CS221 Fall 2018 Homework 3

SUNet ID: 05794739
Name: Luis Perez
Collaborators:

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

Problem 1

(a) To show that this greedy algorithm is suboptimal, simply consider the following set-up. First, we work with a 1-gram model, and consider the input:

"thesecount'

The greedy algorithm will compare the following on the first iteration:

$u($"t"$)$
$u($"th"$)$
$u($"the"$)$
$u($"thes"$)$
$u($"these"$)$
$u($"thesec"$)$
$u($"theseco"$)$
$u($"thesecou"$)$
$u($"thesecoun"$)$
$u($"thesecount"$)$

From the above, it's reasonable to have our 1-gram model such that $u($"the"$) < u($"these"$)$, and both of these will obviously have lower cost than the other non-English words. Therefore, on the first iteration, our greedy algorithm will select the split:

"the secount$''$

On the second iteration, the algorithm will consider:

$u($"s"$)$
$u($"se"$)$
$u($"sec"$)$
$u($"seco"$)$
$u($"secou"$)$
$u($"secoun"$)$
$u($"secount"$)$

Note that non of these are English words, and therefore we define them to have extremely high cost. For the sake of simplicity, we'll have $u(\text{“secount”})$ have the lowest cost amongts the above, but still have an extremely high cost since it's not an English word. As such, the final output of our algorithm will be:

“the secount”

With cost $u(\text{“the”}) + u(\text{“secount”})$. However, note that the optimal split point would actually be:

“these count”

with cost $u(\text{“these”}) + u(\text{“count”})$. We note that:

$$u(\text{“these”}) + u(\text{“count”}) < u(\text{“the”}) + u(\text{“secount”})$$

mainly because of an extremely high cost associated with $u(\text{“secount”})$.

(b) In “submissions.py”.

Problem 2

(a) We simply consider the sequence of vowel-free words:

$$mnysvl.$$

Let us define first possibleFills:

$$\begin{aligned} \text{possibleFills(mny)} &= \{\text{money}, \text{many}\} \\ \text{possibleFills(s)} &= \{\text{is}, \text{sea}\} \\ \text{possibleFills(vl)} &= \{\text{evil, veil}\} \end{aligned}$$

We now consider how our greedy algorithm will proceed above.

- The algorithm will first pick “many”, given that $\text{bigramCost(-BEGIN-, money)} > \text{bigramCost(-BEGIN-, many)}$, since in English, more sentences begin with “many”.
- Next, it will select “sea”, since $\text{bigramCost(many, sea)} < \text{bigramCost(many, is)}$ give that “many is” is grammatically incorrect.
- Next, it will select “veil” since $\text{bigramCost(sea, veil)} < \text{bigramCost(sea, evil)}$, which is a somewhat reasonable assumption.

In this case, the greedy algorithm will therefore end with the output “Many sea vail” with a relatively high bigram cost, due to high cost of (many, sea), and (sea, vail).

However, note that the optimal reconstruction would have been “Money is evil” given the significantly lower costs of (money, is) and (is, evil).

(b) In “submissions.py”.

Problem 3

(a) We present the formal definition of the search problem below. We can define the states as consisting of the tuples (w, text) where w is the fully-vowelized word which was previously selected to be part of our output (or a special token signifying the start, if nothing was previously selected), and text is the sequence of characters that remain to be split into words and for vowels to be inserted. More formally, let our input query be $q = b_0 b_1 \cdots b_n$, a query string of n constants where b_0 is a special token specifying the start of a string. Then our states are:

$$\text{States}(b_0 b_1 \cdots b_n) = \{(b_0, b_1 b_2 \cdots b_n)\} \cup \bigcup_{i=1}^{n} \bigcup_{j=i}^{n} \{(w, b_j b_{j+1} \cdots b_n) \mid w \in \text{possibleFills}(b_i b_{i+1} \cdots b_{j-1} b_j)\}$$

Given the above definition of our states, it's easy to see what our start state s_{start} is:

$$s_{\text{start}} = (b_0, b_1 b_2 \cdots b_n)$$

recalling that b_0 is oure special token specifying the start of a string.

We continue by defining our Actions(s) for some state $s = (w, b_i b_{i+1} \cdots b_n)$. Intuitively, the actions simply consist of two things – picking a "split point" $i \leq j \leq n$ in the remaining characters such that we will have two substring $b_i \cdots b_j$ and $b_{j+1} \cdots b_n$ (which is the empty string when $j + 1 > n$). We can formalize the above by defininining our action space as:

$$\text{Actions}((w, b_i b_{i+1} \cdots b_n)) = \bigcup_{j=i}^{n} \{p_j \mid p_j \in \text{possibleFills}(b_i b_{i+1} \cdots b_j)\}$$

We continue expanding on the above by explicitly defining our successor function, which can be formally defined as:

$$\text{Succ}((w, b_i b_{i+1} \cdots b_n), p_j) = (p_j, b_{j+1} b_{j+1} \cdots b_n)$$

Next, we need to formally define the Cost(s, a) of taking a particular action in state s. This can be directly defined for our problem as:

$$\text{Cost}((w, b_i b_{i+1} \cdots b_n), p_j) = \text{bigramCost}(w, p_j)$$

since it is just the cost of selecting p_j as the next word.

Finally, we now simply need to consider the end test. This too can easily be defined as follows:

$$\text{IsEnd}((w, b_i b_{i+1} \cdots b_n)) = i > n$$

since we reach an end state whenever we have no further characters to split (ie, we have an empty string).

(b) In "submissions.py".

(c) We can define the unigram cost function simply as follows:

$$u_b(w) = \min_{w'} b(w', w)$$

We can pre-compute the above and store in a lookup table. This can be done once, and will take $O(|W|^2)$ where $|W|$ is the size of our word corpus. We assume this pre-computation is done for the below relaxed problem statement.

We now present the formal definition of the relaxed search problem, P_{rel} below. Let our input query be $q = b_1 \cdots b_n$, a query string of n constants.

$$\text{States}_{\text{rel}}(b_1 \cdots b_n) = \bigcup_{i=1}^{n} \{(b_i b_{i+1} \cdots b_n)\}$$

Given the above definition of our states, it's easy to see what our start state s_{start} is:

$$s_{\text{rel}_{\text{start}}} = (b_1 b_2 \cdots b_n)$$

We continue by defining our Actions(s) for some state $s = (b_i b_{i+1} \cdots b_n)$. Intuitively, the actions simply consist of two things – picking a "split point" $i \leq j \leq n$ in the remaining characters such that we will have two substring $b_i \cdots b_j$ and $b_{j+1} \cdots b_n$ (which is the empty string when $j + 1 > n$). We can formalize the above by defininining our action space as:

$$\text{Actions}_{\text{rel}}(b_i b_{i+1} \cdots b_n) = \bigcup_{j=i}^{n} \{p_j \mid p_j \in \text{possibleFills}(b_i b_{i+1} \cdots b_j)\}$$

We continue expanding on the above by explicitly defining our successor function, which can be formally defined as:

$$\text{Succ}_{\text{rel}}((b_i b_{i+1} \cdots b_n), p_j) = (b_{j+1} b_{j+1} \cdots b_n)$$

Next, we need to formally define the Cost(s, a) of taking a particular action in state s. This can be directly defined for our relaxed problem as:

$$\text{Cost}_{\text{rel}}((b_i b_{i+1} \cdots b_n), p_j) = u_b(p_j)$$

since it is just the unigram cost of selecting p_j as the next word.

Finally, we now simply need to consider the end test. This too can easily be defined as follows:

$$\text{IsEnd}_{\text{rel}}((b_i b_{i+1} \cdots b_n)) = i > n$$

since we reach an end state whenever we have no further characters to split (ie, we have an empty string).

With the above relaxed problem, it's quite easy to see that:

$$\begin{aligned}
\text{Cost}_{\text{rel}}((b_i b_{i+1} \cdots b_n), p_j) &= u_b(p_j) && \text{(definition of Cost)} \\
&= \min_{w'} b(w', p_j) && \text{(definition of } u_b\text{)} \\
&\leq b(w, p_j) \\
&\qquad \text{(where } w \text{ is the previous word picked, as in original problem)} \\
&= \text{Cost}((w, b_i b_{i+1} \cdots b_n), p_j)
\end{aligned}$$

(d)

- Yes, UCS is a special case of A^*. We can see this my nothing UCS is simply $A*$ where we have the same states, actions, starting state, end states, and costs. However, for UCS, we must have that the heuristic function h is given by $h(s) = 0$. Therefore UCS is a less generic algorithm.
- Yes, BFS is a special case of UCS. BFS is simply UCS where we have the same states, actions, starting state, and end states. However, for BFS, we have that $\text{Cost}(s, a) = 1$ always (ie, our cost function is one). Therefore, BFS is a special case of UCS. Thefore BFS is a less generic algorithm.