Assignment 3. Search Problem

Hwanjo Yu
CSED342 - Artificial Intelligence

Contact: TA Jinhwan Nam (njh18@postech.ac.kr)
Deadline: 1 Apr 2024 at 2:00 pm

General Instructions

This assignment has been developed in **Python 3.8**, so please use **Python 3.8** to implement your code. we recommend using **Conda environment**.[1]

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

```
# BEGIN_YOUR_ANSWER
```

and

```
# END_YOUR_ANSWER
```

You can add other helper functions outside the answer block if you want, but do not import other libraries and do not make changes to files other than `submission.py`.

Your code will be evaluated on two types of test cases, **basic** and **hidden**, which you can see in `grader.py`. Basic tests, which are fully provided, 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

[1] https://docs.conda.io/projects/conda/en/stable/user-guide/install/index.html

provided in `grader.py`, but the correct outputs are not. To run all the tests, type

```
python grader.py
```

This will tell you only whether you passed the basic tests. The script will alert you if your code takes too long or crashes on the hidden tests, but does not say whether you got the correct output. You can also run a single test (e.g., `2a-1-basic`) by typing

```
python grader.py 2a-1-basic
```

We strongly encourage you to read and understand the test cases, create your own test cases, and not just blindly run `grader.py`.

Problems

Here, we are going to solve two problems, **Text reconstruction** and **Maze search**. Both are search problems, which are defined by **state**, **action**, **cost**, and **successor**. The object is to find the minimum cost path from a starting state to the goal.

Problem 1. Text reconstruction

In this problem, we consider two tasks: word segmentation and vowel insertion. Word segmentation often comes up in processing many non-English languages, in which words might not be flanked by spaces on either end, such as in written Chinese or in long compound German words.

Vowel insertion is relevant in languages such as Arabic or Hebrew, for example, where modern script eschews notations for vowel sounds and the human reader infers them from context. More generally, it is an instance of a reconstruction problem given 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.

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 by estimating the likelihood of text,

$$p(w_1, w_2, ..., w_{N-1}, w_N) = \sum_{i=1}^{N} p(w_i | w_1, ..., w_{i-1}).$$

A very common language model in NLP is an n-gram sequence model, which assumes $p(w_i|w_1, ..., w_{i-1}) = p(w_i|w_{i-(n-1)}, ..., w_{i-1})$. We'll use the n-gram model's negative log-likelihood ($-\log p(w_i|w_{i-(n-1)}, ..., w_{i-1})$) as a cost function $c(w_{i-(n-1)}, ..., w_{i-1}, w_i)$. The cost will always be positive, and lower costs indicate better fluency. 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

$$c(w_1, w_2, w_3, w_4) = u(w_1) + u(w_2) + u(w_3) + u(w_4).$$

For a bigram model b ($n = 2$), the cost assigned to $[w_1, w_2, w_3, w_4]$ is

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

where w_0 is a special token that represents the beginning of the sentence. Here, u and b are provided as a trained unigram and bigram language model. All the costs returned by the models are non-zero. Also, any words not in the corpus are automatically assigned a high cost, so you don't have to worry about cost exploding to infinity.

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.

Problem 1.a [10 points]

Implement an algorithm that finds the optimal word segmentation of an input character sequence. Your algorithm will consider costs based simply on a unigram cost function.

Before jumping into code, you should think about how to frame this problem as a state-space search problem. How would you represent a state? What are the successors of a state? What are the state transition costs?

Uniform cost search (UCS) is implemented for you in `util.py`, and you should make use of it here. Fill in the member functions of the `WordSegmentationProblem` 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 the inputs would be in lowercase. The function `segmentWords` should return the segmented sentence with spaces as delimiters, i.e. `' '.join(words)`.

For convenience, you can run `python 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. For example:

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

Console commands other than `seg` – namely `ins` and `both` – will be used for the upcoming parts of the assignment. Other commands that might help with debugging can be found by typing help at the prompt.

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). For example, `possibleFills('fg')` returns `set(['fugue', 'fog'])`.

Problem 1.b [10 points]

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 an empty string is a valid element of the list.

The argument `queryWords` is the input sequence of vowel-free words. 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; it 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 fills, e.g. `chclt → {}`, where chocolate is a valid fills. 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.

Putting it together

We'll now see that it's possible to solve both of these tasks at once. This time, you are given a whitespace- and vowel-free string of alphabetical characters. Your goal is to insert spaces and vowels into this string such that the result is the most fluent possible one. As in the previous task, costs are based on a bigram cost function.

Problem 1.c [15 points]

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

When you've completed your implementation, the function `segmentAndInsert` should return a segmented and reconstructed word sequence as a string with space delimiters, i.e. `' '.join(filledWords)`.

The argument `query` is the input string of space- and vowel-free words. 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; it takes a word as string and returns a set of reconstructions.

Note: Unlike in problem 1.b, where a vowel-free word could (under certain circumstances) be considered a valid reconstruction of itself, here you should only include in your output words that are the reconstruction of some vowel-free word according to `possibleFills`. Additionally, you should not include words containing only vowels such as "a" or "I"; all words should include at least one consonant from the input string.

Use the command `both` in the program console to try your implementation. Similar to ins command, vowels are striped and spaces are also ignored. For example:

```
>> both imagine all the people
   Query (both): mgnllthppl
   imagine all the people
```

Problem 2. Maze search

In this problem, we consider searching a 2D grid maze. It's a common task to find an exit path with the minimum cost.

Starting from the first state, the agent can take any action from `possibleMoves`. To see how action corresponds to the actual move, please check `util.directions`. If a valid action is chosen, the agent moves toward the direction with predefined cost. However, every maze has a wall that the agent can pass but at a high cost. Also, the size of a maze is finite, which limits its state space.

Problem 2.a [3 points]

Implement an algorithm that finds the minimum cost of a path from the starting point to the goal.

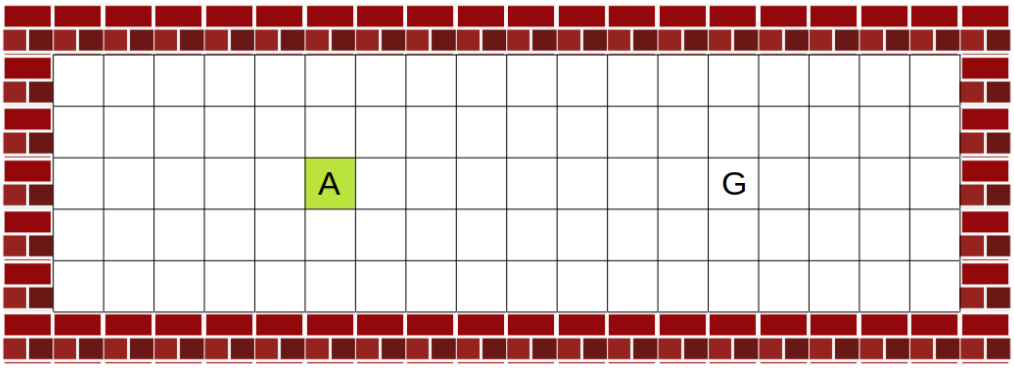
Fill in the member functions of the `MazeProblem` and the `UCSMazeSearch` function. When you've completed your implementation, the function `UCSMazeSearch` should return a cost of the shortest path. The arguments `start` and `goal` are the coordinates of the starting point and the goal in a maze. As we are going to explore a 2D maze, each coordinate is represented as a tuple of two integer numbers. The argument `moveCost` is a function that takes in two

coordinates and outputs a cost from the first coordinate to the second coordinate. The argument `possibleMoves` is a function that takes in a single coordinate and outputs a set of possible moves and their costs.

Maze search with A* search algorithm

Until now, maze search is not that different from text reconstruction. However, with uniform cost search used in text reconstruction, the agent has to search a lot of states that might be meaningless in a high-level perception.

Consider the following figure showing a grid maze, where **A** and **G** represent the current positions of the agent and the goal.

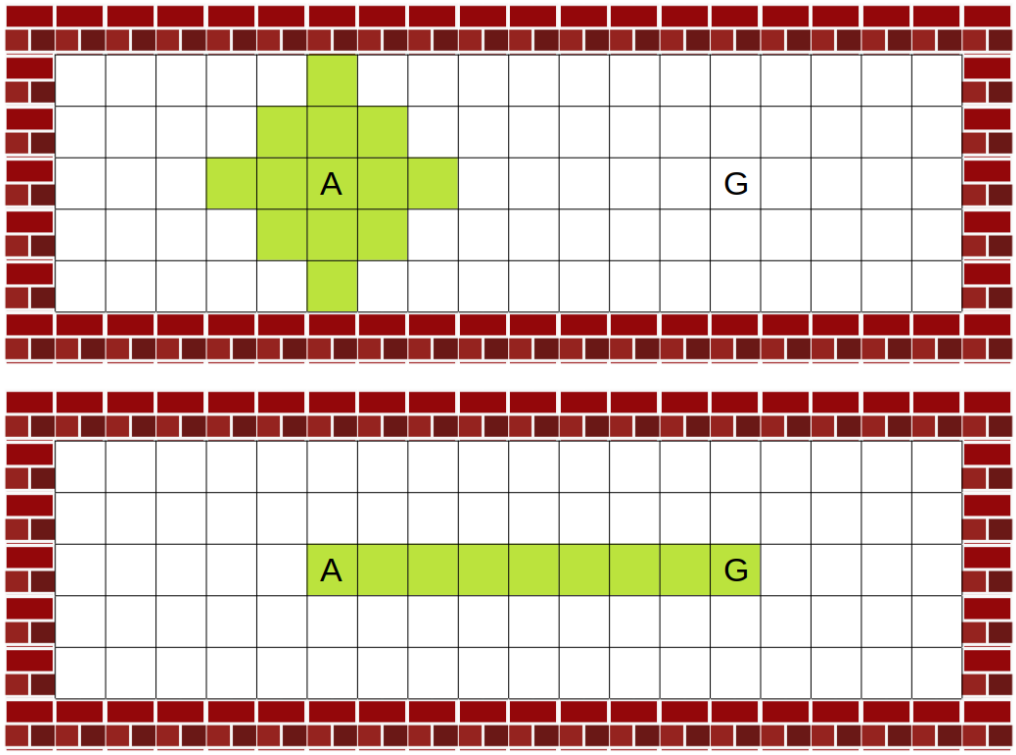


With a given figure, humans can notice that going straight to the goal is the shortest path. However, with uniform cost search, the agent does not know about such a strategy. Instead, it will move around all the nearby cells.

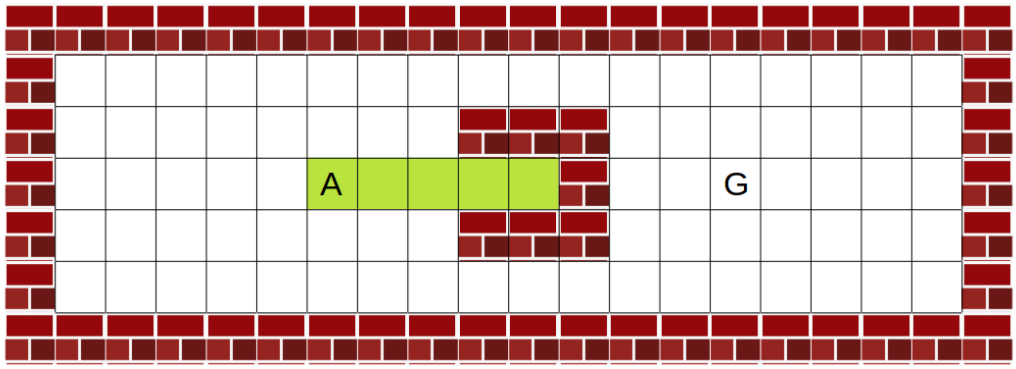
To prevent such explorations, we can use A* search with a heuristic cost function, instead of uniform cost search. If heuristic cost is consistent, A* search always finds the shortest path with less exploration compared to uniform cost search.

As the term 'heuristic' implies, depending on a heuristic cost function, A* search differs in time to find the shortest path.

With a naïve heuristic, finding the shortest path can be non-optimal. Still, it will find the shortest path if the given heuristic cost function is consistent.



up: with UCS, down: with A* search. Green tiles represent visited states.



Heuristic used for A* search can be non-optimal. Still, it will find the shortest path if it is consistent.

Problem 2.b [12 points]

Implement a heuristic cost function that is consistent with the maze search problem.

Fill in the function `consistentHeuristic`. It is used as a heuristic cost function for this problem, which is an input argument of `util.UniformCostSearch.solve`. When you've completed your implementation, the function `AStarMazeSearch` should return a cost of the shortest path. Note that if your heuristic cost function is well-defined, A* search should be faster than UCS.