CS 205

Lecture Worksheet
November 17, 2015

Generating Permutations: Recursive Solution

Last time, we generated a distances matrix that contained the distances from cities in our input:

`distances`	New York, NY	Chicago, IL	San Francisco, CA
New York, NY	0	1,271,382	4,677,494
Chicago, IL	1,270,079	0	3,431,581
San Francisco, CA	4,675,822	3,429,242	0

As a reminder, we set this up as a dictionary of dictionaries so that we can access any distance with the following code:

```
distances["New York, NY"]["Chicago, IL"]
```

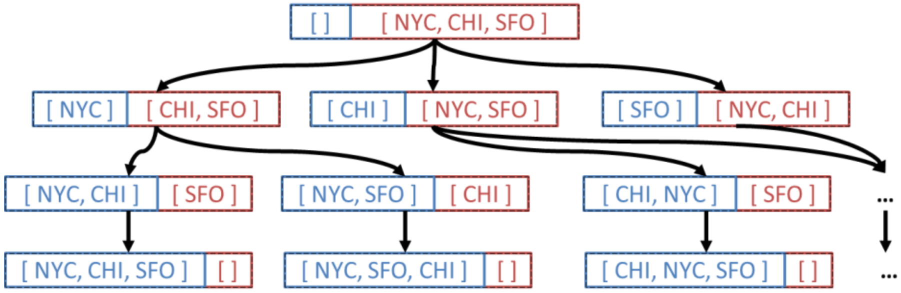
The puzzle that we left with is how we generate every permutation? To set up this problem, we will use a reclusive function with two arguments:

- `path`: The current path through the graph (as a List)
- `unused`: The cities not part of the current path (as a List)

Every recursive solution almost always has three components:

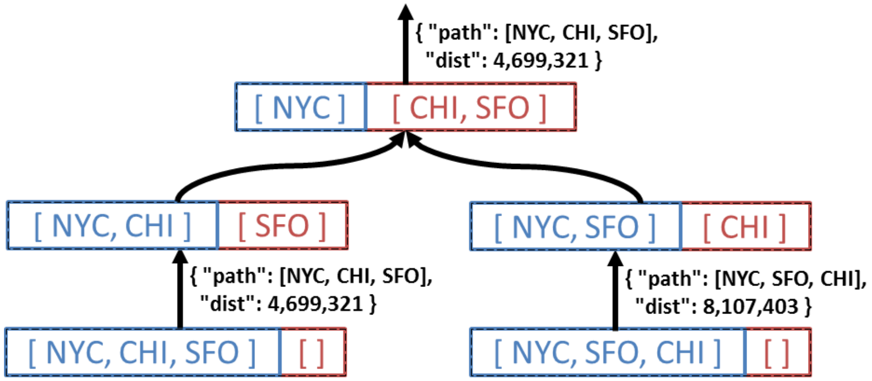
1. **Recursive Case**: If there is at least one unused city, loop through all the unused cities. For each of these unused cities, make a recursive call with the city appended to the end of the `path` list and removed from the `unused` list.

Visually, we can represent the recursive step as the following tree:



The second step to a recursive solution is the base case:

2. **Base Case**: When no cities remain in the `unused` list, return the distance and the path.
3. **Reduction**: When multiple results are returned, return the minimum of all the results.



Let's program the `makePath` function to complete this recursion:

```
def makePath( path, unused, distances ):
  if len( unused ) == 0:
    # Base case




  else:
    # Reduction result
    min = None

    # Reclusive Case
    for city in unused:








    return min
```