

Re: efficiently accumulating values

Source: <http://coding.derkeiler.com/Archive/Lisp/comp.lang.lisp/2006-07/msg00323.html>

- *From:* liyer.vijay@xxxxxxxxxx
 - *Date:* 7 Jul 2006 22:15:11 -0700
-

Ken Tilton wrote:

liyer.vijay@xxxxxxxxxx wrote:

Any other suggestions? What am I doing wrong?

Maybe nothing, but ever since you found that timing mistake that caused one Lisp version to only seem to go blazingly fast I keep wondering if you had the same bug in the Python timing. :)

Do you have a full standalone example others could run? Toss in the Python version as well. Maybe the ACL profiler will reveal something, though to be honest I have a wicked hard time understanding its output (hint to Franz <g>).

I too had a hard time with SBCL's profiler, though, admittedly, I didn't spend too much time reading the SBCL manual.

I would also be interested in the counts, shy of a full reproducible. What are the dimensions, how many words found, etc, etc.

kenny

Here is the lisp I wrote when I first came with my questions.

```
(defpackage :vijay.boggle
  (:use :cl)
  (:export :boggle :read-dictionary-file)
  (:nicknames :boggle))

(in-package :boggle)
```

Re: efficiently accumulating values

```
(defvar *words* (make-hash-table :test #'equal)
"The actual dictionary of words.")
(defvar *prefixes* (make-hash-table :test #'equal)
"The prefixes of all the words.")
(defparameter *minimum-length* 3 "The minimum length for any word.")
```

```
(defun read-dictionary-file (filename)
"Reads the dictionary in FILENAME and stores it."
(with-open-file (file filename) (read-dictionary file)))
```

```
(defun read-dictionary (stream)
"Reads in STREAM and stores its words in the dictionary."
(loop as line = (read-line stream nil nil)
while line
as word = (string-downcase (string-trim #(\Space #\Tab) line))
if (> (length word) 2) do (add-word word)))
```

```
(defun add-word (word)
"Add WORD to the dictionary."
(setf (gethash word *words*) t)
(loop as i from 2 below (1- (length word)) ; prefix ends at
penultimate char
as prefix = (subseq word 0 i)
do (setf (gethash prefix *prefixes*) t)))
```

```
(defun sanitize (letters)
"Changes LETTERS into the more commonly used form for the boggle
puzzle. It returns a list of strings of each alphabet in LETTERS.
This is mainly because `q' is actually `qu' for the puzzle."
;; There is a bug in this function. If given `aqcd' it will
;; (correctly) make it `aqucd'. Likewise, `aqucd' will be taken as
;; `aqucd' itself. So if the board has `q' and `u' in adjacent
;; squares, then it MUST be specified as `quu'.
(loop as char across (substitute-substring "q" "qu" letters)
collect (if (char= char #\q) "qu" (string char))))
```

```
(defun make-graph (letters)
"Forms the boggle puzzle from LETTERS."
(let* ((chars (sanitize letters))
(side (isqrt (length letters))))
(make-array (list side side)
:displaced-to (make-array (* side side)
:initial-contents chars))))
```

```
(defun boggle (letters &optional (*minimum-length* 3))
"Find all possible words in LETTERS."
(delete-duplicates (find-words (make-graph (string-downcase
letters)))
:test #'equal))
```

```
(defun find-words (graph)
```

Re: efficiently accumulating values

```
"Given GRAPH find all possible words."  
(destructuring-bind (m n) (array-dimensions graph)  
  (loop with result = ()  
    as i from 0 below m  
    do (loop as j from 0 below n  
      do (setq result (nconc result (get-words graph i  
j))))  
    finally (return result))))
```

```
(defun get-words (graph i j)  
  "Find all possible words starting from GRAPH position I J"  
  (labels ((rec (i j prefix acc)  
    (loop for (ni nj) in (adjacent graph i j)  
      as next = (aref graph ni nj)  
      as word = (concatenate 'string prefix next)  
      if (and (>= (length word) *minimum-length*) (wordp  
word))  
        do (push word acc)  
          when (prefixp word)  
            do (let ((old (aref graph i j)))  
              (setf (aref graph i j) 'nil  
acc (rec ni nj word acc)  
(aref graph i j) old))  
              finally (return acc))))  
    (rec i j (aref graph i j) '()))
```

```
(defun wordp (word)  
  "Is WORD a dictionary word?"  
  (gethash word *words*))
```

```
(defun prefixp (word)  
  "Are there dictionary words beginning with WORD ?"  
  (gethash word *prefixes*))
```

```
(defun adjacent (graph i j)  
  "Returns a pair (ai aj) of adjacent positions to GRAPH[i j]"  
  (let ((side (array-dimension graph 0))  
    (adjacent '()))  
    (labels ((legalp (i) (and (>= i 0) (< i side)))  
      (add (i j)  
        (if (and (legalp i) (legalp j) (aref graph i j))  
          (push (list i j) adjacent))))  
      (let ((i-1 (1- i)) (i+1 (1+ i)) (j-1 (1- j)) (j+1 (1+ j)))  
        (add i-1 j-1) (add i j-1) (add i+1 j-1)  
        (add i-1 j) (add i+1 j)  
        (add i-1 j+1) (add i j+1) (add i+1 j+1))  
        adjacent)))
```

```
(defun substitute-substring (new old string &key (test #'string=))  
  "Substitutes OLD by NEW in STRING."
```

Re: efficiently accumulating values

```
(let ((pos (search old string :test test)))
  (if pos
    (concatenate 'string
      (subseq string 0 pos)
      new
      (substitute-substring new
        old
        (subseq string (+ pos
          (length old)))
        :test test)
      string)))
```

```
;;; *eof*
```

```
;;;;; Python code follows
```

```
from math import sqrt
```

```
WORDS = { }
```

```
def read_dictionary(stream):
    'Read the file for words for the dictionary.'
    for line in stream.readlines():
        word = line.lower().strip()
        if len(word) > 2: add_word(word)
```

```
def add_word(word):
    'Add a word to the dictionary.'
    global WORDS
    WORDS[word] = True
    for i in range(2, len(word)):
        prefix = word[:i]
        if not is_prefix(prefix):
            WORDS[prefix] = 0
```

```
def make_graph(letters):
    "Form the boggle puzzle given LETTERS."
    letters.replace('qu', 'q')
    side = int(sqrt(len(letters)))
    graph = [[] for i in range(side)]
    for i in range(side):
        graph[i] = [[] for j in range(side)]
    k = 0
    for i in range(side):
        for j in range(side):
            graph[i][j] = letters[k].replace('q', 'qu')
            k += 1
    return graph
```

```
def boggle (letters):
    'Solve the boggle puzzle for the given letters'
```

Re: efficiently accumulating values

Re: efficiently accumulating values

```
return list(set(find_words(make_graph(letters.lower()))))
```

```
def find_words(graph):
    m, n = len(graph), len(graph[0])
    results = []
    for i in range(m):
        for j in range(n):
            results.extend(get_words(graph, i, j))
    return results
```

```
def get_words(graph, i, j):
    def rec(i, j, prefix, acc):
        for ni, nj in adjacent(graph, i, j):
            next = graph[ni][nj]
            word = prefix + next
            if len(word) > 3 and is_word(word):
                acc.append(word)
            if is_prefix(word):
                old_val = graph[i][j]
                graph[i][j] = ""
                acc = rec(ni, nj, word, acc)
                graph[i][j] = old_val
            return acc
    return rec(i, j, graph[i][j], [])
```

```
def is_word(word):
    global WORDS
    return is_prefix(word) and WORDS[word]
```

```
def is_prefix(word):
    global WORDS
    return WORDS.has_key(word)
```

```
def adjacent(graph, i, j):
    m = len(graph)
    adjacent = []
    is_legal = lambda i: i >= 0 and i < m
    def add(i, j):
        if is_legal(i) and is_legal(j) and graph[i][j]:
            adjacent.append((i, j))
    add(i-1, j-1); add(i, j-1); add(i+1, j-1)
    add(i-1, j); add(i+1, j)
    add(i-1, j+1); add(i, j+1); add(i+1, j+1)
    return adjacent
```

```
### eof
```

As to my bug in the FIND-WORDS
(loop ... if (and (zerop i) (zerop j)) do (get-words graph i j) ...)
was because I had included the IF for debugging and I'm so used to C-k
killing sexp (because I use paredit.el) that I forgot that LOOP dealt

Re: efficiently accumulating values

Re: efficiently accumulating values

with lines. Hence the mistake. I corrected it in other posts. I apologize for the confusion caused.

The python code is almost the exact same as the lisp code. I was bored one day and translated lisp to python. Right now I am not very concerned about efficiency or style in python (after this, I'll ask the fine folks at comp.lang.python :-)

However, the python code runs faster (for this particular input) than the lisp code. It is also shorter but not by much. The advantage in python is that I can use (in find_words) results.extend to accumulate the values.

About the input, I saw on Peter Norvig's site that "rstcsdeiaegnlrpeatesmssid" gives the most number of words (2905) for the same dictionary that Sidney Markowitz refers to <http://www.gtoal.com/wordgames/yawl/word.list> This is the same dictionary that I use and I get the same number of words. For this input, the python code is faster by a factor of 2. When I discard the results in both of them, lisp is faster by a factor of about 3. For inputs which don't give many results like "abcdefghijklmnopqrstuvwxy" the lisp code is faster by about 1.5 to 2. For "abcd...wxy" lisp takes a mere 0.013 seconds.

Coming back to lisp, I don't believe that the bottleneck is in GET-WORDS. The timings when I don't accumulate the results from GET-WORDS (as Sidney Markowitz asked) I get the best time of 0.15 seconds. So even if GET-WORDS can be further optimized (as noted by Kaz Kylheku in several places), I believe the greatest time is taken when so many results need to be stitched together.

I restarted my computer (it's been on for a few weeks now) and ran all these tests again. Here are the new numbers with corresponding code.

BOGGLE>

```
(run
(defun find-words (graph)
  "Original FIND-WORDS"
  (destructuring-bind (m n) (array-dimensions graph)
    (loop with result = ()
      as i from 0 below m
      do (loop as j from 0 below n
          do (setq result (nconc result (get-words graph i
j))))
        finally (return result))))
```

```
(defun find-words (graph)
  "use tail pointer"
  (destructuring-bind (m n) (array-dimensions graph)
    (loop with result = () and tail = ()
      as i from 0 below m
```

Re: efficiently accumulating values

Re: efficiently accumulating values

```
do (loop as j from 0 below n
as gotten-words = (get-words graph i j)
if result do (rplacd tail gotten-words)
else do (setf result gotten-words)
do (setf tail (last gotten-words)))
finally (return result))))
```

```
(defun find-words (graph)
"loop nconcs"
(destructuring-bind (m n) (array-dimensions graph)
(loop as i from 0 below m
nconcing (loop as j from 0 below n
nconcing (get-words graph i j))))))
```

```
(defun find-words (graph)
"collect and flatten"
(destructuring-bind (m n) (array-dimensions graph)
(flet ((get-ij (num)
(let ((i (floor (/ num m))))
(list i (- num (* i m))))))
(loop as num from 0 below (* m n)
as (i j) = (get-ij num)
collecting (get-words graph i j) into results
finally (return (apply #'nconc results))))))
```

```
(defun find-words (graph)
"hash-table"
(destructuring-bind (m n) (array-dimensions graph)
(loop with result = (make-hash-table :test #'equal)
as i from 0 below m
do (loop as j from 0 below n
do (loop as word in (get-words graph i j)
do (setf (gethash word result) nil)))
finally (return (loop as key being the hash-key in result
collect key))))))
```

```
(defun find-words (graph)
"vector-push-extend"
(destructuring-bind (m n) (array-dimensions graph)
(loop with result = (make-array 0 :fill-pointer 0 :adjustable t)
as i from 0 below m
do (loop as j from 0 below n
do (loop as word in (get-words graph i j)
do (vector-push-extend word result)))
finally (return (coerce result 'list))))))
```

```
(defun find-words (graph)
"concatenate"
(destructuring-bind (m n) (array-dimensions graph)
(loop with result = (make-array 0 :fill-pointer 0 :adjustable t)
as i from 0 below m
```

Re: efficiently accumulating values

```
do (loop as j from 0 below n
do (setq result
(concatenate 'vector result
(get-words graph i j))))
finally (return (coerce result 'list))))
```

```
(defun find-words (graph)
"discard results"
(destructuring-bind (m n) (array-dimensions graph)
(loop as i from 0 below m
do (loop as j from 0 below n
do (get-words graph i j)))))
```

```
(defun find-words (graph)
"change order of nconc arguments"
(destructuring-bind (m n) (array-dimensions graph)
(loop with result = '()
as i from 0 below m
do (loop as j from 0 below n
do (setf result (nconc (get-words graph i j)
result)))
finally (return result))))
```

```
(progn
(defvar *collect* nil)
```

```
(defun find-words (graph)
"collect with special variable"
(destructuring-bind (m n) (array-dimensions graph)
(loop with *collect* = '()
as i from 0 below m
do (loop as j from 0 below n
do (get-words graph i j))
finally (return *collect*))))
```

```
(defun get-words (graph i j)
"find all possible words starting from graph position i j"
(labels ((rec (i j prefix)
(loop for (ni nj) in (adjacent graph i j)
as next = (aref graph ni nj)
as word = (concatenate 'string prefix next)
if (and (>= (length word) *minimum-length*)
(wordp word))
do (push word *collect*)
when (prefixp word)
do (let ((old (aref graph i j)))
(setf (aref graph i j) 'nil)
(rec ni nj word)
(setf (aref graph i j) old))))))
(rec i j (aref graph i j))))
```

Re: efficiently accumulating values

Original FIND-WORDS --> 0.79526
use tail pointer --> 0.77866
loop nconcs --> 0.7815
collect and flatten --> 0.77976
hash-table --> 0.5194
vector-push-extend --> 0.77916
concatenate --> 0.78652
discard results --> 0.1086
change order of nconc arguments --> 0.73612
collect with special variable --> 0.73658

I look forward to more of your insightful comments.

Cheers
Vijay

Can we quote you on that?

A long time ago, someone in the Lisp industry told me it was poor form
quote people; it suggests that they lack value.
-- Kent M Pitman <pitman@xxxxxxxxxxxxxx> in comp.lang.lisp

No way! It suggests that they have transcended ordinary existence to
dwell among the symbols. Moreover, it shows that we are actually
interested in /them/ and not merely in what they can evaluate for us.
-- Kaz Kylheku in comp.lang.lisp

.