CS 135 Fall 2008
Byron Weber Becker, Ian Goldberg, Brad Lushman, Daniel Roche, Troy Vasiga

Assignment 10

Due Monday, December 1, 10:30am
Files to submit: `book.ss, edge.ss, puzzle.ss` and (for the bonus) `decline.ss, 2dpuzzle.ss`.

Language level: Intermediate Student with Lambda
Warmup Exercises: (Not to be submitted) 28.1.4, 30.1.1, 31.3.2
Extra practice exercises: (Not to be submitted) 28.2.1-4, 31.3.8, 32.2.1-8

1. Do HtDP Exercises 31.3.6 and 31.3.7, using accumulators, and place your solution in *book.ss*.

2. We have seen the adjacency list representation of graphs. This question deals with another representation, called the *edge list representation* of graphs. An edge list is a pair of lists: the first list is a list of all vertices, and the second list is a list of pairs, where each pair is two symbols, representing the start (head) and end (tail) of the edge (respectively). For example, if graph *G* has an adjacency list representation of '((A (B C D)) (B (C D)) (C ()) (D ())), then *G* would have an edge list representation of '((A B C D) ((A B) (A C) (A D) (B C) (B D))).

   In the file *edge.ss*, place your solutions to the following problems:

   (a) Write the function *adj-to-edge* which consumes a graph in adjacency list representation and produces an edge list representation for the graph.

   (b) Write the function *edge-to-adj* which consumes a graph in edge list representation and produces the corresponding adjacency list representation. Note that edges may not be in sorted order: that is, '((A B C D) ((B D) (A B) (B C) (A D) (A C))) is a valid edge list representation of the graph *G* described above.

   (c) Write the function *reverse-graph* which consumes a graph in adjacency list representation and produces a graph in adjacency list representation with all the edges reversed: that is, the direction of each edge has been reversed. For example, (*reverse-graph G*) $\Rightarrow$ '((A ()) (D (A B)) (C (A B)) (B (A))).

3. The *find-route* function we studied in class finds a route or path from a starting node to an ending node in an explicit graph. Games often use a similar function but with an implicit graph – a graph in which the neighbours of a node are calculated as needed rather than looked up in an explicit data structure.

   One such game is peg solitaire. For an example of 2-D peg solitaire, see `http://www.mazeworks.com/peggy/index.htm`. We will work with a 1-D version that is similar (but less interesting). The rule for moving from one configuration of the game to the next is that one peg jumps over another peg (either to the right or the left), landing in an open spot on the 1-D board. The jumped peg is removed.

   For example, a starting board configuration could be represented by '(O O _ O) where 'O represents a peg and '_ represents an open space. With this configuration, only one move is

possible: the first peg jumps to the only open space, resulting in '(_ _ O O). Again, there is only one possible move, resulting in the ending configuration '(_ O _ _).

The function *solve-puzzle* will solve a one-dimensional version of the peg solitaire puzzle. "Solve" means that it will consume a starting configuration and an ending configuration for a game. It will produce one of:

- *false*, if it is impossible to reach the ending configuration according to the game's rules,
- a list of intermediate game configurations showing how to get from the starting configuration to the ending configuration.

(*solve-puzzle* '(O O _ O) '(_ O _ _)) will produce '((O O _ O) (_ _ O O) (_ O _ _)) while (*solve-puzzle* '(O O _ O) '(O _ _ _)) will produce *false*.

A data definition will be useful:

- A puzzle configuration (*pzl-config*) is (*listof* (*union* '_ 'O)).

You have two options in solving this question.

- **Option 1:** In the file `puzzle.ss` write the following function:

  ;; solve-puzzle: pzl-config pzl-config → (union false (listof pzl-config))
  ;; Solve a 1D peg puzzle where start is the starting configuration, end is the desired
  ;; ending configuration.
  (**define** (*solve-puzzle start end*) ...

  You may write whatever helper functions you wish, but your entire automarking grade will be based on *solve-puzzle*. In order to indicate your selection for this option, define a constant *grade-helper-functions* to *false* in *puzzle.ss*. Of course, your solution will also be marked for style, understandibility, etc..

- **Option 2:** If Option 1 seems daunting, you may wish to choose this option: this option gives you a more structured approach to solve this problem. Define the constant *grade-helper-functions* to *true* if you choose this option. Half of your autograding marks will come from correctly implementing all the helper functions described below, and the remaining half will come from correctly implementing *solve-puzzle*. Of course, your work will also be marked for style, understandability, etc. If you find this option too constraining, pick Option 1. There is no Option 3.

  We begin breaking down the problem of *solve-puzzle* by noting *solve-puzzle* is almost identical to *find-route* except:

  – Instead of representing nodes in the graph with symbols, as in class, nodes are *pzl-config*s (see the data definition above). Therefore a path from the "origination node" to the "destination node" is a list of *pzl-config*s instead of a list of symbols. Some parts of *find-route* will need minor changes to account for this.

  – The big change is in *neighbours*. Rather than looking up the out-neighbours in an explicit graph, we'll need to generate them from a given *pzl-config*. The hard part of writing *solve-puzzle* is writing *gen-neighbours*. (Note the new name, to recognize the fact that we're generating the neighbours.)

There are many ways to write *gen-neighbours*. One strategy uses the following steps:

- We need to check for a move at each possible place in a given puzzle configuration. So we'll begin by making a list of all the possible ways to split a configuration.
- Now check the second part of each split to see if it begins with either '(O O _) or '(_ O O). If it does, replace those three symbols with '(_ _ O) and '(O _ _), respectively. If it does not, replace the entire split with a failure value such as *false*.
- Remove all of the failure values from the list of splits.
- Put the two parts of each split back together again, resulting in the list of neighbours.

These four steps for the configuration '(O O _ O O) result in:

'((() (O O _ O O)) ((O) (O _ O O)) ((O O) (_ O O)) ((O O _) (O O)) ((O O _ O) (O)) ((O O _ O O) ()))	'((() (_ _ O O O)) false ((O O) (O _ _)) false false false)	'((() (_ _ O O O)) ((O O) (O _ _)))	'((_ _ O O O) (O O O _ _))

Write the following helper functions in the file `puzzle.ss`:

(a) (*split cfig*) consumes a *pzl-config* and produces a list of pairs. Each pair represents one possible way to split the configuration into two pieces. This works out nicest if the first member of the pair is the reverse of the configuration up to the split point and the second member of the pair is what comes after the split point (slightly different from what is shown above). The following example uses numbers for clarity. For example, (*split* '(1 2 3)) produces

```
'((() (1 2 3))
  ((1) (2 3))
  ((2 1) (3))
  ((3 2 1) ()))
```

An accumulator is useful for this function; abstract list functions are not.

(b) (*make-move split*) consumes a "split", defined as (*list pzl-config pzl-config*). If the second configuration in the split begins with either '(O O _) or '(_ O O), a new *pzl-config* is produced with those three items replaced with either '(_ _ O) or '(O _ _), as appropriate. Otherwise, the function produces *false*.

(c) (*prepend rev end*) consumes two *pzl-config*s and produces a *pzl-config* that is *end* appended to the end of *rev* reversed. You should *not* use *append* or *reverse*.
For example, (*prepend* '(O O _) '(_ _ _)) ⇒ '(_ O O _ _ _).

(d) Here's the big one: (*gen-nbrs cfig*) consumes a *pzl-config* and produces a (*listof pzl-config*), the list of neighbours of *cfig*. For example, (*gen-nbrs* '(_ O O _)) produces '((O _ _ _) (_ _ _ O)).
Use the functions defined previously, plus abstract list functions and appropriate anonymous functions to implement the strategy outlined above.

(e) Finally, rework *find-route* into *solve-puzzle* using your new *gen-nbrs* function.

4. **2% Bonus**: In the file *decline.ss*, write the function *decline* which should find the longest declining sequence in a given list of integers. That is, efficiently find the longest continuous subsequence in which each element but the last is one larger than the element immediately following it.

   For example: (*decline* '(3 2 1)) ⇒ '(3 2 1), (*decline* '(1 3 2 1 3 4 5 4 3 2)) ⇒ '(5 4 3 2) and (*decline* '(1 2 5)) ⇒ '(1)

5. **5% Bonus**: One-dimensional peg solitaire is okay, but the puzzle is much more interesting in two dimensions. In a separate file called `2dpuzzle.ss`, modify your code from question 3 to handle two-dimensional peg solitaire.

   We will represent a two-dimensional puzzle configurations by a list of lists of symbols, where each list of symbols has the same length, similar to matrices from Assignment 9. Besides the symbols 'O and '_ as before, we also allow the symbol 'X to indicate an unused portion of the board (that is, positions which contain neither a peg nor a space.

   For example, the following represents the "plus" configuration on the popular cross-shaped solitaire board:

```
(define plus '((X X _ _ _ X X)
               (X X _ O _ X X)
               (_ _ _ O _ _ _)
               (_ O O O O O _)
               (_ _ _ O _ _ _)
               (X X _ O _ X X)
               (X X _ _ _ X X)))
```

   The rules for moves are the same as in one-dimensional solitaire, except that they may be applied to any row or column (note that diagonal jumps are not allowed).

   To change your *solve-puzzle* function to work with two-dimensional peg solitaire, the only change really required is in the *gen-neighbours* function. The solution will be especially simple if you make use of the *gen-neighbours* you already wrote for question 2, and the *transpose* function from Assignment 9.

The remaining material in this document deals with enhancements that do not need to be submitted.

Many "natural" problems can be solved by finding a path through some sort of implicit graph, and for this reason graph search techniques form the basis for many artificial intelligence (or "AI") algorithms.

In this assignment, we have examined one such problem, that of a simple peg solitaire game. Other one-player puzzles such as sudoku can be solved in a similar manner. Graph search can even be used to make intelligent decisions in two-player games such as tic-tac-toe, chess, or checkers, by examining every possible sequence of moves.

But simple games are not the only problems which can be solved by performing a graph search. Consider a graph where "nodes" are Scheme programs, and edges between node represent small transformations from one program to another. We might use a graph search technique in such an implicit

graph to identify cheaters on CS 135 assignments, by finding two different assignment submissions which a short "path" connecting them in the graph.

Yet another example is automated theorem proving. Each node in the graph represents a set of "facts" or equations known to hold, and the out-neighbours of a node are facts that can be concluded by applying some known "rule" (e.g. an axiom, definition, or theorem) to the set of facts in the originating node. In fact, automated theorem proving has been successfully applied to generate many non-trivial mathematical proofs, and similar techniques are sometimes even used to "prove" that a computer program is correct (of course, this begs the question of proving that the prover is correct, but we will ignore this issue for now).

Although we know our graph-search algorithm will work in theory for all these problems, in practice it will not be very useful for any but the simplest questions (those which we could probably solve without the use of a computer anyway). Luckily, there are a number of improvements to the graph search algorithm that makes tackling more complex problems possible. Let's examine a few of these, back in the familiar context of peg solitaire.

The page at `http://www.mazeworks.com/peggy/index.htm` gives a number of initial board configurations of the standard cross-shaped board and defines a "perfect game" as a series of moves ending with a single peg in the center hole. Your *solve-puzzle* function from question 5 should be able to solve the smallest configuration, "cross" and possibly "plus" as well. You could probably also solve these manually if you wanted to. But if you try to tackle a more difficult one, such as "fireplace" or "arrow", your program will probably take a very long time to run. Try implementing the following improvements:

- **Breadth-first search**. The type of search performed by the graph search algorithm discussed in class is called "depth-first" because we follow any branch of the graph all the way until it terminates before trying any other branch. This is very wasteful because we will end up revisiting each intermediate configuration many times on different branches.

  The breadth-first search approach is to examine all nodes at depth 1, and then all nodes at depth 2, etc. For our problem, the nodes at depth 1 are all the neighbours of the starting configuration, and then those at depth two are all the neighbours of any node at depth 1, and so forth. By removing duplicate configurations at every step, we avoid revisiting nodes. However, this does use more memory than depth-first recursion, so try to use accumulative recursion wherever possible unless your computer has a terabyte of RAM.

- **Symmetries** The ending configuration of a single peg in the center of the cross-shaped board has many lines of symmetry through it (specifically, four of them). When two intermediate configurations are symmetrical to each other, there is no need to examine both of them because they are equivalent if we just flipped the board around. So the removal of duplicates at each step can be extended to remove all symmetrical configurations at each step of the algorithm.

- **Bi-directional search** We have been examining the peg solitaire game in the "forward" direction, where each move reduces the number of pegs by one. But we might also approach the game from the "reverse direction", where the rules are applied in reverse so that each move increases the number of pegs by one. We could then search from the desired ending configuration to the desired starting configuration, instead of the other way around. This might give some improvement, but probably not much.

To get an improvement, perform a simultaneous search in both directions, until at the same step both searches have a number of possible intermediate configurations with the same number of pegs. Then simply find a common intermediate configuration from the backward and forward searches, and the problem is solved.

Many more enhancements can of course be utilized to gain even more efficient graph-search algorithms, to solve more and more interesting problems, and this is the subject of further study into AI if you are interested.