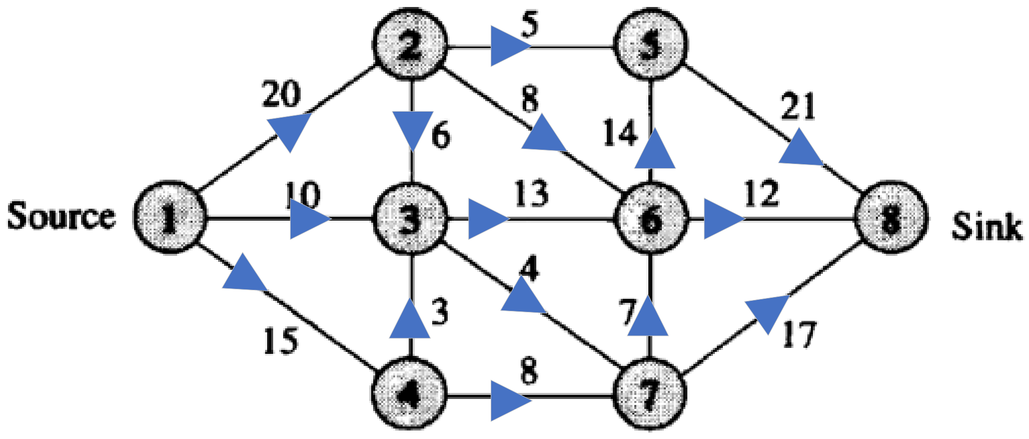
In the network optimization problems we did in Section 5 (logistics and transportation applications) however, once the data is defined, and it is clear, they tend to have a single clear optimization goal (eg: minimize total traveling time) and therefore can be formulated as a clean optimization problem. However, computational challenges may prevent us from finding one (as is the case for vehicle-routing problems). We often have to be satisfied with a good-enough answer. The second question is a practice question handling such a challenge.

Look up before attempting question 2: on the Google page (link in question (2b) or Gurobi site (link in 2c)

Theory

1. (10 points)
   a. Formulate the problem of sending the maximum amount of flow on the following network (the edges have capacities as shown in the figure) as a linear program and solve it (Excel Solver will work)



- Define a s-t *Cut* as a partition of V into disjoint sets S, T (i.e., S $\cup$T= V and S$\cap$ T = $\emptyset$) with s in S and t in T. (*read definition carefully---cuts will reappear a few times in the course*)
- Define *capacity of the cut*, Cap(S, T) = sum of the capacities of edges from S to T

- Define *Flow across the cut*, Flow(S,T): net flow out of S = sum of flows out of S - sum of flows into S.

b. A very famous and central result of network flow theory states: Max-flow = Value of Min-cut (Max-flow/Min-cut theorem). Proving *Max-flow ≤ Value of Min-cut* is easy (proving the other direction is harder). Argue why it is so. By inspection, find the min-cut in your solution.

Practice

2. [30 or 35 points] The data **HW3_qatar_tsp.txt** contains latitude and longitude data of 194 cities from Qatar. Calculate the distance matrix (use an approximation like here which is quite accurate for short distances; or use packages like `haversine` or `geopy`). The x and y-cordinates are the latitude and longitude in decimal form multiplied by 1000. EUC_2D means take these as Cartesian co-ordinates. Can also use haversine treating them as longitude and latitude. You are free to use any other functions you find.
   a. (10 points) Plot the latitude and longitude as a scatter plot using a drawing package (some options are: `matplotlib` basemap toolkit (the most advanced, but also the most difficult to use), `geopy, gmplot, plotly` …). It should look roughly like this.
   (Aside: It is not necessary but it would look great if you could plot it on an actual map isn't it? This can be done with a few lines of code with many packages. Try `bokeh` *with Google maps. (You need to get an API key from Google Maps first). Follow the sample code here. Many alternatives on the web, and feel free to explore. I added a code snippet at the end.)*

   DO EITHER (b) OR (c) --- no need to do both; if you submit both only (c) will be graded

   b. (10 points) Install the or-tools and modify the traveling salesman routine to find a tour of the cities (Hamiltonian tour---visit each city exactly once). If it is not finding a tour fast enough, try a subset of 10 cities (then 15 and so on). My suggestion is to try to understand the solved example given there and try to recreate it for this data.

   c. [Challenging: 15 points] The most powerful integer programming solver is Gurobi (along with CPLEX) and are used for serious industry optimization problems. They give a free one-year license if you download and install from a University IP address.

   Connect with the Python interface of Gurobi and find an optimal tour using the sub-tour elimination integer programming formulation we did in class (tutorial).

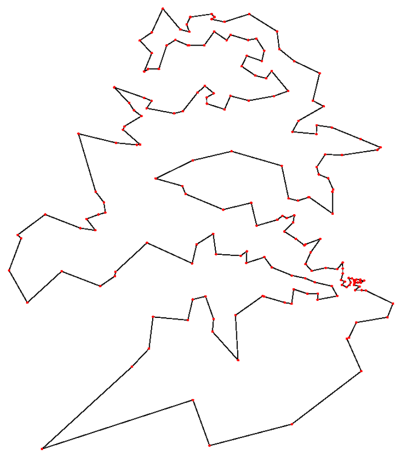
   You are free to use the code found on the Gurobi site, but you may want to read and try the interactive example here first.

You cannot generate all the sub-tour elimination constraints at once (too many). So what we want to do instead is the following computational strategy:

i. Start with a problem with only the degree =2 constraints.
ii. Check the resulting solution for sub-tours (how? Check connectivity using a BFS routine). If there is a sub-tour, add the sub-tour elimination constraints corresponding to a cut defined by the sub-tour (i.e. you are eliminating that sub-tour in the next round).
iii. Repeat till you find a tour (in which case, it is optimal). Make sure you do not discard the solution from the previous round, but start from what you had found.

This method of generating constraints on the fly is suitable when the number of constraints are exponential in the problem size. [1]

d. (10 points) Plot the resulting tour on the scatter plot. The optimal tour should look (more or less) like this.



[1] The dual version of this is called column generation. For very large problems this can go on forever. So as an approximation we just stop it when we run out of time and patch together the final solution into a workable tour as a heuristic.

```
from bokeh.models import ColumnDataSource, GMapOptions,  Arrow, OpenHead
from bokeh.plotting import gmap, output_notebook, show

output_notebook()

min_lat = min(location_data['latitude'])
max_lat = max(location_data['latitude'])
min_lon = min(location_data['longitude'])
max_lon = max(location_data['longitude'])

map_options = GMapOptions(lat=min_lat + (max_lat - min_lat) / 2,
lng=min_lon + (max_lon - min_lon) / 2,
                          map_type="roadmap", zoom=8)
For GMaps to function, Google requires you obtain and enable an API key:
# They ask for your cc but you get $300 credit for free

#     https://developers.google.com/maps/documentation/javascript/get-api-
key
#
# Replace the GOOGLE_API_KEY below with your personal API key:

map_plot = gmap("GOOGLE_API_KEY", map_options, title="Qatar")

map_plot.circle(x="longitude", y="latitude", size=6, fill_color="red",
fill_alpha=0.8, source=location_data)

show(map_plot)
```