CHEAT SHEETS HERO

Graph Algorithms Cheatsheet

A quick reference guide to graph algorithms, commonly used in coding interviews. Covers fundamental algorithms, their complexities, and common use cases.

Graph Representations & Basics

Graph Representations

Adjacency Matrix	A 2D array where `matrix[i][j]` represents whether an edge exists between vertices `i` and `j`. • **Space Complexity:** O(V^2) • **Good for:** Dense graphs (many edges).
Adjacency List	An array of lists, where each list `adj[i]` stores the neighbors of vertex `i`. • **Space Complexity:** O(V + E) • **Good for:** Sparse graphs (few edges).
Edge List	A list of tuples, where each tuple `(u, v, w)` represents an edge from vertex `u` to vertex `v` with weight `w`. • **Space Complexity:** O(E) • **Good for:** Simple graph representation, useful for certain algorithms.

Basic Graph Properties

Vertex (Node)	A fundamental unit in a graph. Represented by a unique identifier.
Edge	A connection between two vertices. Can be directed or undirected. • **Directed Edge:** `(u -> v)`: Edge from `u` to `v` only. • **Undirected Edge:** `(u <-> v)`: Edge between `u` and `v` in both directions.
Weight	A value assigned to an edge, representing cost, distance, or other metric.
Path	A sequence of vertices connected by edges.
Cycle	A path that starts and ends at the same vertex.
Connected Graph	A graph where there is a path between every pair of vertices.

Breadth-First Search (BFS)

BFS Overview

BFS is a graph traversal algorithm that explores the graph level by level, starting from a given source vertex. It uses a queue to maintain the order of vertices to visit.

- **Time Complexity:** O(V + E)
- **Space Complexity:** O(V)
- **Use Cases:** Finding the shortest path in unweighted graphs, web crawling, social networking searches.

BFS Algorithm Steps

1. Initialize a queue and add the source vertex to it.
2. Mark the source vertex as visited.
3. While the queue is not empty:
   - Dequeue a vertex `u` from the queue.
   - For each neighbor `v` of `u`:
     - If `v` is not visited:
       - Enqueue `v`.
       - Mark `v` as visited.

BFS Example (Python)

```python
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        vertex = queue.popleft()
        print(vertex, end=" ")  # Process the vertex

        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

# Example graph
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

bfs(graph, 'A') # Output: A B C D E F
```

Depth-First Search (DFS)

DFS Overview

DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It uses a stack (implicitly through recursion) to keep track of the vertices to visit.

- **Time Complexity:** O(V + E)
- **Space Complexity:** O(V) (in the worst case, for recursive calls)
- **Use Cases:** Detecting cycles in a graph, topological sorting, solving mazes.

DFS Algorithm Steps

1. Mark the current vertex as visited.
2. For each neighbor `v` of the current vertex:
   - If `v` is not visited:
     - Recursively call DFS on `v`.

DFS Example (Python)

```python
def dfs(graph, vertex, visited):
    visited.add(vertex)
    print(vertex, end=" ")  # Process the vertex

    for neighbor in graph[vertex]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Example graph
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

visited = set()
dfs(graph, 'A', visited) # Output: A B D E F C
```

Dijkstra's Algorithm

Dijkstra's Overview

Dijkstra's algorithm is used to find the shortest paths from a source vertex to all other vertices in a weighted graph (with non-negative edge weights).

- **Time Complexity:** O(V^2) (with adjacency matrix), O(E log V) (with priority queue)
- **Space Complexity:** O(V)
- **Use Cases:** Finding shortest routes in navigation systems, network routing.

Dijkstra's Algorithm Steps

1. Initialize distances to all vertices as infinity, except the source vertex which is set to 0.
2. Create a set of unvisited vertices.
3. While the set of unvisited vertices is not empty:
   - Select the unvisited vertex with the smallest distance (using a priority queue for efficiency).
   - For each neighbor `v` of the selected vertex `u`:
     - Calculate the distance to `v` through `u`.
     - If this distance is shorter than the current distance to `v`:
       - Update the distance to `v`.

Dijkstra's Example (Python)

```python
import heapq

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

    while pq:
        dist, vertex = heapq.heappop(pq)

        if dist > distances[vertex]:
            continue

        for neighbor, weight in
graph[vertex].items():
            distance = dist + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance,
neighbor))

    return distances

# Example graph (weighted)
graph = {
    'A': {'B': 5, 'C': 2},
    'B': {'A': 5, 'D': 1, 'E': 4},
    'C': {'A': 2, 'F': 9},
    'D': {'B': 1, 'E': 6},
    'E': {'B': 4, 'D': 6, 'F': 3},
    'F': {'C': 9, 'E': 3}
}

start_node = 'A'
shortest_paths = dijkstra(graph, start_node)
print(f"Shortest paths from {start_node}:
{shortest_paths}")
# Expected output (order may vary slightly due
to heapq):
# Shortest paths from A: {'A': 0, 'B': 5, 'C':
2, 'D': 6, 'E': 9, 'F': 11}
```