CS 486/686 Introduction to Artificial Intelligence—Fall 2016

Assignment 1: Natural Language Generation using Search

Due Date: Friday October 7, 2016 at 5:00pm.

Purpose: In class, you learned about the A* search algorithm. **But search is useful for far more than path-finding in games or route-planning.** In this assignment, you will be implementing a natural language (in our case, everyday English language) text generator that uses search to find grammatically-correct sentences.

NOTE: This assignment may be done in teams of two people.
*****Please attach a copy of the Assignment 1 CoverSheet (available on our course LEARN site) signed by each team member to the front of your assignment hand-in.**

INTRODUCTION/OVERVIEW OF THE ASSIGNMENT

Input file specification:
We have provided for you an **input.txt** file that contains pairs of words that have been extracted from a text corpus of 11 fairy tales. Each pair of words is associated with the *probability* that the second word follows the first word in this text corpus. Each of the words is labelled with its grammatical *"part-of-speech"* (PoS) tag (e.g., NN for noun, VB for verb, JJ for adjective). You can find a part-of-speech tag reference at:

http://www.winwaed.com/blog/2011/11/08/part-of-speech-tags/

Each line in **input.txt** has the format:

```
word1/PartOfSpeech1//word2/PartOfSpeech2//probability
```

For example, the following line indicates that when the word "hans" appears, 5% of the time it is followed by the word "saw". "hans" here has the part-of-speech (PoS) tag NNP (which means "singular proper noun"), and "saw" has the PoS tag VBD (which means "past-tense verb"). [Note: During our pre-processing of the **input.txt** file all words, including proper nouns and names, were converted to lower-case.]

```
hans/NNP//saw/VBD//0.05
```

Note that, in general, it is possible for the same word to function as more than one part-of-speech. For example, "saw/VBD" is a verb, but "saw/NN" can also be a noun.

Determining how "good" a sentence is:
In any search problem, it is necessary to have some way to compare candidate solutions to determine which is "better." For A*, one candidate solution is "better" than another if it reaches the goal state with a shorter path. For this assignment, one candidate solution is "better" than another if it is a complete, grammatically correct, sentence with a **higher conditional probability**. In other words, which sentence is more likely to occur in this corpus (i.e., sample of texts)? For example, in fairy tales we are more likely to come across sentences involving imaginary creatures than in news articles.

Important to note: We distinguish between a "sequence" of words, and a "sentence"—a sentence must have an acceptable grammatical form. During the search process we will be building up a sequence of words, but this sequence may not yet be a fully complete, grammatically correct, sentence. However, our search algorithm needs to have some way of assigning a probability to each intermediate sequence in order to compute the probability ***P*** of the final completed sentence at the end of the search.

So, for our corpus of fairy tales we calculate probability ***P***(<this sequence occurs in our corpus>) as follows:

1. For any 1-word sequence where this word occurs in our corpus vocabulary (e.g., "hans", "all"):
   - Probability ***P*** (<this word occurs in the corpus>) is intuitively 1.0

2. For any 2-word sequence (e.g., <"hans", "saw">):
   - ***P***(<"hans", "saw">) occurs in our corpus is 0.05, using the information in the **input.txt** file.

     Reminder: input.txt contains pairs of words that have been extracted from our corpus of 11 fairy tales. Each pair of words is associated with the *probability* that the second word follows the first word in this text corpus.

3. For the 3-word sequence <"hans", "saw", "all">:
   - ***P***(< "hans", "saw", "all"> occurs in our corpus>) is calculated as the product of the probabilities of the successive word-pair sequences <"hans", "saw"> and <"saw", "all"> so:

     P(< "hans", "saw", "all">) = ***P***(<"hans", "saw">) ∗ P(< "saw", "all">) where "*" indicates multiplication.
     Again, ***P***(<"hans", "saw">) and P(< "saw", "all">) can be obtained from the **input.txt** file.

4. In general then, for any *n*-word sequence **P**(< "hans", "saw", $word_3$, $word_4$, …, $word_{n-1}$, $word_n$>:
   - ***P***(<probability this sequence occurs in our corpus>) is calculated as the product of the probabilities of the (*n*-1) successive adjacent word-pair sequences.

     For example, the probability that the sequence <"hans saw all danger"> (which is a complete grammatically correct sentence) occurs in our corpus would be calculated as:

     P (<"hans", "saw">) ∗ ***P***(< "saw", "all">) * ***P***(< "all", "danger">) where each of these probabilities can be obtained from the **input.txt** file.

Example search:
In this example, we use a **breadth-first search** to look for a valid sentence starting with the name "hans."
Our definition of a **valid sentence** is that it must have the grammatical form: **NNP-VBD-DT-NN**
That is, a proper noun (NNP), followed by a past tense verb (VBD), followed by a determiner (DT), followed by an ordinary single or mass noun (NN). If we print all word sequences considered during the search, we obtain something like the following:

```
hans // probability 1.0
hans had // probability 0.05
...
hans saw // probability 0.05
hans had died // probability 0.00025
hans had been // probability 0.0033
...
hans saw all // probability 0.0013
hans had that glance // probability 0.0000053
...
hans saw all danger // probability 0.000024
...
```

The first node we consider contains only the word "hans", with a probability of 1.0 (the probability of a word given itself is always 1.0).

Since "hans/NNP" could be the start of a valid sentence, we look at the pairs starting with "hans/NNP," and choose one (e.g., "had/VBD") to consider next. We find that the probability of "had/VBD" given "hans/NNP" is 0.05, so we say that the probability of "hans saw" given "hans" is 1.0 * 0.05 = 0.05.

We consider all 2-word sequences before looking at 3-word sequences. If the second word of any 2-word sequence does not have the necessary part-of-speech (VBD) then we drop that sequence from further consideration.

Proceeding to look at pairs starting with "had/VBD", we find the pair <"had", "died">. However, the parts-of-speech in this sentence ("NNP," "VBD," "VBD") cannot help form a valid sentence according to our stated grammatical criteria (since we require a "DT" instead for the third word), so we do not consider this sentence any further and continue on to the next word "been".

As we continue the search process, we eventually find the sentence "hans saw all danger," which meets our validity requirement. Thus, the sentence, "hans saw all danger" represents a *valid* goal node (although not necessarily the *most probable* goal node).

THE ASSIGNMENT: **A SIMPLE SEARCH-BASED SENTENCE**

Part 1 [30 marks] WRITING THE BASIC CODE

Write a **breadth-first search** function named **`generate`** that takes the following parameters:

1. **`startingWord`**: The first word of the sentence. E.g., takes an argument like "hans".
2. **`sentenceSpec`**: A list of its parts-of-speech. E.g., takes an argument like ["NNP", "VBD", "DT", "NN"].
    - **Note:** For this Part 1 of the assignment you should assume that a valid sentence can **only** have the form: NNP-VBD-DT-NN. You will experiment with other possible valid sentence forms in Part 2.
    - Syntactically, the way you will specify a list will vary depending on what programming language you are using (see allowed languages below).
3. **`graph`**: The text of **`input.txt`**.
    - **Tip:** Your function **`generate`** should "parse" this input text to create an explicit data structure to represent this information.

The output of the function should be a **`string`** that contains:

1. The highest-probability sentence found.
2. The number of nodes (i.e., word sequences) considered during the search.

Example:

With **`startingWord = "hans"`** and **`sentenceSpec = ["NNP","VBD","DT","NN"]`**, your output should be something like the following:

```
"hans found a man" with probability 0.0003416856492027335
Total nodes considered: 3471
```

Do not worry if the total nodes considered is different from 3471. The exact number will vary depending on when you consider a node to be "counted."

You may implement your solution in any of the following languages: Python, Javascript, Java, or C++.

Ideas for helper functions you may want to write:

- **Sequence addWordToSequence(Sequence s, String word, String partOfSpeech, Float probability)**
  Adds a word to the end of a sequence
- **Sequence duplicateSequence(Sequence s)**
  Returns a copy of a word sequence (Sequence is a class or structure you will have to define)
- **Boolean isValidSentence(Sequence s)**
  Returns true if a word sequence matches the given **`sentenceSpec`**
- **Boolean mayFormValidSentence(Sequence s)**
  Returns true if it is possible for a sequence to match **`sentenceSpec`** if more words are added. If this function returns false, then the sentence can be dropped from the search.

Submit for Part 1:

- **The source code for your search function.**

Part 2 [20 marks] TESTING YOUR SENTENCE GENERATOR

For this part of the assignment you will consider different possible grammatical forms for a valid sentence. You will still continue to use a simple breadth-first search strategy.

Run your function **`generate`** to find the **highest-probability sentence** for each of the following argument sets of starting word and grammatical sentence form:

1. **`startingWord = "benjamin", sentenceSpec = ["NNP","VBD","DT","NN"].`**
2. **`startingWord = "a", sentenceSpec = ["DT","NN","VBD","NNP"].`**
3. **`startingWord = "benjamin", sentenceSpec = ["NNP","VBD","DT","JJS","NN"].`**
4. **`startingWord = "a", sentenceSpec = ["DT","NN","VBD","NNP","IN","DT","NN"].`**

Check to make sure that all outputs look like they have probabilities and node counts that make sense to you. If the outputs don't make sense, your function may have a bug!

If your function is too slow you may want to check the following:

- Is your function rejecting all sequences that cannot match **`sentenceSpec`**?
- Is your function rejecting all sequences with probabilities that are smaller than the highest-probability sentence found so far?

Submit for Part 2:

- **The output of your function for each of these inputs.**

Part 3 [35 marks] OPTIMIZING YOUR SENTENCE GENERATOR: EXPERIMENTING WITH DIFFERENT SEARCH STRATEGIES

In this part of the assignment you will have your function **`generate`** take a **search strategy** as another **`string`** parameter named **`searchStrategy`** between **`sentenceSpec`** and **`graph`**. Your function signature should now be:

```
String generate(
     String startingWord,
     List<String> sentenceSpec,
     String searchStrategy,
     String graph);
```

Again, your function signature may look different depending on what language you're using.

Implement the following search strategies:

1. **"BREADTH_FIRST"**. You've already implemented this strategy.
2. **"DEPTH_FIRST"**. The difference between breadth-first and depth-first is really just the difference between using a *queue* and a *stack*.
3. **"HEURISTIC"**. In a heuristic-guided search, the choice of which sentence to consider next is determined by a *heuristic function*. The standard heuristic function for A* is the Euclidean distance from the current node to the goal node, but for this assignment **you must create a heuristic function** (Euclidean distance has no meaning for this assignment).

Hints for creating a heuristic:

- Think about the "shortest possible distance" between any (incomplete) sentence and a goal (complete) sentence. Remember that the "distance" for this search problem is a probability. Think: What is the largest possible probability from any given sentence to a goal sentence?
- Recall from class that a heuristic is a "relaxed" version of the original search problem. Think: In what ways can this search problem be relaxed?
- **There are many "right" answers.** You will get points based on your reasoning process and the clarity of your code, not on whether your heuristic is "right."

Submit for Part 3:

- **Source code for your function.**
- **For each input from Part 2, and for each search strategy, submit the count of word sequences (i.e., nodes) considered during the search.**

Part 4 [15 marks] EXPLAINING AND EVALUATING YOUR SENTENCE GENERATOR

Answer the following questions:

1. Is breadth-first generally a better search strategy than depth-first for these inputs? Explain why or why not.
2. Describe the heuristic you chose for the directed search.
3. Does your heuristic guarantee that the highest-probability sentence will always be found? Explain why or why not.
4. Is it possible for your heuristic search to have worse run-time performance than a depth-first or breadth-first search? If worse performance is possible, explain what would cause a worse performance to occur. If not, explain why worse performance would never occur.

Hints: You may want to consider the following: the cost of evaluating the heuristic, the cost of prioritizing nodes, the effect of graph structure, and the effect of sentence length.

Note: There is not a single "right" answer to these questions—the mark you receive will depend on the quality of your explanations, not on whether you answer "yes" or "no." To get full marks on this part, your explanations must be clear.

Observation: Even if your sentence generator did not work perfectly, explaining the reasons why is worth marks, i.e., you "own the science" for your experiment. In real-life, scientists can get null hypotheses (i.e., failed experiments) published in top journals or conferences by rigorous explanation and evaluation. The real value in scientific research is enabling replication and reuse of your methods by other scientists.

Submit for Part 4:

- **Your answers to these questions.**

REMINDER: Please attach a copy of the Assignment 1 CoverSheet (available on our course LEARN site) signed by each team member to the front of your assignment hand-in.