Module 7

Graphs

- What is a graph?
  - a collection of points called vertices, some of which are connected by line segments called edges
  - in formal mathematical notation, a graph G is an ordered pair of a set V of vertices and a set E of edges, given as G = (V, E)
- What is a node or vertex?

  A point, usually represented by a dot in a graph
- What is a loop in a graph?

  When an edge from a node is incident on itself
- What is the degree of a vertex?

  The number of vertices that are incident on a given vertex
- What is adjacency?

  The connection(s) between a node and its neighbor
- What is a path?

  A sequence of vertices where each adjacent pair is connected by an edge

- What is an undirected graph?

  it represents edges as lines between the nodes
- What is a directed graph?

- the edges provide orientation in addition to connecting nodes
- edges, which will be drawn as lines with an arrow, will point in which direction the edge connects the two nodes
- a weighted graph adds a bit of extra information to the edges

▼ An edge should support what methods?

- endpoints(): Return a tuple (u,v) such that vertex u is the origin of the edge and vertex v is the destination; for an undirected graph, the orientation is arbitrary.
- opposite(v): Assuming vertex v is one endpoint of the edge (either origin or destination), return the other endpoint.

▼ What are some basic graph algorithms?

- graph-traversal algorithms (how can one reach all the points in a network?)
- shortest-path algorithms (what is the best route between two cities?)
- topological sorting for graphs with directed edges (is a set of courses with their prerequisites consistent or self-contradictory?)

▼ What is the graph-coloring problem?

- it seeks to assign the smallest number of colors to the vertices of a graph so that no two adjacent vertices are the same color
- this problem arises in several applications, such as event scheduling: if the events are represented by vertices that are connected by an edge if and only if the corresponding events cannot be scheduled at the same time, a solution to the graph-coloring problem yields an optimal schedule

▼ How can a graph be represented with an adjacency list?

- A simple list can be used to present a graph. The indices of the list will represent the nodes or vertices in the graph. At each index, the adjacent nodes to that vertex can be stored.
- Or a dictionary may be used. Using a list for the representation is quite restrictive because we lack the ability to directly use the vertex labels.

```
graph = dict()
graph['A'] = ['B', 'C']
graph['B'] = ['E','A']
graph['C'] = ['A', 'B', 'E','F']
graph['E'] = ['B', 'C']
graph['F'] = ['C']
```

▼ How can a graph be represented with an adjacency matrix?

- A matrix is a two-dimensional array. The idea here is to represent the cells with a 1 or 0 depending on whether two vertices are connected by an edge.
- The length of the keys is used to provide the dimensions of the matrix, which are stored in cols and rows.
- We then set up a *cols* by *rows* array, filling it with zeros. The *edges_list* variable will store the tuples that form the edges of in the graph. For example, an edge between node A and B will be stored as (A, B). The multidimensional array is filled using a nested for loop.
- The neighbors of a vertex are obtained by graph[key]. The key in combination with the neighbor is then used to create the tuple stored in *edges_list.*
- Fill the multidimensional array, adjacency_matrix, by using 1 to mark the presence of an edge with the line adjacency_matrix[index_of_first_vertex] [index_of_second_vertex] = 1
- The *matrix_elements* array has its *rows* and *cols* starting from A through to E with the indices 0 through to 5. The *for* loop iterates through our list of tuples and uses the index method to get the corresponding index where an edge is to be stored.
- In the adjacency matrix produced:
  - At column 1 and row 1, the 0 there represents the absence of an edge between A and A.
  - On column 2 and row 3, there is an edge between C and B.

```
matrix_elements = sorted(graph.keys())
cols = rows = len(matrix_elements)

adjacency_matrix = [[0 for x in range(rows)] for y in range(cols)]
```

```
edges_list = []

for key in matrix_elements:
  for neighbor in graph[key]:
      edges_list.append((key,neighbor))

"""
edges_list:
[('A', 'B'), ('A', 'C'), ('B', 'E'), ('B', 'A'), ('C', 'A'),
 ('C', 'B'), ('C', 'E'), ('C', 'F'), ('E', 'B'), ('E', 'C'),
 ('F', 'C')]
"""

for edge in edges_list:
  index_of_first_vertex = matrix_elements.index(edge[0])
  index_of_second_vertex = matrix_elements.index(edge[1])
  # mark the presence of an edge
  adjacecy_matrix[index_of_first_vertex][index_of_second_vertex] = 1

"""
adjacency matrix:
[0, 1, 1, 0, 0]
[1, 0, 0, 1, 0]
[1, 1, 0, 1, 1]
[0, 1, 1, 0, 0]
[0, 0, 1, 0, 0]
"""
```

▼ What is breadth-first search?

- The breadth-first search algorithm starts at a node, chooses that node or vertex as its root node, and visits the neighboring nodes, after which it explores neighbors on the next level of the graph.
- In the worst-case scenario, each vertex or node and edge will be traversed, thus the time complexity of the algorithm is O(|V| + |E|), where |V| is the number of vertices or nodes while |E| is the number of edges in the graph.

▼ What is depth-first search?

- This algorithm traverses the depth of any particular path in the graph before traversing its breadth.
- As such, child nodes are visited first before sibling nodes.

- It works on finite graphs and requires the use of a stack to maintain the state of the algorithm.

▼ What is Dijkstra's algorithm?

- Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source.
- This algorithm is applicable to undirected and directed graphs with nonnegative weights only.
- First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on.

▼ What are greedy algorithms?

- The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.
- Greedy algorithms are much easier to conceive because the guiding rule is for one to always select the solution that yields the most benefit and continue doing that, hoping to reach a perfect solution.
- This technique aims to find a global optimal final solution by making a series of local optimal choices.
- The local optimal choice seems to lead to the solution. In real life, most of those local optimal choices made are suboptimal. As such, most greedy algorithms have a poor asymptotic time complexity.

▼ What is Prim’s algorithm?

- A greedy algorithm that constructs a minimum spanning tree through a sequence of expanding subtrees.
- The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set of the graph's vertices.
- On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree.

- The algorithm stops after all the graph's vertices have been included in the tree being constructed.
- Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is n − 1, where *n* is the number of vertices in the graph.
- The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.

▼ What is the knapsack problem?

- Given *n* items of known weights w1,w2,...,wn and values v1,v2,...,vn and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.
- The exhaustive-search approach to this problem leads to generating all the subsets of the set of *n* items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them.

▼ What is the traveling salesman problem (TSP)?

- TSP asks to find the shortest tour through a given set of *n* cities that visits each city exactly once before returning to the city where it started.
- The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then it can be stated as the problem of finding the shortest Hamiltonian circuit of the graph.