Project Motivation

Over the last year I've been using a relatively new (2009 announce, first "stable" release 28 Mar 2012) programming language called Go, for various projects. As it's a new language it lacks a lot the niche type libraries common in older languages. Go has an existing Porter stemming algorithm implementation and while it is old and well tested, it isn't very flexible or customizable. I decided to create a Go implementation of the Paice/Husk rule based stemmer to help fill a gap in the languages libraries, and make it freely available under a MIT style license. Asimple website hosted on Google App Engine was also created to demonstrate use of the library, as well as a separate evaluation program.

Architecture and Implementation

The project was split into 3 separate parts:

1. The paicehusk stemming library
2. Ademonstration website
3. The evaluation program

The Paice/Husk library:

The library was developed using only functionality provided by the Go standard Library. It consists of around 450 lines of Go. The stemmer follows the algorithm defined at http://www.comp.lancs.ac.uk/computing/research/stemming/Links/paice.htm. The exposed API consists of the following functionality:

NewRuleTable

Takes a slice of strings in rule format, validates them and returns a new RuleTable.

```
// NewRuleTable returns a new RuleTable instance
func NewRuleTable(f []string) (table *RuleTable)

// Example usage:
var DefaultRules = NewRuleTable(strings.Split(defaultRules, "\n"))
```

DefaultRules

Provides a default set of rules for a user.

```
// DefaultRules is a default ruleset for the english language.
var DefaultRules = NewRuleTable(strings.Split(defaultRules, "\n"))

// Example Usage:
for i := range testFile {
        fmt.Println(paicehusk.DefaultRules.Stem(testFile[i]))
}
```

ValidRule

Extracts a rule from the provided string using the regexp `[a-zA-Z]*\\*?[0-9][a-zA-z]*[.>]gim`

```
// Validates a rule
func ValidRule(s string) (rule string, ok bool)

// Example Usage:
s, ok := ValidRule(s)
if !ok {
        return nil, false
}
```

ParseRule

Converts the textual representation of a rule to a rule struct

```
// ParseRule parses a rule in the form:
// |suffix|intact flag|number to strip|Append|Continue flag
func ParseRule(s string) (r *rule, ok bool)

// Example Usage:
table = &RuleTable{Table: make(map[string][]*rule)}
for _, s := range f {
        if r, ok := ParseRule(s); ok {
                table.Table[r.suf[:1]] = append(table.Table[r.suf[:1]], r)
        }
}
```

Stem

Method on RuleTable that returns the stem of the param word using the stored rules.

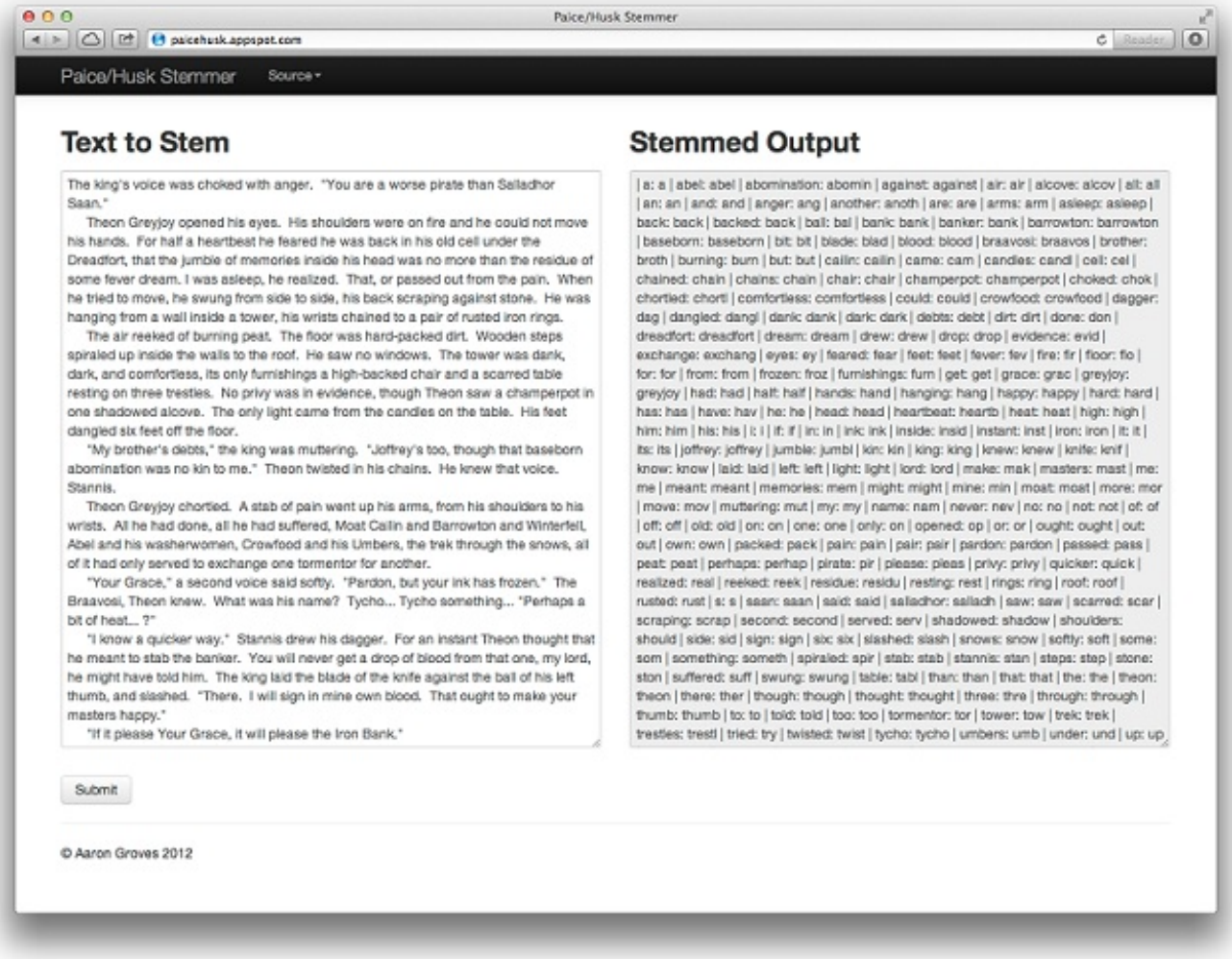
```
// Stem a string, word, based on the rules in *RuleTable r, by following
// the algorithm described at:
// http://www.comp.lancs.ac.uk/computing/research/stemming/Links/paice.htm
func (r *RuleTable) Stem(word string) string

// Example Usage:
for i := range testFile {
        fmt.Println(paicehusk.DefaultRules.Stem(testFile[i]))
}
```

Demonstration Site:

To demonstrate the use of the library and to provide an easy to use stemming service, I created a simple website hosted on Google App Engine. Go is one of the 3 run-times supported by App Engine, and as such, allowed me to integrate my library very quickly and easily.

I used the Bootstrap front-end framework so I didn't have to employ my terrible design skills.



The page communicates with the server via Ajax requests.

```
$.ajax({
        type: "POST",
        url: "/stem",
        data: JSON.stringify(input),
}).done(function(data) {
        var res = "| ";
        var raw = "\n\nRaw output:\n";
        $.each(data, function(key, value){
                res += key + ": " + value + " | ";
                raw += value + "\n";
        });
        $("#output").val(res + raw);
});
```

The Contents of the Input text box is converted to a JSON object, then posted to the server. The server un-marshals the JSON and stems the input word by word, storing the resulting stems in a map, de-duplicating the input. This map is then converted to JSON and returned to the page, where it is written to the out put text box.

```
if err := decoder.Decode(text); err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
}

words := wordreg.FindAllString(text.Text, -1)
for _, word := range words {
        out[strings.ToLower(word)] = paicehusk.DefaultRules.Stem(word)
}

encoder := json.NewEncoder(w)
```

```
11 if err := encoder.Encode(out); err != nil {
12         http.Error(w, err.Error(), http.StatusInternalServerError)
13 }
```

Evaluation

Astemmers strength is based on how likely it is to merge words together. Aweak stemmer will only merge clearly related words, eg plurals by dropping the 's' or verb variants such as dropping the 'ing' of 'fighting'. Astrong stemmer will be much more aggressive, however this increases the likely hood of stemming errors. The Paice/Husk stemmer is generally considered a "strong" stemming algorithm.

The Go implementation was evaluated against 3 word lists. It was also compared to an older Pascal implementation to test implementation accuracy, as well as the Porter Stemmer to provide another comparison.

The following metrics were calculated:

1. Mean number of words per conflation class
2. Index Compression Factor
3. Number of words and stems that differ
4. Mean number of characters removed
5. Median number of characters removed
6. The mode number of characters removed
7. The mean Hamming distance
8. The median Hamming distance
9. The mode Hamming distance
10. Fox and Frake Similarity Metric
11. Chris O'Neill Similarity Metric
12. Character removal distribution table
13. Hamming distance distribution table

Implementation Accuracy

I believe my implementation of the Paice/Husk algorithm to be accurate, however it does have some differences from the older Pascal version, and another C implementation that I tested against (in which I discovered several bugs). I've went over the algorithm and rule files many times, and have come to the conclusion that the older versions have errors, or have modifications to the algorithm.