Crawling and Indexing the World Wide Web

This problem set explores some issues that arise in constructing a “spider” or a “web agent” that crawls over the documents in the World Wide Web. For our purposes, the Web is an extremely large (and growing!) collection of documents. Each document contains some text and also links to other documents, in the form of URLs.

In this problem set, we’ll be working with programs that can start with an initial document and follow the references to other documents to do useful things. For example, we could construct an index of all the words occurring in documents, and make this available to people looking for information on the web (as do many of the search engines on the web, such as www.google.com, www.excite.com, www.yahoo.com, www.lycos.com, or www.altavista.com).

Just in case you aren’t fluent with the details of HTTP, URLs, URIs, HTML, XML, XSL, HTTP-NG, DOM, and the rest of the alphabet soup that makes up the technical details of the Web, here’s a simplified version of what goes on behind the scenes:

1. The Web consists of a very large number of things called documents, identified by names called URLs. For example, the 6.001 home page has the URL http://sicp.ai.mit.edu/. The first portion of a URL (http://) reveals the name of a protocol (in this case hypertext transmission protocol, or HTTP) that can be used to fetch the document, and the rest of the URL contains information needed by the protocol to specify which document is intended.

2. By using the HTTP protocol, a program (most commonly a browser but any program can do this—“web agents” and spiders are examples of such programs that aren’t browsers) can retrieve a document whose URL starts with HTTP:. The document is returned to the program, along with information about how it is encoded, for example, ASCII or Unicode text, HTML, images in GIF or JPG or MPEG or PNG or some other format, an Excel or Lotus spreadsheet, etc.

3. Documents encoded in HTML (HyperText Markup Language) form can contain a mixture of text, images, formatting information, and links to other documents. Thus, when a browser (or other program) gets an HTML document, it can extract the links from it, yielding URLs for other documents in the Web. If these are in HTML format, then they too can be retrieved and will yield yet more links, and so on.
4. A spider is a program that starts with an initial set of URLs, retrieves the corresponding documents, adds the links from these documents to the set of URLs and keeps on going. Every time it retrieves a document, it does some (hopefully useful) work in addition to just finding the embedded links.

5. One particularly interesting kind of spider constructs an index of the documents it has seen. This index is similar to the index at the end of a book: it has certain key words and phrases, and for each entry it lists all of the URLs that contain that word or phrase. There are many kinds of indexes, and the art/science of deciding what words or phrases to index and how to extract them is at the cutting edge of research (it’s part of the discipline called information retrieval). We’ll talk about full text indexing, which means that every word in the document (except, perhaps, the very common words like “and,” “the,” “a,” and “an”) is indexed.

In this project, we’ll be interested in three tasks related to searching the World Wide Web. First, we will develop a way to think about the “web” of links as a directed graph. Second, we will build procedures to help in traversing or searching through graphs such as the Web. Third, we will consider ways to build an index for some set of web pages to support fast retrieval of URLs that contain a given word.

1. Directed Graphs

The essence of the Web, for the purpose of understanding the process of searching, is captured by a formal abstraction called a directed graph. A graph (like the one in Figure 1), consists of nodes and edges\(^1\). In this figure, the nodes are labelled U through Z. Nodes are connected to other nodes via edges. In a directed graph, each edge has a direction so that the existence of an outgoing edge from one node to another node (e.g. node X to node Y) does not imply that there is an edge in the reverse direction (e.g. from node Y to node X). Notice that there can be multiple outgoing edges from a node as well as multiple incoming edges to a node, e.g. there are edges from both Y and Z to W. The set of nodes reachable via an outgoing edge from a given node is referred to as the node’s children. For example, the children of node W are nodes U and X. Lastly, a graph is said to contain a cycle if you start from some node and manage to return to that same node after traversing one or more edges. So for example, the nodes W, X and Y form a cycle, as does the node V by itself.

A second example of a directed graph is shown in Figure 2. This particular directed graph happens to be a tree: each node is pointed to by only one other node and thus there is no sharing of nodes, and there are no cycles (or loops).

In order to traverse a directed graph, let’s assume that we have two selectors for getting information from the graph:

- **(find-node-children graph node)** returns a list of the nodes in graph that can be reached by outbound edges from node. For example, in Figure 2 the children of node B are C, D, E, and H – things that can be reached in one hop by an outgoing edge.

- **(find-node-contents graph node)** returns the contents of the node. For example, when we represent the web as a graph, we will want the node contents to be an alphabetized list of all of the words occurring in the document at node.

---

\(^1\)You’ve already encountered a simpler type of graph called a tree.
Figure 1: An example of a general graph.

Figure 2: An example of a tree, viewed as a directed graph.
1.1 The Web as a Graph

The Web itself can be thought of as a directed graph in which the nodes are HTML documents and the edges are hyperlinks to other HTML documents. For example, in Figure 2 the node labeled B would be a URL, and a directed edge exists between two nodes B and E if the document represented by node B contains a link to the document represented by node E (as it does in this case).

As mentioned earlier, a web spider (or web crawler) is a program that traverses the web. A web spider might support procedures such as:

- \( \text{find-URL-links web URL} \) returns a list of the URLs that are outbound links from \( URL \).
- \( \text{find-URL-text web URL} \) returns an an alphabetized list of all of the words occurring in the document at \( URL \).

In a real web crawler, \( \text{find-URL-links} \) would involve retrieving the document over the network using its URL, parsing the HTML information returned by the web server, and extracting the link information from \(<a \text{ HREF}=\ldots>\), \(<\text{image src}=\ldots>\) and similar tags. Similarly, in a real web crawler, \( \text{find-URL-text web URL} \) would retrieve the document, discard all of the mark-up commands (such as \(<\text{body}>, <\text{html}>, <\text{ul}>, \) etc.), alphabetize (and remove duplicates from) the text, and return the resulting list of words.

For this project our programs will not actually go out and retrieve documents over the web. Instead, we will represent a collection of web documents as a graph as discussed earlier. When you load the code for this project, you will have available a global variable, \( \text{the-web} \), which hold the graph representation for a set of documents for use in this project.

Our implementation of \( \text{find-URL-links} \) and \( \text{find-URL-text} \) will simply use the graph procedures to get web links (children) and web page contents:

\[
\begin{align*}
(\text{define (find-URL-links web url)})
& = (\text{find-node-children web url}) \\
(\text{define (find-URL-text web url)})
& = (\text{find-node-contents web url})
\end{align*}
\]

1.2 Directed Graph Abstraction

We will build a graph abstraction to capture the relationships as shown in Figures 1 and 2, as well as enable us to have some contents at each node. You should study the code in \text{search.scm} provided with the project very closely; parts of it are described in the following discussion.

We will assume that our graph is represented as a collection of graph-entries. Each graph-entry will itself consist of a node (represented as a symbol – the name of the node), a list of children nodes, and some contents stored at the node (which in general can be of any type). The constructors, type predicate, and accessors for the \text{graph-entry} abstraction are shown below:

\[
\begin{align*}
&& \text{Graph Abstraction} \\
&& \text{Graph} & \text{a collection of Graph-Entries}
\end{align*}
\]
Graph-Entry

Node = symbol

Contents = anytype

Graph-Entry

; make-graph-entry: Node,list<Node>,Contents -> Entry
(define (make-graph-entry node children contents)
  (list 'graph-entry node children contents))

(define (graph-entry? entry) ; anytype -> boolean
   (and (pair? entry) (eq? 'graph-entry (car entry))))

; Get the node (the name) from the Graph-Entry
(define (graph-entry->node entry) ; Graph-Entry -> Node
   (if (not (graph-entry? entry))
       (error "object not entry: " entry)
       (first (cdr entry)))))

; Get the children (a list of outgoing node names) from the Graph-Entry
(define (graph-entry->children entry) ; Graph-Entry -> list<Node>
   (if (not (graph-entry? entry))
       (error "object not entry: " entry)
       (second (cdr entry)))))

; Get the contents from the Graph-Entry
(define (graph-entry->contents entry) ; Graph-Entry -> Contents
   (if (not (graph-entry? entry))
       (error "object not entry: " entry)
       (third (cdr entry)))))

Given this representation for a graph-entry, we can build the graph out of these entries as follows:

(define (make-graph entries) ; list<Entry> -> Graph
   (cons 'graph entries))

(define (graph? graph) ; anytype -> boolean
   (and (pair? graph) (eq? 'graph (car graph))))

(define (graph-entries graph) ; Graph -> list<Graph-Entry>
   (if (not (graph? graph))
       (error "object not a graph: " graph)
       (cdr graph)))

(define (graph-root graph) ; Graph -> Node|null
   (let ((entries (graph-entries graph)))
     (if (null? entries)
       #f
       (graph-entry->node (car entries)))))

In the above implementation, we will arbitrarily consider the first graph-entry to hold the “root” for the graph. The procedure graph-root returns the root node.
Given these abstractions, we can construct the graph in Figure 2 (with node a as the root) using:

```
(define test-data
  (make-graph (list
    (make-graph-entry 'a '(b i m) '(some words))
    (make-graph-entry 'b '(c d e h) '(more words))
    (make-graph-entry 'c '() '(at c node some words))
    (make-graph-entry 'd '() '())
    (make-graph-entry 'e '(f g) '(and even more words))
    (make-graph-entry 'f '() '())
    (make-graph-entry 'g '() '())
    (make-graph-entry 'h '() '())
    (make-graph-entry 'i '(j k l) '(more words yet))
    (make-graph-entry 'j '() '())
    (make-graph-entry 'k '() '())
    (make-graph-entry 'l '() '()))))
```

Note that several of the nodes have no children, and that several have no contents.

We would like to have some accessors to get connectivity and contents information out of the graph. We first define a procedure to find a graph-entry in a graph, given the node:

```
; Find the specified node in the graph
(define (find-graph-entry graph node) ; Graph,Node -> Graph-Entry|null
  (define (find entries)
    (cond ((null? entries) '())
          ((eq? (graph-entry->node (car entries)) node)
           (car entries))
          (else (find (cdr entries))))
  (find (graph-entries graph)))
```

We are often more interested in the node children or node contents, rather than the graph-entry. The `find-node-children` and `find-node-contents` accessor procedures can be implemented as follows:

```
; Find the children of the specified node in the graph
(define (find-node-children graph node) ; Graph,Node -> list<Node>|null
  (let ((entry (find-graph-entry graph node)))
    (if (not (null? entry))
      (graph-entry->children entry)
      '())))
```

```
; Find the contents of the specified node in the graph
(define (find-node-contents graph node) ; Graph,Node -> contents|null
  (let ((entry (find-graph-entry graph node)))
    (if (not (null? entry))
      (graph-entry->text entry)
      '())))
```

In our representation above, we use node names (Node = symbol) to reference a graph-entry in a graph; the children of a node are represented as a list of other node names. An alternative to this
approach would be to make the node itself a full abstract data type, so that a node object would have identity, and the children of a node could be, for example, a list of the actual children node objects. The tradeoff would be more work in building the graph (e.g. to link together actual node objects as nodes and edges are added to a graph), but substantial savings when nodes are requested from the graph (i.e. by avoiding a linear search of the graph-entries for the matching node name). With such an alternative abstraction, when requesting a child node one can achieve constant time access (in the size of the graph), as opposed to linear time access as in the current implementation. The alternative implementation would also be a much better model for the world wide web, where constant time access of a named URL link is achieved. This is an incredibly important property of the web that our graph abstraction does not have: the web would surely collapse of its own weight if web page access time grew linearly with the number of pages existing on the web!

2. Searching a Graph

How can we search a graph? The basic idea is that we need to start at some node and traverse the graph in some fashion looking for some goal. The search might succeed (meaning that the goal is found), or it might fail (meaning that goal was not-found). This very basic and abstract search behavior can be captured in the following procedure:

```scheme
;; search: Seeker, Node -> symbol
(define (search next-place start-node)
  (if *debugging*
      (write-line (list 'start start-node)))
  (define (loop node)
    (let ((next-node (next-place node)))
      (cond ((eq? next-node #t) 'FOUND)
            ((eq? next-node #f) 'NOT-FOUND)
            (else
              (if *debugging*
                  (write-line (list 'from node 'to next-node))
              (loop next-node))))))
  (loop start-node))
```

Note the use of the *debugging* flag. If we set this global variable to #t, we will see the order in which the procedure is traversing the graph.

2.1 Seeker Procedures

The real flexibility of the search procedure come from the Seeker procedure that is passed to search. Conceptually, a seeker procedure captures the knowledge about the order in which the nodes in the graph will be examined, as well as how to detect when the goal has been found.

A seeker procedure to be used in a search should have the following type: Seeker = Node -> (Node | boolean), and must return one of three values:

- returns #t if the node is a goal
- returns #f if there are no more nodes to visit
A seeker procedure is difficult enough that we will not directly implement them. Instead, we will write different procedures, called strategies, that create seeker procedures for us.

### 2.2 Search Strategies

There are two common approaches for searching directed graphs, called depth-first search and breadth-first search. In a depth-first search we start at a node, pick one of the outgoing links from it, explore that link (and all of that link’s outgoing links, and so on) before returning to explore the next link out of our original node. For the graph in Figure 2, that would mean we would examine the nodes (if we go left-to-right as well as depth-first) in the order: a, b, c, d, e, f, g, h, i, j, k, l, and finally m (unless we found our goal earlier, of course). The name “depth-first” comes from the fact that we go down the graph (in the above drawing) before we go across.

In a breadth-first search, we visit a node and then all of its “siblings” first, before exploring any “children.” For Figure 2, we’d visit the nodes in the order a, b, i, m, c, d, e, h, j, k, l, f, g.

We can abstract the notions of depth-first, breadth-first, and other kinds of searches using the idea of a search strategy. A search strategy is a procedure that creates a seeker procedure which knows how to walk or traverse a graph in some desired fashion. A seeker governs how the graph is searched, but the strategy configures the seeker in terms of how to detect when the goal node has been found and the order of graph traversal.

A strategy procedure typically takes three arguments:

- a graph
- a goal? procedure, where goal?: Node -> boolean
- a children procedure, where children: (Graph,Node) -> list<Node>

The goal? procedure is a predicate or test to see if a node is the goal node. It takes a node as argument and tests whether the node is the goal of the search. The goal? procedure might examine the contents at the node, or the name of the node, it might just say whether the searcher has run out of time, or it might just never return true (so the searcher will visit the entire graph), or whatever. The children procedure delivers the nodes that one can get to through the edges from a given node.

The hard part, of course, is writing a strategy that makes a seeker. The seeker procedure that it creates should take a node as input and return the next node to be examined. The complexity comes from the fact that the current node may not have any children, so the seeker may need to “remember” that there are still other nodes to be visited (e.g. children or siblings of previously visited nodes). This is a tip-off that there must be state and mutation in the definition of the seeker procedure. In fact, we’ll see that there are two different places where side-effects occur in order to support correct general graph traversal.
2.3 A Depth-First Strategy

Here’s an initial attempt at a depth-first search strategy. It doesn’t quite work on all cases, but it’s a good place to start. The idea is that it maintains (via side-effects to the variable \*to-be-visited*) a list of nodes that need to be visited, in the order in which they should be visited. Whenever a new node is visited, all of its children are added to \*to-be-visited*. When \*to-be-visited* is empty, we’ve visited all the nodes we can get to, so we give up:

```
(define (depth-first-strategy-1 graph goal? children)
  (let ((*to-be-visited* '()))
    (define (where-next? here)
      (set! *to-be-visited* (append (children graph here) *to-be-visited*))
      (cond ((goal? here) #T)
            ((null? *to-be-visited*) #F)
            (else
             (let ((next (car *to-be-visited*)))
               (set! *to-be-visited* (cdr *to-be-visited*))
               next))))
    where-next?))
```

Notice that `depth-first-strategy-1` returns `where-next?` as a seeker procedure; that seeker procedure continues to have access to the \*to-be-visited* variable (as well as the `graph` information) whenever the seeker procedure is used during an actual search.

This simple algorithm does not work in general (see Warmup Exercise 3), but it does work for the graph in Figure 2.

3. An Index Abstraction

We will also be interested in constructing an index of web pages. To do this, we will first construct a general purpose index abstraction, and then use it for our purpose of web indexing.

An Index enables us to associate values with keys, and to retrieve those values later on given the key. Here we will assume that a key is a Scheme symbol (i.e. `Key = symbol`), and that a value is also a symbol (i.e. `Val = symbol`). Our index will be a mutable data structure.

A concrete implementation for an index is as follows. An Index will be a tagged data object that holds a list of Index-Entries. Each Index-Entry associates a Key with a list of values for that Key, i.e.

```
; Index = Pair<'index, list<Index-Entry>>
; Index-Entry = Pair<Key, Pair<list<Val>, null>>
```

Thus our index implementation is shown (partially) below. You will be asked in the exercises to complete the implementation. The index implementation makes use of the Scheme procedure `assv`; you will find it helpful to consult the Scheme manual as to what these procedures do.
(define (make-index) ; void -> Index
(list 'index))

(define (index? index) ; anytype -> boolean
(and (pair? index) (eq? 'index (car index))))

; This is an internal helper procedure not to be used externally.
(define (find-entry-in-index index key)
(if (not (index? index))
(error "object not an index: " index)
(assq key (cdr index))))

; returns a list of values associated with key
(define (find-in-index index key) ; Index,Key -> list<Val>
(let ((index-entry (find-entry-in-index index key)))
(if (not (null? index-entry))
(cadr index-entry)
'())))

(define (add-to-index! index key value) ; Index,Key,Val -> Index
(let ((index-entry (find-entry-in-index index key)))
(if (null? index-entry)
;; no entry -- create and insert a new one...
... TO BE IMPLEMENTED

;; entry exists -- insert value if not already there...
... TO BE IMPLEMENTED
)
index))

An example use of the index is shown below

(define test-index (make-index))
(add-to-index! test-index 'key1 'value1)
(add-to-index! test-index 'key2 'value2)
(add-to-index! test-index 'key1 'another-value1)

(find-in-index test-index 'key1)
;Value: (another-value1 value1)

(find-in-index test-index 'key2)
;Value: (value2)

4. Warmup Exercises

These exercises will get you thinking about the project. We suggest that you start on them early. You should turn them in during tutorial the week of March 17th.

Warmup Exercise 1: Index Implementation.
In order to simply use the index abstraction, one should not need to understand the underlying implementation (both the structure of the data representation and the implementation of the abstraction procedures). In one of the programming exercises, however, you will be asked to complete the implementation of \texttt{add-to-index!} partially shown above. In order to do this, you will need to understand the index implementation.

Draw a box and pointer diagram, and show the corresponding printed representation, to illustrate the implementation of an \texttt{Index} as defined in Section 3. Describe how you want the following expressions to create and then mutate your data structure:

(define test-index (make-index))
(add-to-index! test-index 'key1 'value1)
(add-to-index! test-index 'key2 'value2)
(add-to-index! test-index 'key1 'another-value1)

**Warmup Exercise 2:** State and search strategies.

The following variation of \texttt{depth-first-strategy-1} has a bug that causes it to not work, even on \texttt{test-data}.

(a) Explain what is wrong. In particular, why does it matter where \texttt{*to-be-visited*} is declared?

(b) Use environment diagrams to visually illustrate where the problem is\textsuperscript{2}. Assume that you have two procedures \texttt{goal?} and \texttt{find-node-children} already defined in the global environment. Show how the environment diagram changes when you evaluate the definitions of \texttt{depth-first-strategy-1} and \texttt{depth-first-strategy-2}. Show how each of the following expressions differ in how they change the environment diagram, and identify where the source of the problem is:

(define seeker-1
  (depth-first-strategy-1 test-data goal? find-node-children))

(seeker-1 'a)

(define seeker-2
  (depth-first-strategy-2 test-data goal? find-node-children))

(seeker-2 'a)

\textsuperscript{2}When working with environment diagrams, you will find it very helpful to use different colored pens
Warmup Exercise 3: The Web as a General Graph

Although we’ve been presenting the concepts and ideas in this problem set in the context of the Web, for the project you will be using data structures that have been pre-built. Thus, you will not be interfacing or touching the Web directly; instead, you will be dealing with a graph data structure we’ve created for you called the-web.

The following partial definition of the-web mimics a subset of the graph of web pages at the 6.001 web site. Each node here is the URL of a web page and the children nodes are the URLs referenced in the links on the page.

```
(define the-web
  (make-graph (list
    (make-graph-entry
      'http://sicp.ai.mit.edu/
      'http://sicp.ai.mit.edu/SchemeImplementations
      http://sicp.ai.mit.edu/psets)
    '(... words extracted from http://sicp.ai.mit.edu/ ...))
  (make-graph-entry
    'http://sicp.ai.mit.edu/SchemeImplementations
    (http://sicp.ai.mit.edu/getting-help.html
    http://sicp.ai.mit.edu/lab-use.html
    *the-goal*)
    '(... words extracted from http://sicp.ai.mit.edu/SchemeImplementations ...)))
  (make-graph-entry
    'sicp.ai.mit.edu/getting-help.html
    'sicp.ai.mit.edu/
    sicp.ai.mit.edu/SchemeImplementations
    '(... words extracted from http://sicp.ai.mit.edu/getting-help.html))
  ...)))
```

Explain why depth-first-strategy-1 will create a seeker that fails on this graph. (If it helps, note that this graph has the same kind of form as Figure 1.) What is the essential difference between the test-data and the-web examples that causes depth-first-strategy-1 to fail here?

5. Programming assignment: A web spider

Begin by loading the code for project 2. This will define the search and data structure procedures listed above. Just to make sure everything is working, evaluate

```
(search (depth-first-strategy-1
  test-data ; graph
  (lambda (node) (eq? node 'l)) ; goal?
  find-node-children ; children
  'a ; start-node)
```

This should traverse the test-data graph until the search finds node l (lowercase L), and you should see the nodes being visited in depth-first order.
Computer Exercise 1: A breadth-first search.

*depth-first-strategy-1* creates a depth-first seeker procedure. A breadth-first search strategy can be obtained by modifying *only one line* of the procedure *depth-first-strategy-1*, leaving the total number of characters in the procedure unchanged! Do this (name your procedure *breadth-first-strategy-1*), demonstrate that it works on test-data, and write a short (but clear) explanation of why it works.

Marking nodes

In Warmup Exercise 3, you discussed a problem with *depth-first-strategy-1*. One way to fix this problem is to use the following pair of procedures to “mark” or remember visited nodes.

- *(node-visited! node)*: remembers that we’ve visited *node*
- *(deja-vu? node)*: test whether or not we’ve visited *node*

The implementations of the search and seeker procedures are careful in that they use an internal variable (e.g. *to-be-visited*) to store their state. Thus we could run several searches concurrently (for example searching for different goals or using different strategies to see which finishes first) if we create a new seeker procedure for each search.

This means that it is very important that *node-visited!* and *deja-vu?* also be capable of having their own private state that will not get mixed up between different searches. Our approach here will be to produce a new pair of procedures for use in each seeker procedure; that pair of procedures will have access to a shared internal state variable. In this case, the “data” about what nodes have already been visited is accessible in a subtle fashion — the *node-visited!* and *deja-vu?* procedures share access to the environment in which they were created and where this variable lives, so no previously-visited-nodes data structure needs to be explicitly passed as arguments to these procedures.

The following version of a depth-first strategy calls *make-mark-procedures* to get new marking procedures, and then uses them as follows:

```
(define (depth-first-strategy graph goal? children)
  (let ((mark-procedures (make-mark-procedures)))
    (let ((deja-vu? (car mark-procedures))
          (node-visited! (cadr mark-procedures))
          (*to-be-visited* '()))
      (define (try-node candidates)
        (cond ((null? candidates) #F)
              ((deja-vu? (car candidates))
               (try-node (cdr candidates)))
              (else
               (set! *to-be-visited* (cdr candidates))
               (car candidates))))
      (define (where-next? here)
        (node-visited! here)
        (set! *to-be-visited* (append (children graph here) *to-be-visited*)))
      (try-node (children graph graph))))
```
(if (goal? here)
  #T
  (try-node *to-be-visited*)
  where-next?)))

Note that the procedure used here, (make-mark-procedures), returns a list whose first element is the deja-vu? procedure and whose second element is the node-visited! procedure.

You should also look carefully at the try-node procedure. Its job is to look through the *to-be-visited* list until the first node in the list is found that has not been previously visited. At that point, it first mutations the *to-be-visited* variable to point just to remaining nodes in that list (appearing after the first non-visited node), and then returns that first non-visited node.

Computer Exercise 2: Marking visited nodes.

Implement make-mark-procedures. The procedure should be very short (less than a dozen lines of code), and you can make use of the built-in Scheme procedure memv. (See SICP or the Scheme manual if you are unsure how memv works.) Do not worry about efficiency. We're concerned here only with the correctness of the program and with making sure that you understand how to introduce the local state necessary to make the procedures work correctly.

To show that your implementation of make-mark-procedures works, use search and a depth first strategy (i.e. depth-first-strategy) to walk the sample graph test-cycle which is defined for you in search.scm. Then you should modify your simple breadth first search strategy (breadth-first-strategy-1) from Computer Exercise 1 to create a better procedure (call it breadth-first-strategy) that correctly makes sure that the search only visits nodes once. Demonstrate that this procedure works by showing that it can also traverse test-cycle.

Once you are sure these procedure are working, give the order in which the nodes are visited for depth-first search and for breadth-first search of the-web. You should provide a goal? procedure that always returns false so that the entire web is traversed, and start the walk at the node labeled http://sicp.ai.mit.edu/.

6. Indexing the web

Now let’s turn to the problem of creating a full-text index of documents on the Web, like the one created by Google or other search engines. We’ll assume that we have a graph that represents the World Wide Web, and this graph uses node names that correspond to URLs (as in the-web sample given earlier). Remember, we’re assuming that we have a procedure find-URL-text which, for this web representation, gets us the alphabetized text at the node. For example, (find-URL-text the-web 'http://sicp.ai.mit.edu/) yields the list:

(18:30:02 1999 6.001 6001-WEBMASTER@AI.MIT.EDU 8
ABOUT ALL AM AND ANNOUNCEMENTS ANSWERS ARE ASSIGNMENT
ASSIGNMENTS BY CALENDAR CAN CHANGE COLLABORATIVE
COMMENTS COMPUTER COPYRIGHT CURRENT DO DOCUMENTATION
EDT FALL FIND FOR GENERAL GET GETTING GUIDELINES HELP
HOW I IN INDIVIDUAL INFORMATION INSTITUTE INTERPRETATION
IS LAST LECTURE MASSACHUSETTS ME MICROQUIZZES MODIFIED

Computer Exercise 3: The Index Abstraction.

Your first task is to complete the implementation of the index abstraction. Complete the definition of add-to-index! so that we have available the following procedures:

- (make-index): Creates a new index.
- (add-to-index! index key value): Add the value under the given key in the index.
- (find-in-index index key): Returns a list of all the values that have been entered into the index under the specified key.

Verify that your add-to-index! works with the other index procedures by showing the result of evaluating the insertions and finds presented in Warmup Exercise 1.

Computer Exercise 4: A Web Index

Given this index data abstraction, we want to write a procedure to use the index to keep track of all the documents (URLs) that contain a particular word:

- (add-document-to-index! index web url): Add an entry in the index for each word in the contents of the URL (so that the key for that entry is a word, and the data in the entry is the url).

This will enable you later on to get back the list of all documents (URLs) that appear under that key (and thus contain that word). Here’s an example of how it should work:

(define the-web-index (make-index))

(add-document-to-index! the-web-index
  the-web
  'http://sicp.ai.mit.edu/)

(find-in-index the-web-index 'help)
;Value: (http://sicp.ai.mit.edu/)

(find-in-index the-web-index '*magic*)
;Value: #f

Note: A professionally written version of these procedures would pay careful attention to the efficiency of the algorithm used, and would probably involve alphabetical order and complicated data structures. You should not worry about this unless you find yourself with spare time on your hands — and if you do have that time, you might want to look at the Scheme reference manual definition of sort, string<=?, and symbol->string. Your tutor will thank you for making your solution easy to understand, even at the expense of performance (and, in later life, many of your colleagues will feel the same way!).
**Computer Exercise 5:** Crawling the Web to Build an Index.

Now let’s simulate what a typical search engine’s spider does: Crawl the entire web (recall: use a goal procedure that always returns false) and produce a full-text index of everything you find.

Write a procedure, `make-web-index`, similar to the procedure `search`, that creates a new index, finds all the URLs that can be reached from a given web and initial URL, indexes them, and returns a procedure that can be used to look up all the URLs of documents containing a given word. Your procedure should use `breadth-first-strategy` from Computer Exercise 2 to actually traverse the-web. You will also want to make use of the index manipulating procedures from above.

You can test your program by trying the following example. Which document(s) do you find?

```scheme
(define find-documents (make-web-index the-web 'http://sicp.ai.mit.edu/))
(find-documents 'collaborative)
```

**Computer Exercise 6:** A dynamic web search.

Let’s put everything together by comparing the performance of crawling the web to find a desired document, versus using a full-text index of the web. Note the difference: a dynamic search would traverse URLs in real-time when a user initiates a search, while the full-text index is the result of some precomputation stage that has occurred prior to any user initiated search. This won’t be a full or fair comparison, but it should give you some ideas about tradeoffs in designing real systems that analyze the contents of the actual Web.

To investigate crawling, write two procedures:

1. `(search-any web start-node word)` searches or traverses the indicated web (using `breadth-first-strategy`) and returns the first document that it finds that contains the given word. It should stop searching as soon as it finds such a document.

2. `(search-all web start-node word)` searches the entire web (using `breadth-first-strategy`) and returns all documents that contain the given word.

Show that your procedures work by using `search-any` and `search-all` to look for the word 'collaborative' in the-web structure. Make sure this is consistent with what you found in Computer Exercise 5.

**Computer Exercise 7:** Comparison – Web Index vs. Dynamic Search.

Let’s compare the technique of dynamic web searching with web indexing using `make-web-index` and `find-documents`. We’ve provided a program, `(generate-random-web size)`, that you can use to create test webs of different sizes (total number of nodes) with some randomly generated text. Use this to build several test webs. You don’t have to make them too big; the procedure will in fact not build anything larger than size 200. For each web, measure the amount of time it takes, starting from the node named \*start*:

- to use `search-any` to find a document containing the word “help”;
• to use search-any to find a document containing a word that is not in the test web: “Chuckvest”;

• to use search-all to find all documents containing the word ”help”;

• to run make-web-index to create an index for the test web;

• to use find-documents to find all documents containing the word “help”, not including the time needed to create the index.

• to use find-documents to find all documents containing the word “Chuckvest” (there won’t be any), not including the time needed to create the index.

Note that although find-documents was originally created in Exercise 5 to deal with the-web, you can use the same method (utilizing make-web-index) to apply this idea to any random test web.

To enable you to do timing, we’ve included a special function called timed. To use it, place the symbol timed at the beginning of a combination that you want to evaluate, e.g.

(timed factorial 25)

will apply the procedure factorial to the argument 25, print out the time (in seconds) it takes to compute this, and return the result.

Write a few short paragraphs explaining the measurements you made and what conclusions you might want to draw about searching and crawling the real Web. If you were building a service to help people find information on the Web, what kinds of factors would you consider in deciding which method to use?

Extra Credit Exercise 1: Using a stack or a queue.

In lecture, we discussed abstract data types for stacks and queues. If one thinks carefully about our depth first and breadth first searches, one realizes that we are essentially reimplementing these concepts inside our strategy and seeker procedures.

As extra credit, implement the stack and queue abstractions as discussed in lecture, and then modify your various strategy procedures to use these abstractions instead of directly manipulating a *to-be-visited* variable. You might also find it useful to add a pop-stack! and pop-queue! procedure to the stack and queue abstractions, which each remove the top (or front) of the queue and then return that top (or front) item.

Extra Credit Exercise 2: An alternative graph abstraction.

As discussed in Section 1.2, our representation of a graph as a list of graph-entries, with symbolic names for nodes is inefficient for access of children nodes. If you are up to the challenge, you might try implementing a different graph abstraction using an explicit node abstract data type that gets around this problem. Note: this would require substantial programming effort; since the graph abstraction itself changes, these changes may impact many other aspects of the system!