while open  ≠ [ ] do                              % states remain
      begin
         remove leftmost state from open, call it X;
            if X is a goal then return SUCCESS                    % goal found
        else begin
            generate children of X;
            put X on closed;
            discard children of X if already on open or closed;  % loop check
            put remaining children on right end of open          % queue
         end
       end
   return FAIL                               % no states left
end.
```