**Algorithms**

Programming Assignment #4

Graph: Maximum Flow

**Introduction:**

Our commercial software *graphlab®* was highly appreciated by the customers. However, they felt the graph functions in *graphlab®* are not enough. As the chief programmer of this software, you are asked to add one more function, maximum flow. What you have to do is implement a good maximum flow algorithm with as little runtime and memory usage as possible.

**Input/output Files:**

In this PA, we still use the *DOT* language to describe our graph. However, this time we have to support reading directed graph. Below is an example directed acyclic graph (dg6.dot) that has six vertices (v0~v5) and 9 edges. The first line starts with a keyword ‘*digraph*’, followed by its name ‘*dg6*’. The description of this graph is enclosed in a pair of curly brackets {}. Each line between the curly brackets represents an edge. For example, ‘v0 -> v1’ represents an edge from *v0* to *v1.* In the square bracket, the edge capacity is described as a label, which will be shown in the figure. For example, the edge capacity v0 -> v1 is 4.

|  |
| --- |
| digraph dg6 {  v0 -> v1 [label = "4"];  v0 -> v2 [label = "1"];  v0 -> v3 [label = "3"];  v1 -> v4 [label = "7"];  v2 -> v4 [label = "3"];  v2 -> v5 [label = "5"];  v3 -> v4 [label = "1"];  v4 -> v5 [label = "6"];  } |

The new graph *dg6\_mf* is the maximum flow of the original graph *dg6*. In the maximum flow graph, the label of *edge (u, v)* represents the flow *f(u, v)*. A flow *f(u, v)* must follow two properties, i.e., capacity constraint and flow conservation. At the end, we also show the corresponding information, including the number of vertices, edges, total flow in the comments. The last two lines show the runtime (user) in seconds and memory usage (peak) in MB. Note: Please show your results (red parts) in correct format because our automatic grading tool will check these numbers.

|  |
| --- |
| digraph dg6\_mf {  v0 -> v1 [label = "4"];  v0 -> v2 [label = "1"];  v0 -> v3 [label = "1"];  v1 -> v4 [label = "4"];  v2 -> v5 [label = "1"];  v3 -> v4 [label = "1"];  v4 -> v5 [label = "5"];  }  // vertices = 6  // edges = 7  // max flow = 6  // runtime = 0.0 sec  // memory = 1.0 MB |

We can also use the same *dot* command to convert the output file into a png picture. Please note that *gn6\_mf* looks different from the original graph *gn6*, because the labels of edges have been changed, and some edges (or vertices, not shown in this case) even disappeared. The look of the graph does not matter as long as your maximum flow is correct.

**Commands required:**

In the graphlab tool, you need to support the following system commands. All commands have been already implemented.

|  |  |
| --- | --- |
| **System Commands** | **Description** |
| **ls** | List directory contents |
| **exit** or **quit** | Exit graphlab |
| **help** | Show all commands |
| **source** <batch\_filename> | Execute commands in batch mode |
| **dot** –Tpng <dot\_filename> –o <png\_filename> | Convert dot file into png file |
| **display** <png\_filename> | Show picture in X-window |
| **read\_graph** <dot\_filename> | Read the undirected graph in dot format |
| **write\_tree\_dfs** –s <sourcenode> –o <dot\_filename> | Perform depth first search starting from source node. Then write to a dot file. |
| **write\_tree\_bfs** –s <sourcenode> –o <dot\_filename> | Perform breadth first search starting from source node. Then write to a dot file. |
| **write\_tree\_mst** –a prim –r <rootnode> –o <dot\_filename> | Perform Prim’s MST starting from the root node. Then write to a dot file. |
| **is\_spanning\_tree** –i <dot\_filename> | Check if the dot file is a spanning tree of the graph.  The output is simply Yes or No. |

You need to implement the following user commands.

|  |  |
| --- | --- |
| **System Commands** | **Description** |
| **read\_graph** –d <dot\_filename> | Read the directed graph in dot format. |
| **write\_max\_flow** –s <sourcenode> -t <sinknode> -o <dot\_filename> | Perform maximum flow starting from the source node to the sink node. Then write to a dot file. |
| **is\_flow** –i <dot\_filename> | Check if the dot file is a flow of the graph.  The output is simply Yes or No. |

To simplify our problem, we assume that the given graphs are always acyclic and the source node can always reach the sink node.

Your graphlab will be tested in batch mode. An example command file (*dg6.bat)* is provided in the input directory.

|  |
| --- |
| read\_graph –d inputs/dg6.dot  dot –Tpng inputs/dg6.dot –o outputs/dg6.png  display outputs/dg6.png &  write\_max\_flow –s v0 –t v5 –o outputs/dg6\_mf.dot  dot –Tpng outputs/dg6\_mf.dot –o outputs/dg6\_mf.png  display outputs/dg6\_mf.png &  is\_flow –i outputs/dg6\_mf.dot  exit |

**Question:**

1. Please write a pseudo code for maximum flow and analyze its complexity.

**Requirements:**

1. Your source code must be written in C or C++. The code must be executable on EDA union lab machines.

2. In your report, compare the results, running time, and memories different input sizes. Please fill in the following table and also plot figures showing the memory and running time. Please use –O2 optimization and turn off all debugging message.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Maximum Flow** | | | | |
| **Input Size**  **(# of Vertices)** | | **Vertices/Edges (Input)** | **Vertices/Edges**  **(Output)** | **Runtime**  **(s)** | **Memory**  **(MB)** |
| 6 | |  |  |  |  |
| 10 | |  |  |  |  |
| 100 | |  |  |  |  |
| 1000 | |  |  |  |  |
| 5000 | |  |  |  |  |

**Output File Format:**

You should refer to dg6\_mf.dot file and treat it as golden output file. Write your own .dot output files in the same format.

The order of vertices or edges does not matter. Being sure that you obtain the correct max flow with good efficiency is all you have to focus on.

If there are output syntax errors, you may lose some scores.

**Grading:**

60% correctness

20% file format and location (checkSubmit)

20% report

**5% efficiency (bonus)**

<note> We will evaluate your max flow efficiency using the following formula:

We will pick top 10 efficiency scores of all students and give them the extra bonus.

**Submission:**

Please submit a hardcopy of your report in class. Also, please submit a single *\*.tgz* file to CEIBA system. Your submission must contain:

1. your source codes and a make file. By simply typing ‘make’, an executable binary is produced under the *bin* directory. Please use ‘-O2’ optimization. Please turn off all debugging messages.

2. a README file that explains your files.

3. a report in the *doc* directory. Please also submit a printed report in class.

4. We will use our own test cases so do NOT include the input files.

The submission filename should be compressed in a single file <student\_id>-<pa4>.tgz. (e.g. b90901000-pa4.tgz). If you have a modified version, please add \_v2 as a postfix to the filename and resubmit it (e.g. b9090100 0-pa4-v2.tgz). You can use the following command to compress a whole directory:

tar zcvf <filename>.tgz <dir>

You are required to run the checksubmitPA4 script to check if your .tgz submission file is correct. To use this script, simply type

./checkSubmitPA4.sh b99901000-pa4.tgz

We have so many students in the class so we need automatic grading. Any mistake in the submission will result in cost 20% off your score. Please be very careful in your submission.

**NOTE:**

TA will check your source code carefully. Copying other source code can result in zero grade for all students involved.