CS1132 Spring 2009 Assignment 2

Adhere to the Code of Academic Integrity. You may discuss background issues and general solution strategies with others and seek help from course staff, but the homework you submit must be the work of just you. When submitting your assignment, be careful to follow the instructions summarized in Section 4 of this document.

Note: In this last homework in the course, you will *design* the solutions to the problems posed. You did some design in the blackjack question of Assignment 1, but in this assignment the questions are more complex and you need to think carefully about how to *decompose* the problem into subproblems or tasks—use subfunctions. The focus here is for you to turn a problem statement written in English into a solution written as a MATLAB program. ***Read carefully*** and take some time to think about the *design*—don't just jump into coding immediately.

1 Educational Game

For this exercise let's imagine that you are hired by a company that specializes in educational software for children. Your task is to create an interactive "geography" game. This piece of software will allow users to learn interactively the locations of different US cities.

At the start of the game a USA map is displayed with small circles marking the locations of a number of cities. The cities are labeled but the labels may be wrong! The city marker is red if its label is wrong and green if the label is correct. The goal of the game is to assign a correct label to every city marker. This will be achieved by repeatedly selecting a pair cities with labels that should be swapped. A city is selected by a mouse click. When a first city is selected the marker turns yellow. After the player selects the second city, the labels are swapped. If a city's newly swapped label is still incorrect, its marker size grows bigger (and is red)! If a city's newly swapped label is correct then the marker turns green and goes back to its original size. You will develop an interactive MATLAB application `CityGame` that implements this educational game! The game will look something like what you see in Figure 1.

The function `CityGame` has the following specifications:

```
function CityGame (positions, trueNames)
% An interactive game to learn the geography of the USA
% positions: x and y coordinates of n cities in an n-by-2 matrix
% trueNames: cell array of length n containing the names of n cities
% So positions(i,1) and positions(i,2) are the x and y coordinates of
%   the city whose name is trueNames(i), where i<=length(trueNames).
%   These coordinates work with the given map USA.JPG.
```

At the start of the game markers are put on the map based on the values in `positions` to indicate the position of each city. Each marker is labeled with a *randomly* chosen name from `trueNames`. Then the application allows the user to swap the labels of the different cities by clicking on the figure. A click outside the white map box indicates that a user wants to leave the game. Throughout the game, messages appear at the top of the figure (the title) to give instructions or indicate the progress to the user. If all cities are correctly labeled then the game ends and a congratulatory message is displayed as the title.

1.1 Detailed specifications & instructions

- The given functions `usaData1` and `usaData2` return the data that you will need to run `CityGame`. `usaData1` contains the data of only five cities, so this is suitable for initial development of your code. `usaData2` gives you more cities to play with. Here's how you would run the game:

```
[positions, trueNames] = usaData1();  % Get the game data
CityGame(positions, trueNames)        % Start the game
```

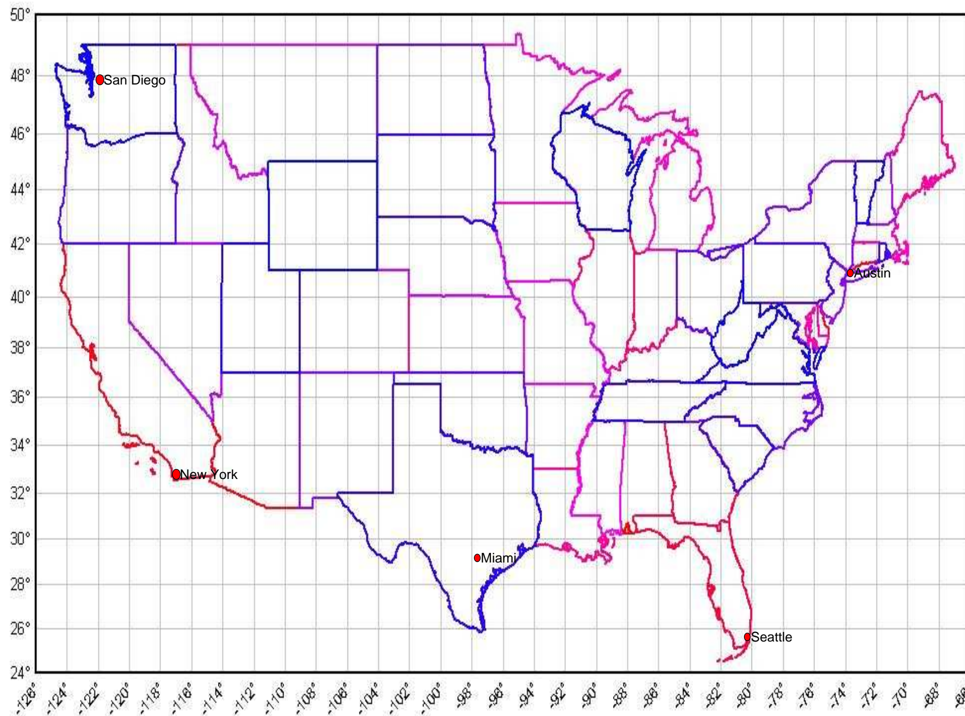

Figure 1: Example of `CityGame` after a few steps.

- Use the given map `USA.JPG`. The following code fragment will display it in a figure window:

```
clf      % Clear figure
hold on  % Hold subsequent plot commands on current axes
USA=imread('USA.JPG');  % USA is an array of numbers
USA = flipdim(USA, 1);  % Reverse the vertical coordinates: Usually in images the
                        %   origin (0,0) is the top left corner, therefore we need
                        %   to flip the data to use our Cartesian coordinates.
[n,m]=size(USA);  % Map dimensions:  x-coordinates in [1,m] and y-coordinates in [1,n]
image(USA);       % Display the map in a figure window
axis off          % Hide default axes

%%%% Add your code here to draw the cities and labels %%%%

hold off % Subsequent plot commands appear on new axes
```

- At the start of the game, display the map of USA, draw the city markers according to the data but label them *randomly* using the names in the cell array of city names. Make sure that no city name appears more then once. The markers should be colored as described earlier.
- Ask the user (using the plot title) to select a city whose label should to be swapped. Clearly mark the selected node.
- Ask the user to select another city to be swapped, swap the labels, and redraw the markers according to the rules described earlier. If the user does not (successfully) select the second city to be swapped but makes a mouse click on the map, deselect the first city.
- Go back to asking the user to select a city for swapping, and also display in the title the names that have just been swapped.

- Stop the game and display an appropriate message if the user clicks outside of the white map box at any time.
- Throughout the user interaction, refresh the plot title with appropriate instructions or messages. E.g., "New York and San Diego have just been swapped. Select a city to be swapped.", "New York was selected, select another city.", ..., etc.

1.2 Hints

- Take a close look at (and experiment with) the example in Module 2 Part 3 to get a better feel for how to use graphic commands and interactivity.
- The "Graphics example files" section in Module 2 Part 3 shows commonly used controls. "Text Alignment" is especially relevant.
- The statement `[a,b]= ginput(1)` stores the location of a user mouse click in variables `a` (x-coordinate) and `b` (y-coordinate).
- Remember that you can read the documentation on MATLAB commands by typing `help` <*command_name*> in the Command Window.
- To refresh a figure, use the command `clf` followed by `hold on` to make sure that all subsequent graphics commands add to the current set of axes. (As shown in the given fragment above.)
- *Encapsulate pieces of your code in subfunctions!* (Subfunctions are simply functions written in the same file as the main function, `CityGame` in this case.) Consider writing subfunctions for each of the following tasks:
  - "Randomize" the order of the city names in a cell array
  - Draw (refresh) the network plot with given positions for the cities and labels
  - Check if the current labels are correct
  - Determine whether a position clicked is inside or outside map box
  - Determine whether a position clicked is close enough to any of the city to consider that mouse click to have selected a city.
  - Update marker radii
- To makes things easier for you we provide function `DrawDisk` for drawing colored disks. Use it wisely.

2 Finding Similar Genomic Sequences

Many questions in biology are investigated with the help of efficient computational methods for searching, comparing, and organizing huge amounts of sequence data. Both DNA sequences and protein sequences can be treated as character strings. For DNA there are 4 possible letters: A, C, G, and T corresponding to the 4 bases adenine, cytosine, guanine, and thymine, respectively. For proteins there are 20 possible letters: A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y, each representing one of the 20 amino acids. One frequently used technique in computational biology is the comparison of sequences to find a "match"—or just similarity. For example, scientists have successfully predicted some proteins' function and/or 3-dimensional structure by extrapolating from similar proteins with known function and 3-d structure.

Consider the proteins that appear in, say, a mouse. For many of them there are similar proteins that appear in a rat or a cat. Although such proteins accomplish the same or similar functions, they are not identical; there are substitutions, insertions, and deletions of amino acids. There are corresponding substitutions, insertions, and deletions in the portions of the DNA that encode the information a cell uses to build the proteins.

You will write an application for finding subsequences (in a long sequence) that are similar to a query sequence. Solving this problem in full generality requires the use of Dynamic Programming, an algorithm design technique that is studied in upper level courses such as CS4820, Introduction to Analysis of Algorithms. In this assignment, we restrict our definition of "similar" to allow just substitutions and a limited number of insertions. You will write a function `proteinMatch` such that the statement

```
n= proteinMatch('genome.txt', 'protein.txt', 'result.txt')
```

reads sequence data from a file called `genome.txt` in the current directory, reads sequence data from a file called `protein.txt`, identifies all subsequences in the genome data that are similar to the protein sequence, and writes the findings to a file called `result.txt`. The number of similar subsequences found is stored in variable `n`.

2.1 Defining a Match

1. Allow one nucleotide to substitute for another. T is similar to A and C is similar to G. Consider the following example:

```
AATGCCCAACCG
 ATCC
```

The top string is the data and the bottom string is the query. If we consider C and G to be similar then there is a match that begins at position 2 of the data string. We assign a "penalty" for each substitution: an A-T substitution draws a penalty of 1 and a C-G substitution draws a penalty of 2. We can say a match is found based on some allowed penalty. For the above example:

Allowed Penalty	*Match Position(s)*
0	*none*
1	8
2	2
3	2, 8

A "Match Position" is the position (index) in the data where the beginning of the match occurs.

2. Allow the insertion of nucleotides. This is commonly described as a *gap*. In this example we do not consider substitution.

```
AATGCCCATGGG
 AT    AT
```

Our query string is ATAT, which is not found in the data. However, if we allow a gap of length 4, as laid out above, then we say that we have a match. We restrict our search to allow only one continuous gap. The penalty for each gap space is one. (So a gap of length 4 draws a penalty of 4.)

Combining these two ideas, we define two (sub)sequences to be similar—we say they match—if the gap (if one is needed) has a length of no more than 3 and the combined penalty for matching is less than 5.

2.2 Example

Example of the genome data file:

```
//A bogus example genome
ATGAAAGGTCATGGGTATTAGTCATAGCTGATCATGATCATGAT
TGATGATAGATTATATGCTCGCGTAGATGCTATATTATGCGCTA
```

Example of the protein data file:

```
//A bogus example protein
TGAATC
```

Example of output, `result.txt`:

```
2
TGAAAG
TGAATC
2
TGAAAGGTC
TGAA***TC
2
TGAAAGG
TGAAT*C
2
TGAAAGGTC
TGAAT***C
29
TGATCATG
TGA**ATC
32
TCATGATC
TGA**ATC
35
TGATCATG
TGA**ATC
41
TGATTG
TGAATC
48
TGATAG
TGAATC
```

2.3 File Format

Your program writes the results in an ASCII (plain text) file. Write three lines of text for each match found:

1. The "match position," which is the position (index) in the genome string at which the match occurs
2. The subsequence of the genome string that matches (starting from the "match position")
3. The protein sequence with any inserted gap filled in by the `*` character

We provide you with several files, which can be used for testing your program. Files `genome.txt` and `genome1.txt` can be matched against `protein.txt` and the performance can be easily measured. File `humanC5_1.txt` holds a string representing the bases for a portion of homo sapiens chromosome one; this data was downloaded from NCBI, the National Center for Biotechnology Information. Unfortunately we don't have a real protein for you to match against this gene. The rules for similarity are more complicated and a penalty is bigger when we are dealing with real matchings. You can try `humanC5_1.txt` against our made up proteins `protein.txt` and `protein1.txt` both should terminate in 2 to 5 minutes of runtime with a number of positive matches.
All given files contain ASCII characters (plain text). Each file begins with a line of text that identifies the sequence but *is not* part of the nucleotide sequence. The remaining lines in the file contain the sequence and are of varying lengths. Your code should work with the given files—do not modify the files in any way!

2.4 Hints

- Break down the problem! There are three main tasks in finding a match: calculate the penalty, find/set a gap, and find the match position. Encapsulate individual tasks in subfunctions.
- You can use the example sequence above in developing your matching algorithm.

- Here is some relevant reading to help with file I/O (input/output): Module 2 Part 4 on reading formatted input; Gilat pages 98, 99 on writing to a file.

3 Self-Check List

The following is a list of the minimum *necessary* criteria that your assignment must meet in order to be considered *satisfactory*. Failure to satisfy any of these conditions will result in an immediate request to resubmit your assignment. Save yourself and the graders time and effort by going over it before submitting your assignment for the first time.
Note that although all of these are necessary, meeting all of them might still not be *sufficient* to consider your submission satisfactory. We cannot list everything that could be possibly wrong with any particular submission!

- Δ Comment your code! If any of your functions is not properly commented, regarding function purpose and input/output arguments, you will be asked to resubmit.
- Δ Suppress all unnecessary output by placing semicolons (;) after assignment statements. At the same time, make sure that all output that your program intentionally produces is formatted in a user-friendly way.
- Δ Make sure your functions names are *exactly* the ones we have specified, *including* case.
- Δ Check that the number and order of input and output parameters for each of the functions match exactly the specifications we have given. In particular, the functions required in this project are
  1. Function `CityGame` — takes a matrix of numbers and a cell array of strings.
  2. Function `proteinMatch` — takes three file names (i.e., strings) as arguments and returns a single integer number, the number of matches found.
- Δ Test each one of your functions independently, whenever possible, or write short scripts to test them.
- Δ Check that function `CityGame` for problem 1 does everything that is listed in the detailed specification. Test it extensively, emulating realistic user behavior.
- Δ Check that function `proteinMatch` for problem 2 conforms to the specification. In particular, the function must read from two files and produce a new file with the matches.
- Δ Check that your functions do not crash (i.e., end unexpectedly with an error message) or run into infinite loops. Check this by calling each function from the Matlab prompt several times in a row. Before each test run, you should type the commands `clear all; close all;` to delete all variables in the workspace and close all figure windows.

4 Submission Instructions

1. Upload files `CityGames`, `proteinMatch`, `DrawDisk` onto the Assignment 2 area in CMS.
2. Please don't make another submission until you have received and read the grader's comments.
3. Wait for the grader's comments and be patient.
4. Read the grader's comments carefully and think for a while.
5. If you are asked to resubmit, fix all the problems and go back to Step 1! Otherwise you are done with this assignment. Relax.