MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.891 Spring 2006
Problem Set 5

Issued: Wed. 8 Mar 2006 Due: Wed. 22 March 2006

Reading: Lots of code!

Code: search.scm, maze.scm, patdb.scm, attached.

Organizing Search Strategies

Most complex problems have a component of search, because the problem solver must make choices among alternatives that are not totally determined until the problem under consideration has had more details elaborated. For example, in solving a numerical or an algebraic problem one may have to choose a sign for a square root before enough is known to rule out one of the choices. Or in the analysis of an electrical circuit one may have to assume states (on or off) for diodes in the circuit in order to complete an analysis that may ultimately prove (by finding a contradiction) that the states assumed cannot be the correct ones in the situation specified. Unfortunately, if a search strategy is not incorporated into a program neatly it can become "the tail that wags the dog," distorting the rest of the program so that one cannot easily modify the search strategy without rewriting the whole program.

In these exercises we investigate the extent to which a search strategy can be separated from the other parts of a program, so that one can interchange search strategies without greatly modifying the program. We will also see how a program can be enhanced by clever integration of a search strategy.

Example: Running a maze

Imagine the problem faced by a rat who runs a maze looking for rewards that are stashed at particular places in the maze. We can abstract this problem as a graph with nodes representing the places in the maze and edges connecting the places. We can arrange, with trapdoors, that some paths can be traversed in only one direction. Thus we build a directed graph. So if a path can be run in both directions we must put in two edges to describe that possibility.

Now the rat has to think about the order in which it will explore the maze. Once it finds the goodie at one node, it needs to be able to continue searching, because there may be goodies in other places.
It also must have a way of avoiding running in circles, never finding anything, and it must have a way to know when it has exhausted the maze and give up. We will have to think about all of these things, and we will discover that we need to make a quantum rat, not a classical one.

Consider the following maze:

```
             START
             ^  |
             |  -----
         -----      |
         |          V
 --------A<---------B<------>D*
 |       ^          |        ^
 |       |          |        |
 |       |    -------        |
 |       ---  |              |
 V          \ V              |
*E----------->C---------------
 ^            |
 |            |
 |            V
 -------------F
```

This maze has seven nodes and twelve edges. (The path between nodes B and D is bidirectional, so we need two edges to represent it.) The rat is dropped into the maze at START and the reward is present at nodes D and E. (This is indicated by an asterisk.)

Representing the maze

We represent a maze by its nodes and links. However, to prepare for a more general setting we make the links COMPUTE the target node when needed, rather than just specify it statically. This gives us the flexibility to make a maze incrementally, and thus to explore arbitrarily large mazes, such as the infinite ones that occur in mathematical situations. We use two procedures here:

```
(make-node <name> <value> <link1> ... <linkn>)
```

Makes a node with the given name and value. It is connected to other nodes by the given links. The name given to a node here is arbitrary and is used only to identify the node for debugging printout. The value given to the node is the reward that the rat is looking for. False (#f) signifies that there is no reward at this node.

```
(make-link <thunk> [priority])
```

Makes a link that finds the target node by executing the given thunk. (In Scheme jargon, a thunk is a procedure of no arguments.) An optional priority is provided to allow a search algorithm to put an ordering on the exploration of links that have been seen.

The particular maze shown above is specified in the file maze.scm.

Searching the maze

The general method of searching a maze from a particular node is summarized as follows:

> If the current node is one that we want, return it, with a way of proceeding to find more, otherwise proceed: Choose one of the links that we can proceed from and use it to get its target node. Make that the current node, if possible, and iterate. If we run out of such links, fail.

This description does not specify a particular strategy. A depth-first strategy specifies that the first links we should try are descendents of the current node (the ones emanating from the current node), and if we run out of those we should go back to the node we came from and try the unexplored links emanating from that parent node. We backtrack to parents until we run out of unexplored links from the initial node. The fact that we can backtrack to a parent node (apparently traversing an edge in the wrong direction) is the reason why our rat is a quantum rat rather than a classical rat!

A breadth-first strategy, by contrast, specifies that we must explore sibling links of the current node before descendent links of the current node. (A sibling link is a link emanating from the parent of the current node.).

The file search.scm contains three search strategies, and common support for them that allows us to interchange them as needed. The simplest is depth-first search, and here is the code:

```
(define (depth-first-search node)
  (depth-first-walk node (loop-avoidance-node-filter)))
```

The walker takes as its second argument a link filter that can be used to rule out traversing that link. The default link filter implements a means of preventing visiting the same node twice.

```
(define (depth-first-walk node link-filter)
  (let per-node ((node node) (k (lambda () #f)))
    (maybe-accumulate
     node
     (lambda ()
       (let per-link ((links (node-links node)))
         (if (pair? links)
             (let ((node
                    (dereference-link (car links)
                                      link-filter)))
               (if node
                   (per-node node
                             (lambda ()
                               (per-link (cdr links))))
                   (per-link (cdr links))))
             (k)))))))
```

Each node visited by the walker is checked for acceptance by maybe-accumulate: if the node has a value, the node is returned, consed with a procedure that can be called to proceed with the search. That procedure is described by the lambda expression passed to maybe-accumulate. It gets the links of our node and dereferences them as needed (checking with the link filter first). Each time a link is dereferenced it delivers a descendent node, if possible. This may cause the node to be constructed dynamically. If there is a descendent node, it becomes the current node, with a continuation function, k, that will examine the next link.

```
(define (maybe-accumulate node get-next)
  (if *debug-trace?*
      (begin
        (fresh-line)
        (write-string ";trace: ")
        (write (node-name node))
        (newline)))
  (if (node-value node)
      (cons node get-next)
      (get-next)))

(define *debug-trace?* #f)
```

If we set *debug-trace* to #t we get to see the action of the walker.

The loop-avoidance filter marks each node that is the result of following a link. It is false if the node has already been marked. This prevents it from being the result of following a link twice. In this way the walker implements "Theseus and Ariadne's thread."

```
(define (loop-avoidance-node-filter)
  (let ((table (make-eq-hash-table)))
    (lambda (node)
      (if (hash-table/get table node #f)
          #f
          (begin
            (hash-table/put! table node #t)
            node)))))
```

We can try this out on our maze graph:

```
(define foo (depth-first-search (maze-graph)))

(pp foo)
((node e winner! #[compound-procedure 35]) . #[compound-procedure 34])

(define bar ((cdr foo)))

(pp bar)
((node d winner! #[compound-procedure 37]) . #[compound-procedure 36])

(define mum ((cdr bar)))
;Value: mum

(pp mum)
#f
```

The procedure search-values->list gets all of the nodes with rewards.

Problem 5.1:

Pick a classic puzzle, such as the "Missionaries and Cannibals Puzzle" (see http://www.jimloy.com/puzz/cannibal.htm), or any other one that you find fun. Represent it as a (perhaps implicit) graph to be searched, and express it in our description system. How many steps does it take to solve your problem with each kind of search? Can you use the priorities in the best-first-search to help get better performance on your problem?

Problem 5.2:

Reconstruct the rule interpreter of Problem Set 4 to use Chris's matcher/discrimination-net database. You will discard the matcher, rewrite rule-simplifier.scm, and make some changes to rule-compiler.scm.

Problem 5.3:

Often in a game like chess we want a "plausible-move generator" that sorts the legal moves so as to arrange that locally better moves are considered first. We also want a "static evaluator" that evaluates positions for their value. Can the searching paradigm provided here be used or extended to incorporate such ideas? Explain your opinions on this matter in a coherent paragraph or two. You may illustrate your explanation with code, if that is useful.

Chris Hanson wrote an elegant merger of the pattern matcher with search engine mechanism to make an active discrimination network database. (See patdb.scm) With his system you can store patterns in a discrimination network (made by patdb:make). You can then use the appropriate walker to match the patterns against your data expressions, and only the successful matches are returned (by calling a procedure with the successful terminal node). The node-value of that node is the dictionary.

This system optimizes the database lookup as follows: when several patterns share leading substructure, the matching for that substructure is performed once for all of the patterns rather than once for each.

Problem 5.4: (optional!)

If you are feeling in a "matcho" programming mood, pick a complex game, where you cannot enumerate the states explicitly. (Examples of such games are Chess, Go, Backgammon, various card games, etc.) Express its legal moves as a node and link generator in our description system. (You probably want to make a very simplified version of one of these games... Chess is too big a project for a weekend hackathon, unless your name is Richard Greenblatt!). Demonstrate your code.

WARNING: This is not a small job: don't do it if it prevents you from studying for some other class! I have known people to spend months on problems like this.
