22C:21 Project 3

Due date and time: See submission schedule below.

Introduction. This project consists of two parts. In Part I, which you will write in `C`, you will read words from the file `words.dat`, and build the ladders graph. In Part II, which you will write in `Java`, you will enhance your graph classes, so that you can use these to play a more sophisticated variant of the ladders game.

Part I: Building the ladders graphs. You have already written `Java` code to build a ladders graph. In this project, you will implement a more sophisticated and hopefully more efficient algorithm, in `C`, for building the ladders graph. The main steps of this algorithm are as follows.

1. Read the words from `words.dat` into an array, say `wordsList`.
2. Copy the words from `wordsList` into a *hash table*. This is simply a data structure that can be searched very efficiently to determine the existence of an item in a collection. More details of how to implement this data structure are given below.
3. For each word, say w, in `wordsList`, generate all 5×25 strings that differ from w in exactly one position. For each such generated string, say y, check if y is in the hash table of words. If so, y is a proper word and the graph should contain an edge between w and y.

Hash table. The idea of a *hash table* is simple. Let U be the set of *all* possible length-5 strings of lower case letters. Let $h : U \rightarrow [0, M-1]$ be a function that maps each string in U to an integer in the range 0 through $M-1$. Then any subset S of 5-letter strings can be stored in an array called `table` of size M by using the rule that a string $s \in S$ should be stored in slot `table[h(s)]`. The only problem with this method is that there can be collisions between strings. For example suppose that for two distinct 5-letter strings s and s' in S, $h(s)$ and $h(s')$ are both equal to some integer k. This means that both s and s' have to be stored in slot `table[k]`. To handle such a situation, `table` should be defined not as an array of strings. Instead, it should be defined as an array of singly linked lists of strings. Then all strings that hash to a single slot, say `table[k]`, can be stored in a singly linked list whose head is being pointed to by `table[k]`. This approach to resolving collisions is called *hashing with chaining*. The only aspect of a hash table left to discuss is the hash function h itself.

Any string of lower case letters can be viewed as a base-26 number as follows. Start by thinking of each letter as the number obtained by subtracting 97 (the ASCII value of 'a') from the ASCII value of the letter. Thus each letter is mapped onto a number in the range 0 through 25. For example, the string `next` can be thought of as the following sequence of numbers: 13 4 23 19. Thus a string $s = s_{n-1}s_{n-2}\ldots s_1 s_0$ of letters has an equivalent decimal value defined as:

$$D(s) = 26^{n-1} \cdot x_{n-1} + 26^{n-2} \cdot x_{n-2} + \cdots + 26^1 \cdot x_1 + 26^0 \cdot x_0,$$

where x_i equals the ASCII value of s_i minus 97 for each i, $0 \leq i \leq n-1$. Now for any string s define the hash function $h(s)$ as $h(s) = D(s) \bmod M$. Thus the function h maps any string onto the range 0 though $M-1$. You will have to watch out for possible integer overflow while computing $D(s)$. The only thing left to decide is the value of M. It is common practice to use a hash table whose size is approximately 10 times the number of items being stored in it. It is also common practice to choose the size of the hash table to be a prime number. These two considerations lead me to pick 57571 (the smallest prime larger than 5757×10) as the value for M. So your hash table is an array of size 57571 of singly linked list pointers.

Output of the program. Your `C` program, should produce as many lines of output as there are vertices in the ladders graph. Therefore, there should be 5757 lines in the output. Let us call the i th word in `words.dat`, w_i. For each i, $1 \leq i \leq 5757$, Line Number i of the output should contain information about w_i. Specifically, Line Number i should contain (i) the number of neighbors of w_i followed by (ii) the neighbors of w_i in some arbitrary order. To reduce the size of the output file, you may want to represent each neighbor of w_i by a number. Specifically, if w_j is a neighbor of w_i, then you may want to output j rather than w_j. I leave this choice up to you.

Organization of the program. Your `C` program should be in a single file called `ladders.c`. I suggest that your program contain at the minimum, the following functions:

1. A function that computes the hash value $h(s)$ of a given string s.
2. A function to insert into a hash table.
3. A function to search the hash table for a given string.
4. A function to read the words from `words.dat` into an array.
5. A function to transfer these words from the array into a hash table.
6. A function to generate (possibly into an array) the 125 potential neighbors of a given 5-letter word.

This is a suggestion, not a requirement; you should feel free to organize your code in some other way if it makes more sense to you. In any case, an organization like this should help in debugging your code more quickly.

Part II: A more sophisticated ladders game. For this part of the project, you are required to write a more sophisticated version of the program from Project 2 that played the ladders game. As before, your program will prompt the user for a pair of 5-letter words. Suppose that in response, the user types words x and y. Your program should output a sequence of 5-letter strings

$$x = w_0, w_1, \ldots, w_k = y$$

such that each w_i, $1 \leq i \leq k$, is obtained from w_{i-1} by replacing exactly one letter in one position with some other letter. Not all words in this sequence are required to be in

`words.dat`. In other words, the sequence may contain some nonsensical 5-letter words. Suppose that out of the $k+1$ words in the above sequence, t words belong to `words.dat` and the remaining $(k+1)-t$ words don't. Then the *cost* of the above sequence is $1 \times t + c \times ((k+1)-t) = c(k+1) - t(c-1)$. Here $c \geq 1$ is some fixed value that denotes the cost of using a nonsensical word. Your program is required to produce a sequence of words connecting x to y with *smallest cost*. Clearly, if c is 1, then the program has no incentive to use proper words - all words except the first and the last could be nonsensical. On the other hand if c is large compared to 1, then your program has more incentive to use proper words.

The problem of computing a cheapest sequence words from x to y amounts to solving the shortest path problem on an appropriately defined graph. Let L_5 denote the graph whose vertices are *all* 5-letter strings and whose edges connnect pairs of strings that differ in exactly one letter in one position. As in the previous project, we'll call this the ladder graph. Note that L_5 contains all the words in `words.dat` plus a whole bunch more and so storing L_5 explicitly will be an issue. For each edge $\{x, y\}$ in this graph define $distance(x, y)$ as follows:

$$distance(x,y) = \begin{cases} 1 & \text{if both } x \text{ and } y \text{ belong to } \texttt{words.dat} \\ \frac{1+c}{2} & \text{if exactly one of } x \text{ and } y \text{ belong to } \texttt{words.dat} \\ c & \text{if neither } x \text{ nor } y \text{ belong to } \texttt{words.dat} \end{cases}$$

It is easy to check that given a pair of 5-letter strings w_1 and w_2, a shortest path between w_1 and w_2 is also a cheapest sequence of words connecting w_1 and w_2.

Define a class called `LadderGraph` that extends `MyListGraph`. As you will see below, `LadderGraph` has some features that that are particular to the ladder graph L_5. In addition to the default constructor, the `LadderGraph` class also has a constructor that takes an integer argument c and stores the value of c as part of the description of the graph. Note that the value c is used to calculate the distance between the given pair of vertices. The main function in `LadderGraph` is a function called `shortestPath` that takes a start vertex (5-letter string) and an end vertex (5-letter string) and returns a shortest path between the start and the end vertices. The shortest path is returned as a `Vector` of vertices. This function should implement *Dijkstra's shortest path algorithm* which will be discussed in class. An efficient implementation of Dijkstra's algorithm is obtained by using a *binary heap*. A binary heap has been implemented in a class called `VectorHeap`, in Bailey's `structure` package. You are required to use this class after modifying it by adding a function with the following header:

```
void IncreasePriority(int i, Comparable v);
```

This function replaces the item in the i th slot in the heap by v, with the pre-condition that the priority of v is at least as large as the priority of the item in slot i. Notice that inserting v into the heap may force the heap to reorder itself.

An important aspect of the `LadderGraph` class is that it stores part of the ladder graph L_5 explicitly and part of it implicitly. Specifically, the portion of the graph that is defined on words in `words.dat` is stored explicitly in the class, but the rest of the graph, which is defined on 5-letter strings not in `words.dat` is not explicitly stored. The main reason for not storing the entire graph is that it is too big (it has 26^5 vertices). Therefore, relevant portions of the

graph will have to be generated "on the fly". For example, in the `shortestPath` function we have to repeatedly scan all neighbors of a vertex. To scan the neighbors of a vertex v, we first call `getNeighbors(v)` and obtain all of its neighbors in `words.dat`. After scanning this `Vector` of neighbors, we still have to generate and scan all of v's neighbors that are not in `words.dat`.

The `main` function of your program, which should be called `Ladders.java`, will start by reading the file `words.dat` and the output produced by the `C` program `ladders.c`. It will then prompt the user for a cost to be associated with nonsensical words. The user is expected to respond by typing some positive integer. Using this information, it will construct the portion of the ladders graph defined on words in `words.dat`. It will then repeatedly prompt the user for pairs of 5-letter strings and will respond to each such pair by outputting a cheapest sequence of 5-letter strings connecting the two given strings.

Submission Schedule.

Wednesday, Dec 1st Part I of the project, `ladders.c` is due.

Monday, Dec 13th Part II of the project, `LadderGraph.java`, `VectorHeap.java`, and `Ladders.java` are due.
