Python Graphs & Trees Cheat Sheet

Prof. Laio Oriel Seman - laio.seman@ufsc.br

📊 Graph Representations

Adjacency List

Dict mapping vertices to lists of neighbors

Space: O(V + E)

Example:

```
# Using dict with lists
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D'],
    'D': ['B', 'C']
}
```

Operations:

Add vertex: O(1)
Add edge: O(1)
Check edge: O(degree)
Remove edge: O(degree)

✓ Memory efficient for sparse graphs • Fast neighbor iteration
✗ Slower edge existence check

Adjacency Matrix

2D array where matrix[i][j] = 1 if edge exists

Space: $O(V^2)$

Example:

```
# Using 2D list
# Vertices: A=0, B=1, C=2, D=3
matrix = [
    [0, 1, 1, 0],   # A
    [1, 0, 0, 1],   # B
    [1, 0, 0, 1],   # C
    [0, 1, 1, 0]    # D
]
```

Operations:

Add vertex: $O(V^2)$
Add edge: O(1)
Check edge: O(1)
Remove edge: O(1)

✓ Fast edge existence check • Simple implementation
✗ Memory intensive • Slow vertex addition

Edge List

List of tuples representing edges

Space: O(E)

Example:

```
# List of edge tuples
edges = [
    ('A', 'B'),
    ('A', 'C'),
    ('B', 'D'),
    ('C', 'D')
]
```

Operations:

Add vertex: O(1)
Add edge: O(1)
Check edge: O(E)
Remove edge: O(E)

✓ Memory efficient • Good for edge iteration
✗ Slow neighbor queries • Slow edge existence check

🌐 NetworkX Essentials

Graph Creation

```
import networkx as nx

# Undirected graph
G = nx.Graph()
G.add_edge('A', 'B')
G.add_edges_from([('B', 'C'), ('C', 'D')])

# Directed graph
DG = nx.DiGraph()
DG.add_edge('A', 'B')

# Weighted graph
WG = nx.Graph()
WG.add_edge('A', 'B', weight=5)
```

Basic Operations

```
# Node operations
G.add_node('E')
G.remove_node('E')
list(G.nodes())  # ['A', 'B', 'C', 'D']

# Edge operations
G.add_edge('A', 'E')
G.remove_edge('A', 'E')
list(G.edges())  # [('A', 'B'), ...]

# Properties
G.number_of_nodes()  # 4
G.number_of_edges()  # 3
G.degree('A')        # 1
```

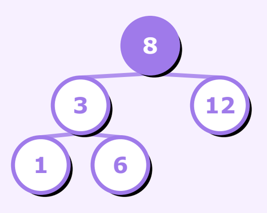
Node & Edge Attributes

```
# Node attributes
G.add_node('A', color='red', size=100)
G.nodes['A']['color']  # 'red'

# Edge attributes
G.add_edge('A', 'B', weight=3, type='road')
G.edges['A', 'B']['weight']  # 3

# Bulk attribute setting
nx.set_node_attributes(G, 'blue', 'color')
nx.set_edge_attributes(G, 1, 'weight')
```

🌳 Tree Structures

🌲 Binary Tree

Each node has at most 2 children



Time Complexity

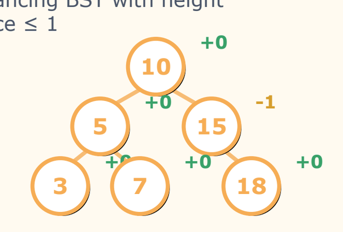
Insert: O(log n) avg, O(n) worst
Traversal: O(n)
Search: O(log n) avg, O(n) worst
Delete: O(log n) avg, O(n) worst

Key Features

- Simple structure
- In-order traversal gives sorted data
- Foundation for other trees

⚖️ AVL Tree

Self-balancing BST with height difference ≤ 1



Time Complexity

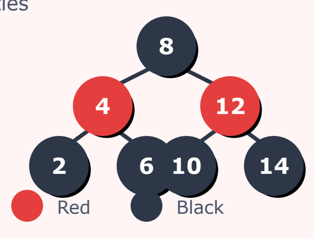
Insert: O(log n)
Rotation: O(1)
Search: O(log n)
Delete: O(log n)

Key Features

- Self-balancing
- Height difference ≤ 1
- Guaranteed O(log n) operations

🔴 Red-Black Tree

Self-balancing BST with color properties



Time Complexity

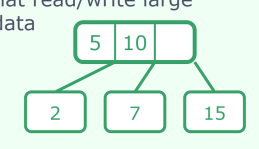
Insert: O(log n)
Recolor: O(1)
Search: O(log n)
Delete: O(log n)

Key Features

- Every node is red or black
- Root is black
- Red nodes have black children

📚 B-Tree

Multi-way tree optimized for systems that read/write large blocks of data



Time Complexity

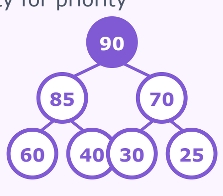
Search: O(log n)
Split: O(1)
Insert: O(log n)
Merge: O(1)
Delete: O(log n)

Key Features

- All leaves at same level
- Minimum degree t ≥ 2
- Each node has at most 2t-1 keys

⛰️ Binary Heap

Complete binary tree satisfying heap property for priority operations



Time Complexity

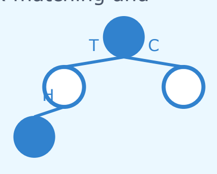
Insert: O(log n)
Heapify: O(log n)
Extract-Max: O(log n)
Build-Heap: O(n)
Peek: O(1)

Key Features

- Complete binary tree structure
- Max-heap: parent ≥ children
- Min-heap: parent ≤ children

📝 Trie (Prefix Tree)

Tree structure for efficient string prefix matching and retrieval



Time Complexity

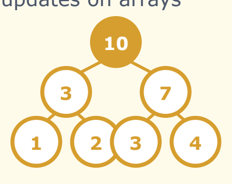
Insert: O(m)
Prefix-Search: O(p + k)
Search: O(m)
Auto-complete: O(p + k)
Delete: O(m)

Key Features

- Each path represents a string
- Common prefixes share paths
- Space efficient for large dictionaries

📊 Segment Tree

Binary tree for efficient range queries and updates on arrays



Time Complexity

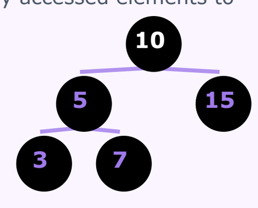
Range-Query: O(log n)
Build: O(n)
Point-Update: O(log n)
Range-Update: O(log n)

Key Features

- Each node represents a range
- Leaves represent single elements
- Supports range sum/min/max queries

🌳 Splay Tree

Self-adjusting BST that moves frequently accessed elements to root



Time Complexity

Search: O(log n) amortized
Splay: O(log n) amortized
Insert: O(log n) amortized
Delete: O(log n) amortized

Key Features

- Recently accessed nodes near root
- Excellent for temporal locality
- Self-optimizing structure

⚡ Graph Algorithms

Depth-First Search

Explores as far as possible before backtracking

Time: O(V + E)
Space: O(V)

Use Cases:

- Topological sort
- Cycle detection
- Connected components

Implementation:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited

# NetworkX version
nx.dfs_tree(G, source='A')
```

Breadth-First Search

Explores level by level using queue

Time: O(V + E)
Space: O(V)

Use Cases:

- Shortest path
- Level traversal
- Bipartite check

Implementation:

```
from collections import deque

def bfs(graph, start):
    visited = set([start])
    queue = deque([start])
    while queue:
        vertex = queue.popleft()
        print(vertex)
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

# NetworkX version
nx.bfs_tree(G, source='A')
```

Dijkstra's Algorithm

Shortest path in weighted graphs (non-negative weights)

Time: O((V + E) log V)
Space: O(V)

Use Cases:

- GPS navigation
- Network routing
- Social networks

Implementation:

```
import heapq

def dijkstra(graph, start):
    distances = {v: float('inf') for v in graph}
    distances[start] = 0
    pq = [(0, start)]

    while pq:
        curr_dist, curr = heapq.heappop(pq)
        if curr_dist > distances[curr]:
            continue
        for neighbor, weight in graph[curr].items():
            dist = curr_dist + weight
            if dist < distances[neighbor]:
                distances[neighbor] = dist
```

Bellman-Ford Algorithm

Shortest path with negative weights, detects negative cycles

Time: O(VE)
Space: O(V)

Use Cases:

- Currency arbitrage
- Network with negative costs

Implementation:

```
def bellman_ford(graph, start):
    distances = {v: float('inf') for v in graph}
    distances[start] = 0

    # Relax edges V-1 times
    for _ in range(len(graph) - 1):
        for u in graph:
            for v, weight in graph[u].items():
                if distances[u] != float('inf'):
                    distances[v] = min(distances[v],
                                       distances[u] + weight)

    # Check for negative cycles
    for u in graph:
        for v, weight in graph[u].items():
```

Floyd-Warshall Algorithm

All-pairs shortest paths using dynamic programming

Time: $O(V^3)$
Space: $O(V^2)$

Use Cases:

- Dense graphs
- Transitive closure
- All-pairs distances

Implementation:

```
def floyd_warshall(graph):
    vertices = list(graph.keys())
    n = len(vertices)
    dist = [[float('inf')] * n for _ in range(n)]

    # Initialize distances
    for i, u in enumerate(vertices):
        dist[i][i] = 0
        for j, v in enumerate(vertices):
            if v in graph[u]:
                dist[i][j] = graph[u][v]

    # Floyd-Warshall main loop
    for k in range(n):
        for i in range(n):
```

Kruskal's Algorithm

Minimum Spanning Tree using Union-Find

Time: O(E log E)
Space: O(V)

Use Cases:

- Network design
- Clustering
- Circuit design

Implementation:

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py: return False
        if self.rank[px] < self.rank[py]:
            self.parent[px] = py
```

🎯 Special Graph Types

Bipartite Graphs

Vertices can be divided into two disjoint sets

Properties:

- No odd cycles
- 2-colorable
- Perfect matching possible

Key Algorithms:

- Bipartite matching
- König's theorem
- Hall's theorem

Implementation:

```
# Check if graph is bipartite
def is_bipartite(graph):
    color = {}
    for start in graph:
        if start in color:
            continue
        queue = deque([start])
        color[start] = 0
        while queue:
            node = queue.popleft()
            for neighbor in graph[node]:
                if neighbor in color:
```

Directed Acyclic Graph

Directed graph with no cycles

Properties:

- Topological ordering exists
- Used in scheduling
- No cycles

Key Algorithms:

- Topological sort
- Longest path
- Critical path

Implementation:

```
def topological_sort(graph):
    in_degree = {v: 0 for v in graph}
    for v in graph:
        for neighbor in graph[v]:
            in_degree[neighbor] += 1

    queue = deque([v for v in in_degree if in_degree[v] == 0])
    result = []

    while queue:
        vertex = queue.popleft()
        result.append(vertex)
```

📊 Algorithm Complexity Comparison

Algorithm	Time Complexity	Space Complexity	Use Case
DFS	O(V + E)	O(V)	Cycle detection, topological sort
BFS	O(V + E)	O(V)	Shortest path (unweighted), level traversal
Dijkstra	O((V + E) log V)	O(V)	Shortest path (weighted, non-negative)
Bellman-Ford	O(VE)	O(V)	Shortest path (negative weights)
Floyd-Warshall	$O(V^3)$	$O(V^2)$	All-pairs shortest paths
Kruskal's MST	O(E log E)	O(V)	Minimum spanning tree
Prim's MST	O((V + E) log V)	O(V)	Minimum spanning tree
Topological Sort	O(V + E)	O(V)	Task scheduling, dependency resolution

💡 Practical Examples

Social Network Analysis

Find mutual friends and connections

```
import networkx as nx

# Create social network
G = nx.Graph()
G.add_edges_from([
    ('Alice', 'Bob'), ('Bob', 'Charlie'),
    ('Alice', 'David'), ('Charlie', 'David')
])

# Find mutual friends
```

Web Crawler

Crawl websites using BFS

```
from collections import deque
import requests
from bs4 import BeautifulSoup

def web_crawler(start_url, max_depth=3):
    visited = set()
    queue = deque([(start_url, 0)])

    while queue:
        url, depth = queue.popleft()
```

Shortest Path in City

Find optimal route using Dijkstra

```
import networkx as nx

# Create city map with distances
city = nx.Graph()
city.add_weighted_edges_from([
    ('Home', 'Store', 5),
    ('Home', 'School', 10),
    ('Store', 'School', 3),
    ('Store', 'Work', 8),
    ('School', 'Work', 2)
```

Dependency Resolution

Topological sort for task scheduling

```
import networkx as nx

# Create dependency graph
deps = nx.DiGraph()
deps.add_edges_from([
    ('compile', 'link'),
    ('test', 'deploy'),
    ('compile', 'test'),
    ('design', 'compile')
])
```

🔧 NetworkX Quick Reference

Graph Creation & Modification

```
G = nx.Graph() # Undirected
G = nx.DiGraph() # Directed
G.add_node("A")
G.add_edge("A", "B")
G.add_weighted_edges_from([("A","B",5)])
G.remove_node("A")
G.remove_edge("A", "B")
G.clear() # Remove all nodes/edges
```

Graph Properties

```
len(G) # Number of nodes
G.number_of_edges()
G.degree("A") # Node degree
nx.density(G) # Edge density
nx.is_connected(G)
nx.number_connected_components(G)
nx.diameter(G) # Longest shortest path
nx.radius(G) # Shortest eccentricity
```

Algorithms & Analysis

```
nx.shortest_path(G, "A", "B")
nx.shortest_path_length(G, "A", "B")
nx.all_shortest_paths(G, "A", "B")
nx.minimum_spanning_tree(G)
nx.pagerank(G) # PageRank centrality
nx.betweenness_centrality(G)
nx.clustering(G) # Clustering coefficient
nx.triangles(G) # Triangle count
```

💡 Tips & Best Practices

- 🔸 Use adjacency lists for sparse graphs (few edges)
- 🔸 Use adjacency matrices for dense graphs (many edges)
- 🔸 NetworkX graphs are unhashable - convert to frozenset if needed
- 🔸 For large graphs, consider using scipy.sparse matrices
- 🔸 DFS for connectivity, BFS for shortest paths (unweighted)
- 🔸 Dijkstra for weighted shortest paths (non-negative weights)
- 🔸 Use Union-Find for dynamic connectivity queries
- 🔸 Cache expensive computations (shortest paths, centrality)
- 🔸 Consider graph libraries: igraph, graph-tool for performance

⚡ Performance Notes

Sparse Graph ($|E| << |V|^2$)
→ **Adjacency List: Memory efficient, fast neighbor iteration**
Dense Graph ($|E| \approx |V|^2$)
→ **Adjacency Matrix: Fast edge existence check, simple operations**
Frequent Edge Queries
→ **Adjacency Matrix: O(1) edge existence check**
Dynamic Graph (many additions)
→ **Adjacency List: Easy to add nodes and edges**
Memory Constrained
→ **Edge List: Minimal memory overhead**

🌐 Master graphs to solve complex connectivity problems!

Choose the right representation • Know your algorithms • Practice with NetworkX