AI - Word Ladder Lab

Gabor – Fall, 2020

The word ladder lab is to do the following tasks, displaying each item on a separate line such as:

```
Word count: 2345
Edge count: 3421
Degree list:  1324 654 321 20 13 0 8 3 2

Construction time: 12.3s
```

An example file is located at

`https://academics.tjhsst.edu/compsci/ai/words.txt`

You should save this word list to a local file so that you don't have subsequent internet connectivity issues and so that the speed of reading it in is faster. Each word in this particular file is 6 letters long, and there is one word per line (and if there are any empty lines, your code should ignore (skip over) such lines).

Your command line script will take either one or three arguments. The first argument will always be the path to the file containing the words. If it is the only argument, then print out exercises 1-4 (example format shown above) and no other exercises, as your script is also being measured on time. If there are three arguments, the second and third will be a pair of words, and you should do exercises 5-13.

1. Construct a graph based on the words as nodes and where there is an edge between a pair of words if a change in exactly one letter at any single position, leads from one word to the other. Print out the number of words in the format shown above.

2. The number of edges in your graph according to the format shown above.

3. A degree distribution list, starting from the number of singletons up to the number of words of highest degree.

4. The length of time that your script took to three significant figures for problems 1-3.

See the next side for exercises 5-13

5. An example of a word of 2nd highest degree (Second degree word)

6. Number of different connected component sizes (Connected component size count)

7. Largest connected component size (Largest component size)

8. The number of connected components that are K_2 (K2 count)

9. The number of connected components that are K_3 (K3 count)

10. The number of connected components that are K_4 (K4 count)

11. The neighbors of the first input word (Neighbors)

12. A word that is farthest from the first input word (Farthest)

13. A shortest path from the first input word to the second input word. (Path)

Upon a run without script errors in the first part, there will be a summary string at the end with a partial score. Each item has an associated character. The possibilities are:

`S:` Section 2 missing
`X:` No output at all
`E:` Probable script Error
`T:` Probable Timeout
`K:` Keyword not found
`C:` No Colon after the keyword
`M:` Keyword appears Multiple times
`N:` No value found after the colon
`L:` The path Length is incorrect
`U:` Units missing
`F:` Significant Figures issue or word not Found
`V:` The answer (Value) is incorrect
`-:` Not processed as it followed a script error or timeout
`1:` Hurrah, it's correct

The first two items have point values of 13 each, while the rest are at 5. The remaining 19 points will be based on the time and the efficiency of the algorithm used to construct the graph. 60 seconds are allocated to the first four, while the remaining items get 75s in total.

One of the fundamental routines that you will need is a breadth first search (BFS), which will be our workhorse algorithm for the search unit in this class. Essentially, this algorithm says: start from the root and go to all neighbors (the children), now inspect all the neighbors of those neighbors that you haven't already seen (the grandchildren), etc. In other words, on each pass, find the next generation (those items one additional level away from the root).

The following, while being a grossly inefficient way of carrying out the BFS, introduces two elements which will be with us for a long while. `parseMe` holds all those vertices which still need to be explored/examined. `dctSeen` is a dictionary whose keys are vertices and the value for a key is a parent vertex, which is to say a vertex whose distance to the root is one less. The parent of a root will be the empty string (which we presume is not the label of any vertex):

```
def BFS(root):
  dctSeen, parseMe = {}, {root: ""}

  while parseMe:
    dctSeen.update(parseMe)
    parseMe = {nbr:prnt for prnt in {*dctSeen}
                        for nbr in Nbrs(prnt)
                        if nbr not in dctSeen}
```

One of the major problems with the above is that on every pass it must examine all the vertices that it's already handled. Instead, what is usually done is to put newly encountered vertices into a queue (`parseMe`), and then update the queue with the unexamined neighbors of that vertex when it is taken out of the queue. Note that `parseMe` will no longer be a dictionary. It looks like this.

```
def BFS(root):
  parseMe = queue with root
  dctSeen = {root: ""}

  while parseMe:
    prnt = remove next element from parseMe

    for nbr in Nbrs(prnt) not in dctSeen:
      add nbr to parseMe
      add nbr:prnt to dctSeen
```

This is a workhorse algorithm with very many extensions. For example, it is often the case that one would like to find a shortest path from point A to point B. With a very simple modification, BFS can be amended to do just that:

```
def shortestPath(root, goal):
  # return a list of the shortest path from root to goal
  # return the empty list if no path is possible
  parseMe = queue with root
  dctSeen = dict with single key of root and val of ""

  while parseMe:
    item = remove next element from parseMe
    if item == goal: return pathFromRootToGoal(dctSeen,goal)

    for nbr of item not in dctSeen:
      add nbr to parseMe
      add nbr:item to dctSeen

  return []
```