CS 205 **Lecture Worksheet**
November 12, 2015

Going on a Road Trip!

One of the most shared geographical visualization recently that I have noticed have been road trips! They can take on various forms:

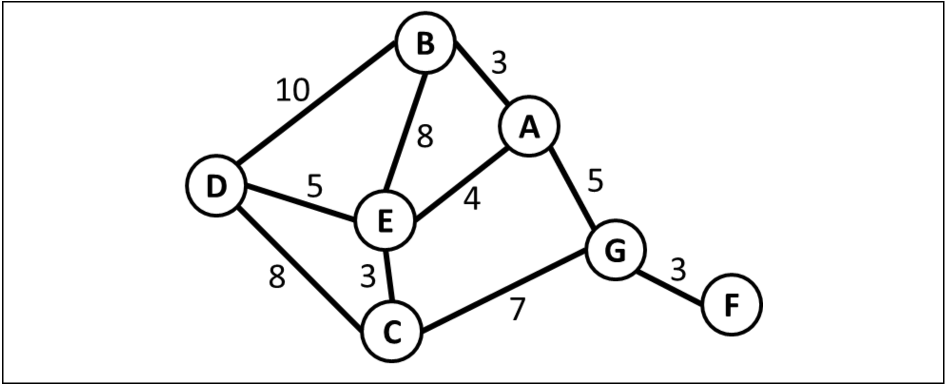
- Shortest road trip to visit every state capital
- Optimal route to visit every national park
- Best route to visit the most interesting city in every state

Traveling Salesman Problem (TSP)

In Computer Science (and Mathematics), the Traveling Salesman Problem (TSP) asks

"What is the shortest path to visit every location exactly once?"

Consider the following graph:



Solution #1: Greedy Path Algorithm

1. Start at a random node
2. Find the edge from your current node to an unvisited node that has the minimal weight
3. Repeat Step 2 until a complete path is found

What is the shortest path using a greedy path algorithm?

Is this the shortest path?

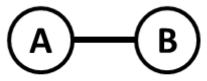
Solution #2: Brute Force Algorithm

One of the only ways to test if our shortest path is really the shortest path is to try **every single path**. This can be a lot of work:

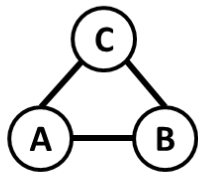
Possible Path							Total Distance
A	B	C	D	E	F	G	
…							
A	B	D	E	C	G	F	
A	B	E	D	C	G	F	
…							
B	A	E	D	C	G	F	
…							
C	E	D	B	A	G	F	
…							

How many paths are there?

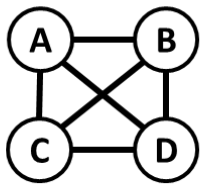
In the worst case, the graph may be **fully connected** – every node is connected to every other node via an edge:



Fully connected graph with 2 nodes



Fully connected graph with 3 nodes

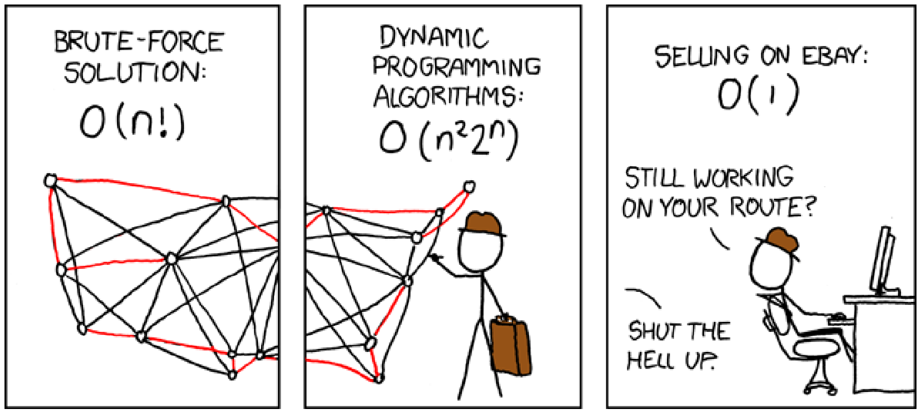


Fully connected graph with 4 nodes

Thinking about these graphs, how many Traveling Salesman Problem paths need to be checked to find a solution?

.

Nodes:	2	3	4	n	50
Paths:					



Source: http://xkcd.com/399

Task #1: Construct a 2D array from the Google API data
Google Distance Matrix API provides us the distance between our locations in a set format:

```
{ ...,
  "rows": [ { "elements": [
                { "distance": { "value": 1234 } },
                { "distance": { "value": 3456 } },
                { "distance": { "value": 4321 } },
              ] },
              { "elements": [
                { "distance": { "value": 1234 } },
                { "distance": { "value": 3456 } },
                { "distance": { "value": 4321 } },
              ] },
              ...
          ]
}
```

The only thing we are guaranteed is that the number of rows and the number of elements is always equal to the number of cities. Our goal is to construct the following:

`distances`	New York, NY	Chicago, IL	San Francisco, CA
New York, NY	…	…	…
Chicago, IL	1270079	…	…
San Francisco, CA	…	…	…

…so `distances["New York, NY"]["Chicago, IL"]` is `789.2`.

Puzzle #1: Suppose we have the following Python code that loops through all pairs of cities, create the

```
distances = defaultdict(dict)
for i, origin in enumerate(cities):
  for j, dest in enumerate(cities):
    matrix[origin][dest] = json___________________________

```

Puzzle #2: The next step is to create every possible route between the cities and calculate their distances. To do this, we will build an algorithm that is simple to describe, but recursively defined.

We'll call the function **`makePath`** and it will take three arguments:

- **`path`**: A list of the cities in our path
- **`unused`**: A list of cities not yet used in our path
- **`matrix`**: The distance matrix for calculations

The **`makePath`** function should do the following:

1. If **`unused`** is empty, we have a complete path.
   a. Calculate the distance between the cities in the **`path`**
   b. Return the distance calculated
2. If **`ununsed`** is not empty:
   a. Create a variable to store the minimum distance (you can start it equal to 999999) and path
   b. Loop through each unused city, each time:
      i. Add the currently visited unused city to a copy of the **`path`**
      ii. Remove the unused city from a copy of the **`unused`** list *(Google how to do it!)*
      iii. Calling **`makePath`** with the new path/unused, storing the return value in a variable
      iv. Checking if the return variable is smaller than the minimum variable; if so, update minimum
   c. After looping through each city, return the minimum distance and path. *(Python tuples help here)*

```
def makePath( path, unused, matrix ):
  if len( unused ) == 0:
    # Calculate distance
  else:
    # Reclusively find the minimum
```