Four Frequency-Strategy Solutions to Hangman

Peter Danenberg <danenberg@post.harvard.edu>

September 3, 2011

Abstract

I'd like to present four solutions to the hangman problem which are sub-strategies of an arch-strategy called *frequency-strategy*: frequency-strategy reduces the word-space by filtering on the most common unguessed letter and guesses words when the remaining-words/remaining-guesses ratio looks auspicious. Three sub-strategies, the *regex-strategy*, *predicate-strategy* and *trie-strategy*, which differ by how they calculate the most common letter, are discussed below. There are also *deterministic* and *sampling* variants on these sub-strategies.

The strategies in order of decreasing performance are: deterministic regex-strategy; sampling regex-strategy; predicate-strategy; trie-strategy.

Contents

1 Frequency Strategy **1**

2 Substrategies **2**
2.1 Trie strategy 2
2.2 Predicate strategy 2
2.3 Regex strategy 3
2.3.1 Sampling regex strategy 3
2.3.2 Deterministic regex strategy 4

3 Invoking `hangman` **4**

4 Conclusions and Possible Improvements **5**

1 Frequency Strategy

The frequency strategy reduces the space of possible solutions by filtering on the most common letter, guessing a word when the remaining-words/remaining-guesses ratio looks auspicious:

1. Filter on the last guessed letter (if it exists).

	Trie	Predicate	Determ. rgx.	Sampl. rgx.
Average time (ms)	315.19897	69.23657	22.884363	22.892752
Average score	11.089	7.214	7.082	7.077

Table 1: Relative performance of the trie-, predicate-, deterministic- and sampling-regex-strategies over 1000 random words.

2. Is the ratio of words to remaining guesses auspicious?
   (a) If so, guess a remaining word.
   (b) Otherwise, guess the most common unguessed letter.
3. Is the game over?
   (a) If so, return the score.
   (b) Otherwise, return to step 1.

2 Substrategies

The sub-strategies of the frequency arch-strategy are variations on step 2b; table 1 has a comparison of their relative performance when run over the entire dictionary.

2.1 Trie strategy

The trie-strategy encodes the dictionary in a sparse, 26-ary trie (see figure 1). I had high hopes for the trie-strategy because of the $O(\log_{26} n)$ lookup and deletion; it both performed the worst, however, and was least accurate.

I suspect that, on the one hand, constant factors intervened; on the other, I decided to sacrifice certain trie-invariants for the sake of performance. As a result of pathological trie-degeneration, the letter-counting algorithm counted spurious letters; and adjusting the algorithm to maintain trie-invariants gave even worse performance.

Average time (ms) 315.19897

Average score 11.089

2.2 Predicate strategy

The predicate-strategy is a linear strategy which is less general than the regex-strategy below: it filters integer-encoded[1] string-representations (see figure 2) with ad-hoc negative and positive predicates. The predicates corresponding to a game-state of e.g. '`__R__`' with a last guess of '`E`' (4) are $(\neg 4 \wedge \neg 4 \wedge 17 \wedge \neg 4 \wedge \neg 4)$.

[1] The integers are simply ASCII(*letter*) − 97.

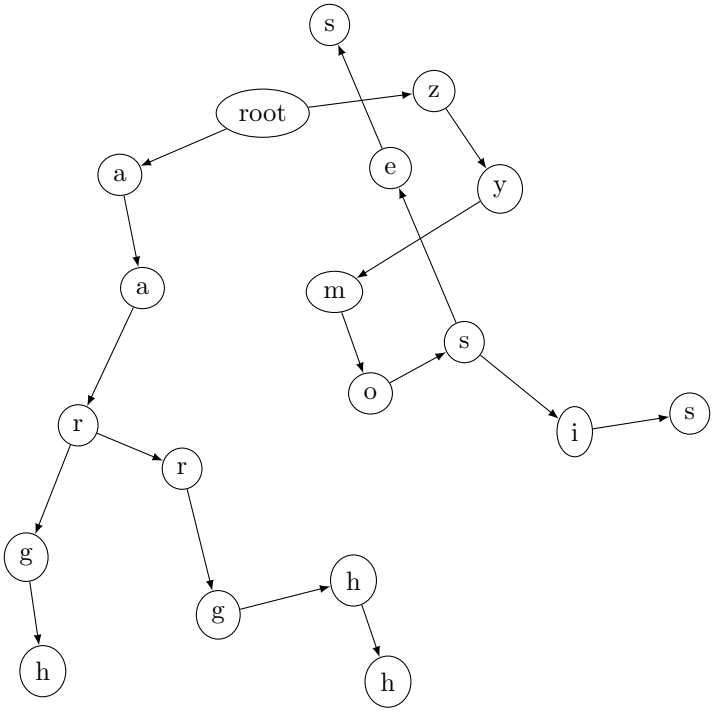

Figure 1: The sparse 26-ary trie encoding ‘aargh’, ‘aarrgh’, ‘aarrghh’, ‘zymoses’ and ‘zymosis’.

Despite the fact that these ad-hoc predicates are less general than regular expressions, they fail to outperform them; I suspect that the overhead of *lambda*-application is to blame.

Average time (ms) 69.23657

Average score 7.214

2.3 Regex strategy

The regex-strategy, like the predicate strategy, is linear; it stores it dictionaries, however, as lists of character-encoded strings and filters them with negative and positive regular expressions. The regex corresponding to a game-state of e.g. ‘`__R__`’ with a last guess of ‘`E`’ is `^[^E][^E]R[^E][^E]$`.

2.3.1 Sampling regex strategy

The sampling regex strategy differs from its deterministic sibling in that, instead of making an exhaustive count of letters frequencies, it samples the word space

(0 0 17 6 7)
(0 0 17 17 6 7)
(0 0 17 17 6 7 7)
(25 24 12 14 18 4 18)
(25 24 12 14 18 8 18)

Figure 2: Predicate-encoding for 'aargh', 'aarrgh', 'aarrghh', 'zymoses' and 'zymosis'.

until a stable ratio of letters appears. The notion of stability has three tuning parameters:

1. ϵ, or `delta-percentage-tolerance`, the percentage below which changes in frequency are considered negligible;
2. `sampling frequency`, the number of iteration between sampling; and
3. `ratio-to-n-of-minimum-iterations`, the ratio of the total words; after which to begin sampling.

It should be possible to tune these parameters to outperform deterministic regex with some sacrifice in accuracy.

Average time (ms) 22.892752

Average score 7.077

2.3.2 Deterministic regex strategy

The deterministic regex-strategy differs from its sampling sibling in that it does a full frequency count of all letters before making a guess. It should, in theory, be more accurate than sampling and slightly slower; in practice, it is practically indistinguishable from sampling with respect to speed and score. In table 1, for instance, is happened to be both faster than sampling and less accurate.

Average time (ms) 22.884363

Average score 7.082

3 Invoking `hangman`

The basic invocation is:

```
./hangman <dictionary-file> <word>+
```

e.g.

```
./hangman words.txt comaker cumulative eruptive factual
```

which should present you with some summary statistics at the end:

```
{:data
  {"comaker" {:time-in-ms 349.20504, :score 11},
   "cumulative" {:time-in-ms 57.731637, :score 5},
   "eruptive" {:time-in-ms 47.931958, :score 5},
   "factual" {:time-in-ms 70.195218, :score 7}},
  :average-time-in-ms 131.26596,
  :average-score 7.0}
```

By default, it runs the deterministic regex strategy; to run other strategies, or for more verbose output, see the usage:

```
Usage: hangman [--deterministic-regex|--sampling-regex|--predicate|--trie]
  [--max-wrong-guesses|-m GUESSES] [--all|-a] [-v|--verbose] DICTIONARY
  [WORD]...
Options
  --deterministic-regex, -d      Use the deterministic regex strategy.
                                   [default true]
  --sampling-regex, -s           Use the sampling regex strategy.
  --predicate, -p                Use the predicate strategy.
  --trie, -t                     Use the trie strategy.
  --max-wrong-guesses, -m <arg>  Set max wrong guesses to GUESSES.
                                   [default 4]
  --all, -a                      Run all the words in the dictionary.
  --verbose, -v                  Verbose output (NB: affects reported times)
```

4 Conclusions and Possible Improvements

I'm a little ashamed to admit that I couldn't beat linear regex, even with a more sophisticated trie-algorithm; it may be that I shouldn't have been so dogmatic about using functional algorithms: a little mutation doesn't hurt once in a while.

Some TODOs:

- Don't merely guess a remaining word randomly: rank them by frequency of constituent letters.
- Tune sampling regex so that it outperforms deterministic regex.
- Beat linear regex!