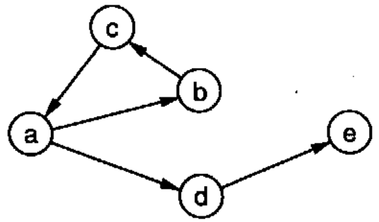

Figure 22.11: A directed graph with a loop.

representing the search of the remaining cities.

If we find a box with a coin in it, we call `cut`, which sets `*paths*` back to the value it had at the time of the `mark`. The effects of the cut are not visible until the next call to `fail`. But when it comes, after the `display`, the next `fail` sends the search all the way up to the topmost `choose`, even if there would otherwise have been live choice-points lower in the search tree. The upshot is, as soon as we find a box with a coin in it, we resume the search at the next city:

```
> (find-boxes)
(LA 1 1)(LA 1 2)C
(NY 1 1)C
(BOS 1 1)(BOS 1 2)(BOS 2 1)(BOS 2 2)C
@
```

In this case, we open seven boxes instead of twelve.

22.6 True Nondeterminism

A deterministic graph-searching program would have to take explicit steps to avoid getting caught in a circular path. Figure 22.11 shows a directed graph containing a loop. A program searching for a path from node `a` to node `e` risks getting caught in the circular path ⟨`a`, `b`, `c`⟩. Unless a deterministic searcher used randomization, breadth-first search, or checked explicitly for circular paths, the search might never terminate. The implementation of `path` shown in Figure 22.12 avoids circular paths by searching breadth-first.

In principle, nondeterminism should save us the trouble of even considering circular paths. The depth-first implementation of *choose* and *fail* given in Section 22.3 is vulnerable to the problem of circular paths, but if we were being picky, we would expect nondeterministic *choose* to be able to select an object


```
(define (path node1 node2)
  (bf-path node2 (list (list node1))))

(define (bf-path dest queue)
  (if (null? queue)
      '@
      (let* ((path (car queue))
             (node (car path)))
        (if (eq? node dest)
            (cdr (reverse path))
            (bf-path dest
                     (append (cdr queue)
                             (map (lambda (n)
                                    (cons n path))
                                  (neighbors node))))))))
```

Figure 22.12: Deterministic search.

```
(define (path node1 node2)
  (cond ((null? (neighbors node1)) (fail))
        ((memq node2 (neighbors node1)) (list node2))
        (else (let ((n (true-choose (neighbors node1))))
                (cons n (path n node2))))))
```

Figure 22.13: Nondeterministic search.

which meets any computable specification, and this case is no exception. Using a correct *choose*, we should be able to write the shorter and clearer version of `path` shown in Figure 22.13.

This section shows how to implement versions *choose* and *fail* which are safe even from circular paths. Figure 22.14 contains a Scheme implementation of true nondeterministic *choose* and *fail*. Programs which use these versions of *choose* and *fail* should find solutions whenever the equivalent nondeterministic algorithms would, subject to hardware limitations.

The implementation of `true-choose` defined in Figure 22.14 works by treating the list of stored paths as a queue. Programs using `true-choose` will search their state-space breadth-first. When the program reaches a choice-point, continuations to follow each choice are appended to the end of the list of stored paths