1 Approximation Algorithms

In the last notebook, we saw one algorithm to solve the Traveling Salesperson Problem. This time, we'll implement some algorithms which will give us approximate solutions to the TSP problem in polynomial time.

1.0.1 If you're using Datahub:

- Run the cell below **and restart the kernel if needed**

1.0.2 If you're running locally:

You'll need to perform some extra setup. #### First-time setup * Install Anaconda following the instructions here: https://www.anaconda.com/products/distribution * Create a conda environment: `conda create -n cs170 python=3.8` * Activate the environment: `conda activate cs170` * See for more details on creating conda environments https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html * Install pip: `conda install pip` * Install jupyter: `conda install jupyter`

Every time you want to work

- Make sure you've activated the conda environment: `conda activate cs170`
- Launch jupyter: `jupyter notebook` or `jupyter lab`
- Run the cell below **and restart the kernel if needed**

In [1]:
```
# Install dependencies
!pip install -r requirements.txt --quiet
```

In [2]:
```
import otter
assert (otter.__version__ >= "5.4.1"), "Please reinstall the requirements and restart your kern

grader = otter.Notebook("hw12-coding.ipynb")
import networkx as nx
import pickle

with open('tests_1.pkl', 'rb') as f:
    test_data = pickle.load(f)

rng_seed = 42
```

2 The Traveling Salesperson Problem

In this notebook, we will revisit the Traveling Salesperson Problem, which asks the following question: *Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?*

The problem can be formally defined as follows:

```
Input: An n x n matrix of distances, where M[i, j] corresponds to the distance from city i to city j.
Output: An ordered list of cities [c1, c2, ... cn] that defines the shortest tour which passes through a
```

TSP is an NP-complete problem, and unless P=NP, there is no polynomial-time algorithm that finds exact solutions to the problem. You may remember that the Dynamic Programming algorithm we implemented in the last homework was very slow :)

This time, we will focus on efficient ways to find approximate solutions.

We have provided a convenience function that, given an input matrix and a list of cities, evaluates the length of the path that passes through all of the cities in the list in order.

```
In [3]: def validate_tour(tour, matrix):
            """Returns the length of the tour if it is valid, -1 otherwise
            """
            n = len(tour)
            cost = 0
            for i in range(n):
                if matrix[tour[i-1]][tour[i]] == float('inf'):
                    return -1
                cost += matrix[tour[i-1]][tour[i]]
            return cost
```

2.0.1 Q1a) Greedy Solution from Designated Home

Implement a greedy solution, which starts at city `home` and greedily chooses the closest city that has not been visited yet, until all cities have been visited. Return the path as a list of cities on that path, starting and ending at `path[0]`. **For example, to represent the cycle `0 -> 1 -> 2 -> 3 -> 0`, return the list `[0, 1, 2, 3]`.** You may break ties arbitrarily.

```
In [4]: def tsp_greedy(matrix, home):
            """
            A greedy implementation of TSP, starting and ending at home.
```

```
    Args:
        matrix: List[List[float]]
            An n x n matrix of distances, where M[i, j] corresponds to the distance from city i
        home: int
            The index of the city to start and end at.

    Returns:
        path: List[int]
            A list corresponding to the order in which to visit cities, starting from path[0] a
            at path[-1] before returning to path[0]. path[0] should be home.
    """
    # BEGIN SOLUTION
    n = len(matrix)
    path = []
    nodes_to_visit = list(range(n))

    curr_idx = home
    curr = home
    path.append(nodes_to_visit.pop(curr_idx))
    while nodes_to_visit:
        curr_idx = min(
            range(len(nodes_to_visit)),
            key = lambda i: matrix[curr][nodes_to_visit[i]]
        )
        curr = nodes_to_visit[curr_idx]
        path.append(nodes_to_visit.pop(curr_idx))

    return path
    # END SOLUTION
```

```
In [ ]: grader.check("q1a")
```

2.0.2 Q1b) Greedy Solution

An easy way to improve over the original greedy solution is to try your greedy solution on all of the possible starting locations and choose the best one. Implement a general greedy solution, which runs the Q1a implementation on all possible home locations, and returns the best overall path.

_Your solution should take around 8 lines of code.___

```
In [6]: def tsp_greedy_general(matrix):
    """
    A generalized greedy implementation of TSP.

    Args:
        matrix: List[List[float]]
            An n x n matrix of distances, where M[i, j] corresponds to the distance from city i
```

```
    Returns:
        path: List[int]
            A list corresponding to the order in which to visit cities, starting from path[0] a
            at path[-1] before returning to path[0].
    """
    # BEGIN SOLUTION

    best_path, best_length = None, 1e9
    for home in range(len(matrix)):
        path = tsp_greedy(matrix, home)
        path_length = validate_tour(path, matrix)
        if path_length < best_length:
            best_path = path
            best_length = path_length
    return best_path

    # END SOLUTION
```

```
In [ ]: grader.check("q1b")
```

2.0.3 Q2) Approximation Algorithm for Metric TSP

When NP-complete problems are given specific constraints, they are sometimes easier to approximate. For this question, we will focus on a special variant of TSP called the **metric TSP**, where distances satisfy the following three properties: 1. Distances are non-negative: $d(i, j) \geq 0$ 2. Distances are symmetric: $d(i, j) = d(j, i)$ 3. Distances satisfy the following inequality:

$$\forall i, j, k \in V,\ d(i, k) \leq d(i, j) + d(j, k)$$

(This is called the triangle inequality, and all mathematical distance metrics obey it, which is why this is called the metric TSP!)

In other words, the graph is complete and the shortest path from one city to another city is always the direct path.

The Metric TSP problem is still NP-complete, but the following approximation returns a path that is guaranteed to be **at most twice the length of the optimal path**:

- Generate a minimum spanning tree of the graph.
- Run depth-first search on the minimum spanning tree.
- Return the nodes in the order that you found them with depth first search (i.e. by preorder number).

See DPV Section 9.2 for more details: https://people.eecs.berkeley.edu/~vazirani/algorithms/chap9.pdf#page=12

Implement this approximation algorithm below.

For this problem, run depth-first search starting at node 0, and explore neighbors in numerical order.

Feel free to reuse code from previous coding homework assignments, but please don't use any external library imports for this part. If you want to reuse your Kruskal's code to generate an MST, you should reconfigure it to take in an adjacency matrix instead of an adjacency list.

In [8]:
```
def metric_tsp_approximation(matrix):
    """
    An algorithm for solving the Metric TSP using minimum spanning trees and depth first search

    Args:
        matrix: List[List[float]]
            An n x n matrix of distances, where M[i, j] corresponds to the distance from city i

    Returns:
        path: List[int]
            A list corresponding to the order in which to visit cities, starting from path[0] a
            at path[-1] before returning to path[0].
    """
    # BEGIN SOLUTION

    class UnionFind:
        def __init__(self, n):
            self.n = n
            self.parents = [i for i in range(n)]
            self.rank = [1]*n

        def find(self, i):
            assert i >= 0 and i <= self.n-1, f"Node {i} is not in the data structure. Only node
            if i != self.parents[i]:
                self.parents[i] = self.find(self.parents[i])
            return self.parents[i]

        def union(self, i, j):
            assert i >= 0 and i <= self.n-1, f"Node {i} is not in the data structure. Only node
            assert j >= 0 and j <= self.n-1, f"Node {j} is not in the data structure. Only node
            pi, pj = self.find(i), self.find(j)
            if pi != pj:
                if self.rank[pi] < self.rank[pj]:
                    self.parents[pi] = pj
                elif self.rank[pi] > self.rank[pj]:
                    self.parents[pi] = pi
                else:
                    self.parents[pi] = pj
                    self.rank[pi] += 1

    def kruskal(matrix):
        T = []
```

```
        edges = []
        # modification to work on a matrix
        for u in range(len(matrix)):
            for v, w in enumerate(matrix[u]):
                edges.append((w, u, v))
        edges.sort()
        UF = UnionFind(len(matrix))
        for e in edges:
            u, v = e[1], e[2]
            if UF.find(u) != UF.find(v):
                UF.union(u, v)
                T.append((u, v))
        if len(T) != len(matrix) - 1:
            return None
        return T

    def make_adj_list(n, edge_list):
        adj_list = [[] for i in range(0,n)]
        for edge in edge_list:
            adj_list[edge[0]].append(edge[1]) # need to include both directions for the edge
            adj_list[edge[1]].append(edge[0])
        return adj_list

    def preorder(adj_list, s):
        def explore(adj_list, curr, visited):
            visited.append(curr)
            for v in sorted(adj_list[curr]):
                if v not in visited:
                    explore(adj_list, v, visited)
        visited = []
        explore(adj_list, s, visited)
        return visited

    mst = kruskal(matrix)
    path = preorder(make_adj_list(len(matrix), mst), 0)
    return path

    # END SOLUTION
```

```
In [ ]: grader.check("q2")
```

2.1 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit.

```
In [ ]: grader.export(pdf=False, force_save=True)
```