COMP 322: Assignment 4 - Winter 2014

Due at 11:30pm, April 8th 2014

1 Introduction

In this assignment we will leverage the classes that we built in the previous homework for graphs of Wikipedia pages to generate useful information about individual articles. In particular, we would like to find the most important articles that are related to some input title. To do this, we will use the concept of random walks: starting at an individual article, send multiple particles that will randomly explore the graph. This randomness is governed by the weights that we associated with each edge: that is, when a particle has to choose an outgoing edge, it will give more preference to edges of higher weight.

Using the strategy described in the previous paragraph, it is not hard to see that after a while most particles will start walking over the most popular pages of the entire Wikipedia graph. The second part of the assignment will "trim" down this graph to subgraphs that are more likely to give meaningful results. We will write algorithms to generate both a "Breadth First Search" subgraph and a "Spanning Tree" graph, very popular constructs in graph theory. We will describe in detail how to build these in short.

Important references:

- `http://www.cplusplus.com/doc/tutorial/classes/`
- `http://www.cplusplus.com/reference/stl/vector/`
- `http://www.cplusplus.com/reference/stl/list/`
- `http://www.cplusplus.com/reference/stl/map/`
- `http://www.cplusplus.com/reference/cstdlib/rand/`
- `http://www.cplusplus.com/reference/queue/queue/`
- `http://www.cplusplus.com/reference/queue/priority_queue/`

2 Assignment Resources

- `wikiFinal.h` : a header file with all class and method definitions for methods that have been implemented already and can be used at will, or have to be implemented by the student in this homework
- `wikiFinal.cpp`: contains implementations for methods that the student is not responsible for
- `wiki_cs322` : a folder containing

- `wpcd` : a smaller version of Wikipedia. Feel free to browse it, so you can have a feel of the graph you are working with
- `wg_edges.txt` and `wg_wikis.txt` : which contain the input Wikipedia graph over which we are working. Feel free to modify the main function to build the graph using your own methods, but, depending on the machine you are using, it might take a while!

3 Assignment Requirements

Please add all your code in a file named `wikiFinalStudent.cpp`. Test your code using these and the other files provided: `wikiFinal.h`, `wikiFinal.cpp` and `simpleFinalTest.cpp`. The TAs will test your code by compiling your submission using the provided `Makefile` and an additional `gradingFinalTest.cpp` (not provided). NOTE: You do not have to use any code from the first three assignments. All you need is already in `wikiFinal.cpp`.

`wikiFinal.h` contains class definition for exceptions that you should use in this homework, for `class Graph` and for `class WikiGraph`. **Please read the class definitions before you read the remainder of the document**. Note that some of the constructors/functions are already implemented, based on the first three assignments. You are responsible for the following:

- In `class Graph`:

  1. `Graph::Edge Graph::sampleEdge(list<Edge> lst)` which takes as input a list of edges and returns a sample Edge from the list, where each edge is picked with probability "weight of edge / total weight". To achieve this:
     (a) Sample a random integer between *0* and *total weight -1*. For simplicity, use `rand()` from `<cstdlib>`.
     (b) Calculate a vector `cumul` storing the cumulative distribution function: at item i you should store the sum of the weights of the first i edges in the list
     (c) Return the k^{th} edge iff `cumul[k-1]` $<$ random integer $\leq$ `cumul[k]`.

     Exceptions: If any of the edges has weight ≤ 0, throw an appropriate exception.
  2. `map<int,int> Graph::random_walks(int start_node)`, which takes as input a start node and repeats a random walk through the graph: jump from one node to a neighbour (i.e. adjacent node) by sampling an edge from the adjacency list, using `sampleEdge`. A walk should be `rw_walk_length` long, and you should perform `rw_num_walks` such walks (both variables are private variables of the class `Graph`). As you perform these walks, maintain, in a map data structure, a count for the number of times a node was visited. We use a map (as opposed to a vector) to maintain these counts because we only care about nodes that were visited (i.e. count > 0)
     Return value: a map that "maps" every node that was visited by the random walks to the number of times it was visited.
     Exceptions: If the start node is invalid, throw an appropriate exception (`invalid_graph_id`).
  3. `list<int> Graph::breadth_first_search(int start_node, int number_nodes)` , which generates a list of nodes related to `start_node` by performing what is known in graph theory as Breadth First Search. You should return `number_nodes` such related nodes, as a list data structure. To compute the BFS:

(a) Use a `set<int>` to keep all elements that should be returned. A nice property of `set` is that it does not allow duplicates, insertions and both checking whether an element is already in the set are all fast operations.

(b) Use a `queue<int>` to maintain a waiting list (i.e. queue) for the nodes that should be added to the set. The first element in the queue is `start_node`

(c) While your still have to add elements to the set, remove the first in line in the `queue`, add it to the `set`, and add all its neighbours to the back of the `queue`. **Only add neighbours that are not already in the set of visited nodes!**

Note, for some articles, the queue might become empty before you visited the required number of nodes. In that case, you can return the fewer nodes that you visited.
Exceptions: If the start node is invalid, throw an appropriate exception (`invalid_graph_id`). If `number_nodes` is ≤ 0, throw an appropriate exception (`invalid_param`).

4. `list<int> Graph::spanning_tree(int start_node, int number_nodes)` , which generates a list of nodes which are related to `start_node` by performing what is known in graph theory as Spanning Tree construction. You should return `number_nodes` such related nodes, as a list data structure. To compute the Spanning Tree:

   (a) Use a `set<int>` to keep all elements that should be returned. A nice property of `set` is that it does not allow duplicates, insertions and both checking whether an element is already in the set are all fast operations.

   (b) Use a `priority_queue<Edge>` to maintain a waiting list (i.e. queue) for the nodes that should be added to the set of visited nodes. Start by adding to the queue all edges in the adjacency list of `start_node`. The difference from BFS is that each `Edge` enters the queue with a priority: higher weights will move ahead of lower ones!

   (c) While your still have to add elements to the set, remove the `Edge` with highest priority in the `priority_queue`, and add the destination of the edge to the set. If the destination was not already in the set, add all edges in the adjacency list of the latter to the `priority_queue`. **Only add edges with destination that is not already in the set of visited nodes!**

   Note, for some articles, the queue might become empty before you visited the required number of nodes. In that case, you can return the fewer nodes that you visited.
   Exceptions: If the start node is invalid, throw an appropriate exception (`invalid_graph_id`). If `number_nodes` is ≤ 0, throw an appropriate exception (`invalid_param`).

- Destructors both for `Graph` and `WikiGraph`, which should take care of all memory that your methods might have allocated dynamically

- A user interface, which will have the following features:

  1. Outputs a nice greeting message to the user!
  2. *Politely* asks the user the amount of related pages that should be printed for each "search" item. This should be positive.
  3. *Politely* asks the user what should be the size of the subgraph considered for Breadth First Search and Spanning Tree
  4. *Politely* asks the user the number of random walks to perform, AND the length of each random walk.

5. *Politely* asks for a Wikipedia article name and outputs the search results (similar to the simple test method in the provided main function) based on the user's preferences

6. Repeat previous item until the Wikipedia article name that the user enters is "exit".

NOTE: the user interface should properly handle all exceptions, by printing appropriate error messages to the user. *The use interface is worth 15% of the assignment.*