FIT1045 Algorithmic Problem Solving – Assignment (18%).

Due: 23:55:00, Friday 6th Oct, 2017.

Submission Procedure

1. Put you name and student ID as a comment on each file (and in the footer of the discussion file).
2. Make sure your programs strictly follow the instructions given at the end of each task.
3. Save your files into *a single folder* called `assignment` and create a zip file of this folder called yourFirstName_yourLastName.zip.
4. Submit your zip file containing your solution to Moodle. Make sure that your assignment is not in "Draft" mode. You need to click "Submit" to successfully submit the assignment.
5. Your assignment will not be accepted unless it is a readable **zip** file.
6. Your submitted zip file should only include the following files:
   - graphFileOps.py
   - intersection.py
   - union.py
   - difference.py
   - supergraph.py
   - connectivity.py
   - greedy.py
   - greedy.pdf (or greedy.rtf if you prefer)
   - bruteforce.py
   - any other files included will not be assessed

Important Notes:

1. Please ensure that you have read and understood the university's policies on plagiarism and collusion available at `http://www.monash.edu.au/students/policies/academic-integrity.html`. You will be required to agree to these policies when you submit your assignment.
   - The assignments will be **checked for plagiarism** using an advanced plagiarism detector
   - Students will be **interviewed by their demonstrators** to demonstrate the understanding of their code during their workshop in week 11
   - If you are **unable to attend** your regular class to be interviewed you must **make alternate arrangements** with us via the role account `mailto:FIT1045.clayton-x@monash.edu`
   - Students who do not attend their interview will have one week from the release of assignment results to make alternate arrangements, **otherwise they shall receive 0 marks for the assignment**
   - Last year, many students were detected by the plagiarism detector and almost all got zero mark for the assignment and, as a result, failed the unit. "Helping" others is NOT okay. Please do not share your solutions with others. If someone asks you for help, ask them to visit us during consultation hours for help
2. Your demonstrator will check in on your progress on the assignment in week 8, make sure you have something to show them as they will use the opportunity to give you some valuable feedback on your work (which may help you improve your final mark!)

3. You are not allowed to import any Python library or module and should not be using any inbuilt python functions that simplify the tasks you are told to implement.

4. Your program will be checked against a number of test cases. Do not forget to include comments in your code explaining your algorithm. If your implementations have bugs, you may still get some marks based on how close your algorithm is to the correct algorithm. Also, make sure that your program handles the boundary cases properly.

5. This assignment requires an understanding of greedy algorithms and brute force solutions to problems in order to complete. These are covered in weeks 5 and 6 respectively but you can begin working on the assignment as soon as it is released.

6. This assignment is non-trivial and is designed to give you the opportunity to practice a large component of the unit content. It's okay if you can't finish everything, but try the best you can. You are strongly encouraged to start as early as possible and get assistance in consultation sessions and from your tutors and demonstrators.

7. It is possible that you may like to revisit this assignment as we cover more in the unit as you may discover other better ways of approaching the tasks given.

Marks: This assignment has a total of 100 marks and contributes to 18% of your final mark. Late submission will have 10% off the total assignment marks per day (including weekends) deducted from your assignment mark, i.e., 10 marks per day. So, if you are 1 day late, you will lose 10 marks. Assignments submitted 7 days after the due date will normally not be accepted.

Marking Guide:

Graph problems: 100 marks

(a) Code readability (Non-trivial comments where necessary and meaningful variable names) – 10 marks

(b) Code decomposition (appropriate decomposition and avoiding duplicate code or duplicate functions) – 10 marks

(c) reading from a file – 5 marks

(d) writing to a file – 5 marks

(e) intersection of two graphs – 5 marks

(f) union of two graphs – 5 marks

(g) difference of two graphs – 5 marks

(h) supergraph – 15 marks

(i) connectivity – 15 marks

(j) greedy cheapest connection – 10 marks

(k) greedy discussion – 5 marks

(l) bruteforce cheapest connection – 10 marks

Context

Graphs apply to many problems and are common in issues of connectivity and networks. For this assignment we are imagining ourselves as professional programmers already and have taken on a client, a Ms. Fathima Arda Moor. Ms. Moor runs a large wheat farm in western Victoria (Australia) and has recently purchased some of her neighbouring farms; she wishes to combine together the water systems to allow for more efficient irrigation of each farm's crops.
Fortunately each of these farms had their piping systems mapped in blueprints (which are relatively easy to digitise); Unfortunately some of these farms include multiple separate piping systems for handling irrigation so she has turned to you for help.

1 Simple Graph-Joining

In this task you will be considering different ways of combining two graphs together into a single graph. This will help Fathima connect together some of the more straightforward piping systems between farms.
For each of the problems in this task, you can expect to receive weighted graphs in adjacency matrix format (with the weight representing the relative likelihood of that pipe breaking), for example the graph G (found in `G.txt`) as below:

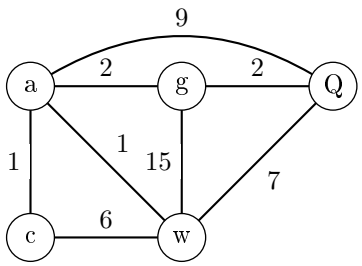

Figure 1: graph G

corresponds to the adjacency matrix:

```
[0,1,2,1,9]
[1,0,0,6,0]
[2,0,0,15,2]
[1,6,15,0,7]
[9,0,2,7,0]
```

1.1 Interpreting file input

The assignment will include several example graphs to test your code with, each will be found within their own file with each file representing an adjacency matrix and the format followed is as such:

```
firstVertexName:...:lastVertexName
space separated values for edges from firstVertexName
...
space separated values for edges from lastVertexName
```

For the example graph G, the file would look like so:

```
a:c:g:w:Q
0 1 2 1 9
1 0 0 6 0
2 0 0 15 2
1 6 15 0 7
9 0 2 7 0
```

Prepare a python program (`graphFileOps.py`) which accepts a file name and produces an adjacency matrix based on the data in that file. Make sure that you can identify vertices by their name *(which need not be one character)* and be sure to close the file when you are done with it.

1.2 Writing to a file

Later on in this assignment you will be required to write adjacency matrices to a file. Add to your existing python file (`graphFileOps.py`) a program which accepts a file name and a table representing an adjacency matrix which writes the adjacency matrix to that file. This should follow the same format as the input files.
A good way to test this would be to read in one of the input files and write to another file. If you code is correct, these files should be identical. Be sure to close the file when you are done with it.

1.3 Intersection

Write a python program (`intersection.py`) which accepts two file names (representing graphs) and finds the intersection of both graphs. In this context, the intersection refers to set of vertices and edges that are shared by any two graphs. Where an edge is shared but has different weights in each graph, the weight of the edge in the intersection graph is the **larger** of the weights. Where an intersection includes a vertex with no edges, that vertex is removed from the intersection
For example consider also the graph H (found in `H.txt`) below:

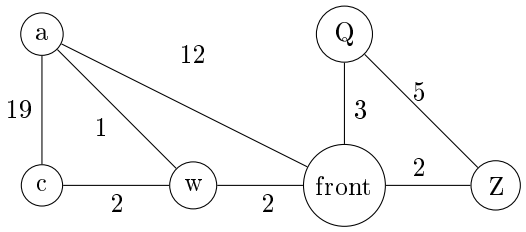

Figure 2: graph H

If we were to take the intersection of the two graphs G and H, we would receive the following graph ($G \cap H$)

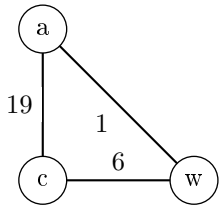

Figure 3: graph $G \cap H$

G and H share the following vertices: a,c,w and Q
G and H share the following edges: $<a,c>$,$<a,w>$ and $<c,w>$
As the intersection does not include any edges connecting Q, the vertex Q is not included in the intersection
Once your program has determined the intersection between two graphs, it should write the resultant graph to a file; this file should be named in the following format: `graphOne_and_graphTwo.txt` so for example if graph G and H were input (from files `G.txt` and `H.txt` respectively), the resultant graph would write the following:

```
a:c:w
0 19 1
19 0 6
1 6 0
```

to the file named `G_and_H.txt`
The intersection graph will allow Ms. Moor to remove the most unreliable pipes from the system where they are duplicated.

1.4 Union

Write a python program (`union.py`) which accepts two file names (representing graphs) and finds the union of both graphs. In this context, the union refers to set of vertices and edges that are in either of the two graphs. Where an edge is shared but has different weights in each graph, the weight of the edge in the union graph is the **smaller** of the weights.
For example, if we used the graphs G and H from before the union graph $G \cup H$ would be:

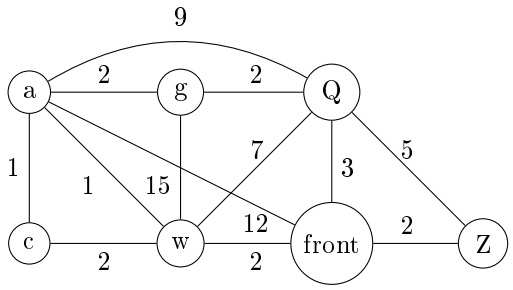

Figure 4: graph $G \cup H$

the union graph of G and H includes all of the vertices and edges of the graphs G and H however the graphs share the following edges: $<a,c>$,$<a,w>$ and $<c,w>$ and so only the cheaper of these edges is included in the final union graph.
Once your program has determined the union between two graphs, it should write the resultant graph to a file; this file should be named in the following format: `graphOne_or_graphTwo.txt` so for example if graph G and H were input, the resultant graph would write the following:

```
a:c:g:w:Q:front:Z
0 1 2 1 9 12 0
1 0 0 2 0 0 0
2 0 0 15 2 0 0
1 2 15 0 7 2 0
9 0 2 7 0 3 5
12 0 0 2 3 0 2
0 0 0 0 5 2 0
```

to the file named `G_or_H.txt`
The union graph gives Ms. Moor insight into the complete piping network across a set of farms.

1.5 Difference

Write a python program (`difference.py`) which accepts two file names (representing graphs) and finds the difference between both graphs. In this context, the difference refers to set of edges that are in only one of the two graphs. Where a vertex has no edges in the difference it is not included in the resultant graph.
For example, if we used the graphs G and H from before the difference graph $G - H$ (or equally $H - G$) would be:

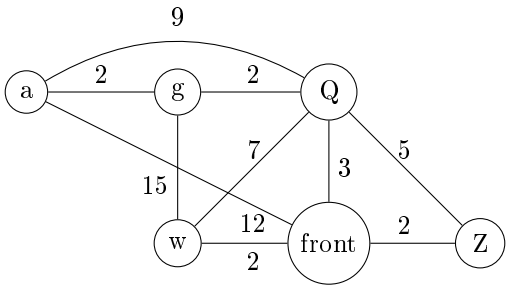

Figure 5: graph $G - H$

The difference graph of G and H includes all of the vertices of the graphs G and H however the graphs share the following edges: $<a,c>$,$<a,w>$ and $<c,w>$ and so these edges are removed from the final difference graph (regardless of their weight). Since removal of these edges results in c having no edges, c is not included in the difference graph
Once your program has determined the difference between two graphs, it should write the resultant graph to a file; this file should be named in the following format: `graphOne_xor_graphTwo.txt` so for example if graph G and H were input, the resultant graph would write the following:

```
a:g:w:Q:front:Z
0 2 0 9 12 0
2 0 15 2 0 0
0 15 0 7 2 0
9 2 7 0 3 5
12 0 2 3 0 2
0 0 0 5 2 0
```

to the file named `G_xor_H.txt`
This will show Ms. Moor what new piping is introduced between two systems.

2 Advanced Graph-Joining

This section will have you looking at some methods from connecting two graphs via another (intermediate) graph. In each of these cases you will consider three graphs as input A,B and C (where $A \cap C = \emptyset$ but $A \cap B \neq \emptyset$ and $B \cap C \neq \emptyset$). In this case $\emptyset$ represents a graph which is empty (has no edges or vertices). In other words, graphs A and C have no edges or vertices in common but both have edges and vertices in common with graph B.
The purpose of this is to produce a means of connecting vertices in graph A with vertices in graph C.
Note: that this will only be guaranteed where A, B and C are themselves connected graphs, you are not expected to find ways to join disjoint graphs together via other disjoint graphs as this may not be successful.

2.1 Super-graph

Write a program (`supergraph.py`) which accepts three file names for different graphs and, using your work from previous sections, creates a super-graph. In this context a super-graph of N graphs is a graph which includes all of the vertices and edges from all N graphs (using the cheapest edge where edges are shared).
For example, we shall introduce the graph I which is as follows:

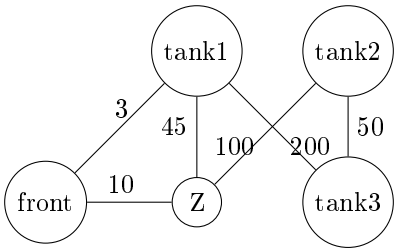

Figure 6: graph I

A comparison of Graphs G and I side by side:

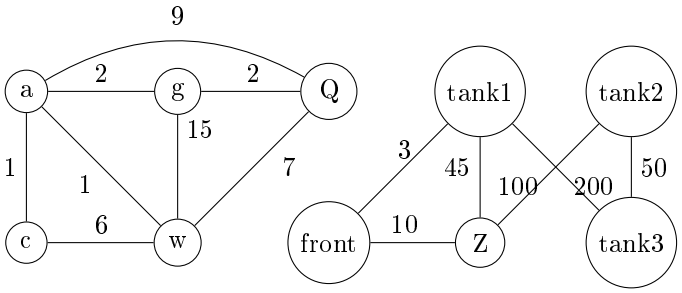

Figure 7: graph G and I in one diagram

As you can see, graphs G and I do not have any shared vertices or edges.
The resultant super graph of G and I via H would be:

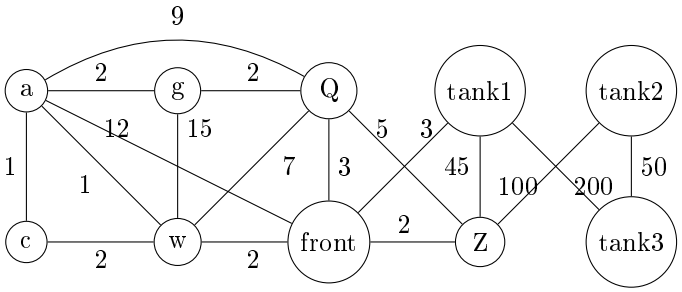

Figure 8: super-graph of graphs G,H, and I

Once you have computed the super-graph for any three graphs you should write it to a file; named format: `supergraphGraphOne-TwoIntermediateGraphTwo-Three.txt` where the graphs are listed in order received but with the intermediary graph in the middle. For example, for the graphs G, H and I we would write:

```
a:c:g:w:Q:front:Z:tank1:tank2:tank3
0 1 2 1 9 12 0 0 0 0
1 0 0 2 0 0 0 0 0 0
2 0 0 15 2 0 0 0 0 0
1 2 15 0 7 2 0 0 0 0
9 0 2 7 0 3 5 0 0 0
12 0 0 2 3 0 2 3 0 0
0 0 0 0 5 2 0 45 100 0
0 0 0 0 0 3 45 0 0 200
0 0 0 0 0 0 100 0 0 50
0 0 0 0 0 0 0 200 50 0
```

to the file named `supergraphGHI.txt`. If we received the graphs in a different order (eg. G, I, H and H, I,G) we would write to a file called `supergraphGHI.txt` and `supergraphIHG.txt` respectively as H is the intermediate graph. If there is no intermediate graph, you may simply list them in the order received.
The supergraph gives Ms. Moor a full piping system covering multiple systems wither everything (ideally) connected (although it does include superfluous pipes).

2.2 checking connectivity

Now that you have code in place to produce a super-graph, we should confirm that this super-graph actually does allow the vertices in the base graphs to reach each other via the intermediate graph.
Write a program (`connectivity.py`) which reads in a supergraph (or any combined graph) and determines whether every vertex in this combined graph can reach every other vertex in that graph.
If this is possible, you should write to a file "`graph IS connected`", otherwise you should write to a file "`graph IS NOT connected`". In both cases the filename should be the filename of the combined graph with "connectivity" at the end.
For example given the supergraph GHI as input we would write:

```
graph IS connected
```

to a file named `supergraphGHIconnectivity.txt`

As another example, let's consider graph J which is as follows:

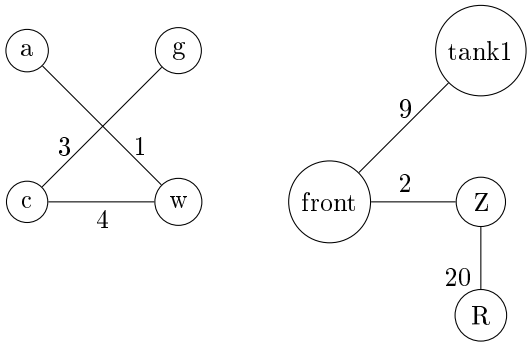

Figure 9: graph J

The supergraph GJI would look like this:

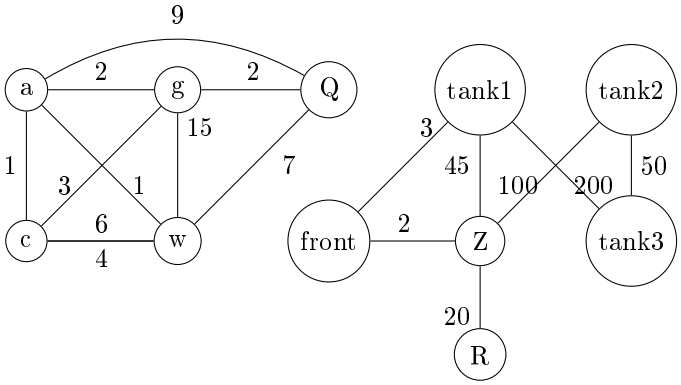

Figure 10: super-graph of graphs G,J, and I

Given the supergraph GJI and G, J, and I as input we would write:

```
graph IS NOT connected
```

to a file named `supergraphGJIconnectivity.txt`
This is as there are some vertices which cannot reach other vertices in the supergraph (ex. a cannot reach Z).
Note: we can run connectivity on any combined graph; this means for example you can run it on `G_or_I.txt` by using passing that graph in (eg. $G_o r_I$ as input) which would print:

```
graph IS NOT connected
```

to the file `G_or_Iconnectivity.txt`
The connectivity checking allows Ms. Moor to be convinced that the network suggested doesn't leave any sections isolated.

2.3 Cheapest Connection

Like in the previous part where you were asked to produce a supergraph, this part focuses on ways to connect two graphs together. In this case however we do not wish to include every single edge. Instead, using edges found in B but not A or C, create a graph with all of the vertices and edges of A and C with a single connecting path between vertices in A and vertices in C. We also want the weight of this path (likelihood of breakage) to be minimal. Where an edge is included in multiple base graphs (eg. appearing in both A and B), like with the union, we only include the cheapest weight edge.
To illustrate, let us introduce one final graph K.

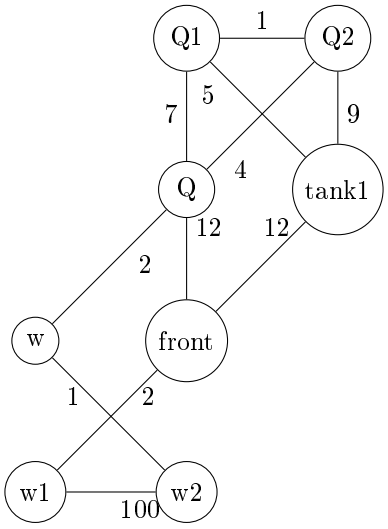

Figure 11: graph K

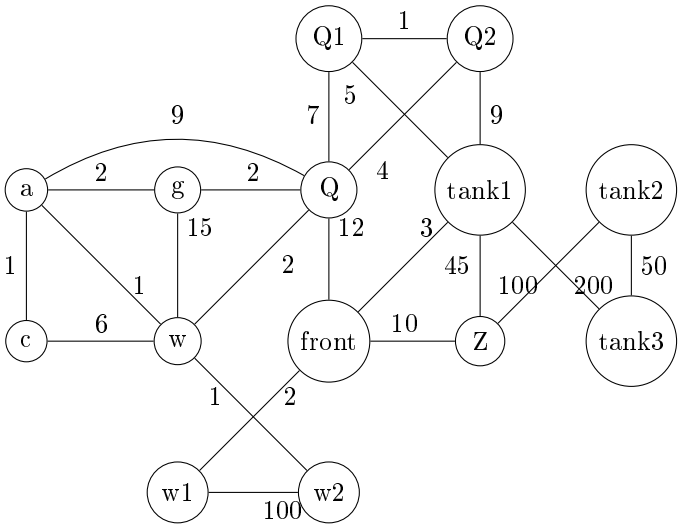

Figure 12: super-graph of graphs G,K, and I

There are many paths to connect the graphs G and I together, we could use the edge $<Q,front>$ for a cost of 12, or the path $<w,w2>$,$<w2,w1>$,$<w1,tank1>$ for a cost of $1 + 100 + 2 = 103$ but the cheapest way to connect these graphs together is with the path $<Q,Q2>$,$<Q2,Q1><Q1,tank1>$ for a cost of $4 + 1 + 5 = 10$
This makes the cheapest connection graph:

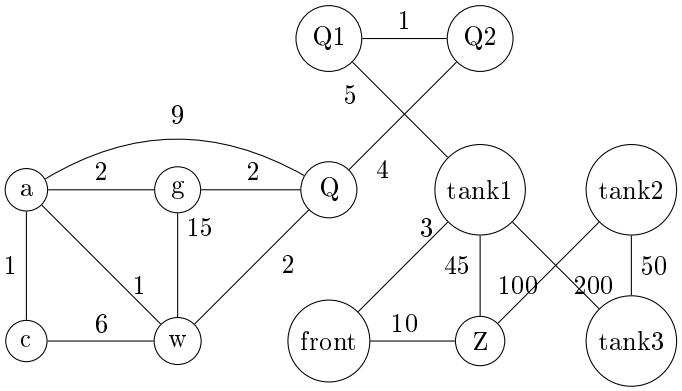

Figure 13: cheapest connection of graphs G and I via K

Notice how the **only** edges and vertices **added** from graph K are those needed to connect the two graphs. Graph K and graph G both include the edge $<w,Q>$ and graph K's edge is cheaper, so in that case we use the cheaper, however we do not include the vertex $w1$ or edge $<Q,Q2>$ or any other edges or vertices only found in K, only those that are necessary to connect graphs G and I together

We will ask you to attempt two different approaches here: greedy and brute-force.
In both cases your code would be expected to produce three files, one being the adjacency matrix of the resultant graph, another would be the file produced by your connectivity checking code for that resultant graph and the same base files, and the very last file would write to a file the cost of the path chosen.
For example for the graphs G, I, and K the file for the resultant graph would be:

```
a:c:g:w:Q:Q1:Q2:front:Z:tank1:tank2:tank3
0 1 2 1 9 0 0 0 0 0 0 0
1 0 0 6 0 0 0 0 0 0 0 0
2 0 0 15 2 0 0 0 0 0 0 0
1 6 15 0 2 0 0 0 0 0 0 0
9 0 2 2 0 0 4 0 0 0 0 0
0 0 0 0 0 0 1 0 0 5 0 0
0 0 0 0 4 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 10 3 0 0
0 0 0 0 0 0 0 10 0 45 100 0
0 0 0 0 0 5 0 3 45 0 0 200
0 0 0 0 0 0 0 0 100 0 0 50
0 0 0 0 0 0 0 0 0 200 50 0
```

and the connectivity check would say
`graph IS connected`
and finally the file for the cost of the path chosen would be:
`10`
The cheapest connection graph will show Fathima the best pipes to use to connect any two existing pipe networks together.

2.3.1 greedy

Write a program (`greedy.py`) that accepts three graphs as input (via their file-names) and follows a greedy approach to determine the cheapest path from the intermediate graph to connect the other two graphs together.

As mentioned your program should write to files for the resultant graph, the connectivity and path cost. You should name these files as:

`greedyGraphOneGraphTwoGraphThree.txt`
`greedyGraphOneGraphTwoGraphThreeconnectivity.txt`
`greedyGraphOneGraphTwoGraphThreepathCost.txt`

In the case of the graphs G, I, and K that would be:

`greedyGKI.txt`
`greedyGKIconnectivity.txt`
`greedyGKIpathCost.txt`

In addition, you should include with your submission a pdf called `"greedy.pdf"` (you may include an `.rtf` file if you prefer) where you describe what your approach was and why you chose it. Finally, you should also include in your discussion if and where any issues might arise with your application of greed (as greedy algorithms are not guaranteed to yield optimal results for all problems).
Make sure you include your full name and student ID in this file.

2.3.2 brute-force

Write a program (`bruteforce.py`) that accepts three graphs as input (via their file-names) and applies brute-force to determine the cheapest path from the intermediate graph to connect the other two graphs together.

As mentioned your program should write to files for the resultant graph, the connectivity and path cost. You should name these files as:

`bruteforceGraphOneGraphTwoGraphThree.txt`
`bruteforceGraphOneGraphTwoGraphThreeconnectivity.txt`
`bruteforceGraphOneGraphTwoGraphThreepathCost.txt`

In the case of the graphs G, I, and K that would be:

`bruteforceGKI.txt`
`bruteforceGKIconnectivity.txt`
`bruteforceGKIpathCost.txt`

This concludes the assignment