# ECE473: Introduction to Artificial Intelligence

## Assignment: Text Reconstruction

*By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.*

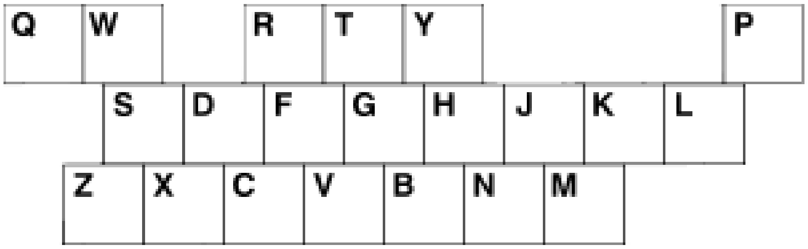

Figure 1: Caption

In this homework, we consider two tasks: *word segmentation* and *vowel insertion.* Word segmentation often comes up when processing many non-English languages, in which words might not be flanked by spaces on either end, such as written Chinese or long compound German words.[1] Vowel insertion is relevant for languages like Arabic or Hebrew, where modern script eschews notations for vowel sounds and the human reader infers them from context.[2] More generally, this is an instance of a reconstruction problem with a lossy encoding and some context.

We already know how to optimally solve any particular search problem with graph search algorithms such as uniform cost search or A*. Our goal here is modeling — that is, converting real-world tasks into state-space search problems.

## General Instruction

You should modify the code in `hw4_submission.py` between

*# BEGIN_YOUR_CODE*

and

*# END_YOUR_CODE*

but you can add other helper functions outside this block if you want. Do not make changes to files other than `hw4_submission.py`.
You can evaluate your code on two types of test cases, basic and hidden, which you can see in `hw4_grader.py`. Basic tests, which are fully provided to you, do not stress your code with large inputs or tricky corner cases. Hidden tests are more complex and do stress your code. The inputs of hidden tests are provided in `hw4_grader.py`, but the correct outputs are not. To run the tests, you will need to have `graderUtil.py`

[1] In German, *Windschutzscheibenwischer* is "windshield wiper". Broken into parts: *wind* wind; *schutz* block / protection; *scheiben* panes; *wischer* wiper.

[2] See https://en.wikipedia.org/wiki/Abjad.

in the same directory as your code and `hw4_grader.py`. Then, you can run all the tests by typing `python hw4_grader.py` This will tell you only whether you passed the basic tests. On the hidden tests, the script will alert you if your code takes too long or crashes, but does not say whether you got the correct output. We strongly encourage you to read and understand the test cases, create your own test cases, and not just blindly run `hw4_grader.py`.

Actual grading for this homework will be done on different corpus with similar grading metric in `hw4_grader.py`.

## Setup: $n$-gram language models and uniform-cost search

Our algorithm will base its segmentation and insertion decisions on the cost of processed text according to a *language model*. A language model is some function of the processed text that captures its fluency.

A very common language model in NLP is an $n$-gram sequence model. This is a function that, given $n$ consecutive words, provides a cost based on the negative log likelihood that the $n$-th word appears just after the first $n$-1 words.[3] The cost will always be positive, and lower costs indicate better fluency.[4] As a simple example: In a case where $n = 2$ and $c$ is our $n$-gram cost function, $c$`(big, fish)` would be low, but $c$`(fish, fish)` would be fairly high.

Furthermore, these costs are additive: For a unigram model $u(n = 1)$, the cost assigned to $[w_1, w_2, w_3, w_4]$ is

$$u(w_1) + u(w_2) + u(w_3) + u(w_4).$$

Similarly, for a bigram model b $(n = 2)$, the cost is

$$b(w_0, w_1) + b(w_1, w_2) + b(w_2, w_3) + b(w_3, w_4),$$

where $w_0$ is `-BEGIN-`, a special token that denotes the beginning of the sentence.

We have estimated $u$ and $b$ based on the statistics of $n$-grams in text. Note that any words not in the corpus are automatically assigned a high cost, so you do not have to worry about that part.

A note on low-level efficiency and expectations: This assignment was designed considering input sequences of length no greater than roughly 200, where these sequences can be sequences of characters or of list items, depending on the task. Of course, it's great if programs can tractably manage larger inputs, but it's okay if such inputs can lead to inefficiency due to overwhelming state space growth.

## Problem 1: Word Segmentation

In word segmentation, you are given as input a string of alphabetical characters (`[a-z]`) without whitespace, and your goal is to insert spaces into this string such that the result is the most fluent according to the language model.

**Implement an algorithm that, unlike the greedy algorithm, finds the optimal word segmentation of an input character sequence.**

Your algorithm will consider costs based simply on a unigram cost function. `UniformCostSearch` is implemented for you in `util.py`, and you should make use of it here. Solutions that use UCS ought to exhibit fairly fast execution time for this problem, so using A* here is unnecessary.

[3] This model works under the assumption that text roughly satisfies the Markov property.

[4] Modulo edge cases, the n-gram model score in this assignment is given by $\ell(w_1, \ldots, w_n) = -\log(p(w_n \mid w_1, \ldots, w_{n-1}))$. Here, $p(\cdot)$ is an estimate of the conditional probability distribution over words given the sequence of previous $n-1$ words. This estimate is gathered from frequency counts taken by reading Leo Tolstoy's *War and Peace* and William Shakespeare's *Romeo and Juliet*.

Before jumping into code, you should think about how to frame this problem as a statespace **search problem**. How would you represent a state? What are the successors of a state? What are the state transition costs? (You don't need to answer these questions in your writeup.)

Fill in the member functions of the `SegmentationProblem` class and the `segmentWords` function. The argument `unigramCost` is a function that takes in a single string representing a word and outputs its unigram cost. You can assume that all of the inputs would be in lower case. The function `segmentWords` should return the segmented sentence with spaces as delimiters, i.e. `" ".join(words)`.

For convenience, you can actually run `python hw4_submission.py` to enter a console in which you can type character sequences that will be segmented by your implementation of `segmentWords`. To request a segmentation, type seg mystring into the prompt.

```
>> seg thisisnotmybeautifulhouse
   Query (seg): thisisnotmybeautifulhouse
   this is not my beautiful house
```

Type help to see a full list of available commands.

**Hint**: You can refer to `NumberLineSearchProblem` and `GridSearchProblem` implemented in `util.py` for reference. They don't contribute to testing your submitted code but only serve as a guideline for what your code should look like.
**Hint**: The valid actions for the `ucs` object can be accessed through `ucs.actions`.

# Problem 2: Vowel Insertion

Now you are given a sequence of English words with their vowels missing (A, E, I, O, and U; never Y). Your task is to place vowels back into these words in a way that maximizes sentence fluency (i.e., that minimizes sentence cost). For this task, you will use a bigram cost function.

You are also given a mapping `possibleFills` that maps any vowel-free word to a set of possible reconstructions (complete words).[6] For example, `possibleFills('fg')` returns `set(['fugue', 'fog'])`.

**Implement an algorithm that finds optimal vowel insertions. Use the UCS subroutines.**

When you've completed your implementation, the function `insertVowels` should return the reconstructed word sequence as a string with space delimiters, i.e. `" ".join(filledWords)`. Assume that you have a list of strings as the input, i.e. the sentence has already been split into words for you. Note that the empty string is a valid element of the list.

The argument `queryWords` is the input sequence of vowel-free words. Note that the empty string is a valid such word. The argument `bigramCost` is a function that takes two strings representing two sequential words and provides their bigram score. The special out-of-vocabulary beginning-of-sentence word `-BEGIN-` is given by `wordsegUtil.SENTENCE_BEGIN`. The argument `possibleFills` is a function that takes a word as a string and returns a set of reconstructions.

Since we use a limited corpus, some seemingly obvious strings may have no filling, such as `chclt -> {}`, where `chocolate` is actually a valid filling. Don't worry about these cases.

**Note**: If some vowel-free word w has no reconstructions according to `possibleFills`, your implementation should consider w itself as the sole possible reconstruction.

Use the `ins` command in the program console to try your implementation. For example:

[6] This mapping was also obtained by reading Tolstoy and Shakespeare and removing vowels.

```
>> ins thts m n th crnr
   Query (ins): thts m n th crnr
   thats me in the corner
```

The console strips away any vowels you do insert, so you can actually type in plain English and the vowel-free query will be issued to your program. This also means that you can use a single vowel letter as a means to place an empty string in the sequence. For example:

```
>> ins its a beautiful day in the neighborhood
   Query (ins): ts btfl dy n th nghbrhd
   its a beautiful day in the neighborhood
```

## Acknowledgement

Stanford CS221: Artificial Intelligence: Principles and Techniques