

CS15100 Lab 8: Word chains

Fall 2006

November 21, 2006

If you are in the 9:30AM MWF section, hand in your solution by emailing it to jacobm@cs.uchicago.edu. It must be in that mailbox before 3:00 PM on **Friday, December 1** (note the extended time).

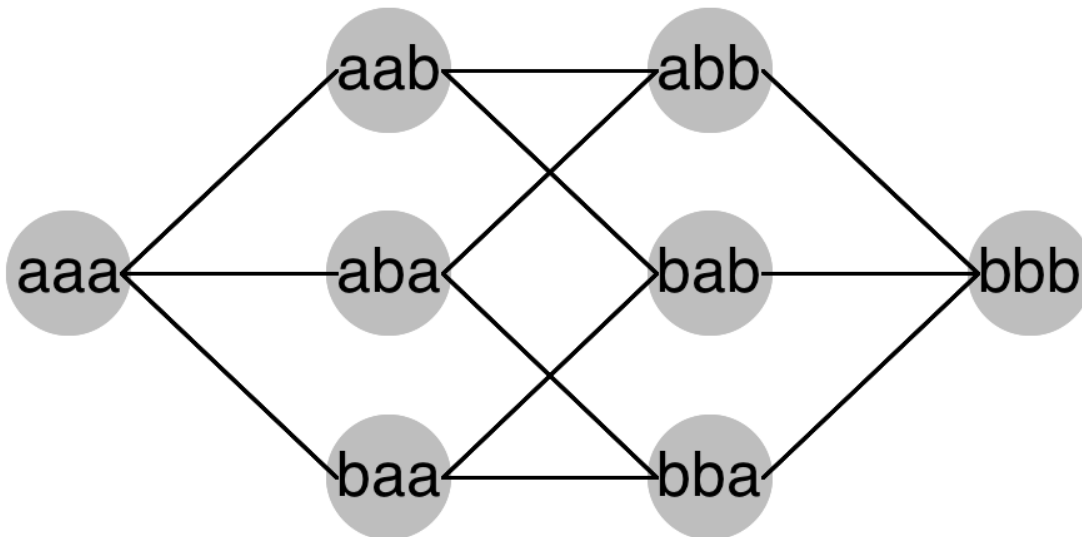
1 Introduction

A *word-chain* is a sequence of English words where each word is one letter away from the words before and after it. For instance,

```
(list "eat" "ear" "bar" "brr" "err" "ere" "are" "ate")
```

is a word chain from eat to ate.

In this lab, you will write a program that finds *the shortest* word chain connecting two given words in a dictionary. You will do that by viewing a dictionary as a graph where each node is a word and edges connect words that are one letter apart. For instance, the dictionary consisting of the words aaa, aab, aba, baa, abb, bab, bba, and bbb would look like this as a graph:



In this graph, every node is a word, and each edge connects a word to all the other words in the dictionary that are exactly one letter away from it (which means that every edge in this graph is bidirectional).

For this lab we will find it useful to use the following definition of a graph:

```
;; a graph is:  
;; (make-graph (string → boolean) (string → (listof string)))  
(define-struct graph (node? neighbors))
```

There are a few differences between this definition of a graph and the one you've seen in class. For one thing, this uses strings instead of symbols (and thus nodes can be manipulated with the functions `string→list : string → (listof character)`, `list→string : (listof character) → string`, and `string=? : string string → boolean`). For another, rather than explicitly containing a list of all valid nodes, instead it holds a function that will tell whether a particular string is a node in the graph.

2 Detect a path, depth first

Design the following function:

```
;; path-exists?/dfs : string string graph → boolean
;; determines if there is a path through G from n1 to n2
;; uses the depth-first traversal strategy
(define (path-exists?/dfs n1 n2 G)
  ...)
```

The “depth-first” traversal strategy means that when your function recurs on a node, the next thing it should do is recur on all of that node’s neighbors, making sure that it does not re-process nodes that it has already visited. (This is just the “normal” way of processing graphs you learned in class and used in homework.)

3 Detect a path, breadth first

Depth-first search is a very important strategy for traversing graphs, but it is not the only strategy. Another traversal strategy that we will find useful for this problem is called *breadth-first search*. If we think of depth-first search as the strategy that tries to go as far as possible from the start node in any particular step, then breadth-first search is the strategy that tries to stay as close to the start node as possible in any step. The breadth-first strategy can be implemented like this, using the central concept of a “to-visit” list of nodes that the algorithm will search in order:

1. Start with a to-visit list that contains just the starting node.
2. Pick the first item on the to-visit list. If it is the destination node, then stop. Otherwise, find all neighbors of the node you picked that haven’t already been visited, and add them to the *end* of the rest of the to-visit list.
3. Repeat the previous step using the newly-created list as the to-visit list.

If the process ever visits the destination node, then there is a path from the start node to the destination node; if it runs out of nodes to visit without reaching the destination then there is no path.

An example of two traversals of the same graph, one using the DFS strategy and the other using the BFS strategy, is available in the online version of this writeup and in this file:

<http://www.cs.uchicago.edu/~jacobm/15100-2006-fall/dfsvsbfs.pdf>

Design the following function:

```
;; path-exists?/bfs : string string graph → boolean
;; determines if there is a path through G from n1 to n2
;; uses the breadth-first traversal strategy
(define (path-exists?/dfs n1 n2 G)
  ...)
```

4 Find the shortest path

One handy property of BFS is that it always takes the shortest possible path from the start node to the destination node. For that reason, it's possible to write a variation on the `path-exists?/bfs` function from the last section that returns the shortest path from start to finish if such a path exists. To do that, you would write a function just like `path-exists?/bfs` but where each element of the to-visit list contains not just a node to visit, but also a path to that node from the start node.

Use this idea to design the function

```
;; shortest-path : string string graph → (union (listof string) false)
;; finds the shortest path between n1 and n2 in G if there is such a path, or false
;; if no path exists
(define (shortest-path n1 n2 G)
  ...)
```

5 Build the graph

To find word chains, the final step is to construct a graph out of a dictionary. Download and install the `dictionary.ss` teachpack here:

<http://www.cs.uchicago.edu/~jacobm/15100-2006-fall/dictionary.ss>

The `dictionary.ss` teachpack provides two functions:

```
;; file→dictionary : string → hash-table
;; converts a file with one word per line into a hash table

;; in-dictionary? : string hash-table → boolean
;; determines if the given word is in the hash table
```

(A hash-table is a data structure available in most programming languages that associates keys with values; one common use of hash tables is to build sets where we can efficiently test for membership. The teachpack functions provided here just build appropriate hash tables from a given file and provide an accessor function for them.)

Using these functions, write a function `file→graph : string → graph` that constructs a graph out of the words in the given file, where each word is a node and two words are connected if (a) they are the same length and (b) they differ by only one letter. For instance, 'bread' and 'broad' are neighbors but 'bread' and 'read' are not, nor are 'sword' and 'words.'

When writing the `neighbors` function, you will find these two support functions helpful:

```
;; other-letters : character → (listof characters)
;; given a letter, gives all the other letters in the alphabet
(define (other-letters char)
  ...)

;; permutations-as-list : (listof character) → (listof (listof character))
;; given a list representation of a word, returns the
;; list representations of all the one-letter-away permutations
;; of that list (whether or not they're legal words)
(define (permutations-as-list chars)
  ...)
```

Once you have finished developing `file→graph`, use the function `shortest-path` you wrote before to find word chains. Using this dictionary:

<http://www.cs.uchicago.edu/~jacobm/15100-2006-fall/words>

find minimal word chains connecting

- eat to ate
- town to hall
- doctor to scheme

if they exist.