Task 1 - Secret Messages

Can you guess what the encoded message below says?

Tha rein in Spein fells meinly in tha mounteins, not pleins

If you got it, nice work! If not, don't worry – you'll have a program to do this very soon! The answer is, "The rain in Spain falls mainly in the mountains, not plains", and we can get this by replacing all of the e's in the encoded message with a's, and vice-versa. (i.e. **A** ↔ **E**)

Let's try another one. Can you guess what the message below says?

I lofe ctudying artivisial intelligense

This one is harder, because there are two pairs of swapped letters: **V** ↔ **F** and **S** ↔ **C**. If we reverse these swaps, then the answer is, "I love studying artificial intelligence", (which we really hope is true!). One more puzzle: if you start with the message, "Cabs are taxis.", and you apply the swaps **A** ↔ **B**, and then **B** ↔ **C**, what encoded message would you get?

The answer is, "Bcas cre tcxis"

- Start: Cabs are taxis
- Swap **A** ↔ **B**: Cbas bre tbxis (adjust colour)
- Swap **B** ↔ **C**: Bcas cre tcxis (adjust colour)

Notice that the encoded messages so far still resemble the original message, because we haven't swapped many letters. However, if we continue to add swaps, the messages will become harder to read, so it would be nice to have a program to help us out.

For this task, you will write a function to encode and decode messages using the above letter swapping method (which is the how the secret message in the introduction was encoded). The function should have three parameters:

1. A string specifying the *key* (i.e. the sequence of letter swaps). For example, "AEGHAG", would mean we should apply the swaps **A** ↔ **E**, **G** ↔ **H**, then **A** ↔ **G** if we're encoding, or the reverse (**A** ↔ **G**, **G** ↔ **H**, then **A** ↔ **E**) if we're decoding. Note that "AEGHAG" is the same as "EAGHAG", since **A** ↔ **E** is the same as **E** ↔ **A**.
2. The name of a text file containing the message to be encoded or decoded.
3. Either 'e' or 'd' indicating whether to encode or decode, respectively.

The function will return the resulting encoded or decoded message as a string, with capitalisation, punctuation and spacing preserved. Here are some example calls to the function:

```
>>> print(task1('AE', 'spain.txt', 'd'))
The rain in Spain falls mainly in the mountains, not plains.
>>> print(task1('VFSC', 'ai.txt', 'd'))
```

```
I love studying artificial intelligence.
>>> print(task1('ABBC', 'cabs_plain.txt', 'e'))
Bcas cre tcxis.
```

Task 2 - Search Space

Congratulations! We can now encrypt and decrypt messages if we have the key (i.e. the sequence of letters to swap). However, what happens if we don't have the key? Well, as the name of this assignment suggests, we'll have to search for one! In this task, we'll look at how we can represent our search space as a tree and we'll also work on a program to generate child nodes for that tree. This will be very helpful when we come to implement our search algorithms later.

Before starting, let's revise the key elements of a search problem from the lecture slides:

Search Problem Formulation

- **A search problem is defined by 4 items**
  - **1) Initial state, e.g. Arad**
  - **2) Goal state (1 or more)**
    - **Stated explicitly, e.g. Bucharest**
    - **Stated implicitly with a goal test, e.g. checkmate state**
  - **3) Operators = actions**
    - **A set of possible actions transforming 1 state to another , e.g. Arad->Zerind, Arad->Sibiu, etc.**
  - **4) Path cost function – assigns a numeric value to each path**
    - **Reflects the performance measure**
    - **E.g. time, path length in km, number of operators executed**
- **Solution – a path from the initial to a goal state**
  - **Its quality is measured by the path cost**
  - **Optimal solution is the one with the lowest path cost**
- **State space - all states reachable from the initial state by operators**

In our case, the initial state is the encrypted message. Can you work out what each of the other elements (i.e. goal state, operators and path cost function) should be?

The answers are—wait! Are you sure you want to read on? Thinking about these questions is a great exercise (and helpful for the exam 😊). If yes, the answers are as follows: 1) the goal state is the decoded message, 2) the operators are the letter swaps (e.g. A ↔ E), since these transform messages into other messages and 3) the path cost is the number of letter swaps (e.g. if we applied A ↔ E, then E ↔ B, that would have a cost of 2.

Now that we have formulated our search problem, we can start setting up tools to help us with the search. In this task, you will write a function to find all of the successors of a state in our search space, given a set of allowed letters to swap. The function should have two parameters:

1. The name of a text file containing the parent state
2. A string containing all letters that are allowed to be swapped. For example, "ABC" would mean A ↔ B, A ↔ C and B ↔ C are allowed, but nothing else. Note that we are adding this condition so we can make the state space smaller, which will help with debugging. This will also be useful when we come to decoding the secret message.

The function will return a string which includes the number of successor states, followed by a list of these states separated by lines. The successors should be generated by applying the allowed operators in alphabetical order. For example, all of the A swaps (e.g. A ↔ B, A ↔ C, A ↔ D… etc.) should come before the B swaps (e.g. B ↔ C, B ↔ D, B ↔ E etc.). Additionally, A ↔ B should come before A ↔ C., since B comes before C. There is no need to include repeats (e.g. we don't need B ↔ A, since it is the same as A ↔ B), or operators that do nothing (e.g. A ↔ A always does nothing, and A ↔ B does nothing if the message doesn't contain any A's or B's). Some examples are given below.

```
>>> print(task2('spain.txt', 'ABE'))
3
Thb rein in Spein fells meinly in thb mounteins, not pleins.

The rain in Spain falls mainly in the mountains, not plains.

Tha rbin in Spbin fblls mbinly in tha mountbins, not plbins.
>>> print(task2('ai.txt', 'XZ'))
0
>>> print(task3('cabs.txt', 'ABZD'))
5
Acbs cre tcxis.

Bcds cre tcxis.

Bczs cre tcxis.

Dcas cre tcxis.

Zcas cre tcxis.
```

Note: you can adapt your code from Task 1 to help you here.

Task 3 - Goal

Excellent work! Now that we have our successor state program, we're almost ready to search! We just need one more ingredient – a goal test! In this task, you will write a function to check if a given message is valid English, by comparing it to a common English word list. The function should take three **inputs**:

1. The name of a text file containing the message
2. The name of a text file containing a list of words, in alphabetical order and each on a separate line, which will act as a dictionary of correct words
3. A threshold, t, specifying what percentage of words must be correct for this to count as a goal (given as an integer between 0 and 100). The threshold is important, because we may need a buffer if our dictionary is missing words, or there are some misspelt words in the message.

The function should return a string containing two lines of text. The first line should be "True" if at least t% of the words in the message are correct according to the dictionary and "False" otherwise. The second line should be the percentage of words that were correct, to 2 decimal places (round off any further decimal places; 0.005 rounds up to 0.01). Some examples are given below.

```
>>> print(task3('jingle_bells.txt', 'dict_xmas.txt', 90))
True
90.00
>>> print(task3('fruit_ode.txt', 'dict_fruit.txt', 80))
False
50.00
>>> print(task3('amazing_poetry.txt', 'common_words.txt', 95))
True
95.65
```

Dictionary matching is case insensitive; if the dictionary contained only the word 'apple', then 'Apple', 'apple', and 'aPPle' in the message should all count as correct words according to the dictionary. Words are separated by whitespace (space and newline characters).

Task 4 - DFS, BFS, IDS, UCS

Fantastic! We now have tools to help us generate children and to perform goal checks. In this task, you will now combine all your work so far to write a function to perform uninformed searches. It should take six **inputs**:

1. A character (d, b, i or u) specifying the algorithm (DFS, BFS, IDS and UCS, respectively)
2. The name of a text file containing a secret message
3. The name of a text file containing a list of words, in alphabetical order and each on a separate line, which will act as a dictionary of correct words
4. A threshold, t, specifying what percentage of words must be correct for this to count as a goal (given as an integer between 0 and 100).
5. A string containing the letters that are allowed to be swapped
6. A character (y or n) indicating whether to print the messages corresponding to the first 10 expanded nodes.

It should then perform DFS, BFS, IDS or UCS to search for a decryption to the given message, reusing your code from previous tasks if you would like to. Note that children should be generated in the same **order** as in Task 2, and you do not need to handle cycles. In the case of UCS, if two nodes have the same priority for expansion, you should expand the node that was added to the fringe first, first. Additionally, you should stop the search if 1000 nodes have been expanded without finding a solution.

The function should **return** a string. This string must contain the following information, in order:

1. The decrypted message, key for generating that message and the path cost, if a solution was found. If no solution was found, the program should print, "No solution found."
2. The number of nodes expanded during the search. Note that the start node counts as an expanded node and, in the case of IDS, the final expanded node count should be the sum of the expanded node counts on each iteration.
3. The maximum number of nodes in the fringe at the same time during the search
4. The maximum search depth reached. That is, the depth of the deepest expanded node. Note that the start node has a depth of 0, and its children have depths of 1.
5. (If indicated with y) the messages corresponding to the first 10 expanded nodes in the search. If less than 10 nodes were expanded, it should print all expanded nodes.

Some examples of function calls and results are given below.

```
>>> print(task4('d', 'cabs.txt', 'common_words.txt', 100, 'ABC', 'y'))

No solution found.


Num nodes expanded: 1000

Max fringe size: 2001

Max depth: 999


First few expanded states:

Bcas cre tcxis.
```

Acbs cre tcxis.

Bcas cre tcxis.

Acbs cre tcxis.

Bcas cre tcxis.

Acbs cre tcxis.

Bcas cre tcxis.

Acbs cre tcxis.

Bcas cre tcxis.

Acbs cre tcxis.
>>> print(task4('b', 'cabs.txt', 'common_words.txt', 100, 'ABC', 'y'))
Solution: Cabs are taxis.

Key: ABAC
Path Cost: 2

Num nodes expanded: 6
Max fringe size: 11
Max depth: 2

First few expanded states:
Bcas cre tcxis.

Acbs cre tcxis.

Bacs are taxis.

Cbas bre tbxis.

Bcas cre tcxis.

Cabs are taxis.

>>> print(task4('i', 'cabs.txt', 'common_words.txt', 100, 'ABC', 'y'))

Solution: Cabs are taxis.

Key: ABAC

Path Cost: 2

Num nodes expanded: 9

Max fringe size: 5

Max depth: 2

First few expanded states:

Bcas cre tcxis.

Bcas cre tcxis.

Acbs cre tcxis.

Bacs are taxis.

Cbas bre tbxis.

Bcas cre tcxis.

Acbs cre tcxis.

Bcas cre tcxis.

Cabs are taxis.

Task 5 - Heuristics

How exciting! We've programmed our very own search algorithms! As a reward, here's a secret: the message in the introduction was generated by only swapping the letters, "A", "E", "N", "O", "S" and "T"!

But there's a problem: if we try running our task 4 program using just these letters, we'll find that none of our four search algorithms actually reaches a solution. We're going to need something more efficient, so let's try some informed search strategies. We need a heuristic. In this task, we will start by developing a heuristic based on the frequency of English letters. This is the idea: imagine you counted the frequencies of the letters in the secret message and found that X was most common. Then, you counted the frequencies of letters in normal English texts, and found that E was most common. Could you guess what X in the secret message stood for? (Yes! E!) We will use this idea when developing our heuristic.

(By the way, the process of comparing letter frequencies to decrypt messages is called frequency analysis, and it can be applied even when the message has no spaces, punctuation or capitalisation).

According to this table, if we sort the English letters from most frequent to least frequent, we get E T A O I N S H R D L… If we limit that to just the letters A E N O S and T (which are the only ones swapped in the secret message), then the ordering becomes E T A O N S. Your task is to write a function that compares this theoretical ordering to the letter ordering in a given message, then estimates how many letter swaps would be needed to make them the same. The function should take two **inputs**:

1. The name of a text file containing the message
2. A boolean (either True or False) indicating whether this message corresponds to a goal node. (We need this because, to be valid, a heuristic must always estimate the cost at a goal node to be 0)

The program should output 0 if this is a goal node. Otherwise, it should count how many times the letters A, E, N, O, S, and T occur in the message and sort them from most common to least common. For example, if T was the most common letter in the message, followed by E, then O, then A, then S, then N, then the sorted string would be TEOASN. Note that, if two letters have the same frequency, you should use alphabetical order to break ties (e.g. A comes before E).

The program should then compare this sorted string to the theoretical goal (ETAONS) and count how many letters are in the wrong place. For example, all 6 letters are in the wrong place in TEOASN, but only three are wrong for TAEONS. Finally, the **output** heuristic value should be ceiling(n/2), where n is the number of letters out of place, and the ceiling function rounds up to the nearest integer. Thus we roughly estimate how many swaps we need to make the ordering the same. Some example function calls and results are given below.

```
>>> print(task5('freq_eg1.txt', False))
```

```
3
>>> print(task5('freq_eg1.txt', True))
0
>>> print(task5('freq_eg2.txt', False))
2
```

Task 6 - Greedy, A*

In this final task, you should modify your solution to Task 4 to include the greedy and A* algorithms. The input and output should be in exactly the same format. The only difference is that the first input can now be d, b, i, u, g or a, where g indicates greedy search and a indicates A* search. Use the heuristic we developed in Task 5 for these informed search strategies.

Once you are finished, try running your greedy and A* searches with the following inputs to decrypt the secret message 😊:

```
>>> task6('g', 'secret_msg.txt', 'common_words.txt', 90, 'AENOST', 'n')
>>> task6('a', 'secret_msg.txt', 'common_words.txt', 90, 'AENOST', 'n')
```