**USACO JAN10 Problem 'wmorph' Analysis**

**by Rob Kolstad**

This task asks us to use a dictionary to 'morph' words, one letter at a time, from one valid dictionary word to another valid dictionary word (e.g., 'cat' -> 'cot' -> 'cog' -> 'dog').

Clearly, we need to choose some searching algorithm. Its goal will be to search from 'cat' to get all the possible one-letter changes (in this dictionary: bat, cat, cit, cot, cut, eat, fat, hat, mat, oat, pat, rat, sat, tat, vat). Those words must somehow be used to seatch for the next single letter change until the chain cat, cot, cog, dog (or shorter!) is found.

Noting that 'cat' and 'dog' share no common letters, it should be fairly clear that a depth-first search will be a catastrophe. Wandering around in the space of (as it turns out) almost 600 three letter words could take a very very long time, especially when searching for a 'shortest' path. In fact, that 'shortest' part is the cue that we should be using a \*breadth-first\* search.

Breadth-first searches are characterized by the use of a queue and a simple algorithm:

Enqueue first element (or first set of elements) to be searched

while (no solution found) {

de-queue a term from the front of the queue

foreach possible next term

solution? --> DONE

enqueue that term at the end of the queue

[optimization: do not enqueue terms already explored]

}

The solution below stores queue elements in a struct q\_f (the \_f reminds us that this is a 'form' -- a datatype) with a word (and plenty of room for its newline and '\0' terminating character) and the length of **the chain that led to that word**, not the number of characters in the word. This data structure should make the code that implements enqueue() and dequeue() clear enough.

The main logic of the program follows these general steps:

* Get the start and end words.
* Determine their length.
* Read the dictionary, keeping only words of that length (without this optimization, longer cases will time out). Note that each word has its terminating newline, if present, removed. **Programming Note:** One should submit the program for debugging at this point to make sure that the dictionary is being read correctly and to peek at some of the entries to make sure one's assumptions about the dictionary are correct. For instance, the dictionary is NOT sorted into any discernible order.
* Mark the start word in the dictionary as having been 'seen'.
* Enqueue the single start word.
* Implement the algorithm fragment shown above with dequeueing, generating of new words, and enqueueing them. **Optimization:** Marking the queued words as 'seen' eliminates the possibility of looking at a word more than once, which is almost a requirement here.

See below for my solution.

[view source](http://ace.delos.com/TESTDATA/JAN10.wmorph.htm#viewSource)



[print](http://ace.delos.com/TESTDATA/JAN10.wmorph.htm#printSource)[?](http://ace.delos.com/TESTDATA/JAN10.wmorph.htm#about)

|  |  |
| --- | --- |
| 001 | #include <stdio.h> |
| 002 | #include <stdlib.h> | |

|  |  |  |
| --- | --- | --- |
| 003 | #include <string.h> | |
| 004 |  |

|  |  |  |
| --- | --- | --- |
| 005 | char dict[25000][20+2]; | |
| 006 | char wordseen[25000]; |

|  |  |
| --- | --- |
| 007 | int ndict = 0; |
| 008 | char startword[20+2]; | |

|  |  |  |
| --- | --- | --- |
| 009 | char endword[20+2]; | |
| 010 |  |

|  |  |  |
| --- | --- | --- |
| 011 | #define NQUEUE 10000 | |
| 012 | struct q\_f { |

|  |  |  |
| --- | --- | --- |
| 013 | char q\_word[20+2]; | |
| 014 | int q\_len; |

|  |  |
| --- | --- |
| 015 | } q[NQUEUE]; |
| 016 | int qin, qout; | |

|  |  |
| --- | --- |
| 017 |  |
| 018 | void | |

|  |  |  |
| --- | --- | --- |
| 019 | enqueue (word, len) | |
| 020 | char \*word; |

|  |  |
| --- | --- |
| 021 | { |
| 022 | strcpy (q[qin].q\_word, (const char \*)word); | |

|  |  |
| --- | --- |
| 023 | q[qin].q\_len = len; |
| 024 | if (++qin >= NQUEUE) qin = 0; | |

|  |  |
| --- | --- |
| 025 | if (qin == qout) { |
| 026 | fprintf (stderr, "queue full\n"); | |

|  |  |  |
| --- | --- | --- |
| 027 | exit (1); | |
| 028 | } |

|  |  |
| --- | --- |
| 029 | } |
| 030 |  | |

|  |  |  |
| --- | --- | --- |
| 031 | struct q\_f \* | |
| 032 | dequeue() { |

|  |  |
| --- | --- |
| 033 | int oldqout = qout; |
| 034 | if (qout == qin)            /\* empty queue? \*/ | |

|  |  |
| --- | --- |
| 035 | return (struct q\_f \*)0; |
| 036 | if (++qout >= NQUEUE) qout = 0; | |

|  |  |  |
| --- | --- | --- |
| 037 | return &q[oldqout]; | |
| 038 | } |

|  |  |
| --- | --- |
| 039 |  |
| 040 | worddiff (w1, w2) | |

|  |  |  |
| --- | --- | --- |
| 041 | char \*w1, \*w2; | |
| 042 | { |

|  |  |
| --- | --- |
| 043 | int ndiffs = 0; |
| 044 | for ( ; \*w1 && \*w2; w1++, w2++) { | |

|  |  |  |
| --- | --- | --- |
| 045 | if (\*w1 == \*w2) continue; | |
| 046 | ndiffs++; |

|  |  |  |
| --- | --- | --- |
| 047 | if (ndiffs > 1) return ndiffs; | |
| 048 | } |

|  |  |  |
| --- | --- | --- |
| 049 | if (\*w1 || \*w2) return -1;          /\* differing lengths \*/ | |
| 050 | return ndiffs; |

|  |  |
| --- | --- |
| 051 | } |
| 052 |  | |

|  |  |  |
| --- | --- | --- |
| 053 | chomp (w) | |
| 054 | char \*w; |

|  |  |
| --- | --- |
| 055 | { |
| 056 | for (; \*w; w++) | |

|  |  |
| --- | --- |
| 057 | if (\*w == '\n') { |
| 058 | \*w = '\0'; | |

|  |  |  |
| --- | --- | --- |
| 059 | break; | |
| 060 | } |

|  |  |
| --- | --- |
| 061 | } |
| 062 |  | |

|  |  |
| --- | --- |
| 063 | main() { |
| 064 | FILE \*fdict = fopen ("dict.txt", "r"); | |

|  |  |
| --- | --- |
| 065 | FILE \*fin = fopen ("wmorph.in", "r"); |
| 066 | FILE \*fout = fopen ("wmorph.out", "w"); | |

|  |  |  |
| --- | --- | --- |
| 067 | int i, wordlen; | |
| 068 |  |

|  |  |  |
| --- | --- | --- |
| 069 | fscanf (fin, "%s", startword); | |
| 070 | fscanf (fin, "%s", endword); |

|  |  |  |
| --- | --- | --- |
| 071 | for (wordlen = 0; startword[wordlen]; wordlen++) | |
| 072 | ; |

|  |  |
| --- | --- |
| 073 |  |
| 074 | /\* get dictionary of words of length wordlen: \*/ | |

|  |  |
| --- | --- |
| 075 | char \*p; |
| 076 | while ( fgets (dict[ndict], 20, fdict))  { | |

|  |  |
| --- | --- |
| 077 | chomp (dict[ndict]); |
| 078 | if (strlen(dict[ndict]) != wordlen) continue; | |

|  |  |  |
| --- | --- | --- |
| 079 | ndict++; | |
| 080 | } |

|  |  |
| --- | --- |
| 081 | for (i = 0; i < ndict; i++) { |
| 082 | if (strcmp (startword, dict[i]) == 0) { | |

|  |  |  |
| --- | --- | --- |
| 083 | wordseen[i] = 1; | |
| 084 | break; |

|  |  |  |
| --- | --- | --- |
| 085 | } | |
| 086 | } |

|  |  |  |
| --- | --- | --- |
| 087 | enqueue (startword, 0); | |
| 088 | struct q\_f \*curword; |

|  |  |
| --- | --- |
| 089 | while (curword = dequeue()) { |
| 090 | if (strcmp (curword->q\_word, endword) == 0) { | |

|  |  |  |
| --- | --- | --- |
| 091 | fprintf (fout, "%d\n", curword->q\_len); | |
| 092 | exit (0); |

|  |  |
| --- | --- |
| 093 | } |
| 094 | /\* find potential new words to morph to \*/ | |

|  |  |
| --- | --- |
| 095 | for (i = 0; i < ndict; i++) { |
| 096 | if (wordseen[i]) continue; | |

|  |  |  |
| --- | --- | --- |
| 097 | if (worddiff(curword->q\_word, dict[i])==1) { | |
| 098 | wordseen[i] = 1; |

|  |  |  |
| --- | --- | --- |
| 099 | enqueue (dict[i], curword->q\_len + 1); | |
| 100 | } |

|  |  |  |
| --- | --- | --- |
| 101 | } | |
| 102 | } |

|  |  |  |
| --- | --- | --- |
| 103 | fprintf (stderr, "failed to morph\n"); | |
| 104 | exit (1); |

|  |  |
| --- | --- |
| 105 | } |