TODO

Peter Danenberg

02 September 2011

Contents

1 TODO Tests

2 DONE Test package

CLOSED: *2011-09-02 Fri 17:35*

```
(use 'hangman.core
     'hangman.frequency-strategy
     'hangman.predicate-strategy
     'hangman.regex-strategy
     'hangman.trie-strategy
     '[clojure.pprint :only (pprint)]
     '[clojure.contrib.generic.functor :only (fmap)]
     '[clojure.contrib.io :only (reader)])

(let [arity->dictionary (make-arity->regex-dictionary "words-head.txt")
      arity->letter->count (fmap deterministic-count-letters arity->dictionary)
      words (line-seq (reader "words-head.txt"))
      words ["this-word-doesn't-exist"]]
  (pprint
   (run-words make-deterministic-regex-strategy
              arity->dictionary
              arity->letter->count
              (constantly nil)
              4
              words
              false)))
```

```
#_ (let [arity->dictionary (make-arity->regex-dictionary "words-head.txt")
      arity->letter->count (fmap sampling-count-letters arity->dictionary)
      words (line-seq (reader "words-head.txt"))]
  (pprint
   (run-words make-sampling-regex-strategy
              arity->dictionary
              arity->letter->count
              (constantly nil)
              4
              words)))

#_ (let [arity->dictionary (make-arity->predicate-dictionary "words-head.txt")
      arity->letter->count (fmap sampling-count-letters arity->dictionary)
      words (line-seq (reader "words-head.txt"))]
  (pprint
   (run-words make-sampling-predicate-strategy
              arity->dictionary
              arity->letter->count
              (constantly nil)
              4
              words)))

#_ (let [arity->dictionary (make-arity->trie-dictionary "words-head.txt")
      arity->letter->count (fmap count-trie arity->dictionary)
      words (line-seq (reader "words-head.txt"))]
  (pprint
   (run-words make-trie-strategy
              arity->dictionary
              arity->letter->count
              identity
              4
              words)))
```

Sampling vs. deterministic on entire dictionary (constant overhead large enough in sampling to slow it down, apparently):

```
{:average-time 22.390905,
 :average-score 7.340243}

{:average-time 17.91233,
 :average-score 7.3121324}
```

According to Aaron: no amnesty for Clojure.

The strategies shouldn't have to deal with reading the file to create dictionaries: they should just be reductions over words.

Non-mutually-exclusive algorithms for comparison?

3 DONE Main

CLOSED: *2011-09-02 Fri 17:35*

'–deterministic' vs. '–sampling'; we just want average scores and times, since we don't have anything to compare it with. Why not '–trie' and '–predicate'? Also: '–whole-dictionary'; not to mention: '–verbose' (may affect running time). '–max-guesses', too.

4 DONE Rationale

CLOSED: *2011-09-02 Fri 17:35*

I went through three implementations before I found one with acceptable performance characteristics: regular-expression based, trie-based and sample-based. They all fell into a common pattern: prune the dictionary on last guess; and, if the ratio of remaining words to guesses looks good, guess a word; otherwise, rank the letters by frequency and guess a letter.

The three implementations differed in the mechanism for storing the dictionary, pruning the dictionary and ranking the letters by frequency.

The trie-based strategy stored the dictionary as a sparse 26-ary trie. I had hoped that the trie-based strategy would be the most efficient, since search, deletion and insertion happen in $O(\log_{26} n)$; but traversal time, nominally $O(n)$, ended up dominating.

The trie-based strategy required significant initialization time; and there was an interesting process of trie-degeneration during pruning that resulted in specious frequency counts.

The regex- and sampling-strategies apply predicates to strings and count letters linearly; the difference is that the sampling-strategy works on integer strings and stops counting when the ratios of letters are sufficiently stable.

Mutually exclusive strategies on the command line?

A little bizarre: we have this tripartite Venn diagram: predicate and regex share strings; predicate and trie share predicates; trie and regex are orthogonal?

Thought I could sacrifice some trie-invariants for the sake of $O(\log n)$ deletion; but this proved fatal. Also, large constants overruled the $O(\log n)$.

4.1 Abstract

I'm presenting four solutions to the hangman problem which are sub-strategies of the arch-strategy I call the **frequency strategy**; the sub-strategies are the so-called **regex-strategy**, the **predicate-strategy** and the **trie-strategy**.

5 DONE Experimental setup

CLOSED: *2011-09-02 Fri 17:35*

Would like to have a table at the end that compares the scores of the regex, trie and string implementations against the reference times (as well as each other's); in addition to score.

Also, let's create some graphs; adding them, for instance, to the TODO. (Just in case they gnuplot, etc. don't exist on the target machine.)

Some kind of matrix, which is: implementation's time and deviation from the best time; implementation's score and deviation from the best score.

This should be overviewable, too.

The performance numbers I've transcribed for testing are meaningless on other machines; that's why we have to measure relative performance.

6 DONE Frequency strategy

`CLOSED:` *2011-09-02 Fri 17:35*

It turns out that we're replicating frequency-based strategy again and again: the only difference is the storage mechanism for dictionaries, retrieval and filtering mechanism for words (given predicates), and the counting mechanism.

We should abstract this into a generic frequency-based strategy with regex, trie and integer-string implementations.

Things to abstract while we're at it: the experimental setup with scores and times.

```
(defn make-frequency-strategy
  [filter-dictionary
   count-letters
   remove-word
   char->letter
   letter->char
   word->string
   get-words
   initial-dictionary
   initial-letter->count]
  (let [dictionary (atom initial-dictionary)
        letter->count (atom initial-letter->count)
        last-guess (atom nil)]
    (reify
      GuessingStrategy
      (nextGuess [_ game]
        (do
          (if @last-guess
            (let [guessed-so-far (.toLowerCase (.getGuessedSoFar game))]
              (reset! dictionary (filter-dictionary @dictionary guessed-so-far @last-guess))
              (reset! letter->count (count-letters @dictionary))))
          (let [words (get-words @dictionary)
                n-words (count words)]
```

```
(let [remaining-guesses (.numWrongGuessesRemaining game)]
  (if (<= n-words remaining-guesses)
    (let [word (nth words (rand-int n-words))]
      (reset! dictionary (remove-word word @dictionary))
      (reset! last-guess nil)
      (new GuessWord (word->string word)))
    (let [guessed-letters
          (map #(-> % Character/toLowerCase char->letter)
               (.getAllGuessedLetters game))
          ;; Devaluate letters already-guessed.
          letter->count
          (merge @letter->count
                 (zipmap guessed-letters (replicate (count guessed-letters) 0)))
          count->letter
          (into (sorted-map-by >) (map-invert letter->count))
          next-guess
          (count->letter (apply max (keys count->letter)))]
      (reset! last-guess next-guess)
      (new GuessLetter (letter->char next-guess)))))))))))
```

7 DONE Trie based guessing strategy

`CLOSED:` *2011-09-02 Fri 17:36*

Generate trie at startup; break links as a filtering mechanism (have to copy the datastructure, therefore). Trie of cardinality 26 corresponding to $a \rightarrow z$; normally used for prefix matching. Can we really beat (the more general) regex by iteration?

See Bagwell's Ideal Hash Tries, by the way.

An initial implementation could use nested `hash-sets` or `sorted-sets`; why not arrays of cardinality 26, though? They might be sparse, and we have to do key-iteration on unknown letters. Let's compare hash-set-, sorted-set-, hash-map-, sorted-map- and vector-based implementations. Let's start with vector.

How do we reject candidates: sentinel at leaves? Segregate tries by cardinality and check depth?

That's right, though: tries need to be of cardinality 26 + 1 to account for sentinels.

Let's do hash-tables; they're more general, though. Otherwise: we have to do the character -> index mapping (which is a form of hashing, by the way).

We need: `insert` and `search`; let's do this utterly ad-hoc to strings; such that we can insert and search for strings. Let's even formalize the `_`-as-wildcard mechanism.

Question between vectors and hash-tables comes down to whether linear search over the vector-cardinality is longer than key-retrieval for the latter (I suspect that it is, by the way, in addition to being a pain-in-the-ass).

And this all before we've done any profiling; hmm. `jvisualvm` didn't yield anything interesting (that I could determine). Manual profiling, on the other hand, revealed that the application of `merge` over map-dyads is expense; reduction over `assoc` yielded ten percent savings.

Let's do a stack-based traversal of the tree (depth-first; I wonder: will depth- or breadth-first make a difference?); a wild-card adds all letters to the traversal stack. As soon as we encounter a letter that doesn't exist, we can sever the tie; can't we?

How are we going to do the letter->frequency histogram with a trie, by the way? Doesn't it require a complete trie traversal? This requires memory overhead, of course: but if each node in the trie stored histography going down, we could delete the node and update the histography immediately.

The storage required is probably pretty big, though: a 26-arity vector for every vector. Or is there some kind of Huffman encoding we could do? It is essentially a 26-ary Huffman encoding. (See n-ary Huffman coding, btw.

Priority queue. Each node is an object containing: the next vector down; a vector of frequencies for each letter down. There's got to be a clever algorithm for this. How fast, though; and do we have egregious initialization times?

7.1 DONE Is comparing chars faster than comparing ints?

`CLOSED:` *2011-08-23 Tue 21:06*

- CLOSING NOTE *2011-08-23 Tue 21:06*
  It turns out they're roughly the same (roughly 500 msecs).

```
(time (reduce = (replicate 1000000 \a)))
(time (reduce = (replicate 1000000 1)))
```

7.2 DONE Test that mutating shallow clones of HashMaps does not mutate clonee.

`CLOSED:` *2011-08-23 Tue 21:08*

- CLOSING NOTE *2011-08-23 Tue 21:08*
  It is indeed the case.

```
(let [hash-map (new java.util.HashMap)]
  (. hash-map put "a" 1)
  (. hash-map put "b" 2)
  (let [hash-map-clone (. hash-map clone)]
    (. hash-map-clone remove "a")
    (assert (= {"a" 1 "b" 2} (into {} hash-map)))))
```

7.3 DONE Trie from word

CLOSED: *2011-09-02 Fri 17:35*

```
;;; Can we keep them as transients? We have to copy and mutate them
;;; later on, though.

;;; The weird thing about this is that it ends on an empty hashmap as
;;; a form of sentinel.
(let [trie (new java.util.HashMap)
      word "harro"]
  (reduce (fn [trie letter]
            ;; Ugly; what's the idiom: chaining? if-let?
            (let [subtrie (. trie get letter)
                  subtrie (if subtrie subtrie (new java.util.HashMap))]
              (. trie put letter subtrie)
              subtrie))
          trie
          word)
  trie)
```

We decided in the car to build the trie up from the leaves using a forest of letters indexed by a vector; build it back up to root (could also go the other way), adding things along the way. This is arbitrary, since we're not doing prefix-stems here.

We're also going to deal with vectors of integers, not hashmaps of chars; though the latter is a space-saving hack.

```
(use 'clojure.contrib.trace)

(reduce (fn [trie letter]
          (let [subtrie (trie letter {})]
            (assoc trie letter subtrie)
            subtrie))
        {}
        "harro")

(defn word->trie [trie word]
  (loop [word word
         trie trie]
    (let [letter (first word)]
      (if letter
        (recur (next word)
               (assoc trie letter (trie letter {})))
        trie))))

(defn make-trie []
```

```
  (transient (vec (repeat 26 nil))))

;; (trace make-trie)

;;; We can't iterate over the transient tree without persisting it;
;;; but we can't mutate the tree once we've persisted it. Lazy
;;; persistent tries?
(defn persistent-trie! [trie]
  (if trie
    (let [trie (persistent! trie)]
      (doseq [subtrie trie]
        (if subtrie
          (persistent-trie! subtrie)))
      trie)))

(defn persistent-trie! [trie]
  (if trie
    (do
      (let [n (count trie)]
        (loop [i 0]
          (if (< i n)
            (let [subtrie (trie i)]
              (if subtrie
                (do
                  (persistent-trie! subtrie)
                  (assoc! trie i (persistent! subtrie))))
              (recur (inc i)))))
        trie))))

(defn hash-word! [trie word]
  (reduce (fn [trie letter]
            (let [subtrie (trie letter)
                  subtrie (if subtrie subtrie (make-trie))]
              (assoc! trie letter subtrie)
              subtrie))
          trie
          word))

;; (trace hash-word!)

;;; Ouch: hashing 200k words (without frequencies) takes 14s! Not to
;;; mention the fact that memory balooned up to 433M, including 31M of
;;; PersistentVectors and 24M of TransientVectors plus 18M of
;;; PersistentVector$Nodes; 8M of ints (but maybe that's not so bad).
;;;
;;; It would be nice if we could build this without going the
```

```
;;; transient route; let's come up with a functional algorithm to do
;;; so (and with maps, maybe, if the memory overhead is egregious).
(time
 (let [words (take 200000 (repeatedly (fn [] (take 7 (repeatedly (fn [] (rand-int 26)))))))
       trie (make-trie)]
   (doseq [word words] (hash-word! trie word))
   (persistent! (persistent-trie! trie))
   "harro"))

#_ (let [trie (make-trie)]
  (hash-word! trie [8 24 3 23 12 20 15])
  (persistent! (persistent-trie! trie)))
```

To do this bottom-up approach, we're going to operate on words of the same arity. We can't do that, though: not all leaves of value n are the same because of the prefix-context. Can we count somehow from the root down?

As opposed to packing a letter/histogram in the same node, let's have two orthogonal tries: frequency-trie and letter-trie.

In terms of a functional algorithm for building trees from words: some sort of reduction which works from the bottom up.

```
(use
 'clojure.contrib.trace
 ;; 'clojure.contrib.singleton
 ;; '[hangman.core :only (debug)]
 'hangman.core
 '[clojure.contrib.io :only (reader)]
 ;; 'clojure.contrib.monads
 'net.dnolen.clj-cont
 ;; 'delimc.core
 '[clojure.set :only (difference map-invert)]
 '[clojure.string :only (join)]
 )

(import '(com.factual.hangman
          GuessingStrategy
          GuessLetter
          GuessWord
          HangmanGame)
        'java.lang.Character)

(def trie-sentinel (gensym))

(defn trie-sentinel? [object]
  (= object trie-sentinel))
```

```
(def wildcard (gensym))

(defn wildcard? [object]
  (= object wildcard))

(defn plumb
  ([word] (plumb {} word))
  ([trie word]
     (if word
       (let [letter (first word)]
         (assoc (trie letter {})
           letter
           (plumb trie
                  (next word))))
       trie-sentinel)))

(defn count-trie
  ([trie] (count-trie {} trie))
  ([frequencies trie]
     (if (not (trie-sentinel? trie))
       (merge-with
        +
        frequencies
        (reduce
         (fn [frequencies key]
           (merge-with
            frequencies
            (count-trie frequencies (trie key))))
         (let [keys (keys trie)]
           (zipmap
            keys
            (replicate (count keys) 1)))
         (keys trie)))
       frequencies)))

;;; Can't we avoid predicates by cleaning the trie more thoroughly on
;;; search-and-dissoc? We're going to have to count on
;;; wildcard-default after a search-and-dissoc pruning.
(defn count-trie-predicates
  ([trie predicates]
     (count-trie-predicates {} trie predicates))
  ([frequencies trie predicates]
     ;; (debug trie
     ;;        (keys trie)
     ;;        predicates)
     ;; (debug (if (map? trie) (keys trie)))
```

```
    (if (or (trie-sentinel? trie)
            (empty? predicates))
      frequencies
      (let [predicate (first predicates)
            keys (filter predicate (keys trie))]
        (debug frequencies predicate keys)
        (merge-with
         +
         frequencies
         (reduce
          (fn [frequencies key]
            (merge-with
             frequencies
             (count-trie-predicates
              frequencies
              (trie key)
              (next predicates))))
          (zipmap
           keys
           (replicate (count keys) 1))
          keys))))))

;;; Let's compare this transient-based implementation with the above
;;; for time-efficiency.
(defn count-trie!
  ([trie]
     (count-trie! (transient (vec (replicate 26 0))) trie))
  ([frequencies trie]
     (doseq [key (keys trie)]
       (assoc! frequencies key (inc (frequencies key)))
       (count-trie! frequencies (trie key)))
     frequencies))

;;; Really what we need to do is a recursive merge of some sort: the
;;; problem is, we have to account for all keys; including ones which
;;; are currently different and future divergence, too. Ouch.
;;;
;;; No, that's not the case (although I wonder if we could dynamically
;;; program this to avoid exponential complexity): the problem is a
;;; multi-key mergens with a single-key mergendum. Therefore, the
;;; first-key hack should abide (, dude).
(defn merge-tries [mergens mergendum]
  (cond (or (trie-sentinel? mergens) (empty? mergens)) mergendum
        (or (trie-sentinel? mergendum) (empty? mergendum)) mergens
        :else (let [mergendum-key (first (keys mergendum))
                    submergens (mergens mergendum-key)
```

```
                    submergendum (mergendum mergendum-key)]
                (if (mergendum mergendum-key)
                  (assoc mergens mergendum-key
                         (merge-tries submergens submergendum))
                  (assoc mergens mergendum-key
                         (mergendum mergendum-key))))))

;;; Yikes: both trie and word are false in a match and mis-match; have
;;; to distinguish between end-of-line and infix. We're going to need
;;; some kind of sentinel, therefore, at the end of plumb. Why not the
;;; word itself? That way, I don't have to thread it during
;;; search. Space-complexity, though.
(defn search-trie [trie word]
  ;; Be careful here: what about mistaking subsets for matching words?
  ;; We won't have to worry about this, for instance, if we can be
  ;; guaranteed to have the same arity. Let's make this assumption.
  (cond
   ;; What about matching the null-trie, though? Only the null-word
   ;; matches it.
   ;;
   ;; This is where we'd have to modify trie, isn't it, with dissoc;
   ;; and pass back the dissociated subtrie for modification of
   ;; counts?
   (not trie) false
   (trie-sentinel? trie) true
   :else
   (let [letter (first word)]
     (search-trie (trie letter) (next word)))))

;;; Do we effectively have to rebuild what matches? Also: we have to
;;; figure out how to deal with wildcards: we're going to need to add
;;; all letters to the search stack.
;;;
;;; Actually, this is not totally distinct from count-trie; we're
;;; basically rebuilding the tree except for one disassociation. Ouch!
;;; That's going to be ~3s for a 200k trie, isn't it?
;;;
;;; This is where it might pay to use mutation: we only have to pay a
;;; penalty at the beginning for copying; we could even use
;;; HashTable's clone method.
;;;
;;; Let's avoid an adjunct trie if at all possible! This entails
;;; tautological self-merges on the trie, I believe.
;;;
;;; We should avoid running this on all wildcalds, of
;;; course. (Negative result, by the way? Trying to beat regex.)
```

```
;;;
;;; Especially with wildcards, there may be many branches that we have
;;; to filter out.
;;;
;;; For wildcards: dispatch on 26 search-and-dissocs.
(defn search-and-dissoc [trie word]
  (if word
    (let [letter (first word)
          subtrie (trie letter)]
      (if subtrie
        (assoc trie letter (search-and-dissoc subtrie (next word)))
        trie-sentinel))
    trie))

;;; Ouch: don't forget that search-and-dissoc needs to return a list
;;; of candidate words, too. Or is 'leaves' a separate function? That
;;; would be too bad: we're already doing the work here. It would be
;;; nice if we could take care of frequency-propagation, too.
(defn search-and-dissoc [trie word]
  (if word
    (let [letter (first word)]
      (if (wildcard? letter)
        (reduce
         (fn [trie letter]
           (assoc trie letter (search-and-dissoc (trie letter) (next word))))
         trie
         (keys trie))
        (let [subtrie (trie letter)]
          (if subtrie
            (assoc trie letter (search-and-dissoc subtrie (next word)))
            trie-sentinel))))
    trie))

(def wildcard-predicate (constantly true))
(defn positive-predicate [predicans]
  (fn [predicandum] (= predicans predicandum)))
(defn negative-predicate [predicans]
  (fn [predicandum] (not (= predicans predicandum))))

;;; Do we have a problem here with positive vs. negative predicates? A
;;; negative predicate, for instance, caps the branch with a sentinel;
;;; and leaves artifacts in e.g. the root.
;;;
;;; Let's think about this: a full trie gets a negative predicate at
;;; root; caps it. This is incorrect, isn't it? It seems like it
;;; should purge the key.
```

```
;;;
;;; It doesn't appear to be the negative predicates, but the positive
;;; ones that create artifacts: negatives are indeed purged. This
;;; makes sense, frankly: none of the initial nodes in e.g. "oses"
;;; have been purged because they didn't contravene the negative
;;; criteria; on the other hand, the second letter of most of them
;;; contravened the positive predicate. count-trie, therefore, has to
;;; have some predicates, I'm afraid.
(defn search-and-dissoc-predicate [trie predicates]
  ;; (debug trie)
  (if (or (empty? predicates)
          (trie-sentinel? trie))
    trie
    (let [predicate (first predicates)
          letters (filter predicate (keys trie))]
      ;; (debug predicate letters)
      (if (empty? letters)
        trie-sentinel
        (reduce
         (fn [subtrie letter]
           (assoc subtrie
             letter
             (search-and-dissoc-predicate (trie letter) (next predicates))))
         {}
         letters)))))

(defn search-and-dissoc-predicate-with-frequencies
  [trie predicates frequencies]
  ;; (debug trie)
  (if (or (empty? predicates)
          (trie-sentinel? trie))
    trie
    (let [predicate (first predicates)
          letters (filter predicate (keys trie))]
      ;; (debug predicate letters)
      (if (empty? letters)
        trie-sentinel
        (reduce
         (fn [subtrie letter]
           (assoc subtrie
             letter
             (search-and-dissoc-predicate (trie letter) (next predicates))))
         {}
         letters)))))

(defn sufficient-depth?-with-cc
```

```
  ([k trie depth current-depth]
     (debug depth current-depth)
     (if (= depth current-depth)
       (k true)
       (if (trie-sentinel? trie)
         false
         (doseq [key (keys trie)]
           (sufficient-depth?-with-cc
             k
             trie
             depth
             (inc current-depth)))))))

(defn sufficient-depth?
  ([trie depth]
     (sufficient-depth? trie depth 1))
  ([trie depth current-depth]
     (with-call-cc
       (let-cc
        k
        (sufficient-depth?-with-cc
          k
          trie
          depth
          current-depth)))))

;;; Just character count on words?
(defn search-and-dissoc-predicate-with-depth
  ([trie predicates depth]
     (search-and-dissoc-predicate-with-depth trie predicates depth 1))
  ;; (debug trie)
  ([trie predicates depth current-depth]
     (debug predicates depth current-depth)
     (if (or (empty? predicates)
             (trie-sentinel? trie))
       trie
       (let [predicate (first predicates)
             letters (filter predicate (keys trie))]
         ;; (debug predicate letters)
         (if (empty? letters)
           trie-sentinel
           (reduce
            (fn [search-trie letter]
              (let [subtrie (trie letter)]
                (if (sufficient-depth? subtrie depth)
                  (assoc search-trie
```

```
                    letter
                    (search-and-dissoc-predicate-with-depth
                     subtrie
                     (next predicates)
                     (- depth current-depth)
                     (inc current-depth)))
                  search-trie)))
            {}
            letters))))))

(defn get-words
  ([trie] (get-words trie '() '()))
  ([trie words word]
     ;; (debug trie words word)
     (if (trie-sentinel? trie)
       (list word)
       (reduce
        (fn [words letter]
          ;; (debug words letter (trie letter))
          (concat (get-words (trie letter) words (cons letter word)) words))
        '()
        (keys trie)))))

;;; We need the arity check here because search-and-dissoc changes the
;;; arity of the trie; and by arity, we mean depth. Shit.
(defn every-predicate? [predicates string]
  (and
   ;; Only match exact strings (not substrings).
   (= (count predicates)
      (count string))
   (every? true?
           (map (fn [letter predicate] (predicate letter))
                string
                predicates))))

(defn get-words-with-predicates
  ([trie predicates]
     (get-words-with-predicates trie predicates '() '()))
  ([trie predicates words word]
     ;; (debug trie words word)
     (if (trie-sentinel? trie)
       (let [word (reverse word)]
         (if (every-predicate? predicates word)
           (list word)
           nil))
       (reduce
```

```
        (fn [words letter]
          ;; (debug words letter (trie letter))
          (concat (get-words-with-predicates (trie letter) predicates words (cons letter wor
        '()
        (keys trie)))))

(defn words->frequencies [words]
  (reduce (fn [frequencies word]
            (merge-with
             +
             frequencies
             (let [word (distinct word)]
               (zipmap word (replicate (count word) 1)))))
          {}
          words))

#_ (defn get-words-and-frequencies-with-predicates
  ([trie predicates]
     (get-words-with-predicates trie predicates '() '() {}))
  ([trie predicates words word frequencies]
     ;; (debug trie words word)
     (if (trie-sentinel? trie)
       (let [word (reverse word)]
         (if (every-predicate? predicates word)
           (list word)
           nil))
       (reduce
        (fn [words letter]
          ;; (debug words letter (trie letter))
          (concat (get-words-with-predicates (trie letter) predicates words (cons letter wor
        '()
        (keys trie)))))

(defn get-words-with-predicates-and-max
  ([trie predicates]
     (get-words-with-predicates-and-max trie predicates nil))
  ([trie predicates max]
     (get-words-with-predicates-and-max trie predicates '() '() 0 max))
  ([trie predicates words word n-words max]
     (with-call-cc
       (let-cc
        k
        ;; (debug trie predicates words word n-words max (count words))
        (cond (and max (>= (count words) max)) (k nil)
              (trie-sentinel? trie) (let [word (reverse word)]
                                      (if (every-predicate? predicates word)
```

```
                                    (list word)
                                    nil))
                :else (reduce
                       (fn [words letter]
                         ;; (debug words letter (trie letter))
                         (concat (get-words-with-predicates-and-max
                                  (trie letter)
                                  predicates
                                  words
                                  (cons letter word)
                                  (inc n-words)
                                  max)
                                 words))
                       '()
                       (keys trie)))))))

(defn get-words-with-predicates-and-max-and-continuation
  ([k trie predicates]
     (get-words-with-predicates-and-max-and-continuation k trie predicates nil))
  ([k trie predicates max]
     (get-words-with-predicates-and-max-and-continuation k trie predicates '() '() 0 max))
  ([k trie predicates words word n-words max]
     ;; (debug k trie predicates words word n-words max)
     (cond (and max (>= (count words) max)) (k nil)
           (trie-sentinel? trie)
           (let [word (reverse word)]
             (if (every-predicate? predicates word)
               (list word)
               nil))
           :else (reduce
                  (fn [words letter]
                    ;; (debug words letter (trie letter))
                    (concat (get-words-with-predicates-and-max-and-continuation
                             k
                             (trie letter)
                             predicates
                             words
                             (cons letter word)
                             (inc n-words)
                             max)
                            words))
                  '()
                  (keys trie)))))

(defn get-words-with-depth-and-max-and-continuation
  ([k trie depth]
```

```
    (get-words-with-depth-and-max-and-continuation k trie depth nil))
  ([k trie depth max]
    (get-words-with-depth-and-max-and-continuation k trie depth '() '() 0 max))
  ([k trie depth words word current-depth max]
    ;; (debug k trie depth words word current-depth max)
    (cond (and max (>= (count words) max)) (k nil)
          (trie-sentinel? trie)
          (if (= current-depth depth)
            (list (reverse word))
            nil)
          :else (reduce
                 (fn [words letter]
                   ;; (debug words letter (trie letter))
                   (concat (get-words-with-depth-and-max-and-continuation
                            k
                            (trie letter)
                            depth
                            words
                            (cons letter word)
                            (inc current-depth)
                            max)
                           words))
                 '()
                 (keys trie)))))

#_ (let [word1 '(1 2 3 4 5)
         word2 '(1 2 3 7 8)
         word3 '(9 2 3 7 8)
         trie (reduce merge-tries {} (map plumb (list word1 word2 word3)))]
     (debug
      ;; (assert (= trie {9 {2 {3 {7 {8 trie-sentinel}}}},
      ;;                 1 {2 {3 {7 {8 trie-sentinel},
      ;;                          4 {5 trie-sentinel}}}}}))
      ;; (assert (= (count-trie trie)
      ;;            {9 1, 1 1, 3 2, 8 2, 7 2, 2 2, 5 1, 4 1}))
      ;; (assert (search-trie trie word1))
      ;; (assert (search-trie trie word2))
      ;; (assert (search-trie trie word3))
      ;; (assert (not (search-trie trie '(1 2 3 4 6))))
      ;; (assert (= (search-and-dissoc trie '(1 2 4))
      ;;            {9 {2 {3 {7 {8 trie-sentinel}}}},
      ;;             1 {2 trie-sentinel}}))
      ;; (assert (trie-sentinel? (search-and-dissoc trie '(2))))
      ;; (assert (= (search-and-dissoc trie `(~wildcard 2 4))
      ;;            {9 {2 trie-sentinel}, 1 {2 trie-sentinel}}))
      ;; (assert (=
```

```
;;          (search-and-dissoc-predicate
;;           trie
;;           [wildcard-predicate
;;            (positive-predicate 2)
;;            (positive-predicate 4)])
;;          {1 {2 trie-sentinel}, 9 {2 trie-sentinel}}))
;; (assert (= (get-words trie)
;;            '((5 4 3 2 1)
;;              (8 7 3 2 1)
;;              (8 7 3 2 9))))
;; (assert (= (search-and-dissoc-predicate
;;             trie
;;             [wildcard-predicate
;;              wildcard-predicate
;;              wildcard-predicate
;;              (negative-predicate 4)])
;;            {1 {2 {3 {7 {8 trie-sentinel}}}},
;;             9 {2 {3 {7 {8 trie-sentinel}}}}}))
;; (assert (every-predicate? [(positive-predicate 1)
;;                            wildcard-predicate
;;                            (negative-predicate 2)
;;                            (positive-predicate 4)
;;                            (positive-predicate 5)]
;;                           '(1 2 3 4 5)))
;; (assert (= (get-words-with-predicates
;;             trie
;;             [(positive-predicate 1)
;;              (positive-predicate 2)
;;              (positive-predicate 3)
;;              wildcard-predicate
;;              wildcard-predicate])
;;            '((1 2 3 4 5)
;;              (1 2 3 7 8))))
;; (get-words-with-predicates-and-max
;;   trie
;;   [(positive-predicate 1)
;;    (positive-predicate 2)
;;    (positive-predicate 3)
;;    wildcard-predicate
;;    wildcard-predicate]
;;   3)
;; (get-words-with-predicates-and-max
;;   trie
;;   [wildcard-predicate
;;    wildcard-predicate
;;    wildcard-predicate
```

```
      ;;    wildcard-predicate
      ;;    wildcard-predicate]
      ;;   2)
      ;; (with-call-cc
      ;;   (let-cc
      ;;    k
      ;;    (get-words-with-predicates-and-max-and-continuation
      ;;      k
      ;;      trie
      ;;      [wildcard-predicate
      ;;       wildcard-predicate
      ;;       wildcard-predicate
      ;;       wildcard-predicate
      ;;       wildcard-predicate]
      ;;      2)))
      (search-and-dissoc-predicate-with-depth
        trie
        [
         wildcard-predicate
         wildcard-predicate
         wildcard-predicate
         wildcard-predicate
         wildcard-predicate
         ]
        5)
      ))

;;; Whew: 200k takes about 1.2 seconds (not sure if the results are
;;; correct, though; and we haven't even tackled the question of a
;;; frequency tree (can it be calculated once, at the beginning, or
;;; progressively?)).
;;;
;;; Yeah: I don't think we have the whole tree: distinct reports, out
;;; of 200k, 199998 distincts; and our chain doesn't look all that
;;; deep.
;;;
;;; Interesting: we have all of the 0s, but only the last of 1--25;
;;; oh: that's because we're using (first (keys ...)), isn't it?
;;;
;;; Ok, solved it: 200k words takes 2.2 seconds; Hephaiste!
;;;
;;; Damn: calculating the frequencies is about 500ms on a 200k trie;
;;; let's try precalculating (whether by duplicate trie or
;;; node-record). (Down to about 500ms, by the way, when not using a
;;; sorted map.)
;;;
```

```
;;; Goes from 1.7 to 1.9 seconds to thread frequencies through plumb;
;;; and up again to 2.1 seconds when using a trie-sentinel instead of
;;; nil. 2.9 to take an initial count, by the way. (This is where we
;;; could save some time by doing a sampled count.)
;;;
;;; Takes 1.1 seconds to retrieve 200k words.
#_ (let [words (take 200000 (repeatedly (fn [] (take 7 (repeatedly (fn [] (rand-int 26))))))
         frequencies (transient (vec (replicate 26 0)))
         trie (time (reduce (fn [trie word] (merge-tries trie (plumb word)))
                            {} words))]
     ;; A debug with time?
     (debug
      ;; (time (persistent! (calculate-frequencies trie)))
      ;; (calculate-frequencies trie)
      ;; (time (count-trie trie))
      ;; (trie 0)
      ;; (time (calculate-frequencies trie))
      ;; (time (count-trie trie))
      ;; (count-trie (time (search-and-dissoc trie '(1 2 3 4 5 6 7))))
      (count-trie (time (search-and-dissoc trie `(1 2 ~wildcard 4 5 6 7))))
      (get-words
       (time (search-and-dissoc-predicate
              trie
              [(positive-predicate 1)
               (positive-predicate 2)
               (positive-predicate 3)
               (positive-predicate 4)
               (positive-predicate 5)
               (positive-predicate 6)
               (positive-predicate 7)])))
      ;; (count (time (get-words trie)))
      ))

;;; Should we spend the time downcasing? It's an essential part of our
;;; algorithm; we might need to.
(defn string->word [word]
  (map (fn [letter] (- (int letter) 97)) word))

(defn make-trie-dictionary [file]
  (with-open [input (reader file)]
    (binding [*in* input]
      ;; I would use read-lines here, but I've run into stack
      ;; overflows.
      (loop [word (read-line)
             trie {}]
        (if word
```

```
          (recur (read-line)
                 (merge-tries trie (plumb (string->word word))))
          trie)))))

(defn make-arity->trie-dictionary [file]
  (with-open [input (reader file)]
    (binding [*in* input]
      ;; I would use read-lines here, but I've run into stack
      ;; overflows.
      (loop [word (read-line)
             arity->trie {}]
        ;; (debug word arity->trie)
        (if word
          (recur (read-line)
                 (let [arity (count word)
                       trie (arity->trie arity)]
                   (assoc arity->trie
                     arity
                     (merge-tries trie
                                  (plumb (string->word word))))))
          arity->trie)))))

(defn char->letter [char]
  (- (int char) 97))

(defn string->predicates [string default-predicate]
  (map (fn [char]
         (if (= char HangmanGame/MYSTERY_LETTER)
           default-predicate
           (positive-predicate (char->letter char))))
       string))

(defn letter->char [letter]
  (char (+ 97 letter)))

(defn word->string [word]
  (join (map letter->char word)))

(defn letter-trie->char-trie
  ([letter-trie]
     (letter-trie->char-trie letter-trie {}))
  ([letter-trie char-trie]
     (if (trie-sentinel? letter-trie)
       trie-sentinel
       (reduce
        (fn [trie letter]
```

```
          (assoc trie
            (letter->char letter)
            (letter-trie->char-trie
             (letter-trie letter)
             {})))
        char-trie
        (keys letter-trie)))))

;; (def letter? integer?)

;; (def word? list?)

(defn make-trie-strategy [dictionary arity]
  ;; We have to resort to atoms here because we're using the
  ;; GuessingStrategy interface.
  (let [dictionary (atom dictionary)
        frequencies (atom frequencies)
        last-guess (atom nil)]
    (reify
      GuessingStrategy
      (nextGuess [_ game]
        ;; (debug (count-trie @dictionary)
        ;;        @last-guess)
        (let [remaining-guesses (.numWrongGuessesRemaining game)
              words (with-call-cc
                      (let-cc
                       k
                       (get-words-with-depth-and-max-and-continuation
                         k
                         @dictionary
                         arity
                         (inc remaining-guesses))))
              n-words (count words)]
          (debug n-words remaining-guesses)
          (if (and (pos? n-words)
                   (<= n-words remaining-guesses))
            ;; Distinguish between words by letter-frequency-weighted
            ;; probability?
            ;;
            ;; Words should decrease monotonically in proportion to
            ;; remaining-guesses; such that, if indeed we get here, we
            ;; should never move on. Therefore, let's not bother
            ;; setting last-guess to the word guessed.
            (let [word (nth words (rand-int n-words))]
              (reset! dictionary (search-and-dissoc @dictionary word))
              (debug 'GuessWord (word->string word))
```

```
  (new GuessWord (word->string word)))
(if @last-guess
  (let [guessed-so-far (.toLowerCase (.getGuessedSoFar game))
        predicates (string->predicates guessed-so-far (negative-predicate @last-
    (reset! dictionary (search-and-dissoc-predicate-with-depth @dictionary predi
    (let [letter->frequency (count-trie @dictionary)
          guessed-letters
          (map (fn [letter] (char->letter (Character/toLowerCase letter)))
               (.getAllGuessedLetters game))
          letter->frequency
          (reduce
           (fn [letter->frequency [letter frequency]]
             (if (.contains guessed-letters letter)
               letter->frequency
               (assoc letter->frequency letter frequency)))
           {}
           letter->frequency)
          #_ (filter (fn [letter-frequency]
                       (not (.contains guessed-letters (first letter-frequency))))
                     letter->frequency)
          frequency->letter
          (into (sorted-map-by >) (map-invert letter->frequency))
          next-guess (second (first frequency->letter))]
      ;; (debug letter->frequency
      ;;        guessed-letters
      ;;        frequency->letter)
      ;; (debug @dictionary
      ;;        guessed-letters
      ;;        frequency->letter)
      (debug letter->frequency
             (map-invert letter->frequency)
             (map letter->char guessed-letters)
             (reduce
              (fn [map frequency-letter]
                (assoc map
                  (letter->char (first frequency-letter))
                  (second frequency-letter)))
              (sorted-map)
              (count-trie @dictionary))
             (reduce
              (fn [map frequency-letter]
                (assoc map
                  (first frequency-letter)
                  (letter->char (second frequency-letter))))
              (sorted-map-by >)
              frequency->letter)
```

```
                        @last-guess
                        (letter-trie->char-trie @dictionary)
                        )
                    (reset! last-guess next-guess)
                    (debug 'GuessLetter (letter->char next-guess))
                    (new GuessLetter (letter->char next-guess))))
                (let [letter->frequency (count-trie @dictionary)
                      frequency->letter
                      (into (sorted-map-by >) (map-invert letter->frequency))
                      next-guess (second (first frequency->letter))]
                  (debug
                   (reduce
                    (fn [map frequency-letter]
                      (assoc map
                        (first frequency-letter)
                        (letter->char (second frequency-letter))))
                    (sorted-map-by >)
                    frequency->letter))
                  (debug 'GuessLetter (letter->char next-guess))
                  (reset! last-guess next-guess)
                  (new GuessLetter (letter->char next-guess)))))))))

(defn make-trie-strategy [dictionary words frequencies]
  ;; We have to resort to atoms here because we're using the
  ;; GuessingStrategy interface.
  (let [dictionary (atom dictionary)
        words (atom words)
        frequencies (atom frequencies)
        last-guess (atom nil)]
    (reify
      GuessingStrategy
      (nextGuess [_ game]
        ;; (debug (count-trie @dictionary)
        ;;        @last-guess)
        (let [remaining-guesses (.numWrongGuessesRemaining game)
              n-words (count words)]
          (debug n-words remaining-guesses)
          (if (and (pos? n-words)
                   (<= n-words remaining-guesses))
            ;; Distinguish between words by letter-frequency-weighted
            ;; probability?
            ;;
            ;; Words should decrease monotonically in proportion to
            ;; remaining-guesses; such that, if indeed we get here, we
            ;; should never move on. Therefore, let's not bother
            ;; setting last-guess to the word guessed.
```

```
(let [word (nth words (rand-int n-words))]
  (reset! dictionary (search-and-dissoc @dictionary word))
  (reset! words (remove #(= % word) words))
  (reset! frequencies (words->frequencies words))
  (debug 'GuessWord (word->string word))
  (new GuessWord (word->string word)))
(if @last-guess
  (let [guessed-so-far (.toLowerCase (.getGuessedSoFar game))
        predicates (string->predicates guessed-so-far (negative-predicate @last-
    (reset! dictionary (search-and-dissoc-predicate @dictionary predicates))
    (reset! words (get-words @dictionary))
    (reset! frequencies (words->frequencies words))
    (let [guessed-letters
          (map (fn [letter] (char->letter (Character/toLowerCase letter)))
               (.getAllGuessedLetters game))
          letter->frequency
          (reduce
           (fn [letter->frequency [letter frequency]]
             (if (.contains guessed-letters letter)
               letter->frequency
               (assoc letter->frequency letter frequency)))
           {}
           frequencies)
          #_ (filter (fn [letter-frequency]
                       (not (.contains guessed-letters (first letter-frequency))))
                     letter->frequency)
          frequency->letter
          (into (sorted-map-by >) (map-invert letter->frequency))
          next-guess (second (first frequency->letter))]
      ;; (debug letter->frequency
      ;;        guessed-letters
      ;;        frequency->letter)
      ;; (debug @dictionary
      ;;        guessed-letters
      ;;        frequency->letter)
      (debug letter->frequency
             (map-invert letter->frequency)
             (map letter->char guessed-letters)
             (reduce
              (fn [map frequency-letter]
                (assoc map
                  (letter->char (first frequency-letter))
                  (second frequency-letter)))
              (sorted-map)
              (count-trie @dictionary))
             (reduce
```

```
                            (fn [map frequency-letter]
                              (assoc map
                                (first frequency-letter)
                                (letter->char (second frequency-letter))))
                            (sorted-map-by >)
                            frequency->letter)
                           @last-guess
                           (letter-trie->char-trie @dictionary)
                           )
                      (reset! last-guess next-guess)
                      (debug 'GuessLetter (letter->char next-guess))
                      (new GuessLetter (letter->char next-guess))))
                  (let [frequency->letter
                        (into (sorted-map-by >) (map-invert @frequencies))
                        next-guess (second (first frequency->letter))]
                    (debug
                     (reduce
                      (fn [map frequency-letter]
                        (assoc map
                          (first frequency-letter)
                          (letter->char (second frequency-letter))))
                      (sorted-map-by >)
                      frequency->letter))
                    (debug 'GuessLetter (letter->char next-guess))
                    (reset! last-guess next-guess)
                    (new GuessLetter (letter->char next-guess)))))))))

(defmacro time-and-value [expr]
  `(let [start# (. System (nanoTime))
         ret# ~expr]
     [(/ (double (- (. System (nanoTime)) start#)) 1000000.0)
      ret#]))

(let [arity->trie (make-arity->trie-dictionary "words.txt")
      arity->frequencies (reduce (fn [arity->frequencies [arity trie]]
                                   (assoc arity->frequencies arity (count-trie trie)))
                                 {}
                                 arity->trie)
      arity->words (reduce (fn [arity->words [arity trie]]
                             (assoc arity->words arity (get-words trie)))
                           {}
                           arity->trie)
      words [
             ;; "aardvark"
             ;; "aardvark"
             ;; "aardvark"
```

```
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "aardvark"
           ;; "comaker"
           ;; "cumulative"
           ;; "eruptive"
           ;; "factual"
           ;; "monadism"
           ;; "mus"
           ;; "nagging"
           "oses"
           ;; "remembered"
           ;; "spodumenes"
           ;; "stereoisomers"
           ;; "toxics"
           ;; "trichromats"
           ;; "triose"
           ;; "uniformed"
           ]
    ]
#_ (debug
    ;; (time (count-trie trie))
    ;; (time (count (get-words trie)))
    ;; (time (count
    ;;        (get-words-with-predicates-and-max
    ;;          trie
    ;;          [
    ;;           wildcard-predicate
    ;;           wildcard-predicate
    ;;           wildcard-predicate
    ;;           ]
    ;;          2)))
```

```
    ;; (time (count
    ;;        (with-call-cc
    ;;          (let-cc
    ;;           k
    ;;           (get-words-with-predicates-and-max-and-continuation
    ;;             k
    ;;             trie
    ;;             [
    ;;              wildcard-predicate
    ;;              wildcard-predicate
    ;;              wildcard-predicate
    ;;              ]
    ;;             2)))))
    ;; (time (count
    ;;        (get-words-with-predicates
    ;;          trie
    ;;          [
    ;;           wildcard-predicate
    ;;           wildcard-predicate
    ;;           wildcard-predicate
    ;;           ])))
    ;; (time (search-trie trie [0 0 7]))
    ;; ((trie 0) 0)
    ;; trie
    )
  (debug
   (map (fn [word]
          (let [arity (count word)]
            (cons word
                  (time-and-value
                   (run (new HangmanGame word 4)
                        (make-trie-strategy
                         (arity->trie arity)
                         (arity->words arity)
                         (arity->frequencies arity)))))))
        words)))
```

We're going to have a huge space penalty for using tries, by the way, that you don't have with strings, apparently; there's this whole node-object that comes along with sequences.

Should we start thinking about an accompanying frequency tree? Or maybe just a record-payload with each node? Two trees require synchronization and replicates the edge-overhead (records, the record-overhead).

One thing to think about is that we only need the max frequency; can we just search the tree seven or eight times? Let's get some times on tree-search. The argument about max frequency is really just an argument about space-

time tradeoff. As we saw previously, keeping a 26-ary vector on each node was prohibitively much space.

Can we defer frequency counting until the merge; or, hell, can we do it at plumb; and come up with some trivial mechanism for updating the counts at merge? We still have to find a mechanism for propagating the counts upward; or does it make sense to propagate downward? Downward would be the most natural but least useful, I think.

How would upward propagation work; and can we associate an integer with the letter (which represents its count from the point down), or do we need to have the full 26-ary vector? Also: can we just maintain a priority queue (heap) of frequencies? I'd like to be able to cheaply modify the frequencies on delete, though.

(The weird thing is, by the way, that we're doing a lot of integer casting according to `jvisualvm`; there's no way to declare `map-of` in the same way as we do `vector-of`, I imagine.)

If we don't maintain a 26-ary vector of frequencies for each node, we're going to have to plumb the tree at the point of deletion, anyway. But maintaining a 26-ary vector for each node is prohibitively expensive, isn't it?

Merely the act of creating of creating a 26-ary vector for each node brings the time for counting to 7000 ms from 500 ms for a 200k list.

Can we at least maintain a frequency table with the tree which can be modified by counting the subtree to be eliminated? Maybe we can even build this table when merging. That way, we only have to count the tree at most twice, I believe: once at creation and each subtree at deletion.

Even visiting a 200k tree without doing any work requires 300 msecs, by the way. (Can we improve that?)

Check out `merge-with`, by the way; we could have used it for `merge-trie`, but let's look at it for a quick calculations of frequencies with `+`.

It looks like we're on track for separating the frequencies from the table itself, and doing things like `(merge-with - frequencies deleted-frequencies)`.

Still need search and delete; can delete, for instance, identify the offending subtrie to prune so that we can pass it do `(merge-with - ...)`?

Ouch: we haven't tried matching paths with words; we could either gather the word on the way down or attach it to the leaves. For search we need some kind of wildcard signification: nil? Then we can't recurse on e.g. `(if letter ...)`; can we explicitly test for the end of vector? Index-based traversal, for instance.

We're going to have to implement a search-and-dissoc that returns, I think, frequency-table of the dissociated subtrie (possibly empty).

Just a thought: are earlier letters more valuable? It seems like the closer the letter is to the root, the more words it determines; even though this seems anti-intuitive. No, I reject this: there would be as many leaves as a filter-criterion as one root (out of twenty six); prefix letters are more efficient, though, as a filter-mechanism: we have to search for leaves.

Would, in that sense, that we had two views of the trie that were inversions of each other: depending on a letter's proximity to the prefix or suffix we could

search from either end.

(Getting excited about this solution: Götter.)

The operations are in place (insert, search, delete, frequencies, leaves, etc.); but we're missing out on the main benefit of the trie if we have to do these all sequentially: i.e. search-and-dissoc should be able to update the frequency table (resort to mutation or return multiple values?); get-words should abandon the search after hitting a certain n; we can save a little time by storing the words as sentinels so we don't have to `cons` on the way down; we should resort to probabilistic sampling of the letter-frequencies.

We're also going to have to prune the tree if we guess a word incorrectly; we can probably prune up from the word until the first point the tree diverges, can't we? That involves some kind of backtracking: `amb`?

Should we use `call-cc` from `monads` to bail out of `count-trie` when `count` is high (for some definition of high)?

Creating, deterministically counting and leaving the dictionary from words.txt take respectively:

```
"Elapsed time: 1952.26851 msecs"
"Elapsed time: 1393.619088 msecs"
"Elapsed time: 656.135302 msecs"
```

Maybe deterministically counting isn't such a big deal if we get it over with during initialization.

`search-and-dissoc` is a positive filtering device; but we need a negative device, too: `search-and-dissoc` should really dispatch on a vector, say, of predicates. This flexibility comes at a cost: function-application is probably more expensive than mere char-comparison.

Here's an unintended consequence: `search-and-dissoc` actually changes the dictionary by adding terminal substrings of disassociated strings; we need to `search-and-dissoc` on a predicate-vector. `string->predicates` will take a default predicate which will be `wildcard-predicate` in the case of `get-words` (which also needs to take a max `n`).

At some point we need to figure out whether propagating the effect of filtering upward on deterministically precalculated frequencies at O(2 * n) is really more efficient than some sort of probabilistic sampling.

This is an optimization, though; let's get the mechanics working first.

Note: there are some embarrassingly parallel aspects of `search-and-dissoc`: each branch is orthogonal and can be parallelized, I think.

Let's construct `run` in such a way that doesn't require mutation of state: let's run the strategies as a reduction, receiving as input its own output; and the specification of some initial state.

The substring problem is very dire: we can't really associate key with a trie-sentinel as well as a subtrie; we can punt, of course, by dealing with tries of a certain arity. Alternatively, we can have trie-sentinels pointing to nil as keys.

Let's punt and restrict our search to tries of a certain arity; why have sentinels at all, then? It's convenient to be able to distinguish between termina and

missing keys. Don't forget, however, that search-and-dissoc changes the arity of the trie; we still need the length checks in `every-predicate?`, therefore. (Shit, this is getting subtle; at some point: too subtle for us? It's a wager.)

By "arity", we sometimes mean "depth;" unfortunately. Better clear this up: trie-arity should probably refer to width; term-arity (i.e. dictionary-term) is actually trie-depth.

While we appear to be a little faster than naïve regexen, we're losing words that we shouldn't ("toxics", "oses").

I'm seeing lots of trie-sentinel artifacts where the key should be removed instead (see 'oses'); I'm guessing `search-and-dissoc-predicate` is the culprit: let's reëxamine the base-case.

Also: what the hell is it doing for so long on things like 'aardvark'? Let's visualize. It's spends a lot of time, predictably, in `merge-tries`; `trie-sentinel?` overwhelmingly, also, which kind of sucks; `count-trie` makes an appearance. Let's probabalize `count-trie`; what about `trie-sentinel?`: we don't have fixed-depth tries because of `search-and-dissoc`: how do we distinguish absent keys from bona fide words?

One of the invariants probably needs to be a depth-consistent trie; otherwise, we get a lot of artifacts. How to purge depth-inconsistent branches? It seems like, at some point, a branch should have an unambiguous terminus which is depth-inconsistent. No: search-and-dissoc should be depth-aware; instead of capping a depth-inconsistent branch, we should delete it. Should we delete it, anyway? Sentinels are for depth-appropriate termini; search-and-dissoc won't create new depth-appropriate terminin. All the depth-appropriate termini should have been created at make-trie-dictionary.

Calculate frequencies at initialization; propagate frequencies to trie-strategy; update them at `get-words`; we can have a `get-words-until-stable-frequencies`, if necessary.

The only way to get frequencies from the trie is to prune non-viable branches (e.g. branches capped at sub-arity).

If we could mutate the trie, I could propagate up and modify dirty branches.

Trie search yields: leaves; another, for those leaves: frequencies.

```
(use
 'clojure.contrib.trace
 'hangman.core
 '[clojure.contrib.io :only (reader)]
 'net.dnolen.clj-cont
 '[clojure.set :only (difference map-invert)]
 '[clojure.string :only (join)]
 )

(import '(com.factual.hangman
          GuessingStrategy
          GuessLetter
          GuessWord
```

```
          HangmanGame)
        'java.lang.Character)

(def trie-sentinel (gensym))

(defn trie-sentinel? [object]
  (= object trie-sentinel))

(defn plumb
  ([word] (plumb {} word))
  ([trie word]
     (if word
       (let [letter (first word)]
         (assoc (trie letter {})
           letter
           (plumb trie
                  (next word))))
       trie-sentinel)))

(defn letter->char [letter]
  (char (+ 97 letter)))

(defn word->string [word]
  (join (map letter->char word)))

(defn words->strings [words]
  (map word->string words))

(defn letter->frequency-->char->frequency [letter->frequency]
  (reduce (fn [char->frequency [letter frequency]]
            (assoc char->frequency
              (letter->char letter)
              frequency))
          (sorted-map)
          letter->frequency))

(defn letter-trie->char-trie
  ([letter-trie]
     (letter-trie->char-trie letter-trie {}))
  ([letter-trie char-trie]
     (if (trie-sentinel? letter-trie)
       trie-sentinel
       (reduce
        (fn [trie letter]
          (assoc trie
            (letter->char letter)
```

```
            (letter-trie->char-trie
             (letter-trie letter)
             {})))
         char-trie
         (keys letter-trie)))))

;;; We have to do all this current-depth business because we don't
;;; have a properly pruned tree; we either have to do the work
;;; up-front during search-and-dissoc, or we pay for it later.
;;;
;;; Let's think about what a proper search-and-dissoc might look like.
#_ (defn count-trie
  ([trie depth debug?] (count-trie trie depth {} 0 debug?))
  ([trie depth frequencies current-depth debug?]
     (if debug?
       (debug (letter-trie->char-trie trie)
              depth
              (letter->frequency-->char->frequency frequencies)
              current-depth))
     (if (trie-sentinel? trie)
       (do
        (debug 'HARRRRRRRO
               trie
               depth
               current-depth
              (letter->frequency-->char->frequency frequencies))
        (if (= depth current-depth)
          frequencies
          {}))
       (merge-with
        +
        frequencies
        (reduce
         (fn [frequencies key]
           (merge-with
            frequencies
            (count-trie (trie key) depth frequencies (inc current-depth) debug?)))
         (let [keys (keys trie)]
           (zipmap
            keys
            (replicate (count keys) 1)))
         (keys trie))))))

(defn count-trie
  ([trie] (count-trie {} trie))
  ([frequencies trie]
```

```
    (if (not (trie-sentinel? trie))
      (merge-with
       +
       frequencies
       (reduce
        (fn [frequencies key]
          (merge-with
           frequencies
           (count-trie frequencies (trie key))))
        (let [keys (keys trie)]
          (zipmap
           keys
           (replicate (count keys) 1)))
        (keys trie)))
      frequencies)))

(defn merge-tries [mergens mergendum]
  (cond (or (trie-sentinel? mergens) (empty? mergens)) mergendum
        (or (trie-sentinel? mergendum) (empty? mergendum)) mergens
        :else (let [mergendum-key (first (keys mergendum))
                    submergens (mergens mergendum-key)
                    submergendum (mergendum mergendum-key)]
                (if (mergendum mergendum-key)
                  (assoc mergens mergendum-key
                         (merge-tries submergens submergendum))
                  (assoc mergens mergendum-key
                         (mergendum mergendum-key))))))

(def wildcard-predicate (constantly true))
(defn positive-predicate [predicans]
  (fn [predicandum] (= predicans predicandum)))
(defn negative-predicate [predicans]
  (fn [predicandum] (not (= predicans predicandum))))

;;; Can we somehow propagate the entire stack, and only assoc when we
;;; have a terminal branch satisfying the predicates? It seems like
;;; this could be accomplished somehow by reducing on the root and
;;; performing a merge-tries.
;;;
;;; If we could crack this, it would solve (I think) the difficulties
;;; we're having with count-trie and get-words w.r.t. arity
;;; (depth). Call the new function 'prune', for instance: it's simply
;;; a leaf-finding exercise with predicates that merges with root;
;;; similar to building up the dictionary to begin with. You give up
;;; the efficencies of partial-tries, though.
;;;
```

```
;;; Partial tries: I wish we could make it work.
(defn search-and-dissoc [trie predicates]
  (if (or (empty? predicates)
          (trie-sentinel? trie))
    trie
    (let [predicate (first predicates)
          letters (filter predicate (keys trie))]
      (if (empty? letters)
        trie-sentinel
        (reduce
         (fn [subtrie letter]
           (assoc subtrie
             letter
             (search-and-dissoc (trie letter) (next predicates))))
         {}
         letters)))))

(defn every-predicate? [predicates string]
  (and
   ;; Only match exact strings (not substrings).
   (= (count predicates)
      (count string))
   (every? true?
           (map (fn [letter predicate] (predicate letter))
                string
                predicates))))

;;; 'Predicates' is somewhat ad-hoc; could do this filtering, for
;;; instance, in 'reduce'. How do we perform this while avoiding
;;; creating the words and iterating through their list?
;;;
;;; Distinguish between outer and inner reduction? We're trying to
;;; force a linear narrative here.
(defn reduce-words [subreduce init trie predicates]
  (letfn [(reduce-words [k reductum trie word]
            (debug 'REDUCE-WORDS reductum trie word)
            (if (trie-sentinel? trie)
              (let [word (reverse word)]
                (debug
                 'TRIE-SENTINEL
                 ;; reductum
                 ;; trie
                 word
                 )
                (if (every-predicate? predicates word)
                  (list word)
```

```
                    ;; Can we avoid propagating this to reduce? This
                    ;; signifies no word.
                    nil))
                (reduce
                 (fn [reductum letter]
                   ;; This is getting applied for each depth of the
                   ;; recursion; really only want to subreduce on words,
                   ;; don't we?
                   #_ (debug 'SUPER-REDUCE
                           trie
                           (reduce-words reductum (trie letter) (cons letter word))
                           reductum
                           letter)
                   ;; (subreduce reductum (reduce-words reductum (trie letter) (cons letter wo
                   (subreduce (reduce-words k reductum (trie letter) (cons letter word)) reduc
                   ;; (reduce-words reductum (trie letter) (cons letter word))
                   )
                 init
                 (keys trie))))]
    (with-call-cc
      (let-cc
       k
       (reduce-words k init trie '())))))

(defn map-trie
  ([map trie predicates]
     (map-trie map trie predicates '() '()))
  ([map trie predicates words word]
     ;; (debug trie words word)
     (if (trie-sentinel? trie)
       (let [word (reverse word)]
         (if (every-predicate? predicates word)
           (list (map word))
           nil))
       (reduce
        (fn [words letter]
          ;; (debug words letter (trie letter))
          (concat (map-trie map (trie letter) predicates words (cons letter word)) words))
        '()
        (keys trie)))))

;;; As long as we're plumbing the depths of the trie: can't we find
;;; suitable leaves, calculate frequencies and merge with root
;;; simultaneously? They're all based on a leaf-finding exercise.
(defn prune [trie predicates]
  (reduce merge-tries {} (map-trie plumb trie predicates)))
```

```
;;; Building the tree from scratch is redoing a lot of work; we should
;;; scrap this mechanism, I think.
;;;
;;; Or: store the dictionary as a list of words; build the trie and
;;; count the frequencies as necessary; apply the predicates (which
;;; are less general than regular expressions). So, we need a:
;;; filter-dictionary; dictionary->trie; count-trie
;;; (probabilistically).
(let [word1 '(1 2 3 4 5)
      word2 '(1 2 3 7 8)
      word3 '(9 2 3 7 8)
      trie (reduce merge-tries {} (map plumb (list word1 word2 word3)))]
  (debug '_____________________________________________________)
  (debug trie
         (count-trie
          (prune trie
                 [wildcard-predicate
                  wildcard-predicate
                  wildcard-predicate
                  (positive-predicate 7)
                  wildcard-predicate]))
         (map-trie
          (fn [word]
            {:word word
             :trie (plumb word)
             :frequencies ()}))))

(defn get-words-with-predicates
  ([trie predicates]
     (get-words-with-predicates trie predicates '() '()))
  ([trie predicates words word]
     ;; (debug trie words word)
     (if (trie-sentinel? trie)
       (let [word (reverse word)]
         (if (every-predicate? predicates word)
           (list word)
           nil))
       (reduce
        (fn [words letter]
          ;; (debug words letter (trie letter))
          (concat (get-words-with-predicates (trie letter) predicates words (cons letter wor
        '()
        (keys trie)))))

(defn get-words
```

```
  ([trie depth] (get-words trie depth '() '() 0))
  ([trie depth words word current-depth]
     ;; (debug trie depth words word current-depth)
     (if (trie-sentinel? trie)
       (if (= current-depth depth)
         (list (reverse word))
         nil)
       (reduce
        (fn [words letter]
          (concat (get-words (trie letter) depth words (cons letter word) (inc current-depth
        '()
        (keys trie)))))

(defn string->word [string]
  (map char->letter string))

(defn make-arity->trie-dictionary [file]
  (with-open [input (reader file)]
    (binding [*in* input]
      ;; I would use read-lines here, but I've run into stack
      ;; overflows.
      (loop [word (read-line)
             arity->trie {}]
        ;; (debug word arity->trie)
        (if word
          (recur (read-line)
                 (let [arity (count word)
                       trie (arity->trie arity)]
                   (assoc arity->trie
                     arity
                     (merge-tries trie
                                  (plumb (string->word word))))))
          arity->trie)))))

(defn char->letter [char]
  (- (int char) 97))

(defn string->predicates [string default-predicate]
  (map (fn [char]
         (if (= char HangmanGame/MYSTERY_LETTER)
           default-predicate
           (positive-predicate (char->letter char))))
       string))

(defmacro time-and-value [expr]
  `(let [start# (. System (nanoTime))
```

```
         ret# ~expr]
     [(/ (double (- (. System (nanoTime)) start#)) 1000000.0)
      ret#]))

(defn make-trie-strategy [arity dictionary words letter->frequency]
  (let [dictionary (atom dictionary)
        words (atom words)
        letter->frequency (atom letter->frequency)
        last-guess (atom nil)]
    (reify
      GuessingStrategy
      (nextGuess [_ game]
        (let [remaining-guesses (.numWrongGuessesRemaining game)
              n-words (count @words)]
          ;; When would n-words be zero? If a word came up not in the
          ;; dictionary, I suppose.
          (if (and (pos? n-words)
                   (<= n-words remaining-guesses))
            (let [word (nth @words (rand-int n-words))]
              (reset! words (remove #(= % word) @words))
              (debug (word->string word))
              (new GuessWord (word->string word)))
            (if @last-guess
              (let [guessed-so-far (.toLowerCase (.getGuessedSoFar game))
                    predicates (string->predicates guessed-so-far (negative-predicate @last-
                (reset! dictionary (search-and-dissoc @dictionary predicates))
                (reset! words (get-words @dictionary arity))
                (reset! letter->frequency (count-trie @dictionary arity true))
                (debug '!!!!!!!!!!!!!!!!!!!
                       (letter->frequency-->char->frequency @letter->frequency)
                       (words->strings @words))
                (let [guessed-letters
                      (map (fn [letter] (char->letter (Character/toLowerCase letter)))
                           (.getAllGuessedLetters game))
                      letter->frequency
                      (reduce
                       (fn [letter->frequency [letter frequency]]
                         (if (.contains guessed-letters letter)
                           letter->frequency
                           (assoc letter->frequency letter frequency)))
                       {}
                       @letter->frequency)
                      frequency->letter
                      (into (sorted-map-by >) (map-invert letter->frequency))
                      next-guess (second (first frequency->letter))]
                  #_ (debug letter->frequency
```

```
                            (map-invert letter->frequency)
                            (map letter->char guessed-letters)
                            (reduce
                             (fn [map frequency-letter]
                               (assoc map
                                 (letter->char (first frequency-letter))
                                 (second frequency-letter)))
                             (sorted-map)
                             (count-trie @dictionary arity true))
                            (reduce
                             (fn [map frequency-letter]
                               (assoc map
                                 (first frequency-letter)
                                 (letter->char (second frequency-letter))))
                             (sorted-map-by >)
                             frequency->letter)
                            (letter->char @last-guess)
                            (letter-trie->char-trie @dictionary)
                            )
                       (reset! last-guess next-guess)
                       (debug 'GuessLetter (letter->char next-guess))
                       (new GuessLetter (letter->char next-guess))))
                  (let [frequency->letter
                        (into (sorted-map-by >) (map-invert @letter->frequency))
                        next-guess (second (first frequency->letter))]
                    (debug
                     (reduce
                      (fn [map frequency-letter]
                        (assoc map
                          (first frequency-letter)
                          (letter->char (second frequency-letter))))
                      (sorted-map-by >)
                      frequency->letter))
                      (debug 'GuessLetter (letter->char next-guess))
                    (reset! last-guess next-guess)
                    (new GuessLetter (letter->char next-guess)))))))))))

#_ (let [arity->trie (make-arity->trie-dictionary "words-head.txt")
      ;; It takes roughly 2 seconds to traverse a fresh trie: at the
      ;; very least, would could couple these two activities.
      arity->words
      (reduce (fn [arity->words [arity trie]]
                (assoc arity->words arity (get-words trie arity)))
              {}
              arity->trie)
      arity->letter->frequency
```

```
      (reduce (fn [arity->letter->frequency [arity trie]]
                (assoc arity->letter->frequency arity (count-trie trie arity false)))
              {}
              arity->trie)
      words [
             ;; "oses"
             ;; "aal"
             "abas"
             ]]
  #_ (debug arity->trie
         arity->words
         arity->letter->frequency)
  (debug
   (map (fn [word]
          (let [arity (count word)]
            (cons word
                  (time-and-value
                   (run (new HangmanGame word 4)
                        (make-trie-strategy
                         arity
                         (arity->trie arity)
                         (arity->words arity)
                         (arity->letter->frequency arity)))))))
        words)))

(use
 'hangman.core
 '[clojure.contrib.generic.functor :only (fmap)])

(defn string->word [string]
  (map char->letter string))

(def trie-sentinel (gensym))

(defn trie-sentinel? [object]
  (= object trie-sentinel))

(defn word->trie [trie word]
  (if word
    (let [letter (first word)]
      (assoc trie
        letter
        (word->trie (trie letter {}) (next word))))
    trie-sentinel))

(defn dictionary->trie [dictionary]
```

```
  (reduce #(word->trie %1 %2) {} dictionary))

;;; Single dispatch; arity-invariant: branch on nil instead of
;;; trie-sentinel?
(defn count-trie
  ([trie] (count-trie {} trie))
  ([frequencies trie]
     (if (trie-sentinel? trie)
       frequencies
       (merge-with
        +
        frequencies
        (reduce
         (fn [frequencies key]
           (merge
            frequencies
            (count-trie frequencies (trie key))))
         (let [keys (keys trie)]
           (zipmap
            keys
            (replicate (count keys) 1)))
         (keys trie))))))

(def wildcard-predicate (constantly true))

(defn positive-predicate [predicans]
  (fn [predicandum] (= predicans predicandum)))

(defn negative-predicate [predicans]
  (fn [predicandum] (not (= predicans predicandum))))

(defn every-predicate? [predicates word]
  (every? true?
          (map (fn [letter predicate] (predicate letter))
               word
               predicates)))

(defn filter-dictionary [predicates dictionary]
  (filter (partial every-predicate? predicates) dictionary))

(defn make-arity->dictionary [file]
  (with-open [input (reader file)]
    (binding [*in* input]
      (loop [string (read-line)
             arity->dictionary {}]
        (if string
```

```
          (recur (read-line)
                 (let [arity (count string)
                       dictionary (arity->dictionary arity '())]
                   (assoc arity->dictionary
                     arity
                     (cons (string->word string)
                           dictionary))))
          arity->dictionary)))))

#_ (let [word1 '(1 2 3 4 5)
      word2 '(1 2 3 7 8)
      word3 '(9 2 3 7 8)
      dictionary [word1 word2 word3]
      trie (dictionary->trie dictionary)]
  (debug (assert (= trie
                    {9 {2 {3 {7 {8 trie-sentinel}}}},
                     1 {2 {3 {7 {8 trie-sentinel}, 4 {5 trie-sentinel}}}}}))
         ;; (assert (= (count-trie trie)
         ;;            {9 1, 1 1, 3 2, 8 2, 7 2, 2 2, 5 1, 4 1}))
         (count-trie trie)
         (assert (= (filter-dictionary [wildcard-predicate
                                        wildcard-predicate
                                        wildcard-predicate
                                        (positive-predicate 7)
                                        wildcard-predicate]
                                       dictionary)
                    '((1 2 3 7 8) (9 2 3 7 8))))))

(let [arity->dictionary (time (make-arity->dictionary "words.txt"))
      ;; arity->trie (time (into {} (for [[arity dictionary] arity->dictionary]
      ;;                              [arity (dictionary->trie dictionary)])))
      arity->trie (time (fmap dictionary->trie arity->dictionary))
      arity->count (time (fmap count-trie arity->trie))])
```

Clean up the trie:

```
(use
 ;; 'clojure.contrib.trace
 ;; 'clojure.contrib.singleton
 ;; '[hangman.core :only (debug)]
 'hangman.core
 '[clojure.contrib.io :only (reader)]
 ;; 'clojure.contrib.monads
 'net.dnolen.clj-cont
 ;; 'delimc.core
 '[clojure.set :only (difference map-invert)]
```

```
 '[clojure.string :only (join)]
 '[clojure.contrib.generic.functor :only (fmap)] )

(import '(com.factual.hangman
          GuessingStrategy
          GuessLetter
          GuessWord
          HangmanGame)
        'java.lang.Character)

(load-file "make-frequency-strategy.clj")

(def trie-sentinel (gensym))

(defn trie-sentinel? [object]
  (= object trie-sentinel))

(defn plumb
  ([word] (plumb {} word))
  ([trie word]
     (if word
       (let [letter (first word)]
         (assoc (trie letter {})
           letter
           (plumb trie
                  (next word))))
       trie-sentinel)))

(defn count-trie
  ([trie] (count-trie {} trie))
  ([frequencies trie]
     (if (not (trie-sentinel? trie))
       (merge-with
        +
        frequencies
        (reduce
         (fn [frequencies key]
           (merge-with
            frequencies
            (count-trie frequencies (trie key))))
         (let [keys (keys trie)]
           (zipmap
            keys
            (replicate (count keys) 1)))
         (keys trie)))
       frequencies)))
```

```
;;; Really what we need to do is a recursive merge of some sort: the
;;; problem is, we have to account for all keys; including ones which
;;; are currently different and future divergence, too. Ouch.
;;;
;;; No, that's not the case (although I wonder if we could dynamically
;;; program this to avoid exponential complexity): the problem is a
;;; multi-key mergens with a single-key mergendum. Therefore, the
;;; first-key hack should abide (, dude).
(defn merge-tries [mergens mergendum]
  (cond (or (trie-sentinel? mergens) (empty? mergens)) mergendum
        (or (trie-sentinel? mergendum) (empty? mergendum)) mergens
        :else (let [mergendum-key (first (keys mergendum))
                    submergens (mergens mergendum-key)
                    submergendum (mergendum mergendum-key)]
                (if (mergendum mergendum-key)
                  (assoc mergens mergendum-key
                         (merge-tries submergens submergendum))
                  (assoc mergens mergendum-key
                         (mergendum mergendum-key))))))

(def wildcard-predicate (constantly true))

(defn positive-predicate [predicans]
  (fn [predicandum] (= predicans predicandum)))

(defn negative-predicate [predicans]
  (fn [predicandum] (not (= predicans predicandum))))

(defn dissoc-trie [trie word]
  (if word
    (let [letter (first word)
          subtrie (trie letter)]
      (if (trie-sentinel? subtrie)
        trie-sentinel
        (assoc trie letter (dissoc-trie subtrie (next word)))))
    trie-sentinel))

(defn search-and-dissoc [trie predicates]
  (if (or (empty? predicates)
          (trie-sentinel? trie))
    trie
    (let [predicate (first predicates)
          letters (filter predicate (keys trie))]
      (if (empty? letters)
        trie-sentinel
```

```
        (reduce
         (fn [subtrie letter]
           (assoc subtrie
             letter
             (search-and-dissoc (trie letter) (next predicates))))
         {}
         letters)))))

(defn get-words
  ([arity trie] (get-words arity trie '() '()))
  ([arity trie words word]
     (if (trie-sentinel? trie)
       (if (= (count word) arity)
         (list (reverse word))
         nil)
       (reduce
        (fn [words letter]
          (concat (get-words arity (trie letter) words (cons letter word)) words))
        '()
        (keys trie)))))

;;; Should we spend the time downcasing? It's an essential part of our
;;; algorithm; we might need to.
(defn string->word [word]
  (map (fn [letter] (- (int letter) 97)) word))

(defn make-arity->trie-dictionary [file]
  (with-open [input (reader file)]
    (binding [*in* input]
      (loop [word (read-line)
             arity->trie {}]
        (if word
          (recur (read-line)
                 (let [arity (count word)
                       trie (arity->trie arity)]
                   (assoc arity->trie
                     arity
                     (merge-tries trie
                                  (plumb (string->word word))))))
          arity->trie)))))

(defn char->letter [char]
  (- (int char) 97))

(defn string->predicates [string default-predicate]
  (map (fn [char]
```

```
         (if (= char HangmanGame/MYSTERY_LETTER)
           default-predicate
           (positive-predicate (char->letter char))))
       string))

(defn letter->char [letter]
  (char (+ 97 letter)))

(defn word->string [word]
  (join (map letter->char word)))

(defn make-trie-strategy [arity initial-dictionary initial-letter->count]
  (make-frequency-strategy
   (fn [dictionary guessed-so-far last-guess]
     (let [predicates (string->predicates guessed-so-far (negative-predicate last-guess))]
       (search-and-dissoc dictionary predicates)))
   count-trie
   (fn [word dictionary]
     (dissoc-trie dictionary word))
   char->letter
   letter->char
   word->string
   (partial get-words arity)
   initial-dictionary
   initial-letter->count))

(defmacro time-and-value [expr]
  `(let [start# (. System (nanoTime))
         ret# ~expr]
     [(/ (double (- (. System (nanoTime)) start#)) 1000000.0)
      ret#]))

(let [arity->trie (time (make-arity->trie-dictionary "words.txt"))
      arity->letter->count (time (fmap count-trie arity->trie))
      words [
             ;; "aardvark"
             ;; "aardvark"
             ;; "aardvark"
             ;; "aardvark"
             ;; "aardvark"
             ;; "aardvark"
             ;; "aardvark"
             ;; "aardvark"
             ;; "aardvark"
             ;; "aardvark"
             ;; "aardvark"
```

```
            ;; "aardvark"
            ;; "aardvark"
            ;; "aardvark"
            ;; "aardvark"
            ;; "aardvark"
            ;; "aardvark"
            ;; "aardvark"
            ;; "aardvark"
            ;; "aardvark"
            "comaker"
            "cumulative"
            "eruptive"
            "factual"
            "monadism"
            "mus"
            "nagging"
            "oses"
            "remembered"
            "spodumenes"
            "stereoisomers"
            "toxics"
            "trichromats"
            "triose"
            "uniformed"
            ]
    ]
#_ (debug
    ;; (time (count-trie trie))
    ;; (time (count (get-words trie)))
    ;; (time (count
    ;;        (get-words-with-predicates-and-max
    ;;          trie
    ;;          [
    ;;           wildcard-predicate
    ;;           wildcard-predicate
    ;;           wildcard-predicate
    ;;           ]
    ;;          2)))
    ;; (time (count
    ;;        (with-call-cc
    ;;          (let-cc
    ;;           k
    ;;           (get-words-with-predicates-and-max-and-continuation
    ;;             k
    ;;             trie
    ;;             [
```

```
      ;;              wildcard-predicate
      ;;              wildcard-predicate
      ;;              wildcard-predicate
      ;;              ]
      ;;             2)))))
      ;; (time (count
      ;;        (get-words-with-predicates
      ;;          trie
      ;;          [
      ;;           wildcard-predicate
      ;;           wildcard-predicate
      ;;           wildcard-predicate
      ;;           ])))
      ;; (time (search-trie trie [0 0 7]))
      ;; ((trie 0) 0)
      ;; trie
      )
  (debug
   (map (fn [word]
          (cons word
                (time-and-value
                 (let [arity (count word)]
                   (run (new HangmanGame word 4)
                        (make-trie-strategy
                         arity
                         (arity->trie arity)
                         (arity->letter->count arity)))))))
        words)))
```

Forget the trie-stuff: use predicates?

```
(use
 '[hangman.core :only (run debug)]
 '[clojure.contrib.generic.functor :only (fmap)]
 '[clojure.contrib.math :only (abs)]
 '[clojure.set :only (map-invert)]
 '[clojure.string :only (join)]
 '[clojure.contrib.io :only (reader)]
 '[clojure.pprint :only (pprint)])

(import '(com.factual.hangman
          GuessingStrategy
          GuessLetter
          GuessWord
          HangmanGame)
        'java.lang.Character)
```

```
(defn char->letter [char]
  (- (int char) 97))

(defn string->word [string]
  (map char->letter string))

(defn letter->char [letter]
  (char (+ 97 letter)))

(defn word->string [word]
  (join (map letter->char word)))

(defn words->strings [words]
  (map word->string words))

(defn letter->count-->char->count [letter->count]
  (reduce (fn [char->count [letter count]]
            (assoc char->count
              (letter->char letter)
              count))
          (sorted-map)
          letter->count))

(def wildcard-predicate (constantly true))

(defn positive-predicate [predicans]
  (fn [predicandum] (= predicans predicandum)))

(defn negative-predicate [predicans]
  (fn [predicandum] (not (= predicans predicandum))))

(defn string->predicates [string default-predicate]
  (map (fn [char]
         (if (= char HangmanGame/MYSTERY_LETTER)
           default-predicate
           (positive-predicate (char->letter char))))
       string))

(defn every-predicate? [predicates word]
  (every? true?
          (map (fn [letter predicate] (predicate letter))
               word
               predicates)))

(defn filter-dictionary [dictionary guessed-so-far last-guess]
```

```
  (let [predicates (string->predicates guessed-so-far (negative-predicate last-guess))]
    (filter (partial every-predicate? predicates) dictionary)))

(defn make-arity->dictionary [file]
  (with-open [input (reader file)]
    (binding [*in* input]
      (loop [string (read-line)
             arity->dictionary {}]
        (if string
          (recur (read-line)
                 (let [arity (count string)
                       ;; Using a set instead of a list here to take
                       ;; advantage of ‘disj’. Never mind: filter
                       ;; reconverts to lazy seq.
                       dictionary (arity->dictionary arity '())]
                   (assoc arity->dictionary
                     arity
                     (cons (string->word string)
                           dictionary))))
          arity->dictionary)))))

(defn letter->count-->letter->percentage [letter->count]
  (let [total (apply + (vals letter->count))]
    (fmap #(float (/ % total)) letter->count)))

(def *delta-percentage-tolerance* 1.0e-5)

(def *sampling-frequency* 100)

(def *ratio-to-n-of-minimum-iterations* 100)

;;; Should we do some running average to establish stability? Also,
;;; should last-letter->count be from *sampling-frequency* iterations
;;; or one iteration ago?
(defn sufficiently-stable? [last-letter->count letter->count]
  (let [last-letter->percentage (letter->count-->letter->percentage last-letter->count)
        letter->percentage (letter->count-->letter->percentage letter->count)
        delta-letter->percentage
        (merge-with - letter->percentage last-letter->percentage)]
    (< (apply + (vals delta-letter->percentage))
       *delta-percentage-tolerance*)))

(defn count-letters [dictionary]
  (let [minimum-iterations
        (/ (count dictionary) *ratio-to-n-of-minimum-iterations*)]
    (loop [last-letter->count {}
```

```
           letter->count {}
           dictionary (shuffle dictionary)
           iteration 0]
      (if dictionary
        (if (and (> iteration minimum-iterations)
                 (zero? (mod iteration *sampling-frequency*))
                 (sufficiently-stable? last-letter->count letter->count))
          letter->count
          (let [word (first dictionary)]
            (recur letter->count
                   (merge-with
                    +
                    letter->count
                    (zipmap word (replicate (count word) 1)))
                   (next dictionary)
                   (inc iteration))))
        letter->count))))

(defn remove-word [word dictionary]
  (remove #(= % word) dictionary))

(defn make-sampling-strategy [initial-dictionary initial-letter->count]
  (make-frequency-strategy
   filter-dictionary
   count-letters
   remove-word
   char->letter
   letter->char
   word->string
   identity
   initial-dictionary
   initial-letter->count))

(defmacro time-and-value [expr]
  `(let [start# (. System (nanoTime))
         ret# ~expr]
     [(/ (double (- (. System (nanoTime)) start#)) 1000000.0)
      ret#]))

(def words
  '("comaker"
    "cumulative"
    "eruptive"
    "factual"
    "monadism"
    "mus"
```

```
    "nagging"
    "oses"
    "remembered"
    "spodumenes"
    "stereoisomers"
    "toxics"
    "trichromats"
    "triose"
    "uniformed"))

(def reference-scores
  '(25 9 5 9 8 25 7 5 5 4 2 11 5 5 5))

;;; From the initial implementation.
(def naive-regex-scores
  '(25 8 7 9 10 25 7 5 5 4 3 10 5 7 13))

(def naive-regex-times
  '(914.201401
    624.136794
    457.645472
    1164.972
    854.34475
    98.380223
    950.42491
    132.590102
    387.955044
    403.945935
    246.297455
    548.420537
    603.323457
    276.769557
    640.156031))

(def deterministic-count-letter-scores
  '(11 6 8 25 8 25 5 4 6 6 3 9 4 7 9))

(def deterministic-count-letter-times
  '(269.192986
    227.544676
    73.002197
    256.03295
    287.633228
    27.553267
    264.223448
    55.23997
```

```
    66.713347
    92.050017
    49.081876
    194.52453
    142.528282
    83.375216
    211.739498))

(defn average-vals [data key]
  (let [vals (map key (vals data))]
    (float (/ (reduce + vals) (count vals)))))

(pprint
 (let [arity->dictionary (make-arity->dictionary "words.txt")
       arity->count (fmap count-letters arity->dictionary)]
   (let [time-scores
         (map #(time-and-value
                (let [arity (count %)]
                  (run (new HangmanGame % 4)
                       (make-sampling-strategy
                        (arity->dictionary arity)
                        (arity->count arity)))))
              words)
         data
         (zipmap words (map (fn [initial-time
                                 reference-score
                                 [time score]]
                              (let [delta-time (- time initial-time)
                                    delta-score (- score reference-score)]
                                (let [percent-change-time (* 100 (abs (/ delta-time initial-
                                      percent-change-score (* 100 (abs (float (/ delta-score
                                  {:time time
                                   :delta-time delta-time
                                   :percent-change-time percent-change-time
                                   :score score
                                   :delta-score delta-score
                                   :percent-change-score percent-change-score})))
                            deterministic-count-letter-times
                            reference-scores
                            time-scores))
         average-time (average-vals data :time)
         average-score (average-vals data :score)
         average-delta-time (average-vals data :delta-time)
         average-delta-score (average-vals data :delta-score)
         average-percent-change-time (average-vals data :percent-change-time)
         average-percent-change-score (average-vals data :percent-change-score)]
```

```
     {:data (into (sorted-map) data)
      :average-time average-time
      :average-delta-time average-delta-time
      :average-percent-change-time average-percent-change-time
      :average-score average-score
      :average-delta-score average-delta-score
      :average-percent-change-score average-percent-change-score})))
```

7.4 CANCELED Probabilistic sampling for frequency?

CLOSED: *2011-09-02 Fri 17:35*

Can we get a probabilistic idea of what the frequencies are by sampling the tree? Doing an entire sample seems excessive; how do we establish a reasonable N: the point at which (if ever) the relative rations of letters seem to stabilize? Cf. spreading activation: it simply sums over individual deltas and stops when a certain threshold, epsilon, has been reached.

7.5 DONE Trie from dictionary

CLOSED: *2011-09-02 Fri 17:35*

We could trie every word and do a merge-reduction on the root to avoid mutation; expensive? Also, I'd like to keep track of node-histography.

Crafting this

```
(use '[clojure.contrib.io :only (reader)])

;;; We're not going to be able to transient this, because all subtries
;;; will have to be transient and persisted, too; unless we do a
;;; nested persistence.
(defn make-trie-dictionary [file]
  (with-open [input (reader file)]
    (binding [*in* input]
      (let [trie (transient {})]
        (loop [word (read-line)]
          (if word
            (do

              (recur
               (read-line)))
            (persistent! trie)))))))

(make-trie-dictionary "words.txt")
```

8 DONE We are assuming that the dictionary is lower-case.

CLOSED: *2011-09-02 Fri 17:35*

9 DONE 'run' should return the score and total time.

CLOSED: *2011-08-22 Mon 20:13*

- CLOSING NOTE *2011-08-22 Mon 20:13*
  The 'time-not-value' macro; cf. 'time'.

Are we going to have to parse the output of `time`, somehow? Or, just coöpt the time macro.

```
(defmacro time-not-value
  [expr]
  `(let [start# (. System (nanoTime))]
     ~expr
     (/ (double (- (. System (nanoTime)) start#)) 1000000.0)))
```

10 DONE Maps

CLOSED: *2011-08-22 Mon 14:47*

> A Map is a collection that maps keys to values. Two different map types are provided - hashed and sorted. Hash maps require keys that correctly support hashCode and equals. Sorted maps require keys that implement Comparable, or an instance of Comparator. Hash maps provide faster access (log32N hops) vs (logN hops), but sorted maps are, well, sorted. count is O(1). conj expects another (possibly single entry) map as the item, and returns a new map which is the old map plus the entries from the new, which may overwrite entries of the old. conj also accepts a MapEntry or a vector of two items (key and value). seq returns a sequence of map entries, which are key/value pairs. Sorted map also supports rseq, which returns the entries in reverse order. Maps implement IFn, for invoke() of one argument (a key) with an optional second argument (a default value), i.e. maps are functions of their keys. nil keys and values are ok.
>
> Related functions
>
> **Create a new map** hash-map sorted-map sorted-map-by

'Change' a map assoc dissoc select-keys merge merge-with zipmap

Examine a map get contains? find keys vals map?

Examine a map entry key val

11 DONE Regex then trie

CLOSED: *2011-08-22 Mon 14:48*

Let's work with a regex solution that treats the words holistically (i.e. as strings); if we can beat regex with some sort of trie: great.

12 DONE Does the sorted-map-by comparator have access to the values as well?

CLOSED: *2011-08-19 Fri 02:41*

- CLOSING NOTE *2011-08-19 Fri 02:41*
  Apparently not

```
(assert (= "3 2"
           (with-out-str
             (sorted-map-by
              (fn [x y] (print (format "%s %s" x y)) true) 2 \a 3 \b))))
```

13 DONE How do maps behave with key-collision?

CLOSED: *2011-08-19 Fri 02:41*

- CLOSING NOTE *2011-08-19 Fri 02:41*
  Replacement

```
(assert (= {2 4} (assoc (assoc (hash-map) 2 3) 2 4)))
(assert (= {2 4} (assoc (assoc {} 2 3) 2 4)))
```

14 DONE Extract some game mechanics into hangman.core.

CLOSED: *2011-08-18 Thu 03:56*

15 DONE Regex-based GuessingStrategy

CLOSED: *2011-08-22 Mon 14:47*

(New pseudo-code syntax, by the way: parenthesized phrases are potential optizimations; bracketed phrases are commentary. Also: TODOs are full sentences with full stop.)

```
- Create dictionary.
  - Read file and create a list of words.
    - (Order the list by arity, so that pre-filtering is more
      efficient.)
- Initialize filter-strategy on HangmanGame and dictionary.
  - Copy the dictionary.
    - [We're going to have state, unfortunately, since we don't
      control the game loop; or can we use continuations, =yield= or
      coroutines?
  - (Pre-filter dictionary by letter-arity.)
- Next guess
  - Convert =game.getGuessedSoFar= into a suitable regex.
    - Create negative regex: =String.format("[^%s]", new
      String(game.getIncorrectlyGuessedLetters()))=
    - Substitution on =game.getGuessedSoFar= of
      =HangmanGame.MYSTERY_LETTER= for negative regex.
    - Prepend =^=, append =$=.
  - Destructively filter dictionary on regex.
  - If solution is unambiguous, guess word.
    - (Alternatively, some sort of score based on number of possibly
      solutions and number of remaining guesses.)
    - Otherwise, create a sorted map of frequency -> letter (counting
      one letter per word); guess one random sample from the maximally
      frequent letters.
```

```
(use '[clojure.contrib.io :only (reader)]
     '[clojure.contrib.string :only (replace-str)]
     'hangman.core
     '[clojure.set :only (difference map-invert)]
     '[clojure.contrib.math :only (abs)]
     '[clojure.pprint :only (pprint)]
     'clojure.contrib.profile
     '[clojure.contrib.generic.functor :only (fmap)])

(import '(com.factual.hangman
          GuessingStrategy
          GuessLetter
          GuessWord
          HangmanGame
```

```
          HangmanGame$Status)
        'java.lang.Character)

(load-file "make-frequency-strategy.clj")

;;; Possible optimization: sort by arity.
(defn make-dictionary [dictionary-file]
  (with-open [dictionary-input (reader dictionary-file)]
    (binding [*in* dictionary-input]
      ;; I would use read-lines here, but I've run into stack
      ;; overflows.
      (loop [word (read-line) words '()]
        (if word
          (recur (read-line) (cons word words))
          words)))))

(defn make-arity->dictionary [dictionary-file]
  (with-open [dictionary-input (reader dictionary-file)]
    (binding [*in* dictionary-input]
      (loop [word (read-line)
             arity->dictionary {}]
        (if word
          (let [arity (count word)]
            (recur (read-line)
                   (assoc arity->dictionary arity (cons word (arity->dictionary arity '()))))
          arity->dictionary)))))

(defn random-letter []
  (let [letter (+ (rand-int 26) 97)]
    (char letter)))

(def *delta-percentage-tolerance* 1.0e-5)

(def *sampling-frequency* 100)

(def *ratio-to-n-of-minimum-iterations* 100)

;;; Should we do some running average to establish stability? Also,
;;; should last-letter->count be from *sampling-frequency* iterations
;;; or one iteration ago?
(defn sufficiently-stable? [last-letter->count letter->count]
  (let [last-letter->percentage (letter->count-->letter->percentage last-letter->count)
        letter->percentage (letter->count-->letter->percentage letter->count)
        delta-letter->percentage
        (merge-with - letter->percentage last-letter->percentage)]
    (< (apply + (vals delta-letter->percentage))
```

```
       *delta-percentage-tolerance*)))

(defn deterministic-count-letters [dictionary]
  (loop [dictionary dictionary
         letter->frequency (hash-map)]
    (if dictionary
      (let [word (first dictionary)
            ;; 'distinct' was here.
            letters word]
        ;; Also think about zipmap here.
        (recur (next dictionary)
               (reduce (fn [letter->frequency letter]
                         (assoc letter->frequency
                           letter
                           (+ (letter->frequency letter 0) 1)))
                       letter->frequency
                       letters)))
      letter->frequency)))

(defn sampling-count-letters [dictionary]
  (let [minimum-iterations
        (/ (count dictionary) *ratio-to-n-of-minimum-iterations*)]
    (loop [last-letter->count {}
           letter->count {}
           dictionary (shuffle dictionary)
           iteration 0]
      (if dictionary
        (if (and (> iteration minimum-iterations)
                 (zero? (mod iteration *sampling-frequency*))
                 (sufficiently-stable? last-letter->count letter->count))
          letter->count
          (let [word (first dictionary)]
            (recur letter->count
                   (merge-with
                    +
                    letter->count
                    (zipmap word (replicate (count word) 1)))
                   (next dictionary)
                   (inc iteration))))
        letter->count))))

(defn negative-regex [last-guess]
  (format "[^%s]" last-guess))

(defn filter-dictionary [dictionary guessed-so-far last-guess]
  (let [negative-regex (negative-regex last-guess)
```

```
        filtering-regex (format "^%s$" (replace-str "-" negative-regex guessed-so-far))
        filtering-pattern (re-pattern filtering-regex)]
    (filter #(re-seq filtering-pattern %) dictionary)))

(def char->letter identity)

(def letter->char identity)

(def word->string identity)

(def get-words identity)

(defn remove-word [word dictionary]
  (remove #(= % word) dictionary))

(defn make-regex-strategy
  [count-letters initial-dictionary initial-letter->count]
  (make-frequency-strategy
   filter-dictionary
   count-letters
   remove-word
   char->letter
   letter->char
   word->string
   get-words
   initial-dictionary
   initial-letter->count))

(def make-deterministic-regex-strategy
  (partial make-regex-strategy deterministic-count-letters))

(def make-sampling-regex-strategy
  (partial make-regex-strategy sampling-count-letters))

;;; Modify this to return time and value!
(defmacro time-and-value [expr]
  `(let [start# (. System (nanoTime))
         ret# ~expr]
     [(/ (double (- (. System (nanoTime)) start#)) 1000000.0)
      ret#]))

(def words
  '("comaker"
    "cumulative"
    "eruptive"
    "factual"
```

```
    "monadism"
    "mus"
    "nagging"
    "oses"
    "remembered"
    "spodumenes"
    "stereoisomers"
    "toxics"
    "trichromats"
    "triose"
    "uniformed"))

(def reference-scores
  '(25 9 5 9 8 25 7 5 5 4 2 11 5 5 5))

;;; From the initial implementation.
(def initial-scores
  '(25 8 7 9 10 25 7 5 5 4 3 10 5 7 13))

(def initial-times
  '(914.201401
    624.136794
    457.645472
    1164.972
    854.34475
    98.380223
    950.42491
    132.590102
    387.955044
    403.945935
    246.297455
    548.420537
    603.323457
    276.769557
    640.156031))

(defn average-vals [data key]
  (let [vals (map key (vals data))]
    (float (/ (reduce + vals) (count vals)))))

(debug
 (let [arity->dictionary (make-arity->dictionary "words.txt")
       arity->count (fmap letter->frequency arity->dictionary)]
   (let [time-scores
         (map #(time-and-value
                (let [arity (count %)]
```

```
          (run (new HangmanGame % 4)
               (make-sampling-regex-strategy
                (arity->dictionary arity)
                (arity->count arity)))))
     words)
    data
    (zipmap words (map (fn [initial-time
                            reference-score
                            [time score]]
                         (let [delta-time (- time initial-time)
                               delta-score (- score reference-score)]
                           (let [percent-change-time (/ delta-time initial-time)
                                 percent-change-score (float (/ delta-score reference-s
                             {:time time
                              :delta-time delta-time
                              :percent-change-time percent-change-time
                              :score score
                              :delta-score delta-score
                              :percent-change-score percent-change-score})))
                       initial-times reference-scores time-scores))
    average-time (average-vals data :time)
    average-score (average-vals data :score)
    average-delta-time (average-vals data :delta-time)
    average-delta-score (average-vals data :delta-score)
    average-percent-change-time (average-vals data :percent-change-time)
    average-percent-change-score (average-vals data :percent-change-score)]
{:data (into (sorted-map) data)
 :average-time average-time
 :average-delta-time average-delta-time
 :average-percent-change-time average-percent-change-time
 :average-score average-score
 :average-delta-score average-delta-score
 :average-percent-change-score average-percent-change-score}))
```

At some point, letter guesses don't give me any more information; unless I do a diff on the words left. When words are fewer than spaces, start guessing words?

With manual profiling, `merge` is taking a shit-load of time; can we do something with `frequencies`? Can we fuck around with `transient` and `assoc!`?

(# is a removal macro like `#;`, by the way.)

`jvisualvm` reveals that `clojure.lang.Util.hash()` is eating most of the processing time; thoughts for optimizing this?

We have about 1.3M `Objects`, and 0.9M `TreeMap$Entries`; not to mention 0.5M `Strings`, 0.4M `Conses` and 0.16M `TreeMaps`. Hmm. Are these sets orthogonal?

It looks like – oh, shit: `distinct` is taking a lot of time; followed by: `hash`,

`re-seq`, `filter`, `seq`, `re-matcher`, etc. So we do have a lot of regular expression time.

Jesus: taking out `distinct` saves almost %50; it causes certain words to fail, on the other hand.

Here are the results from removing `distinct` (time in milliseconds):

```
{:average-delta-time -283.85242,
 :average-percent-change-time -0.4640968,
 :average-delta-score 1.4666667,
 :average-percent-change-score 0.296532}
```

With distinct, there is only a nominal (10%) difference in `:average-percent-change-score`:

```
{:average-delta-time -79.898224,
 :average-percent-change-time -0.111338414,
 :average-delta-score 0.8666667,
 :average-percent-change-score 0.196532}
```

And with this latest change `re-seq` is now the biggest consumer (behind `swank.util.concurrent.mbox$receive.invoke`; is that an artifact of using swank, however?).

Wow, by implementing the `arity->dictionary` preprocessing, I was able to get an average 70% efficiency:

```
{:average-delta-time -353.4292,
 :average-percent-change-time -0.6696512,
 :average-delta-score 1.4666667,
 :average-percent-change-score 0.296532}
```

Too bad the score still sucks; besides `seq` now, hash still consumes the most time: some kind of object comparison? Is clojure.core/hash the same as Java's `hashCode`? Possibly.

We might want to try in-place filtering, since `seq` gets called a lot; we should do some manual profiling at this point, possibly, though, to determine where the lion's share of time is taken up.

I want to give tries a try, though.

One potential optimization is to do an initial letter-frequency histogram and merely subtract the words that get filtered; maybe you're quibbling over linear (i.e. micro) optizimations now, though.

1. Randomly sample the frequency space until we have stable frequencies;
2.

15.1 DONE Is it slower to access char-based hash-tables than int-based vectors?

`CLOSED:` *2011-08-24 Wed 16:53*

- CLOSING NOTE *2011-08-24 Wed 16:54*
  Vectors are an order of magnitude faster for large n; crossover is roughly 1000000: some of that may have been due to garbage collection, though. The int-based vector is only marginally more efficient than the char->int-based vector, though: 210 ms. vs. 250 ms. for n = 1000000. There's probably some overhead associated with transienting a vector, etc.

  One advantage of using tries is that it can be totally vector-based with ints. (Don't even bother with arrays, by the way; vide infra.)

```
;;; Use doall, etc. instead of loop to force evaluation? Otherwise, we
;;; could just use (take (repeatedly #(rand-int 26))), etc.
(let [n 1000000
      chars (loop [chars '()
                   n n]
              (if (zero? n)
                chars
                (recur (cons (char (+ (rand-int 26) 97))
                             chars)
                       (dec n))))
      ints (loop [ints '()
                  n n]
             (if (zero? n)
               ints
               (recur (cons (rand-int 26) ints)
                      (dec n))))]
  ;; Char-based map
  (time
   (reduce (fn [histogram char]
             (assoc histogram char (+ (histogram char 0) 1)))
           {}
           chars))

  ;; Char->int-based immutable vector
  (time
   (reduce (fn [vector char]
             (let [i (- (int char) 97)]
               (assoc vector i (+ (vector i) 1))))
           (vec (replicate 26 0))
           chars))

  ;; Char->int-based mutable vector
```

```
(time
 (let [vector (transient (vec (replicate 26 0)))]
   (doseq [char chars]
     (let [i (- (int char) 97)]
       (assoc! vector i (+ (vector i) 1))))
   (persistent! vector)))

;; Int-based mutable vector
(time
 (let [vector (transient (vec (replicate 26 0)))]
   (doseq [int ints]
     (assoc! vector int (+ (vector int) 1)))
   (persistent! vector)))

;; Really fucking slow
(time
 (let [array (into-array Integer/TYPE (replicate 26 0))]
   (doseq [int ints]
     (aset-int array int (+ (aget array int) 1)))
   array)))

;; "Elapsed time: 897.958722 msecs"
;; "Elapsed time: 223.132804 msecs"
;; "Elapsed time: 245.723347 msecs"
;; "Elapsed time: 207.313954 msecs"
;; "Elapsed time: 13016.683057 msecs"
```

16 DONE Separate java and clojure dirs under src?

CLOSED: *2011-08-18 Thu 01:59*

Could accomplish this with `:source-path` and `:java-source-path`.

17 DONE Strings as seqs

CLOSED: *2011-08-17 Wed 23:51*

```
;;; Treating strings as seqs of characters.
(assert
 (= (with-out-str
      (doseq [c "abc"] (println c)))
    "a\nb\nc\n"))

;;; Convert a string into a seq.
(assert (= (seq "abc") '(\a \b \c)))
```

18 DONE Implement a trivial GuessingStrategy.

CLOSED: *2011-08-17 Wed 23:52*

```
(import '(com.factual.hangman
          GuessingStrategy
          GuessLetter
          GuessWord
          HangmanGame
          HangmanGame$Status))

(defn can-keep-guessing?
  ([game] (= (.gameStatus game) HangmanGame$Status/KEEP_GUESSING)))

;;; We should abstract some of this into hangman.core.
(defn run
  ([game strategy]
     ;; Can also just wrap this in with-open, since
     ;; assertCanKeepGuessing (in guessLetter and guessWord) will
     ;; throw an exception.
     (while (can-keep-guessing? game)
       (println game)
       (let [guess (.nextGuess strategy game)]
         (.makeGuess guess game)))
     (println game)
     (.currentScore game)))

(defn random-letter []
  (let [letter (+ (rand-int 26) 97)]
    (char letter)))

(defn make-random-strategy []
  (reify
    GuessingStrategy
    (nextGuess [_ game] (new GuessLetter (random-letter)))))

(let [game (new HangmanGame "factual" 4)]
  (assert (= (.currentScore game) 0))
  (assert (= (.gameStatus game) HangmanGame$Status/KEEP_GUESSING))
  (run game (make-random-strategy)))
```

19 DONE Read in words, filter on regex.

CLOSED: *2011-08-17 Wed 16:34*

```
;;; Roughly 50 msecs.
(time
 (let [words
       (binding [*in* (reader "words.txt")]
         (loop [word (read-line) words '()]
           (if word
             (recur (read-line) (cons word words))
             words)))]
   (filter (partial re-find #"^[^a][^a][e]$") words)))
```

20 DONE Read in list of words.

CLOSED: *2011-08-15 Mon 21:22*

```
(use 'clojure.contrib.io)

;;; This is about twice as slow: it has to parse the line; it only
;;; gathers 53754 words, furthermore, whereas `wc words.txt' gives
;;; 173528.
(time
 (with-in-reader
   "words.txt"
   (loop [word (read *in* false nil) words '()]
     (if word
       (recur (read *in* false nil) (cons word words))
       (count words)))))

;;; This gives 173529 (including the blank word).
(time
 (with-open [dictionary (reader "words.txt")]
   (binding [*in* dictionary]
     (loop [word (read-line) words '()]
       (if word
         (recur (read-line) (cons word words))
         (count words))))))

;;; This is slightly slower than `loop'; gives the right result,
;;; though.
(time (count (reduce (fn [x y] (cons y x)) '() (read-lines "words.txt"))))
```

21 DONE Regex

CLOSED: *2011-08-15 Mon 19:51*

```
(use 'clojure.contrib.str-utils)
```

```
(re-find #"[a-z]+" "harro")
(re-gsub #"h" "x" "harro")
```

22 DONE Clojure

CLOSED: *2011-08-17 Wed 23:52*

On namespaces; let's relegate factual's hangman to src/com/factual/hangman, or should we create a separate repo containing a jar?

Vectors vs. lists in namespaces (fuckers). Main from maven, btw.; specifying in MANIFEST.MF, too.

On perusing the API:

`defrecord` like SRFI-9; `comment` like `#;`; but cf `defstruct`; `delay`; `deliver`; `disj` like `delete`; also, `dissoc` for maps; `doall` for side effects; `doc` reads doc strings, also `^{doc: ...}`? cf. `dorun` vs. `doall`; `doseq` appears to be the analogy for `for-each`, though; `doall` and `dorun` force the lazy evaluation of a sequence, apparently; `dorun` ignores the side-effects on the sequence; cf. `unfold`? `doto`: `(doto (new java.util.HashMap) (.put "a" 1) (.put "b" 2))`; `do` like `begin`; `file-seq`; `fnext` like `cadr`; `for`: lazy list-comprehension; `force`; `format` strings (c-style); `gen-class`; `gen-interface` (requires compilation?); `gensym`, cool; `get`; `get-in`: "nested associative structure": alist? `hash-map`; `identity`; `if-let`; `if-not`; `inc`; `interleave` like `zip`; `intern`; `interpose`; `into` like `append`? `into-array`; `iterate`: x, (f x), (f (f x)); `juxt` (ad-hoc, anyone?); `(keep f coll)`: ad-hoc identity filter on non-nil; `lazy-cat`; `lazy-seq`; `letfn`; `list*`; `macroexpand`; `make-hierarchy` relates to `derive`, `isa?`; `map-indexed` (sometimes useful); `memfn`: java method as first-class `fn` (this is cool: `(map (memfn toUpperCase) ["a" "short" "message"])`); `memoize`; `merge` for maps; `meta` for metadata (`^{ ... }`?); `next` like `cdr`; `neg?` instead of `negative?` [analogical learning]; `nfirst` instead of `cdar`; `nnext` instead of `cddr`, etc.; `not-any?`; `not-every?`; `nth` instead of `list-ref`; `nthnext` for pairs? `partition`; `pmap` for parallel mapping; `pr` like `write` (`write` is intended for `read`, too, I think); `pr-str`; `print`, `println`: "human consumption"; `print-str`, `println-str`; `printf` (cf. `format`); `prn`: `pr` and `newline`; `pvalues`: parallel; `reductions`: intermediate values of reductions, too; `reify`: instantiate objects? `remove` also like `delete`? lazy; `repeat`; `repeatedly`: like `tabulate`; `replicate` like `make-list`; `reset!` like `set!`? `rseq`: reverse `seq`; `rsubseq`; `send` for agents; `seq`: seq on the collection; `seque`: queued seq; `sequence`: distinct from `seq`; `set`: distinct elements; `shuffle`; `some`: `(some #{:fred} coll)` (oh, yeah: sets can be used as predicates somehow to test for membership, right?); `sort`; `sorted-set`; `str`: to string; `subseq`; `subvec`; `take`; `take-last`; `trampoline`: mutual recursion without stack consumption; `with-bindings` and `with-bindings*` vs. `binding` (former two appear to be "low level")?

23 DONE Leiningen

`CLOSED:` *2011-08-17 Wed 23:52*

Pretty good intro; workflow:

- you create a project (`lein new`), define dependencies on external libraries and download them with `lein deps` command (you need to run it after each change of dependencies);
- you write your code, periodically running `lein compile`, `lein test`, and may be using `lein repl`, `lein swank` or `lein nailgun` (depending on your personal preferences) for interactive development;
- if you develop a library, that you plan to use in other projects, then you can install it into Maven's local repository with `lein install` command, or you can upload it to Clojars (with scp, as suggested in documentation, or by using the lein-clojars plugin);
- if you develop a program for end-user, then you can pack your code into package with `lein jar` command (only your code, without dependencies), or with `lein uberjar`, with all dependencies included into package it's much easier to distribute such packages to end users.

24 CANCELED Oh, shit: and don't forget about parallelization.

`CLOSED:` *2011-09-02 Fri 17:35*

The regex-based, filtering guessing-strategy is susceptible to parallelization; isn't it? See this, though:

> About pfilter, it should work but run slower than filter, since your predicate is simple. Parallelization isn't free so you have to give each thread moderately intensive tasks to offset the multithreading overhead. Batch your items before filtering them.

25 CANCELED Check TODO.pdf into the repository as documentation?

`CLOSED:` *2011-09-02 Fri 17:36*

It goes against checking derivative things in; maybe we can generate it for releases. Here's a half-ass discussion; 'tis true, though, that PDF is a PS-derivative and, as such, is not wholly binary.

We can .gitignore `TODO.pdf`, such that we only check it in "when we really mean it."

26 CANCELED call-cc

CLOSED: *2011-08-22 Mon 19:34*

Unlike the scheme example (which depends heavily on mutation) runs to the very end of the list.

```
(use 'clojure.contrib.monads)
(run-cont (call-cc (fn [k] (k 2))))
(defn generate-one-element-at-a-time [list]
  (defn control-state [return]
    (doseq [element list]
      (def return
        (call-cc
         (fn [resume-here]
           (def control-state resume-here)
           (return element)))))
    (return 'fuck-this-shit))
  (defn generator []
    (call-cc control-state))
  generator)

(def generate-digit (generate-one-element-at-a-time '(0 1 2)))
(run-cont (generate-digit))
```

See call/cc patterns; same here:

```
(use 'clojure.contrib.monads)

(defn generate-one-element-at-a-time [list]
  (def control-state
    (atom
     (fn [return]
       (let [return (atom return)]
         (doseq [element list]
           (swap!
            return
            (fn [return]
              (call-cc
               (fn [resume-here]
                 (swap! control-state
                        (fn [control-state] resume-here))
                 (return element)))))))
       (return 'fuck-this-shit))))
  (defn generator []
    (call-cc @control-state))
  generator)
```

```
(def generate-digit (generate-one-element-at-a-time '(0 1 2)))
(run-cont (generate-digit))
```

We need a different pattern (see continuation monads in Haskell) that doesn't rely on mutating state variables?

From CPS in Haskell:

```
(use 'clojure.contrib.monads
     'clojure.contrib.math)

((fn [n] (run-cont (call-cc (fn [k] (k (* n n)))))) 2)

;; (def continuation nil)

;; (defn incrementer [x]
;;   (fn [k]
;;     (loop [x (+ x 1)]
;;       (def continuation k)
;;       (recur x))))

;; (def i (incrementer 1))
;; (run-cont i)

(def continuation nil)

(run-cont
 (domonad cont-m
          [x (m-result 1)
           y (call-cc (fn [c] (def continuation c) (c 2)))]
          (+ x y)))

(run-cont (continuation 5))
(run-cont (continuation 42))
(run-cont (continuation -1))
```

```
;; [LISTOF X] -> ( -> X u 'you-fell-off-the-end)
(define (generate-one-element-at-a-time lst)

  ;; Hand the next item from a-list to "return" or an end-of-list marker
  (define (control-state return)
    (for-each
     (lambda (element)
       (set! return (call-with-current-continuation
                     (lambda (resume-here)
                       ;; Grab the current continuation
                       (set! control-state resume-here)
                       (return element)))))
```

```
     lst)
    (return 'you-fell-off-the-end))

  ;; (-> X u 'you-fell-off-the-end)
  ;; This is the actual generator, producing one item from a-list at a time
  (define (generator)
    (call-with-current-continuation control-state))

  ;; Return the generator
  generator)

(define generate-digit
  (generate-one-element-at-a-time '(0 1 2 3 4 5 6 7 8 9)))

(generate-digit)
(generate-digit)
```

27 CANCELED Continuations

`CLOSED:` *2011-08-18 Thu 01:53*

Someone did delimited continuations.

```
(use 'delimc.core)
(assert (= (+ 1 (reset (+ 2 3))) 6))
(assert (= (+ 1 (reset (+ 2 (shift k 3)))) 4))
(assert (= (+ 1 (reset (+ 2 (shift k (+ 3 (k 4)))))) 10))
```

Given the pre-defined interface, though, I'm not sure how we'd fold it in: a global continuation, of course, which is mutated by `reset!`. Is that principally different than mutating the dictionary state of the `GuessingStrategy`?

28 CANCELED Graph

`CLOSED:` *2011-08-17 Wed 23:53*

- CLOSING NOTE *2011-08-17 Wed 23:53*
  We're not going to go with quasi-Markov-chains now; regex, then optimization with tries, if necessary.

Graph based on input words, where the weight is the number of times a letter follows another (or precedes?) (this is Markov, isn't it?); and an frequency table of letters to seed the guesses.

Then we do a Dijkstra weighted–shortest-path and greedy optimization?

At some point we need to do a calculation, though, don't we: given that we have so many guesses left, etc., what's the cost of guessing a word vs. guessing a letter?

We have neither secretWord nor maxWrongGuesses, though.

Going with the Dijkstra analogy, we remove edges for wrong guesses; each edge represents a potential digram. (Could this be improved with n-grams, etc.?)

1. Get the game;
2. determine how many letters;
3. take appropriate words;
4. build a frequency table for initial guesses (randomizing and eliminating ties?);
5. build a weighted digram graph;
6. guess correctly: prune the graph (all paths at index {a1, . . . , an} converge on a) (question is: do we distinguish between letter e.g. 'a' at positions e.g. 1 and 3? If we permit cycles, the links have lower quality; how do we reference index i in the graph, though? We do have some starting nodes and perhaps even a forest (shit: one graph for each starting letter));
7. guess incorrectly: prune the graph (no such letter exists).

We're dealing with a sort of tree here, aren't we, whose leaves are words? In other words, given a sequence of correct guesses; it should be unambiguous that it signifies a given word. Better yet: we should be able to determine the set of signifiable words; we could have some heuristic: possible words vs. `maxWrongGuesses`, but I don't think we have `maxWrongGuesses`.

Not true! We have `numWrongGuessesRemaning()` and `getMaxWrongGuesses()`. `getSecretWordLength()`: nice! That's what I was looking for.

If `numWrongGuessesRemaning` $> |signifiableWords|$, then we can clearly guess all the $\{signifiableWords\}$ one-by-one.

Thing we're missing is: how do we go from an incompletely determined graph to possible words? That's why I suspect we're dealing with, not a tree per se, but a graph with terminal vertices; graph if we converge on letters: i.e. $c_1 \rightarrow a_2 \rightarrow b_3$, $f_1 \rightarrow a_2 \rightarrow b_3$; as opposed to: $c_{1_1} \rightarrow a_{2_1} \rightarrow b_{3_1}$, $f_{1_1} \rightarrow a_{2_2} \rightarrow b_{3_2}$. Convergence seems to make sense from a space point-of-view, as well as determining convergent paths.

Create a special root node: one artificial graph (graph with terminal vertices, not a tree).

```
CLASSPATH := .:$(shell echo lib/*.jar | tr ' ' ':')
SOURCE := $(wildcard *.java) $(wildcard *.scm)
OBJECTS := $(patsubst %.java,%.class,$(wildcard *.java)) \
    $(patsubst %.scm,%.scc,$(wildcard *.scm))

.PHONY: test
```

```
%.class : %.java
    javac -classpath $(CLASSPATH) $<

%.scc : %.scm
    sisc -e "(compile-file \"$<\" \"$@\")"

test: all
    java -classpath $(CLASSPATH) Hangman words.txt factual

all: $(OBJECTS)
```

```
import java.lang.reflect.Array;
import java.util.Arrays;
import java.util.Queue;
import java.io.IOException;

import sisc.interpreter.Interpreter;
import sisc.interpreter.Context;
import sisc.interpreter.SchemeCaller;
import sisc.interpreter.SchemeException;
import sisc.data.EmptyList;
import sisc.data.Symbol;
import sisc.data.Pair;
import sisc.data.Value;
import sisc.data.SchemeString;
import sisc.util.Util;
import sisc.modules.s2j.JavaObject;

public class Hangman {
    public static final int maxWrongGuesses = 5;

    public static int run(HangmanGame game, GuessingStrategy strategy) {
        return 0;
    }

    public static void main(final String[] args) throws SchemeException, IOException {
        String dictionary = args[0];
        String[] words = Arrays.copyOfRange(args, 1, args.length);
        for (String word: words) {
            HangmanGame game = new HangmanGame(word, maxWrongGuesses);
            GuessingStrategy strategy = new GraphGuessingStrategy(dictionary, game);
            run(game, strategy);
        }
    }
}
```

```
(import s2j)

(define-syntax debug
  (syntax-rules ()
    ((_ x ...)
     (begin
       (write `((x ,x) ...))
       (newline)))))

(define-java-classes
  <guessing-strategy>
  <hangman-game>
  <hangman>)

(define-java-proxy
  (guessing-strategy dictionary game)
  (<guessing-strategy>)
  (define x 2)
  #;
  (define (guessing-strategy dictionary game)
    (set! x 3)
    (debug x))
  (define (next-guess game)
    #;(game get-secret-word-length)
    2))

;; (debug (java-proxy-class <hangman-game>))
;; (debug (java-class-name <hangman-game>)
;;        (java-method-name guessing-strategy))

;; (define-generic-java-methods next-guess)
;; (define-generic-java-field-accessors :next-guess)

(define (make-dictionary dictionary)
  (with-input-from-file
      dictionary
    (lambda ()
      (unfold ))))

(define (init strategy dictionary game)
  (debug 2)
  2)

(define (main argv)
  (let ((dictionary (car argv))
        (words (cdr argv)))
```

```
(debug 2 dictionary words)
;; (debug (java-new <java.util.linked-hash-set> (->jint 100)))
;; (debug (java-new <hangman-game> (->jstring "factual") (->jint 4)))
(let* ((game (java-new <hangman-game> (->jstring "factual") (->jint 4)))
       (strategy (guessing-strategy dictionary game)))
  strategy)
#;
(let ((game (java-new <hangman-game> (->jstring "factual") (->jint 4))))
  (debug game))

;; (java-new <hangman-game> (->jstring "oeunth") (->jint 5))
;; (java-new <hangman-game> "oeuhnt" 5)
#;
(with-input-from-file
dictionary
(lambda ()
(let next-word ((word (read)))
(if (eof-object? word)
0
(begin
(debug word)
(next-word (read)))))))))
```

I'm going to assume that the program takes a dictionary file and list of words; and that I have to program the glue.

How is this going to work with initialization, etc.? For clarity, should we implement the interface in Java, there to defer to Scheme?

No, let's do it in Scheme. (Risky.)

We'll define the main run-loop in Java (`run(game, strategy)`); the creation of the strategy, however, will defer to Scheme (`new GraphStrategy(dictionary)`).

```
import java.io.IOException;

import sisc.interpreter.Interpreter;
import sisc.interpreter.Context;
import sisc.interpreter.SchemeCaller;
import sisc.interpreter.SchemeException;
import sisc.data.EmptyList;
import sisc.data.Symbol;
import sisc.data.Pair;
import sisc.data.Value;
import sisc.data.SchemeString;
import sisc.util.Util;
import sisc.modules.s2j.JavaObject;

public class GraphGuessingStrategy implements GuessingStrategy {
```

```
    public Guess nextGuess(HangmanGame game) {
        return new Guess() {
            public void makeGuess(HangmanGame game) {
            }
        };
    }

    // Let this take, instead, a dictionary-graph.
    public GraphGuessingStrategy(final String dictionary, final HangmanGame game) throws Sch
        Context.execute(new SchemeCaller() {
                public Object execute(Interpreter interpreter) throws SchemeException {
                    interpreter.loadSourceFiles(new String[] { "hangman.scc" });
                    try {
                        interpreter.eval(Util.proc(interpreter.eval("init")),
                                         new Value[] {
                                             new JavaObject(this),
                                             new SchemeString(dictionary),
                                             new JavaObject(game)
                                         });
                    } catch (Throwable e) {
                        e.printStackTrace();
                    };
                    return true;
                }
            });
    }
}
```

Let's initialize a "strategy-factory" with the dictionary that creates a graph for each class of words with letters $2..n$. No, fuck that: initialize the `GraphGuessingStrategy` with the dictionary-graph.

How do we share the graphs, though? Either it's some sort of opaque scheme-object corresponding to, say, a hash-table; or it's some Java object (say, `java.util.HashTable`) with which we have to interact from Scheme. What are the performance characteristics of both scenarios?

For convenience, I'd rather have Scheme objects to pass around (a `Value`, essentially).

(Oh, check it out: they have a Hashtable interface.)

```
(require-library 'sisc/libs/srfi/srfi-1)
(import srfi-1)
(import s2j)
;; (load "irregex-0.8.1/irregex.scm")
;; (require-extension (lib scsh-regexp/scsh-regexp))
```

```
;; (import scsh-regexp/scsh-regexp)

(define-java-classes
  <java.lang.string>
  <java.util.regex.pattern>)

(define-generic-java-methods
  matches)

(let ((dictionary
       (with-input-from-file
           "words.txt"
         (lambda ()
           (let next-word ((word (read))
                           (dictionary '()))
             (if (eof-object? word)
                 dictionary
                 (next-word
                  (read)
                  (cons (->jstring word) dictionary))))))))
  #;
  (let ((pattern (->jstring "...")))
    (filter (lambda (word)
              (->boolean (matches word pattern)))
            dictionary))
  ;; Adds 6 sec.
  #;
  (let ((pattern (->jstring ".......")))
    (filter (lambda (word)
              (->boolean (matches word pattern)))
            dictionary))
  ;; Adds 6 sec.
  #;
  (let ((pattern (->jstring "...a...")))
    (filter (lambda (word)
              (->boolean (matches word pattern)))
            dictionary))
  ;; Adds 4 sec; should we bother compiling the regex?
  (let ((pattern (->jstring "...a...")))
    (filter (lambda (word)
              (->boolean (matches word pattern)))
            (let ((pattern (->jstring ".......")))
              (filter (lambda (word)
                        (->boolean (matches word pattern)))
                      dictionary))))
  2
```

```
#;
(for-each (lambda (word)
'harro
;; (irregex-match '(w/nocase ".*") word)
;; (matches (->jstring word) (->jstring ".?"))
#;
(let ((word (java-new <java.lang.string> (->jstring word))))
(matches (java-null <java.util.regex.pattern>) word (->jstring ".?"))))
dictionary))
```

The issue is: either we have to deal with Scheme strings, Java strings or some kind of conversion; the conversion is relatively slow (have to do so for e.g. `java.lang.String.match()`); and irregex doesn't work (otherwise we could stick with Scheme strings).

Can we do slow regex for now, and replace it with some e.g. trie algo later?

I just realized that the chaining doesn't matter; we're only dealing with frequency, I believe. Could roughly:

1. Filter the current dictionary based on `getGuessedSoFar()` (this could be a string->regex substitution for now and would thus carry an O(n * O(regex)) penalty; some sort of e.g. trie-based search might get us log-linear).
2. Is the $|currentdictionary| < numWrongGuessesRemaning$? Guess!
3. Build a frequency list of letters.
4. Guess one of the most frequent letters; remove it.
5. Wrong?
   (a) Guess again.
6. Right?
   (a) Goto 1.

Just met Aaron at the Hacker News meetup: drop Scheme, adopt Clojure.

This is subtle, by the way: regexps are susceptible to false positives. From the same page:

> As far as determining the next character to guess, you probably don't want to select the most frequent character. Instead, you want to select the character that comes closest to being in 50% of words, meaning you eliminate the most possibilities either way.

Also:

> Your regex should look more like this: `^e[^e][^e][^e][^e]e[^e]$`.

Good call.
Frequency and density; also: probability that letter is in a given position!

> From this, you can easily duplicate my algorithm by determining which letter to choose based on which word appears in the most candidate words (highest probability of being right) and break ties by using the highest overall density.

> First, a 1 size fits all algorithm that weighs various different pieces of data to produce an overall score. The advantage here is that you should be able to tune these weights without touching code to produce the best results for a given "dictionary". The second approach would be to have multiple algorithms already fine tuned for a given set of conditions and dispatch to that algorithm based on initial conditions. For instance, it may be ok to take a risk of guessing wrong if the remaining wrong guesses is still high or perhaps utilize the "simple" algorithm already presented unless the probabilities and candidates remaining (if right or wrong) is beyond some threshold.

The Markov chains come up again:

> An higher order approach would be to look at probabilities for chains of letters. E. g. in English the chance for an 'u' should be higher than average after a 'q'.

He's right, actually: looking at mere frequency represents a loss of data ('q' followed by 'u', for instance, is not handled). Let's revisit the Markov idea.
On the other hand, maybe I was right:

> If 'q' is guessed and is correct, then the next time around, 'u' will already jump to the top of the list of letters with the most probability to be correct. You don't need to know any special knowledge about letter frequency (alone or in chains) to get this benefit.

Original post; this guy is suggesting some kind of binary search:

> At least in your example, a letter never appears in more than half of the candidates, so the most frequent letter and the closest-to-half-half letter coincide. But imagine if you found out that the actual word contained Q. Then for sure the next most common letter will be a U; but guessing U and getting it right will not give you much new information. Aiming for half-half lets your correct and incorrect guesses both contribute information.

Position:

> My strategy is optimized not to guess wrong. After only 5 guesses (2 right and 3 wrong) it had narrowed the search space down to

exactly 1 word. This is because I prune by position not just by presence of letter so even successful guesses on a popular letter can still effectively decrease the search space. Your approach would be improved with this strategy as well. I don't think the opportunities stop there.

Note that there might be other ways to score each possible next-letter guess. Number of non-empty buckets comes to mind as an "average case" measure (to be maximized). Again, this is all assuming we're minimizing the total number of guesses. That way, all of the possible outcomes are (i.e., the guessed letter appears or doesn't appear in the word) are treated the same. To minimize the number of wrong guesses, you have to treat the "doesn't appear in the word" outcome differently and weight things in some better way.

Regex should account for position, shouldn't it?

Now let's assume we were going with 'eclectic'. There were 9,638 words in my dictionary with a length of 8. The letter that appeared in the most of those words was the letter 'e' at 6,666. Note that I didn't count total occurrences of 'e' but only 1 per word. This equated to 69%. Now what I set out to do was map the different ways the letter 'e' appeared across those 6,666 words. In the word 'eclectic' it appears in positions 1 and 4 where as in 'envelope' it appears in positions 1, 4 and 8. After guessing and seeing which positions were filled in, I could even eliminate words with letter 'e' even if they shared the common position because they didn't share all positions. That last part (all positions) was the piece I hadn't considered in my previous exercise. So here is the mind blowing part. The most common set of positions across the 6,666 words with the letter 'e' still had less than 1,600 possible words. This means that by selecting the letter 'e' (69% chance of being right) I will reduce the candidate list from 9,638 to less than 1,600 (and probably a lot further). It seemed pointless then to come up with some weights for determining letter based on probability of being correct and degree to which the candidate list is reduced because the "dumb" method was still doing a superb job.

I do have one last final revision to make. I choose the letter that appears in the most candidate words but I don't break ties. Currently it is the result of sort. I plan to add total count as a secondary tie breaking condition to see if that improves results. I should post something later today.

29 Threading and de-structuring

Interesting tips; as well as: `assoc`-reduction.

30 Description

30.1 The Goal

Write a program that plays the Hangman game. A description of the game can be found at http://en.wikipedia.org/wiki/Hangman_(game).

30.2 The Problem

Your goal is to use a provided API to play Hangman efficiently. You need to guess a word using as few guesses as possible, and make no more than maxWrongGuesses incorrect guesses. You are writing the letter/word guessing strategy.

We would like you to use the provided Java APIs and to write your solution in Java. However, if you don't know Java or are not comfortable with it, please feel free to use other reasonably mainstream programming languages (C++, Python, Ruby, Scala, etc.). If you do use another language, please make sure that your program can accept a dictionary file and a list of test words (that are in the dictionary). The output of the program should be a score for each test word and the average score across all test words. Also, if you don't use Java, please provide build instructions if they are not straightforward.

Your score for a word will be:

> you guessed the word right before making maxWrongGuesses incorrect guesses

or

> 25 if you lost the game before guessing the word correctly.

You will need to write an implementation of the GuessingStrategy interface and some code to use your GuessingStrategy on a HangmanGame instance.

The pseudocode to run your strategy for a HangmanGame is:

```
// runs your strategy for the given game, then returns the score
public int run(HangmanGame game, GuessingStrategy strategy) {
    while (game has not been won or lost) {
        ask the strategy for the next guess apply the next guess to
        the game
    }
    return game.score();
}
```

A trivial strategy might be to guess 'A', then 'B', then 'C', etc. until you've guessed every letter in the word (this will work great for "cab"!) or you've lost.

Every word you encounter will be a word from the words.txt file.

30.3 Example

For example, let's say the word is FACTUAL.

Here is what a series of calls might look like:

```
HangmanGame game = new HangmanGame("factual", 4); // secret word is
                                                  // factual, 4 wrong
                                                  // guesses are
                                                  // allowed
System.out.println(game);
new GuessLetter('a').makeGuess(game);
System.out.println(game);
new GuessWord("natural").makeGuess(game);
System.out.println(game);
new GuessLetter('x').makeGuess(game);
System.out.println(game);
new GuessLetter('u').makeGuess(game);
System.out.println(game);
new GuessLetter('l').makeGuess(game);
System.out.println(game);
new GuessWord("factual").makeGuess(game);
System.out.println(game);
```

The output would be:

```
-------; score=0; status=KEEP_GUESSING
-A---A-; score=1; status=KEEP_GUESSING
-A---A-; score=2; status=KEEP_GUESSING
-A---A-; score=3; status=KEEP_GUESSING
-A--UA-; score=4; status=KEEP_GUESSING
-A--UAL; score=5; status=KEEP_GUESSING
FACTUAL; score=5; status=GAME_WON
```

`game.score()` will be 5 in this case since there were 4 letter guesses and 1 incorrect word guess made.

30.4 Sample Data

As a baseline, here are scores for a reasonably good guessing strategy against a set of 15 random words. Your strategy will likely be better for some of the words and worse for other words, but the average score/word should be in the same ballpark.

```
COMAKER = 25 (was not able to guess the word before making more than 5
mistakes)
CUMULATE = 9
ERUPTIVE = 5
```

```
FACTUAL = 9
MONADISM = 8
MUS = 25 (was not able to guess the word before making more than 5
mistakes)
NAGGING = 7
OSES = 5
REMEMBERED = 5
SPODUMENES = 4
STEREOISOMERS = 2
TOXICS = 11
TRICHROMATS = 5
TRIOSE = 5
UNIFORMED = 5
```

30.5 Resources

You should have been provided with a zip file with source code and a dictionary file to get you started. If you were not sent this zip file, or you have any questions about the contents, please let us know right away.

The resources contain a dictionary file called words.txt. You can assume all words that your program will be tested with come from this dictionary file.

30.6 Judging

Your solution will be graded on the following criteria

- code quality, readability, and design
- performance (speed/memory footprint). We are more concerned with average time per game, so expensive one-time initialization is okay (as long as it's not too egregious)
- total score for 100 random words, compared to the total score of several reference implementations for the same 100 words (max wrong guesses is typically set to 5)