CS221 Summer 2019 Homework 3

SUNet ID: jaigupta
Name: Jai Gupta

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) **Greedy is suboptimal**
Consider the following case:

$$query = batmanwon$$

Let the unigram cost function be defined by the following table

word	**cost**
bat	10
man	10
batman	15
won	10

Let us also assume that any word that is not present in the above table has a cost of $1,000,000$ (or infinity), basically so large such that these transitions should never be chosen unless there is no alternative present in the above table.

At the start, we have two choices: "bat" and "batman".

- On choosing "bat" (cost = 10): we will have to choose "man" as the next word and "won" as the following word creating the final segmented sentence as "bat man won". This one has a total cost of 10 + 10 + 10 = 30.
- On choosing "batman" (cost = 15): we will have to choose "won" as the next word. Segmented sentence is this case is "batman won" with a cost of 15 + 10 = 25.

Since the initial cost of choosing "bat" is lower (10) compared to the cost of choosing "batman" (15), the greedy algorithm will choose "bat". This is suboptimal since there is a better way of performing this segmentation as mentioned above to "batman won" which has lower total cost (30 vs 25).

Even if we use a bigram cost function, lets take the following cost:

word	cost
–BEGIN–, bat	10
–BEGIN–, batman	15
bat,man	10
man,won	10
batman,won	10
anything else	1,000,000

In the start, we will have two choices, "bat" (cost = 10) and "batman" (cost = 15). Choosing lower cost option "bat" will lead to the sentence "bat man won" (cost = 10 + 10 + 10 = 30), while choosing the option "batman" will lead to the sentence "batman won" (cost = 15 + 10 = 25).

(b) in submission.py

Problem 2

(a) Consider the following example:
query_words = {"*mn*","*r*","*wrkng*"} And let us assume the following mapping for possibleFills function:

word	possible fills
mn	man, men
r	are
wrkng	working

Also let us assume the following bigram cost function:

pairs	cost
–BEGIN–, man	10
–BEGIN–, men	15
man, are	100
men, are	10
are, working	10
anything else	1,000,000

To start with, we have two choices for expansion of "mn": "man" and "men". According to the bigram cost function, "man" is more favorable (cost = 10) compared to "men" (cost = 15). Hence, greedy approach will choose "man". After that, the choices are forced to get the final sentence as "man are working" with a cost of 10 + 100 + 10 = 120.

On the other hand, if we chose "men" over "man", the initial cost would have been higher (cost = 15). But the eventual sentence "men are working" would have had a much lower total cost of 15 + 10 + 10 = 35 only.

(b) in submission.py

Problem 3

(a) *States*: States must hold the minimum amount of information required to find optimal solution. In this case the optimal solution will require the following in the states:

(a) Number of characters consumed from the input: This ensures that we start the next word from the right position (and also detect end state).

(b) The last word (after vowel filling) in the output: This is required to feed the right words in the bigram cost function to find the least cost choice for the next word.

Actions: An action on a state consists of:

(a) choosing any number of non-zero characters following the end position of the state.

(b) checking possibleFills for a possible completion and using that as an output for next word.

In code, an action is represented by the string denoting the output word.

Costs: Cost is the bigramCost function for the last word of the current output (present in the current state) and the new word being added from possibleFill's outputs.

Initial State: Initial state is the pair (0, '–BEGIN–') since we have consumed 0 characters from the input and '–BEGIN–' denotes the start of sentence.

End State: End state is any state that has consumed the complete input query. This means the first elements of the state is equal to the length of the input query.

As mentioned above, we chose states storing minimal amount of information that is required to find the optimal solution and the rationale is presented above.

(b) in submission.py

(c) As mentioned in hints for the question, we want to find a relaxed problem for creating a A* heuristic function such that:

1. States don't not need to keep track of the last output word.

2. The cost function is independent of the last output word.

But we are given the bigram cost function, which depends on the last output word. So we need to define a new function that is not dependent of the last output word and is

consistent. One way is to make sure that $Cost_{rel}(s, a) \leq Cost(s, a)$. If the new word added by the action is w, we can define the new function as:

$$Cost_{rel}(s, a) = u_b(w) = \min_{w'} b(w', w)$$

We clam that this is a simpler problem to evaluate because the size of the state space has reduced significantly due to removed dependency on the last word of the previous state.

Proof that u_b is consistent:

If at a state s, we have the last output word as w_{prev}, and an action a causes insertion of a new word w, then:

$Cost(s, a) = b(w_{prev}, w)$

but $b(w_prev, w)$ must be greater than or equal to min of $b(w', w)$ over all possible values of w' from corpus. Hence:

$$Cost(s, a) \geq \min_{w'} b(w', w)$$
$$Cost(s, a) \geq u_b(w', w)$$
$$Cost(s, a) \geq Cost_{rel}(s, a)$$

Formalizing the above problem:
States: States store the number of characters of the input consumed. Hence, there are $query\_length + 1$ states.
Actions: Action on a state consists of choosing any number of characters following the end index of current state, passing it through possibleFills and choosing one possible fill.
Cost: Unigram cost u_b defined above for the word being added by the action (chosen output of possibleFills).
Start state: 0, since no input character has been consumed.
End state: $query\_length$, i.e., when all the input characters have been consumed.

Our heuristic function is $h(s) = FutureCost_{rel}(s, a)$.

If $FutureCost\{rel(s, a)$ takes some path $s_1, s_2, ...s_k...$ before reaching the end stage, then it will be the sum of $Cost_{rel}(s_k, a_k)$.

$$\begin{aligned} h(s) &= FutureCost_{rel}(s, a) \\ &= \sum_k Cost_{rel}(s_k, a_k) \\ &\leq \sum_k Cost(s_k, a_k) \\ h(s) &\leq FutureCost(s, a) \end{aligned}$$

This signifies that h(s) is admissible.

Also:

$$\begin{aligned} Cost'(s,a) &= Cost(s,a) + h(Succ(s,a)) - h(s) \\ &= Cost(s,a) - (h(s) - h(Succ(s,a)) \\ &= Cost(s,a) - Cost_{rel}(s,a) \\ &\geq 0 \end{aligned}$$

This too proves that h(s) is consistent.

(d) *Is UCS a special case of A*? Explain why or why not.*

USCS can be considered as a special case of A*. If the heuristic function for A* is zero for all inputs, then A* will essentially boil down to solving just the problem without using any heuristic function which is same as UCS.

Is BFS a special case of UCS? Explain why or why not.

UCS can solve all problems solved by BFS. The problem space of BFS is a subset of problem space for UCS, since compared to UCS, just all edge weights are same for BFS. Note that a small variant of BFS can also handle the case when some of the edge weights are zero, but that does not affect our solution.

Still, BFS takes advantage of the fact that the weights are same and hence can pick the smallest state to process more efficiently by simply replacing the priority queue with a regular queue. Therefore, BFS is not a special case of UCS.