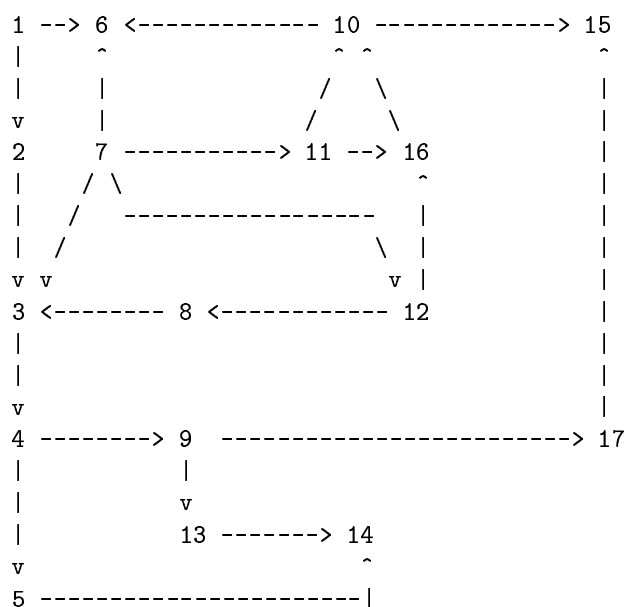


Lecture 25: A Simple Depth First State Space Search

This lecture describes a general depth first search algorithm and its application to the parsing problem. A Scheme implementation is developed. Breadth first search is also described.

An abstract state space

The following is a diagram of a state space. Each number represents a state, and there is an arrow from a state i to a state j if there is an operation that produces state j from state i . (In our example of parsing context free grammars, the states are finite sequences of nonterminal and terminal symbols and each operation replaces the leftmost nonterminal by the righthand side of a production whose lefthand side is equal to the nonterminal.)



Suppose the start state is 7 and the goal state is 14. Our computational task is to discover whether there is a directed path of arrows from state 7 to state 14. (Indeed, there is such a path, namely 7, 3, 4, 5, 14.) We define the successors of state i to be those states reachable by an arrow from state i . For example, the successors of 9 are 13 and 17 in the space pictured above.

We make two assumptions about the state space to simplify the search problem: (1) the number of states reachable from the start state is finite, and (2) there are no cycles, that is, no directed path that leads from a state back to itself. The state space above satisfies both assumptions.

A depth first approach

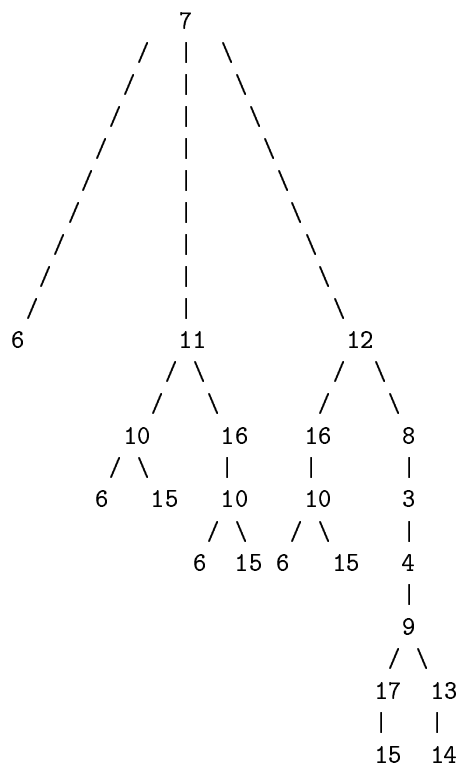
Here is a very simple recursive approach to searching for a path; at each point, there is a current state (reachable from the start state.)

```

If the current state is the goal state, return #t
Otherwise, for each successor of the current state,
  recursively search until one returns #t, at
  which point, return #t.
If none of the successors returns #t, return #f.

```

Before we write the Scheme code for this search, we figure out what the tree of calls of the procedure should look like.



The start state is 7. The first successor of 7 that we try is 6; as this is not the goal state and has no successors to investigate, it returns #f. The next successor of 7 that we try is 11. Recursively, the first successor of 11 that we try is 10, which in turns tries 6 (which has no successors and returns #f) and then 15 (which also has no successors and returns #f.) Thus, 10 returns #f, and the next successor of 11, namely, 16 is tried.

When we finally reach the goal state, 14, the call returns #t to 13, which returns #t to 9, and so on up the tree of calls until the call for 7 returns #t. Note that we never investigate the last successor of 7, because we return #t before that.

The path found by this search is given by the sequence of states between the topmost call and the discovery of the goal state, namely, 7, 12, 8, 3, 4, 9, 13, 14. This is different from (and longer than) the path that we first noted, but it is a correct path from 7 to 14.

Note that this procedure does not keep track of whether it has explored from a given state already. Thus, in the above scenario, we revisit state 10 three times and explore the (unsuccessful) possibilities from there three times. A refinement of the search would keep a list of visited states and avoid exploring again from any previously visited state. This refinement (not pursued here) would make the search into a true depth

first search of the state space graph, which would work for finite graphs with cycles as well as finite acyclic graphs.

Scheme implementation

We choose to implement this in Scheme by means of two mutually recursive procedures, `search` and `search-list`. The procedure `search` takes the current state and the goal state and returns `#t` if and only if there is a path from the current state to the goal state. The procedure `search-list` takes a list of states and the goal state and returns `#t` if and only if there is a path from some state on the list to the goal state.

We assume two auxiliary procedures are available, `successors` and `no-hope?`. The procedure `successors` takes a state and returns the list of successors of the state in the state space. The procedure `no-hope?` takes the current state and the goal state and does some kind of “approximate” check of whether there is no path from the current state to the goal state. This check *must* be correct if it returns `#t`, that is, it must be true that there is no path from the current state to the goal state. However, if the check returns `#f`, there may or may not be a path from the current state to the goal state.

For example, in our parsing problem, because we made the assumption that no nonterminal derives the empty string, if the current state is a string of terminals and nonterminals that is longer than the goal state, there is no way the current state can derive the goal state, and it is safe for `no-hope?` to return `#t`. As another example, suppose the current state begins with a terminal symbol that is different from the initial terminal symbol of the goal state; in this case, too, there is no way that the current state can derive the goal state. The stronger we make `no-hope?`, the less of the state space our procedure will have to search in general.

```
(define search
  (lambda (current goal)
    (cond
      ((equal? current goal) #t)
      ((no-hope? current goal) #f)
      (else (search-list (successors current) goal))))))

(define search-list
  (lambda (ls goal)
    (cond
      ((null? ls) #f)
      ((search (car ls) goal) #t)
      (else (search-list (cdr ls) goal)))))
```

As a simple example in which states are integers, we define `successors` and `no-hope?` as follows.

```
(define successors
  (lambda (state) (list (* 3 state) (+ 6 state))))

(define no-hope?
  (lambda (current goal) (> current goal)))
```

Then if we trace `search` we get the following behavior.

```

> (search 5 57)
CALL search 5 57
  CALL search 15 57
    CALL search 45 57
      CALL search 135 57
        RETN search #f
      CALL search 51 57
        CALL search 153 57
          RETN search #f
        CALL search 57 57
          RETN search #t
        RETN search #t
      RETN search #t
    RETN search #t
  RETN search #t
RETN search #t
#t

```

This indicates that we can get from the start state (5) to the goal state (57) using the operations of multiplying by 3 or adding 6 via the sequence of states 15 (3 times 5), 45 (3 times 15), 51 (45 plus 6), and 57 (51 plus 6). Here is an example of an unsuccessful search.

```

> (search 3 20)
CALL search 3 20
  CALL search 9 20
    CALL search 27 20
      RETN search #f
    CALL search 15 20
      CALL search 45 20
        RETN search #f
      CALL search 21 20
        RETN search #f
      RETN search #f
    RETN search #f
  CALL search 9 20
    CALL search 27 20
      RETN search #f
    CALL search 15 20
      CALL search 45 20
        RETN search #f
      CALL search 21 20
        RETN search #f
      RETN search #f
    RETN search #f
  RETN search #f
RETN search #f
#f

```

Returning the path found

Now we'd like to modify our procedure to return the path it finds when it succeeds. If we look at the tree of calls, if each recursive call returns the path of states from the current state to the goal state when it succeeds,

then all we need to do is add the current state to the beginning of the path. The following modification of the procedures `search` and `search-list` accomplishes this.

```
(define search-start
  (lambda (start goal)
    (search-list (list start) goal)))

(define search
  (lambda (current goal)
    (cond
      ((equal? current goal) '())
      ((no-hope? current goal) #f)
      (else (search-list (successors current) goal)))))

(define search-list
  (lambda (ls goal)
    (if (null? ls)
        #f
        (let ((result (search (car ls) goal)))
          (if (not (equal? result #f))
              (cons (car ls) result)
              (search-list (cdr ls) goal)))))))
```

This is a little different from the procedures we developed in class. The base case (when the current state is equal to the goal) returns the empty list, to avoid two copies of the goal state at the end of the list. And to make sure that the start state is at the beginning of the returned list, we have another top level procedure, `search-start`, which calls `search-list` with a list containing just the start state. Examples of these procedures follow.

```
> (search-start 5 57)
(5 15 45 51 57)
> (search-start 1 33)
(1 3 9 27 33)
> (search-start 4 25)
#f
> (search-start 4 100)
(4 10 16 22 28 34 40 46 52 58 64 70 76 82 88 94 100)
```

Breadth first search

Another method of search involves exploring all the successors of the start state, then all the successors of those states, and so on. This is breadth first search.

In the example, we would first explore the successors of the start state: 3, 6, 11, and 12. As none of these is the goal state, we would next explore their successors: 4, 8, 10, 16. As none of these is the goal state, we would next explore their successors: 3, 5, 9, 10, 15, 16. (If we were keeping track of previously explored states, we would eliminate 3, 10 and 16 as previously explored. This refinement gives breadth first search of a finite graph.) As none of these is the goal state, we would next explore the successors of the remaining states: 13, 14, 17. This includes the goal state, 14, so we return `#t`.

Note that this method finds a shortest path of arrows from the start to the goal state. In this example, the path found is 7, 3, 4, 5, 14. Neither breadth first nor depth first search systematically outperforms the other; both furnish ideas for more sophisticated searching algorithms.