Foster Clark

Patrick Hough

Gary Sidoti

Abe Arce

**Computer Algorithms Project One**

The goal of this assignment is to input a text file containing a hierarchy of

objects, construct a graph, and efficiently traverse this graph. In order to parse this

file into the graph, we first call our import data function. This function reads in a

single line at a time. Then this line is broken down by reading the first string, all the

way up to the colon. We transfer this to all lowercase for uniformity, then this string

is inserted as a vertex. Then in a while loop, we analyze the rest of the line, and

break the string down at every comma. We then erase the space that is present

immediately after the comma. Afterwards we then convert these strings to

lowercase. For each string that is entered, we check if it is already a vertex. If the

string is not a vertex yet, one is created for it. If it is already a vertex, it is a repeat

and does not need to be created. For each vertex that is created, an edge is made

from the source to all of its children. This will result in the graph being created as a

hierarchy. Each vertex will have edges to its children, and those children are

vertexes themselves, with children of their own. The graph stores this information

using adjacency lists, so that it can be traversed easily. An adjacency matrix would

be a waste of space, seeing as the graph is sparse. This makes it possible for us to

traverse the graph with quickly and with ease.

The first problem asks us to traverse from a source vertex, down to a certain

level, and return a number of subtypes the vertex has. This is done using a similar

method to breadth first search. Instead of only inputting a source vertex, it also

inputs the desired level of traversal, and the number of subspecies that the user

wants returned. The idea of this function is to traverse the graph, while keeping

track of what level the current node is on, and comparing it to the desired level.

Once a new level is reached, the node with the highest number of children is

selected (to ensure that we will be able to return the desired number of subspecies),

and the level is checked once again. Once the algorithm arrives on the desired level,

it returns each vertex on the level, until the desired count is reached. This avoids

unnecessary dead ends such as reaching a leaf before the desired level, or not

having enough children to return the desired amount of subspecies.

The second asks what is the most diverse sub species of a species. This is also

found using a breadth first search modification. However in this one, each time a

vertex is reached, the number of children is counted and stored alongside the name

of the vertex. When a vertex is reached, the number of children is compared to the

current max. If the current node has more children, the current max is updated. This

search moves through all of the children of a given vertex. It only stops when there

are no more vertices to reach. At the end of the search, the vertex that has the most

children is given as the output.

The third objective is to find the lowest common ancestor. Contrary to the

last two algorithms, this takes two separate vertices as inputs. Instead of searching

down like a breadth first search, we will work backwards from each vertex. As each

new vertex is reached, it is stored, and all of the current storages are compared. If

there are any matches, the lowest common ancestor has been reached.

All of the problems were solved with the most efficiency we could. By

starting with well-known and working algorithms like Breadth First Search, we are

ensuring that we are starting with an efficient system, and only modifying it for our

needs. This is to encourage the fastest traversal of the graph that we can. The actual

graph is constructed using adjacency lists to also encourage speedy traversal.

Together our project runs as fast as we can possibly make it go, without sacrificing

accuracy.