CS 221 – Autumn 2018
Anand Natu
anatu - 06264867

Homework #3: Reconstruct

Problem 1

Problem 1a

The greedy algorithm is sub-optimal as a result of its iterative nature, and in turn its dependence on the text that the language model has processed in order to develop its fluency. The "blind" forward iteration means that the greedy algorithm misses words which fall between the cracks of its iterative sequence which may actually have a cost that is even lower than what it finds.

Take the example sequence "testingotofmetal". Suppose we have a language model for which n=1 such that some of its costs are defined as:

$$u(w) = \{ testing: 10, test: 11, ingot: 5, metal: 3, of: 5, \{ot, oto, \dots otofmetal\}: 100\ (Not\ in\ vocab)\}$$

The forward pass of the greedy algorithm from the first letter T will find "testing" to be the cost-minimizing split, and then will proceed to look for the next minimizing sequence from the next letter following "testing". However, now that it has made a split at testing, it has placed itself in a position where its forward outlook of splits consists entirely of grams that aren't in the model. Therefore, the minimum cost the greedy model can obtain is

$$u_{min,greedy} = testing + (Not\ in\ vocab) = 10 + 100 = 110$$

Meanwhile, visual inspection quickly reveals that the "true" minimum cost is the natural-English interpretation

$$u_{min,true} = test + ingot + of + metal = 11 + 5 + 5 + 3 = 24$$

Therefore, **the greedy algorithm is suboptimal because it cannot account for a case where one path initially seems cheaper, but actually ends up being more expensive than another path that initially seems expensive, but ends up being cheaper.**

Problem 2

Problem 2a

Consider the example vowel-free sequence "vngr", and suppose that our language model of interest is fluent in the words "avenger" as well as "avantgarde" (assume our model automatically drops non-alphabetical characters in its fluency training). Also assume that the cost of "ave" is lower than the cost of "ava", but the cost of "avenger" is higher than the cost of "avantgarde".

Now, under the greedy algorithm, the vowel completion will first place an "a" and an "e", forming the word "ave" as it is the lowest-cost operation for that vowel placement. Now that the "e" has been placed, the lowest-cost vowel placement it can make afterwards is two more "e"s to spell "avenger". **The greedy algorithm is sub-optimal because it does not account for anything ahead of its current window – therefore, it cannot see the downstream cost impact of any given vowel placement, which**

is why it cannot preemptively place "a" as the second vowel to spell the lowest-cost word, "avantgarde".

Problem 3

Problem 3a

- **States:** The state for the problem is a tuple containing two elements: First, the number of characters from the original string that have already been processed, and second, the last modified word which was inserted into the sequence so the bigram cost can be computed in the state-space. For example, given the input string "mgnllthppl", a sample state might be (3, "imagine").
- **Actions:** The action for each state is a particular combination of a) Splitting the "horizon" string into a particular substring, and b) a population of that substring with possible vowels. For example, a possible action on the frontier string "llthppl" would be splitting into the substring "ll" then populating it with vowels to form "all".
- **Costs:** The cost of each action will be the cost of the bigram produced by the action. The first word in the bigram comes from the state-space (which stores memory of the previously inserted word), and the second word results from the substring formation and vowel insertion described above. Therefore, the cost of the action described above would be the cost of the bigram "imagine all", as assessed by the given bigram cost function.
- **Initial State:** The initial state of the problem is zero (since no characters from the original string have been modified), and the special token signaling the start of a sentence, i.e. (0, "-BEGIN-").
- **End State:** The end state of the problem is when: first, the number of characters processed (the first element of the state tuple) is equal to the length of the original query; second, the working state is not equal to the initial state.

Problem 3c

For the given unigram cost function, we will essentially reduce the problem to only look at a single candidate word at a time, since we are no longer using a bigram function in the relaxed problem. To do so, we will define the new unigram cost function as

$$\boldsymbol{u_b(w) = \min_{i \in (vocab)} \{b(i,w), b(w,i)\}}$$

I.e. the cost of the unigram in the relaxed state (which estimates the future costs) is the cost of the most common bigram which that word appears in. To test for consistency of the heuristic, we require that the relaxed problem always yield lower costs for any given action (i.e. $Cost_{rel}(s,a) \leq Cost(s,a)\ \forall\ (s,a)$). Since we have chosen the most common bigram to approximate future costs, we know that this condition holds for our selection.

Using this new cost function, our state-space representation is as follows:

- **States:** The state for the problem is now a single integer which represents the number of characters in the input string that we have processed. So, given the input string "mgnllthppl", our relaxed sample state goes from (3, "imagine") to just 3.

- **Actions:** The action for the relaxed problem is unchanged, as it still involves splitting the input string into a particular substring, and populating the substring with a set of possible vowels. For example, a possible action on the frontier string "llthppl" would be splitting into the substring "ll" then populating it with vowels to form "all".
- **Costs:** The cost of each action in the relaxed problem is now simply the cost obtained using the cost function described above. Therefore for the word "all" derived from the action described above, the cost is the cost of the most common (lowest-cost) bigram in which "all" appears.
- **Initial State:** The initial state of the problem is simply zero, since at outset no characters from the original string have been modified.
- **End State:** The end state of the problem is when the number of characters processed equals the length of the original input query, so we know that we have explored paths throughout the entire query.

Problem 3d

UCS vs A*

Yes, UCS is a special case of A*. This is because A* is originally formulated as a variation on UCS, in which the search costs can be written as

$$Cost'(s,a) = Cost(s,a) + h\big(succ(s,a)\big) - h(s)$$

Where $Cost(s,a)$ represents the cost of the original UCS problem, and $h(s)$ is a heuristic function which biases or modifies the edge costs to favor search paths which approach the optimal solution. As a result, it follows from the above that **UCS is a special case of A* in which the heuristic, i.e. the degree of bias applied to the edge costs, is zero.**

BFS vs UCS

BFS is a special case of UCS. UCS works by creating a "priority queue" in which nodes that exist on the frontier are ranked according to their cost, and proceeds to visit nodes in ascending order of cost. BFS, on the other hand, stack-ranks nodes according to their level of depth, and searches all nodes at a given level of depth before proceeding to the next. Therefore, **BFS is essentially a special case of UCS in which all of the edge costs of the graph are all equal to 1.**