Graph Theory

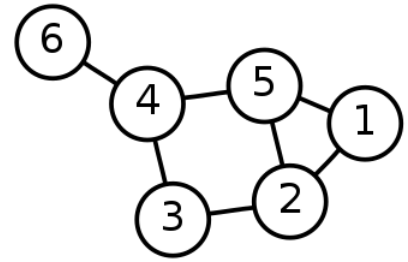
PRESENTED BY: PETER ZHU

MARCH 9, 2018

Topics of Today

1. What is a graph?
2. Graph Traversal (BFS + DFS)
3. Shortest Distance (Dijkstra's Algorithm)
4. Minimum Spanning Tree (Kruskal's Algorithm)
5. Graph Bi-Coloring (Bipartite Checking Algorithm)

What is a Graph?



- A data structure formed by connecting **nodes (a.k.a vertex)** using **edges**
- Types of Graphs:

1. Directed
2. Undirected

1. Weighted
2. Non-weighted

1. Cyclic
2. Acyclic

How is a Graph Represented?

- Two ways:
  1. Adjacency List
  2. Adjacency Matrix

Graph Representation – Adjacency List

- Using a dictionary that maps each node to a list of connected nodes
- Good for both directed and undirected graphs
- Defaults to unweighted, can be weighted through some "hacking"

```
graph = {
          0: [1],
          1: [0, 2, 3, 4],
          2: [1, 5],
          3: [1, 4],
          4: [1, 3, 5],
          5: []
     }
```

Graph Representation – Adjacency Matrix

- Uses a n×n list to represent connection between nodes
- Good for directed/undirected, weighted/unweighted

```
graph = [
    [ 0, 3, None, None],
    [ -1, 0, 0, None],
    [ None, None, 0, None],
    [ None, None, 5, 0]
]
```

Graph Representation – Adj. List vs. Adj. Matrix

- Let *E* be the number of edges in our graph, and *N* be the number of nodes in our graph
- Space complexity of adjacency list: O(E+N)
- Space complexity of adjacency matrix: $O(N^2)$

Graph Traversal - Motivation

- A ton of use cases, just to name a few:
  1. Searching
  2. Graph manipulation
  3. Foundations for other algorithms
  4. Finding shortest path between two nodes
     - Only efficient for unweighted graph

Graph Traversal – Depth First Search (DFS)

- Similar to DFS for trees
- A visited set is used to keep track of nodes already visited
- Pseudocode:

```
def dfs(graph, curr_node, visited):
   print(curr_node)
   visited << curr_node
   for neighbour of graph[curr_node]:
         if (neighbour is not in visited):
            dfs(graph, neighbour, visited)
```

Graph Traversal – Breadth First Search (BFS)

- Similar to BFS for trees
- `queue` is used to keep track of visit order
- Pseudocode:

```
def bfs(graph):
   visited <- list
   queue <- Queue

   queue << graph[0]
   visited[0] = True
   while queue is not empty:
      curr = queue.pop()
      print(curr)
      for adj in graph[curr]:
         if (visited[adj] == False):
            queue << adj
            visited[adj] = True
```

Dijkstra's Algorithm

- Algorithm for finding the shortest path from one node to every other node
- Graph can be weighted/unweighted, directed/undirected, cyclic/acyclic BUT NO NEGATIVE EDGES

```
def dijkstra(adj_matrix, source):
   nodes <- Set
   dist <- dict
   prev <- dict
   for vertex in adj_matrix:
      dist[vertex] <- INFINITY
      prev[vertex] <- None
      nodes << vertex
   dist[source] <- 0

   while nodes is not empty:
      u <- vertex in nodes with min dist[u]
      remove u from nodes

      for neighbour of u:
         new_path = dist[u] + adj_matrix[source][u]
         if new_path < dist[source]:
            dist[source] <- new_path
            prev[source] <- u

   return (dist, prev)
```

Dijkstra’s Algorithm - Complexities

- Let *V* be the number of vertices, *E* the number of edges
- Time complexity: O(V+E)
- Space complexity: O(V)

- Further exploration: Bellman-Ford algorithm, Floyd-Warshall algorithm

Minimum Spanning Trees (MST)

- A MST is a subgraph that connects all vertices together with the minimum possible total edge weight
- We will only consider MST for undirected graphs

- Intuitive example: a telephone company wants to lay cables for a community, a MST will be the most efficient way to lay these cables to reach every home

MST – Kruskal's Algorithm – Disjoint Set

- A set that is partitioned into a number of subsets
- Operations:
  - `makeset(node)` : Adds a node to the disjoint set in its own subset
  - `find(node)` : Finds the representative element (root node) of the node
  - `union(x, y)` : Merges nodes x and y

MST – Kruskal's Algorithm – Disjoint Set

```
class DSNode:
   node <- int
   parent <- DSNode

class DisjointSet:
   ds_set <- Set

   def makeset(node):
      if node not in ds_set:
         ds_set << node

   def find(node):
      if node.parent is not node:
         node.parent = find(node.parent)
      return node.parent

   def union(x, y):
      x_root, y_root = find(x), find(y)

      y_root.parent = x_root
```

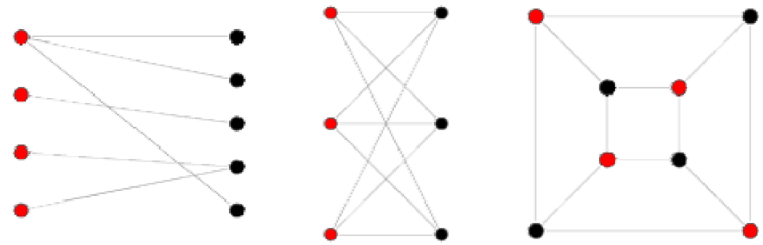
MST – Kruskal's Algorithm

```
def mst(graph):
   edges <- sort(graph.edges)

   vertices <- DisjointSet
   for vertex in graph:
      vertices << vertex

   mst_edges <- list

   for e in edges:
      if (vertices.find(e.pointA) != vertices.find(e.pointB)):
         vertices.union(e.pointA, e.pointB)
         mst_edges << e

   return mst_edges
```

MST – Kruskal's Algorithm – Complexities

- Let *E* be the number of edges, V be the number of vertices
- Average case complexity: O(E logV)
- Space complexity: O(E+V)

- Further exploration: Prim's MST algorithm



Bipartite Graph

- A bipartite graph is a graph whose vertices can be decomposed into two disjoint sets such that no two vertices within the same set are adjacent
- Used on undirected, unwieghted graphs

Bipartite Checking Algorithm

```
def bipartite(graph):
   colors <- dict
   colors[graph[0]] = True

   queue <- Queue
   queue << graph[0]

   while queue is not empty:
      curr_node = queue.pop()
      curr_color = colors[curr_node]

      for child in curr_node.neighbours:
         if colors[child] is None:
            colors[child] = not curr_color
            queue << child
         elif colors[child] == curr_color:
            return False

   return True
```

Bipartite Checking Algorithm

- Let *V* be the number of vertices and *E* be the number of edges
- Time complexity: O(V+E)
- Space complexity: O(V)

Further Exploration

- In addition to the previously mentioned:
  - Tarjan's Algorithm for finding Strongly Connected Components
  - Tarjan's Algorithm for Articulation Points
  - Johnson's Algorithm for finding all the cycles in a directed graph
  - Bellman-Ford Algorithm to detect negative cycles in the graph
  - N-Queen problem
  - Travelling Salesman problem