COT5405 Analysis Of Algorithms (Spring 2021) Report - Assignment 1

Description

This project aims to implement some graph algorithms on small as well as large graphs and analyzing the performance (particularly CPU time and peak memory usage) of each algorithm. The Project is implemented using C++. The project is divided into two parts:

For the first part of project, I have created a collection of three general graph functions namely connected components(), one_cycle(), shortest_paths(). First, the program will generate a way to store the graph and then apply the above functions and record the results. Also, I will analyze the performance of each function.

For the second part of project, we have given a dataset from Netflix that has 4 movie rating files containing over 100 million ratings from 480 thousand randomly-chosen, anonymous Netflix customers over 17 thousand movie titles. The data were collected between October 1998 and December 2005 and reflect the distribution of all ratings received during this period. The ratings are on a scale from 1 to 5 (integral) stars. First, the program will generate a file from the above data containing edges based on some adjacency criteria for the graph (There are three different criteria for the adjacency). Then the program will read the file containing edges and will generate a graph and after will return the number of connected components in the graph and the size of each connected component. Also, I will analyze the performance of each graph.

Design

The project design is as follows:

- **graph_operations.h -** contains the three functions from the first part.
- **graph_simulator.h -** contains functions for generating simulated data using three methods. namely n_cycle(), equivalence_mod_k(), empty_graph().
- **simulated_test.cpp** - contains the main function for generating simulated data and running the operations on it.
- **graph_make.h** - contains functions to read in the movie reviews data and create edges files based on some adjacency criteria. Three functions are tried with different adjacency criteria. The criteria are mentioned later in the report.
- **real-test.cpp** - contains the main function for generating a real graph based on some adjacency criteria and running the operations asked in the second part.

Implementation of graph_operations.h

The file contains a class named **undirected_graph()** that stores the graph and the three algorithm-based functions associated with it. The class stores the graph in an **adjacency list**.
The adjacency list is used to have lesser storage data for the graph as compared to the adjacency matrix. The adjacency list is stored as the **vector of vectors** where the index represents the node and

the vector inside every index holds all nodes adjacent to it. Vector is used basically to allocate the memory in the heap and avoid stack overflow conditions later.

Now, the implementation of three algorithms functions are described below:

1. **connected_components():** The function uses a depth-first search to return the list of connected components. Each component is itself a list. The function returns the connected components as the vector of vectors and the order of the components is not defined. Also, the function uses another function **dfs_cc()** that performs a depth-first search for each component iteratively. Time complexity O(V + E)

2. **one_cycle():** The function uses a depth-first search to return a cycle. If there is one in the graph. If the graph is acyclic, the result is an empty list. Otherwise, the list specifies a cycle as a list of three or more vertices that starts and ends with the same vertex. If there are multiple cycles in the graph then any cycle may be returned. The function uses another function **dfs_one_cycle().** This function stores a **parent array** that will store the parent of each node in the cycle that will help to find the path of that cycle and two variables **cycle_start** & **cycle_end** that stores the start and end of each cycle. Time complexity O(V + E)

3. **shortest_paths():** This function uses Dijkstra's algorithm and returns a map of shortest paths. All the graphs are unweighted so the edge weights are intrinsically 1. The function returns a vector of vectors **sp.** The sp is a list of paths returned by the function for a certain source node. For each vertex v that is reachable from the source, sp[v] is the list of vertices on a path from v back to the source. The path from source to source is just the single node source. There may be several shortest paths but the function provides only one. The vertex v will not appear in sp if there is no path between v and source. The function uses another function **find_path()** that stores a parent array which will help to track the shortest path back to the source from each vertex. Time complexity O(V + E log V) as we use a min-priority queue for implementation

Besides the above three functions, there is one more function **connected_components_real().** This will be used later when processing Netflix data. The function needs a set of nodes that will be passed as arguments so that we will only trace the nodes that are present in the graph.

Implementation of graph_simulator.h

The file contains functions implementation for generating simulated data using three methods namely n_cycle(), equivalence_mod_k(), empty_graph(). All the functions return an **instance of class undirected_graph** after adding edges (i.e adjacency list) to the class object.

The implementation of each function is mentioned below:

1. **n_cycle(long int n):** This function generates a graph where vertices are integers from 0 through n − 1. The function takes the number of nodes as an argument The vertices u and v are connected by an edge if $u - v = \pm 1$ or $u - v = \pm(n - 1)$.
   There is one connected component, every shortest path has length at most n/2, and there is a unique cycle of length n.

2. **equivalence_mod_k(long int n):** For this function, I have taken k as n/3 instead of taking it as an input from the user (i.e. k = n/3). This function generates a graph where vertices are

integers from 0 to n − 1, where k ≤ n. The vertices u and v are connected by an edge if u − v is evenly divisible by k.
There are k components, and each component is a complete graph.

3. **empty_graph(long int n):** This function generates a graph where vertices are integers from 0 through n − 1. There are no edges. There are n connected components, no paths, and no cycles.

Implementation of simulated_test.cpp

This file contains the final main function that imports graph_simulator.h and graph_operations.h to simulate the data of the graph and returns the output of three algorithm functions for three different graphs mentioned above. To compile, execute and test the cpp program, run the following command **(make sure graph_simulator.h and graph_operations.h are in the same current directory as simulated_test.cpp)**

```
g++ -std=c++11 simulated_test.cpp -o simulated_test.out
./simulated_test.out
```

Sample output is provided below for a small input of **5 nodes.**

```
TYPE OF GRAPH = n_cycle
Connected Components:
0 4 3 2 1

Detected Cycle: Yes
0 1 2 3 4 0

shortest_paths from each vertex to source(0)
0
1 0
2 1 0
3 4 0
4 0


TYPE OF GRAPH = equivalence_mod_k
Connected Components:
0 4 3 2 1

Detected Cycle: Yes
0 1 2 0

shortest_paths from each vertex to source(0)
0
1 0
2 0
3 0
4 0


TYPE OF GRAPH = empty_graph
Connected Components:
0
1
2
3
4

Detected Cycle: No


shortest_paths from each vertex to source(0)
0
```

Implementation of graph_make.h

This file contains three adjacency criteria for building up the graph. The file creates an edge file namely (**criteria_1_edges.txt, criteria_2_edges.txt, criteria_3_edges.txt**) that contains edges between nodes of the graph. Each row has two numbers representing an **edge** between them. The description of three criteria is as follows:

1. **criteria_1():** Two viewers have at least three movies in common and they have given the rating to each of the movies in the year “2001” and “2000”.

2. **criteria_2():** Two viewers have at least five movies in common and they have given a full rating to the movie (i.e. 5) and the rating is given in the year “2005”

3. **criteria_3():** Two viewers have at least five movies in common and they have given ratings equal to or greater than 4 and the rating is given between years “2003” and “2005”.

Each criterion creates a map after reading the ratings_data file. The map contains a mapping of each customer to all the movies he/she has watched based on the adjacency criteria. Then we go through each pair of the distinct viewer from the map and then take an intersection of movies they have watched to check further criteria follows or not.

For reducing the size of each edge file, I have used a **map <pair<long, long int>, int>** that will help to keep track of all the node pair(edges) that are already in the file to not include them twice or maybe thrice. This file also improves the execution time when generating a graph from real_test.cpp.

Implementation of real_test.cpp

This file contains the main function that will generate a graph from Netflix data.

It contains a function **load_graph(),** this function read the edges file, import the graph class object from graph_operations.h, and then add edges to the instance of the graph object. Finally, it will call the connected_components_real() function to display all the number of connected components in the graph and also the size of each component.

The main function contains three calls to three different criteria. **It is preferred to call only one criteria at a time and comment out the rest** (also the load_graph functions). So, the system may not crash, not go out of stack memory as a single criterion can take hours to complete.

To compile, execute and test the cpp program, run the following command **(make sure graph_make.h is in the same current directory as real_test.cpp)**

```
g++ -std=c++11 real_test.cpp -o real_test.out
./real_test.out
```

Sample output is provided below for **criteria 3()**

```
Number of Connected Components: 1

Size of Each connected component:
296820
```

Testing

All the readings below can vary depending upon the type of architecture and hardware used in the host.

Performance Analysis Of simulated_test.cpp

Now, we will test simulated_test.cpp for different input nodes and will analyze the CPU time and peak memory usage of the program. Note that for every input simulated_test.cpp will run all the three algorithms function for all the three types of the graph in one go.

- For **n = 100** as input,

  CPU time
  User time (seconds): 0.00
  System time (seconds): 0.00
  Peak Memory
  Maximum resident set size (kbytes): 1008

- For **n = 1000** as input,

  CPU time
  User time (seconds): 0.17
  System time (seconds): 0.02
  Peak Memory
  Maximum resident set size (kbytes): 4460

- For **n = 10000** as input,

  CPU time
  User time (seconds): 15.02
  System time (seconds): 0.70
  Peak Memory
  Maximum resident set size (kbytes): 226312

- For **n = 100000** as input,

  CPU time
  User time (seconds): 87.82
  System time (seconds): .83
  Peak Memory
  Maximum resident set size (kbytes): 2243925

After this increasing n to 1000000 or more my PC was through stack overflow error (basically segmentation fault).

Performance Analysis Of real_test.cpp

Finally, we will test the three criteria function in file graph_make.h and generate a graph from real_test.cpp, and analyze the CPU time and peak memory usage of each criteria.

For generating edges file each criteria took around **1 hour** for each ratings_data_x.txt file. But for my simplicity, I have used all two PCs for reading different ratings_data_x.txt file and generating different edges file for each of them and concatenating all the edges file in the end. You can do that by changing for loop inside every criteria of graph_make.h file. (**Only for criteria 1, I have taken only ratings_data_1.txt and ratings_data_2.txt into account as my system was crashing when generating graph for criteria 1.**)

Now, for generating graphs from edge files and finding connected components for each criteria from real_test.cpp file (basically calling load_graph function for each criteria). The CPU Time and peak memory usage are given below:

- For adjacency **criteria_1():**

  CPU time
  - User time (seconds): 1430.45
  - System time (seconds): 1.03

  Peak Memory
  - Maximum resident set size (kbytes): 532760

- For adjacency **criteria_2():**

  CPU time
  - User time (seconds): 2124.18
  - System time (seconds): 1.96

  Peak Memory
  - Maximum resident set size (kbytes): 896088

- For adjacency **criteria_3():**

  CPU time
  - User time (seconds): 3404.13
  - System time (seconds): 2.34

  Peak Memory
  - Maximum resident set size (kbytes): 1792176