Graphs

- (V, E)

Edge Types

① Directed edge/graph
- ordered pair (u, v)
- ex. a flight, route network

② Undirected edge
- unordered pair (u, v)
- ex. flight route, flight network

Terminology

- incident on a vertex

ex) a, d, b are incident on V

- Path : sequence of alternating vertices and edges
- Cycle : circular sequence of alternating vertices and edges

Properties

① $\sum deg(V) = 2m$

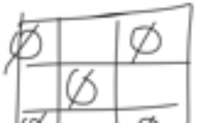
② undirected graph w/ self loops,
$m \leq n(n-1)/2$

Graph Structures

① Edge List

② Adjacent List

③ Adjacency Matrix
- 2D array



1: edge

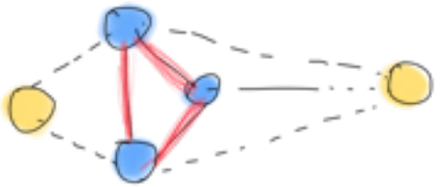
Performance

n: vertices
m: edges

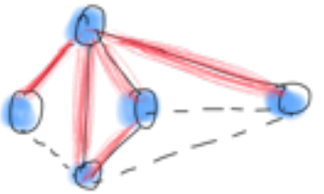
	Edge List	Adjacency List	Matrix
v.incidentEdge()	m	deg(v)	n
u.isAdjacentTo()	m	min(deg(v), deg(w))	1
insertVertex(v)	1	1	n^2
eraseVertex(v)	m	deg(v)	n^2

DFS

Subgraphs



Subgraph



Spanning subgraph

- containing all the

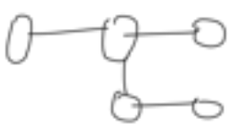
vertices of G.

Connectivity

- If there is a path between every pair of vertices.
- A connected component of a graph G is a maximal connected subgraph of G.

Tree and Forests

- Tree (undirected graph)
  - T is connected
  - T has no cycles



- Forest
  - undirected graph w/o cycles

* A spanning tree isn't unique unless the graph is a tree.

DFS

- visit all the edges & vertices
- determine whether G is connected
- compute a spanning forest of G
- O(n+m) time

- DFS is what Euler tour is to binary trees.

DFS Algorithm

```
Algorithm DFS(G)
    input graph G

for all u ∈ G.vertices()
    u.setLabel(Unexplored)
for all v ∈ G.vertices()
    if v.getLabel() = Unexplored
```

→ discovery edge — unexplored vertex
→ back edge

DFS Properties

① DFS(G, v) visits all the vertices and edges in the connected component of v.

② The discovery edges form a spanning tree of the connected component.

Analysis of DFS

- Setting/getting vertex/edge labels take O(1) time.
- Each vertex < unexplored / visited
- Each edge < unexplored / discovery or back

Path Finding

- Use stack S to keep track of path between the start vertex and the current vertex

Cycle Finding

- Specialize DFS to find a simple cycle
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w.

BFS

- visit all the vertices and edges of G
- determine whether G is connected
- compute connected components
- compute a spanning forest of G
- O(n+m) time
- BFS …
  ① find a path with the minimum number of edges between two given vertices.
  ② find a simple cycle, if there is one.

BFS Algorithm

- use queue to store
→ discovery edge — unexplored edge
→ cross edge

Properties

① BFS visits all the vertices and edges
② The discovery edges labeled by form a spanning tree
③ Every path from s to v has at least i edges

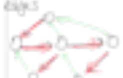
Analysis

- Each vertex < unexplored / visited
- Each edge < unexplored / discovery or cross

DFS vs BFS

Applications	DFS	BFS
spanning forest, connected components, paths, cycles	✓	✓
shortest paths		✓
biconnected components	✓	

DFS
Back edge (v, w)
- w is an ancestor of v in the tree of discovery edges



BFS
Cross edge (v, w)
- w is in the same level as v or in the next level.

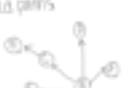
Digraphs

- Each edge goes in one direction
- If G is simple, m ≤ n(n−1)

Reachability

DFS tree rooted at v: vertices reachable from v via directed paths

ex



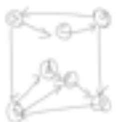
Strong Connectivity

① Pick a vertex in G
② Perform DFS from v in G
- If there's a w not visited, print "no"
③ Let G' be G with edges reversed
④ Perform DFS from v in G'
- If there's a w not visited, print "no"
  else, print "yes"
- Running time: O(n+m)
- [illegible] separate vertices [illegible]

[illegible]

Strongly Connected Components

- Can be done in O(n+m) times using DFS, but is more complicated (similar to biconnectivity).



Transitive Closure

- It provides reachability information about a digraph.

⇩

Dynamic Programming

- Floyd-Warshall Transitive Closure

DAGs and Topological ordering

- DAG (Directed acyclic graph) is a digraph that doesn't have directed cycles.

Topological Sorting

- [illegible]
- use DFS

Algorithm TopologicalSort(G)
  H ← G // copy of G
  n ← G.numVertices()
  while H is not empty do
    Let v be a vertex with no outgoing edges
    Label v ← n
    n ← n-1
    Remove from H

- Running time: O(n+m)
- Topological sorting is not unique.

Minimum Spanning Trees

- Spanning tree of a weighted graph with minimum total edge weight

Cycle Property

for every edge f of C, weight(f) ≤ weight(e)

Kruskal's Algorithm

- A priority queue stores the edges outside clusters
  - Key: weight
  - Element: edge
- [illegible]
- maintains a forest of trees
- Partition-based: $O((n+m)\log n)$
  - PQ operations: $O(m \log n)$
  - UF operations: $O(n \log n)$

Prim-Jarnik's Algorithm

- Similar to Dijkstra algorithm
- pick an arbitrary vertex s and grow MST, starting from s.
- At each step:
  1. Add to the cloud the vertex u outside the cloud with the smallest distance label
  2. we update the labels of vertices adjacent to u.
- [illegible]
- heap-based adaptable priority queue

	Vertex	Edge
start:	B	BA 46
	D	BC 15
	C	~~BD 6~~
	E	~~DC 2~~
	A	DE 23
		~~CE 5~~
		CA 13

B → D → C → A → E

Shortest Paths (BFS)

Properties

1. A subpath of a shortest path is [illegible]

... a shortest path ...

① There is a tree of shortest paths from a start vertex to all the other vertices.

- Dijkstra computes the distances of all the vertices from a given start vertex s.
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative
- Edge Relaxation
  d(z) ← min{d(z), d(u) + weight(e)}

Dijkstra Algorithm

- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- A heap-based adaptable priority queue
- Store 2 labels with each vertex
  - distance, entry in priority queue
- Running time
  - O((n+m) log n) adjacency list
  - O(m log n) graph is connected
- Based on the greedy method
- Why doesn't work for negative weight?

⇒ It could mess up the distances for vertices already in the cloud

Bellman-Ford Algorithm

- works with negative weight edges
- Running time: O(nm)
- must allowed directed edges

DAG-based Algorithm

- Uses topological order
- Faster than Dijkstra's algorithm
- Running time: O(n+m)

* MST example

starting from [A]

* Topological Sort

* Regular Expressions

R.E.: a(ab|ba)*ab

NFA

DFA

Start/Scan	a	b
S	[illegible]	∅
[illegible]	[illegible]	∅
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	∅

(DFA)

R.E.: a(ab|ba)*ab

Start/Scan	a	b
q_0	[illegible]	∅
[illegible]	[illegible]	∅
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	∅