

Contents

1	Types of Graphs	3
1.1	Undirected Graph	3
1.2	Directed Graph(Digraph)	3
1.3	Weighted Graphs	3
1.4	Tree	3
1.5	Rooted Trees	4
1.6	Directed Acyclic Graphs(DAGs)	4
1.7	Bipartite Graph	4
1.8	Complete Graph	4
2	Representing Graphs	4
2.1	Adjacency Matrix	4
2.2	Adjacency List	5
2.3	Adjacency Map	6
2.4	Edge List	6
3	Cons	6
4	Graph problems and algorithms used	7
4.1	shortest path problem	7
4.2	Connectivity	7
4.3	Negative cycles	7
4.4	Strongly Connected Components	7
4.5	Travelling Salesman Problem (TSP)	7
4.5.1	Bridges	7
4.6	Articulation Points	8
4.7	Minimum Spanning Tree(MST)	8
4.8	Network flow: Max flow	8
5	Graph Algorithms	8
5.1	Depth First Search (DFS)	8
5.1.1	Use cases:	8
5.2	Breadth First Search (BFS)	9
5.3	Graph theory on grids	10
5.4	Topological Ordering	11
5.5	SSSP on DAG	12

5.6	Dijkstra's shortest path algorithm	13
-----	--	----

Programmer's HandBook

Akash Tesla

July 2025

Graph Theory

1 Types of Graphs

1.1 Undirected Graph

A Undirected graph is a graph in which edges have no orientation. the edge (u,v) is identical to (v,u)

1.2 Directed Graph(Digraph)

A directed graph is a graph in which edges have orientation. for example edge (u,v) is the edge from u to v

1.3 Weighted Graphs

Weighted graphs are graphs in which it's edges contains a certain value attributed to certain value such as cost, distance, quantity, etc...

1.4 Tree

A tree is a an Undirected graph with no cycles. Equivalently it's a connected graph with n nodes and $n-1$ edges

1.5 Rooted Trees

A rooted tree is a tree with designated root node where every edge either points away from or towards the root node. When edges point away from the root node the graph is called arborescence or out tree and when the edges point towards the root node the graph is called anti-arborescence.

1.6 Directed Acyclic Graphs(DAGs)

Directed Acyclic Graphs are Directed graphs with no cycles. These graphs are commonly used in representing structure with dependencies. Several efficient algorithms exist to operate on DAGS

1.7 Bipartite Graph

A Bipartite graph is one whose vertices can be split into two independent group U,V such that every edge connects between U and V, and there is no odd cycles

1.8 Complete Graph

A complete graph is one where there is a unique edge between every pair of nodes. A complete graph with n vertices is denoted as K_n

2 Representing Graphs

2.1 Adjacency Matrix

An adjacency matrix m is a square matrix used to represent a graph. where each row and column corresponds to a vertex, populate the matrix with condition of $m[\text{from node id}][\text{to node id}] = \text{weight of the edge}$ 1 if it is unweighted

Example:

$$M = \begin{array}{c|ccc} & A & B & C \\ \hline A & 0 & 1 & 0 \\ B & 0 & 0 & 1 \\ C & 1 & 0 & 0 \end{array}$$

Pros

- Space efficient for dense graph
- Takes $O(n)$ for look up

Cons

- Requires $O(v^2)$ space
- Requires $O(v^2)$ time to iterate over all edges
- where, v is the number of vertices in the graph

2.2 Adjacency List

An adjacency list represents graphs with a map with key as vertices that stores a list of neighbour nodes/ list of edges it connects to with its cost if it's weighted

Example: "A":[(C,2),(D,4)], "B":[(A,4),(C,1),(D,7)]

Pros

- Space efficient for sparse graph
- Iterating over all edges is efficient
- Preferred for traversal

Cons

- Edge lookup is $O(\deg(v))$
- Not ideal for representing denser graphs
- where, v is the number of vertices
- $\deg(v)$, is the degree of freedom of the vertices aka number of edges connected to it

2.3 Adjacency Map

A adjacency map represents graphs with a map with key as vertices that stores another map with key as edges and value as it's weight

Example: "A":{"C":2,"D":6,B:"A":7,"C":4

Pros

- Edge weight lookup is $O(1)$
- Space efficient for sparse graph

Cons

- Space inefficient than adjacency list if you don't want $o(1)$ weight lookup
- Not ideal for representing denser graphs

2.4 Edge List

Edge list represents graph as list of unordered edges.

Example: [(C,A,4),(C,A,1),(B,C,6)]

Pros

- Space efficient for sparse graphs
- Iterating over all edges is efficient

3 Cons

- Less space efficient for denser graphs
- Edge weight lookup is $O(E)$
- where, E is number of edges in the graph

4 Graph problems and algorithms used

4.1 shortest path problem

given a weighted graph find the shortest path in the graph

Algorithms: BFS, Dijkstra's algorithm, A*, Bellman-Ford, Floyd-Warshall...

4.2 Connectivity

Is there a path between Node A and Node B.

Algorithms: union find data structure, Any search algorithms (DFS, BFS)

4.3 Negative cycles

Does my weighted graph have a negative cycle if so where, Negative cycle is a path cycle in which the weights add up to a negative number

Algorithms: Bellman-Ford and Floyd-Warshall

4.4 Strongly Connected Components

Strongly connected components are self-contained cycles within a directed graph where every vertex can reach every other vertex in the same cycle

Algorithms: Tarjan's and Kosaraju's algorithm

4.5 Travelling Salesman Problem (TSP)

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits the city exactly once and returns to the origin city, this is an NP-Hard problem. Algorithms: Held-Karp, branch and bound, Ant colony optimization and other approximation algorithms

4.5.1 Bridges

A bridge is an edge in a graph in which its removal would increase the number of connected components. They could hint at weak points, bottlenecks or vulnerabilities in a graph

4.6 Articulation Points

An articulation point is any node in the graph in which its removal would increase the number of connected components, They could hint at weak points, bottle neck or or vulnerabilities in a graph

4.7 Minimum Spanning Tree(MST)

A Minimum spanning tree is a subset of a connected, edge-weighted graph that connects all the vertices together, without any cycles and with minimum possible cost. MSTs are used in designing a least cost network, circuit design, transportation networks and etc..

Algorithms: Kruskal's, Prim's, Boruvka's algorithm

4.8 Network flow: Max flow

With an infinite source how much flow can we push thorough the network consider edges as Pipes with water capacity or roads with car capacity.

Example: Number of cars that can sustain through the traffic.

Algorithms: Ford-Fulkerson, Edmonds-karp, Dinic's algorithm

5 Graph Algorithms

5.1 Depth First Search (DFS)

DFS works by exploring the deepest part of the network from a starting node, then back tracking once you hit the dead-end.

Time complexity : $O(V+E)$

5.1.1 Use cases:

- Compute a graph's minimum spanning tree
- detect and find cycles in an graph
- check if the graph is Bipartite
- Topologically sort the nodes of a graph

- Find the bridges and articulation points
- Generate mazes
- Find augmenting paths in a flow network

Pseudo code

Algorithm 1 Depth First Search (DFS)

```

1: function dfs(node, visited, graph)
2: if Node is visited then
3:   return
4: end if
5: Mark as visited
6: Problem specific logic goes here
7: for next in node.neighbours do
8:   dfs(next)
9: end for

```

Either implement logic in the given section or in a separate function and just call the dfs function

5.2 Breath First Search (BFS)

A BFS traverses node layer by layer, it starts at some arbitrary node of a graph and explores the neighbours nodes first before moving on to the next level of neighbours, it uses queue to traverse the graph, BFS produces a prev array which can be used to construct a spanning tree, a tree which connects all the vertices.

Time complexity: $O(V+E)$

Algorithm 2 Breath First Search (BFS)

```
1: function bfs(node)
2:   q.enqueue(node)
3:   prev = [null,...,null] of size n
4:   while q is not empty do
5:     node = q.dequeue()
6:     for next in node.neighbours do
7:       if next is not visited then
8:         q.enqueue(next)
9:         Mark as visited
10:        prev[next] = node
11:      end if
12:    end for
13:  end while
```

Algorithm 3 Reconstruct Path Algorithm

```
function reconstruct_path(start,target,prev)
  current = target
  prev[start] = null
  path = [ ]
  while current is not null do
    path.append(current)
    current = prev[current]
  end while
  reverse(path)
  if path[0]==start then
    return path
  else
    return [ ]
  end if
```

5.3 Graph theory on grids

Grids can be represented by a graph where each cell is treated as node, and edge connect to it's adjacent cells, depending on the problem a node can 4

neighbours (up, down, left, right) or 8 neighbours (up, down, left, right and other 4 diagonal movements). using these rules a graph can be represented in any forms but a better way to approach the problem is by using direction vectors, we know by the rule set the neighbours of the node can be found by adding 1 in each direction.

Algorithm 4 Direction Vectors

```

1: dr = [+1,-1,0,0]
2: dc = [0,0,+1,-1]
3: for i=0; i<len(dr); i++ do
4:   rr = r + dr[i]
5:   cc = c + dc[i]
6:   if rr < 0 or cc > 0 or rr >=R or cc >= C then
7:     continue {Skip invalid cells}
8:   end if
9: end for

```

5.4 Topological Ordering

Topological Ordering is ordering of vertices in directed acyclic graph (DAG) such that for every edge $u \rightarrow v$, u comes before v. It is used to model dependencies like Task scheduling or course prerequisites.

Topological sort algorithm

The topological sort algorithm can find the a topological ordering in $O(V+E)$ time.

Algorithm 5 Topological sort

```
function top_sort()
N = number of nodes
visited = [false]*N
post order = []
for current = 0; current < N; current++ do
    if current is not visited then
        {Modify the dfs to return the visited nodes}
        visited nodes = dfs(current, visited, graph)
        post order.extend(visited nodes)
    end if
end for
topological order = reverse(post order)
return topological order
```

Algorithm 6 Depth First Search Modified 4 Top sort (DFS)

```
1: function dfs(node, visited, graph)
2: visited_nodes = []
3: if Node is visited then
4:     return
5: end if
6: Mark as visited
7: for next in node.neighbours do
8:     visited_nodes.extend(dfs(next, visited, graph))
9: end for
10: visited_nodes.append(node)
11: return visited_nodes
```

5.5 SSSP on DAG

Single source shortest path on DAG can be solved more efficiently in $O(N+E)$ time than in general graphs as on general graph it's an NP-Hard problem. Since DAG has no cycles, we can exploit the topological ordering of the vertices and make sure every edge is relaxed once in correct order, relaxed in the sense the accumulated weight is the lowest.

Algorithm 7 SSSP on DAG

```
function sssp_on_dag(graph, source)
  N = number of nodes
  order = top_sort(graph)
  dist = [ $\infty$ ]*N
  dist[source] = 0
  for u in order do
    for (neighbour,weight) in u.neighbours do
      if dist[u]+weight < dist[neighbour] then
        dist[neighbour] = dist[u] + weight
      end if
    end for
  end for
```

5.6 Dijkstra's shortest path algorithm

Dijkstra's algorithm is a Single source shortest path algorithm for graph with non-negative weights, the algorithm is typically $O(E \cdot \log(V))$ depends on the data structures used.

Algorithm Prerequisites

A constraint for Dijkstra's algorithm is that all edge weights should be non-negative. This constraint is imposed to ensure that once a node's been visited its optimal distance cannot be improved.

Algorithm 8 Dijkstra's Algorithm

```
function dijkstra(graph,start)
  N = number of nodes
  Visited = [false]*N
  dist = [ $\infty$ ]*N
  dist[start] = 0
  pq = empty priority queue
  pq.insert((s,0))
  while pq.size() != 0 do
    index,minValue = pq.poll()
    mark index as visited
    neighbours = graph[index]
    for neighbour in neighbours do
      if neighbour is visited then
        continue
      newDist = dist[index] + neighbour.cost
      if newDist < dist[neighbour] then
        pq.insert((neighbour,newDist))
      end if
    end if
  end for
end while
return dist
```

TODO implemnet dijkstra's algorithm in the morning