Week 9 - GRAPHS

Brenden West - 2023

Contents

Reading & Videos

- Lafore Ch. 13 - Graphs
- https://www.coursera.org/learn/algorithms-part2/home/week/1

Learning Outcomes

- overview of graphs
- directed & undirected graphs
- graph implementation & traversal
- Spanning trees

Reference

- https://algs4.cs.princeton.edu/41graph/
- https://algs4.cs.princeton.edu/42digraph/
- https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/
- https://www.geeksforgeeks.org/topological-sorting/

Overview

In computer science, a **Graph** is a non-linear data structure consisting of **vertices (nodes)** and **edges** - lines that connect any two vertices.

Graphs are useful for solving real-world network problems. Some examples, with vertices in parens:

- road maps (towns)
- social network (people)
- airline routes (airports)
- mazes (corners & ends)
- college degree programs (courses)

Unlike other data structures, graphs are not modified after being created. Instead they are used to answer questions about how vertices are related.

All trees are **connected graphs** with hierarchical order and without cycles.

Edges & Paths

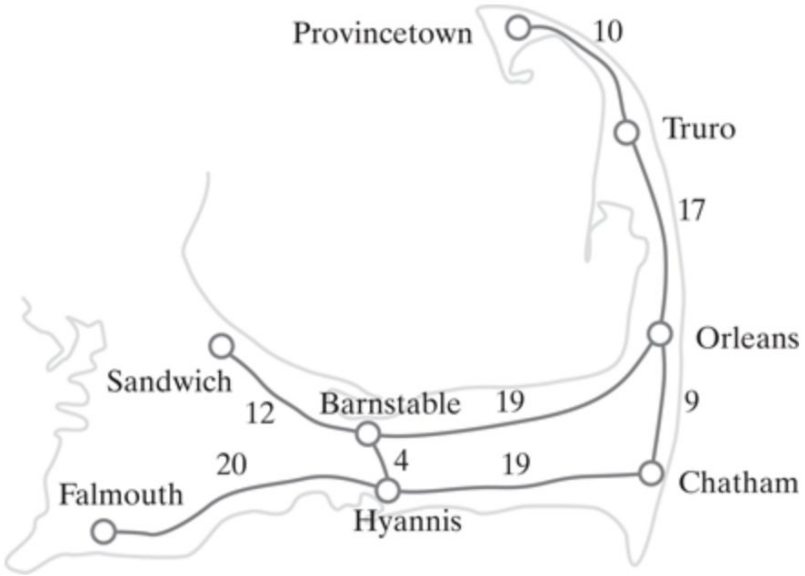


Figure 29-3
A weighted graph

An edge may be **undirected** if there is a two-way relationship between the vertices it connects, or **directed** if the relationship is one-way (e.g. one-way streets).

A graph with directed edges is called a **directed graph** or digraph.

Edges may have an associated weight or cost (e.g. road distance, driving time, etc.).

A **path** between two vertices in a graph is a sequence of edges, where path length is the number of edges.

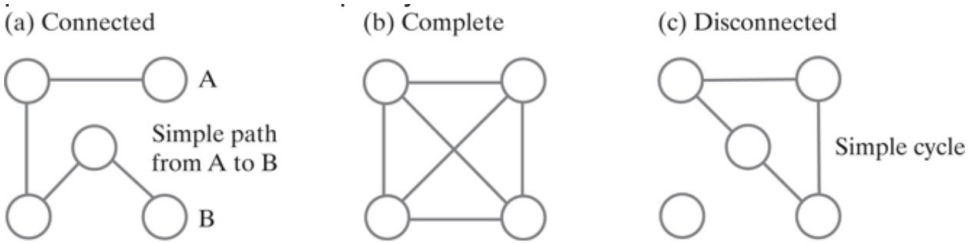
Edges & Paths, cont.

A **cycle** is a path that begins and ends at the same vertex and a **simple cycle** passes through other vertices only once each.

The weight of a path in weighted graph is the sum of its edge weights. Algorithms can use weights to choose from among multiple valid paths (e.g. to minimize cost).

A **connected graph** has a path between every pair of vertices and is complete if every vertex is connected to every other vertex. A disconnected graph has some vertices that can't be reached from all other vertices.

Two vertices in an undirected graph are adjacent (aka neighbors) if joined by an edge.



Traversal

Graph applications focus on the connections between vertices, rather than the contents of vertices.

Graph traversal can begin from any vertex (origin) and visits only vertices reachable from the origin.

Visited vertices are marked to avoid repeated visits.

The order in which neighbors are visited can vary according a graph's implementation.

Breadth-first Search (BFS)

A breadth-first search visits each of a vertex's neighbors before visiting neighbors of neighbors and so on. Level-order tree traversal is an example of BFS.

BFS uses a queue to hold the visited vertices and traversal order is the order that vertices are added to the queue.

Applications:

- Shortest path for an unweighted graph
- Peer-to-peer networks
- Web crawlers
- Social networks
- Garbage collection (in programming)
- Cycle detection

Depth-first Search (DFS)

A depth-first search follows a single path as deeply as possible before following other paths. Inorder, preorder, and postorder traversal are examples of DFS.

DFS traversal can be recursive and uses a queue to track visited vertices. The traversal order is the order in which vertices are added to the queue.

DFS is useful to determine the path between two vertices.

Applications:

- Detecting a cycle
- Path finding
- Topological sorting
- Solving mazes

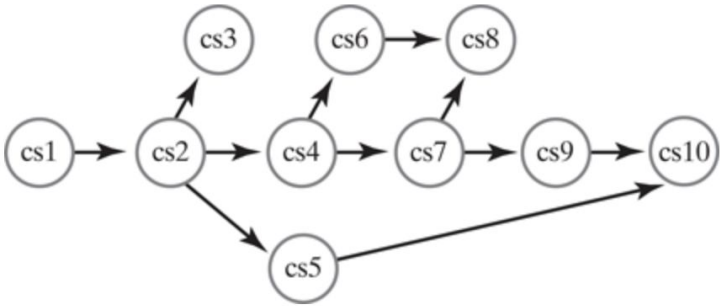
Topological Order

Vertices in a directed graph without cycles can be placed in topological order (in order of their precedence).

Such a graph may have more than one valid topological order.

Graphs with a cycle cannot have a topological order because this would result in circular logic.

Topological sort is the process for discovering the topological order for vertices in a graph. It uses a stack to hold vertices that have no successor or whose neighbors have been visited.



Implementation - Adjacency Matrix

Graphs can be implemented as a two-dimensional array of size V rows and V columns, where V is the number of vertices in the graph.

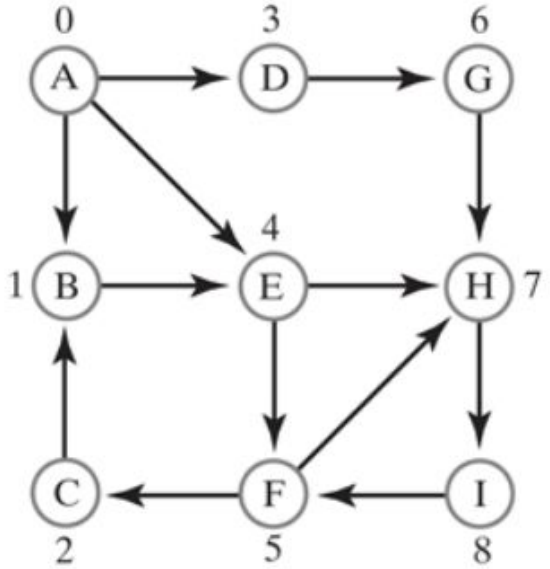
For an unweighted graph, the adjacency matrix would have boolean values for each edge. For a weighted graph, the matrix value would be the edge weight or infinity if no edge exists.

Determining if an edge exists for two vertices is a O(1) operation, but finding all the neighbors of a vertex is O(V).

The adjacency matrix requires fixed space for all possible edges, even though graphs are usually sparse. It can be a good choice for a dense graph.

Adjacency Matrix

(a) A graph



(b) The graph's adjacency matrix

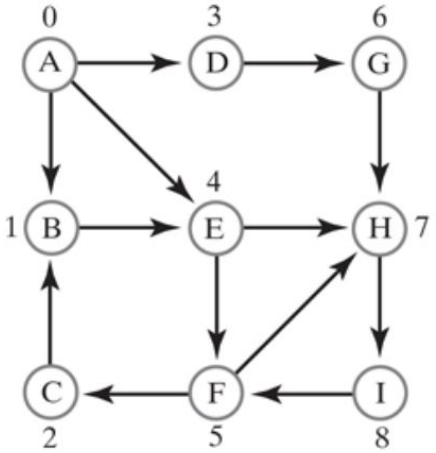
	A	B	C	D	E	F	G	H	I	
A		T		T	T					0
B					T					1
C		T								2
D							T			3
E						T		T		4
F			T					T		5
G								T		6
H									T	7
I						T				8
	0	1	2	3	4	5	6	7	8	

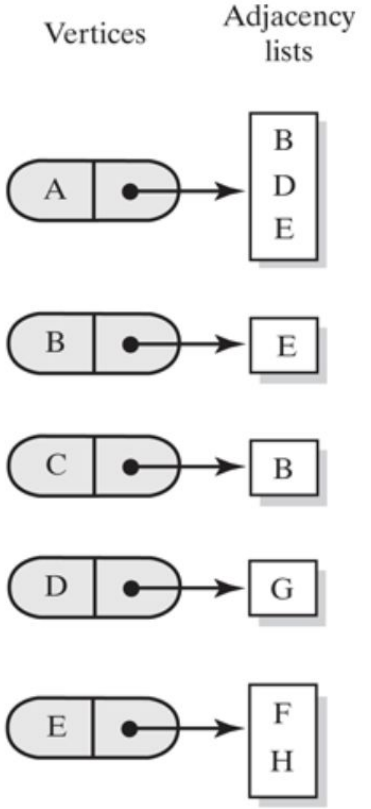
Implementation - Adjacency List

An adjacency list maintains a vertex-indexed array of lists. For each vertex, the array contains a list of its neighbors and represents the edges that originate from this vertex.

Space is reserved only for edges that exist, so this approach uses less memory.

Determining if an edge exists between two vertices, or finding all the neighbors of a vertex, is O(n) at worst but faster on average.





Efficiency

Implementation	Space	Add edge	Find edge btw v and w	Iterate vertices adjacent to v
Adjacency matrix	V^2	O(1)	O(1)	V
Adjacency list	E+V	O(1)	degree(v)	degree(v)

V = number of vertices
E = number of edges

Glossary

- **Graph** - a collection of vertices and edges
- **Vertex** - a node or entity in the graph
- **Edge** - a line that connects two vertices
- **Degree** - number of edges connecting a vertex to neighbors
- **Directed** edge - one-way relationship between two vertices
- **Undirected** edge - two-way relationship between vertices
- Directed graph (**digraph**) - a graph with directed edges
- **Weighted** edge - an edge with a weight or cost value
- **Path** - a sequence of edges between two vertices
- **Cycle** - a path that begins & ends at the same vertex

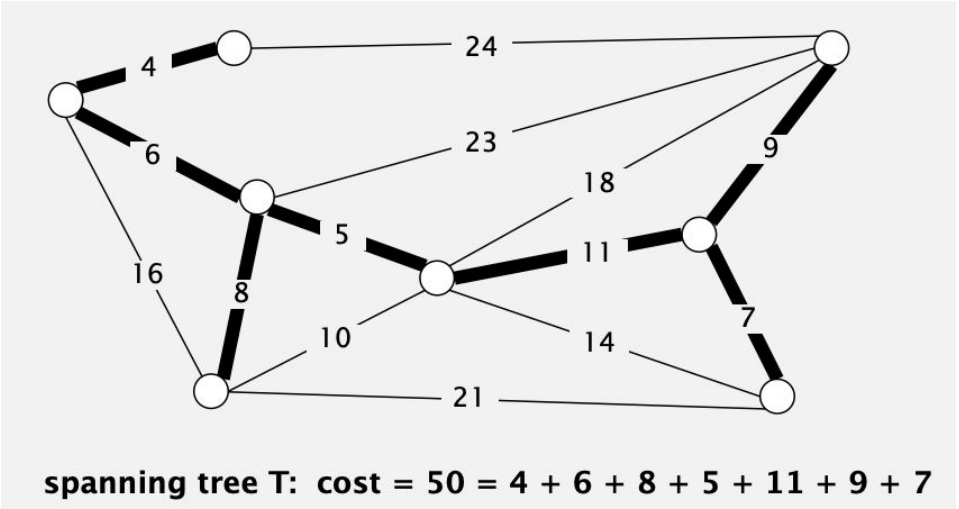
Glossary, cont.

- **Connected** - graph has a path between every vertex
- **Complete** - every vertex is connected to every other vertex
- **Disconnected** - some vertices can't be reached from all other vertices
- **Breadth-first search** - visit each of a vertex's neighbors before visting neighbors of neighbors
- **Depth-first search** - follow a single path to its end before following other paths
- **Topological order** - vertices of a digraph listed in order of precedence
- **Adjacency matrix** - two-dimensional array of graph edges
- **Adjacency list** - graph edges stored as a vertex-indexed array of lists

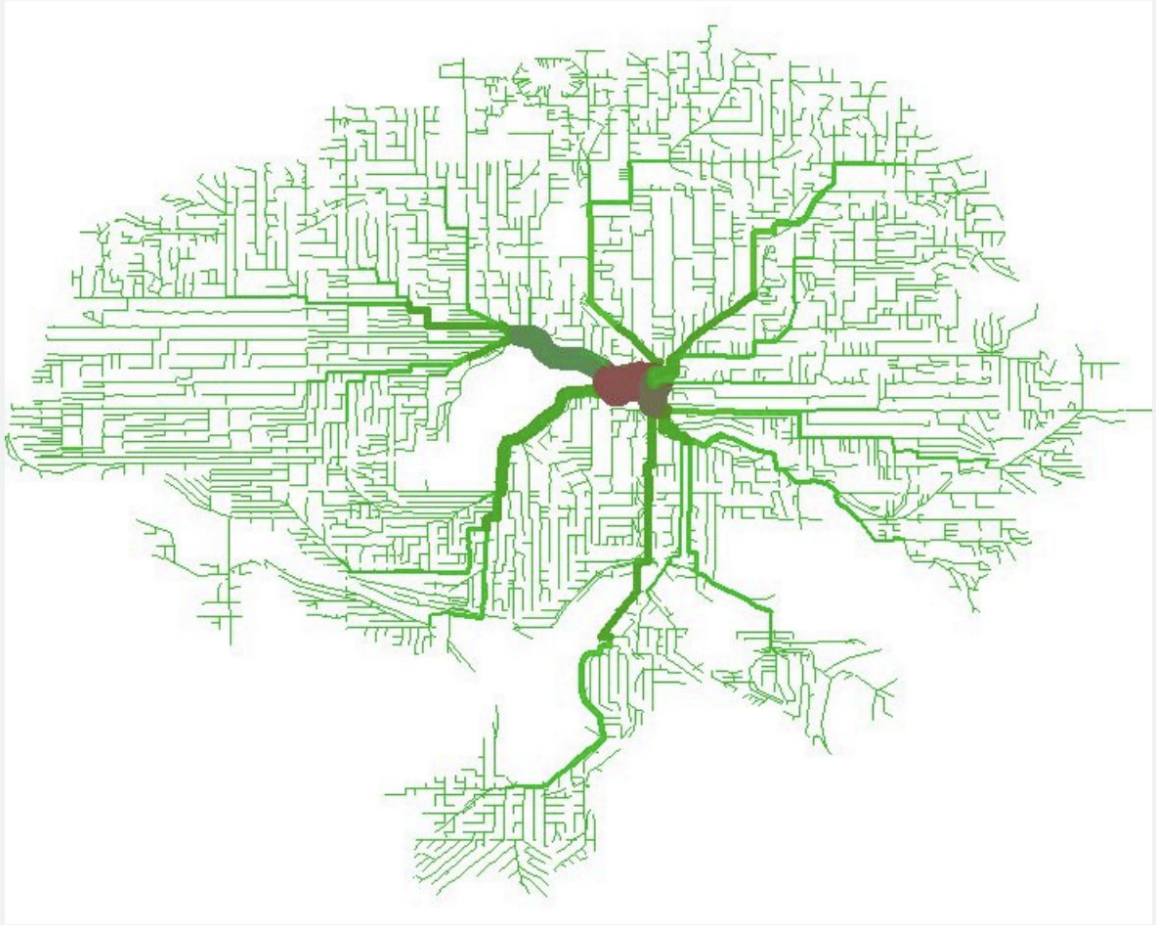
Overview

A **spanning tree** is a **connected**, **acyclical** subgraph that spans all vertices of a graph. A **Minimum Spanning Tree** (MST) is the spanning tree in a weighted-edge graph with the lowest sum of edge weights.

MST's are fundamental to a wide range of applications - e.g. cluster analysis, routing, detecting roads in satellite/aerial imagery.



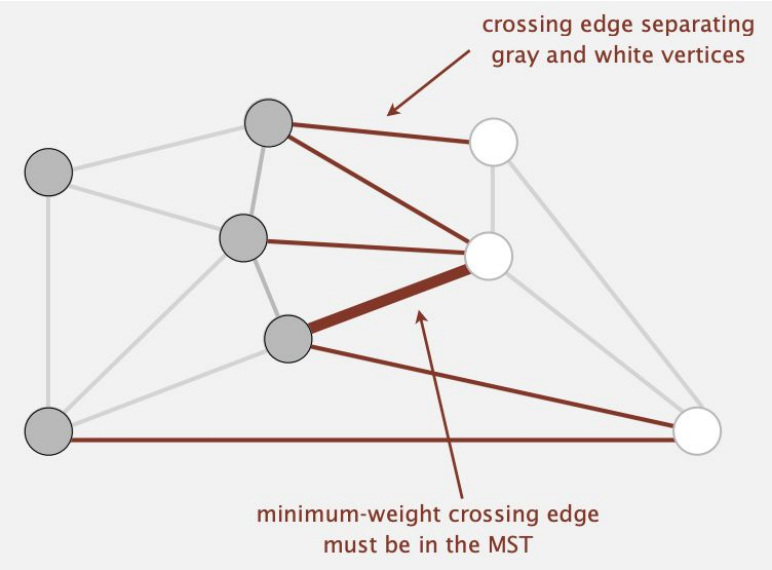
MST of bicycle routes in North Seattle



http://www.flickr.com/photos/ewedistrict/21980840

Finding a MST with a Greedy Algorithm

- **Cut** (partition) the graph vertices into two sets
- Find **crossing edges** that connect any vertex in one set with a vertex in the other set
- Find the **crossing edge** with minimum weight using a **greedy** algorithm
- Connect the target vertex to the origin set
- Repeat until all vertices are connected (e.g. when number of edges is V-1)

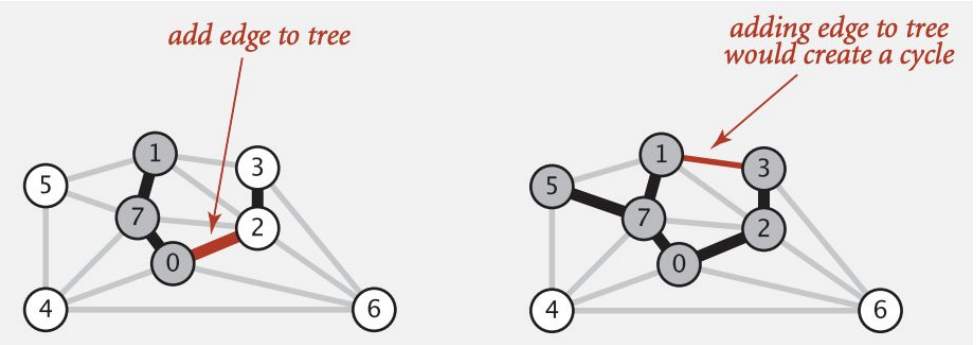


Simplifying Assumptions

- Edge weights are distinct
  - If not all distinct, the graph has multiple MST's
- Graph is connected
  - If not connected, greedy algorithm will compute a **Minimum Spanning Forest** (MSF) of multiple MST's

Kruskal's algorithm (1956)

- Sort graph **edges** in ascending order of weight with a Min Priority Queue
- Maintain a **set** for each **connected component**
- Select next edge with lowest weight and connect its vertices if doing so does not create a cycle (edges are not in same component)
- Merge sets for each component
- Can spawn multiple connected components that gradually merge
- Useful for identifying clusters



https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/KruskalMST.java.html

Prim's algorithm (1957)

- Start with vertex 0 and grow tree greedily
- Add the min weight edge with only one endpoint in the tree
- Add edges incident to the new endpoint to a min priority queue
- Keep track of visited and disregarded edges
- Repeat until tree has V-1 edges
- Array implementation is optimal for dense graphs
- Binary heap is much faster for sparse graphs

https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/PrimMST.java.html

Prim's algorithm (eager)

- Avoids space cost of lazy implementation
- Maintain a priority queue of with one entry for each vertex not on tree
- Priority is weight of shortest edge connecting vertex to tree
- Update edge & priority of a vertex if shorter connection to tree is found (using indexed priority queue)

Running Time

- Kruskal's algorithm computes MST in time proportional to E log E (worst case)
- Prim's algorithm computes MST in time proportional to E log E & extra space proportional to E (worst case)
- Prim's algorithm (eager) computes MST in time proportional to E log V with space proportional to V

https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/PrimMST.java.html

Shortest Path Variants

Which vertices

- **Single source**: from one vertex s to every other vertex
- **Source sink** - from one vertex s to another t
- **All pairs**: between all pairs of vertices

Edge weight restrictions?

- Unweighted
- Non-negative weights
- Euclidean (geometric distance) weights
- Arbitrary (incl. negative) weights

Cycles?

- No directed cycles
- No “negative cycles”

Shortest Paths - Unweighted Graph

In an unweighted graph, the shortest path between two vertices is the path with the fewest edges.

The path is found using a breadth-first search (BFS). Starting at the origin vertex, neighbors are placed on a queue and then the neighbors of each neighbor and so forth.

Each vertex is marked as visited and also with data about its predecessor and the path length traversed to reach it.

Once the target vertex is reached, the full path is derived by adding each predecessor vertex to a stack and returning the stack.

Shortest Paths - Weighted Graph

For a weighted graph, the shortest path between two vertices has the smallest sum of edge weights.

In general, the shortest path is found using a BFS with neighboring edges placed on a min priority queue.

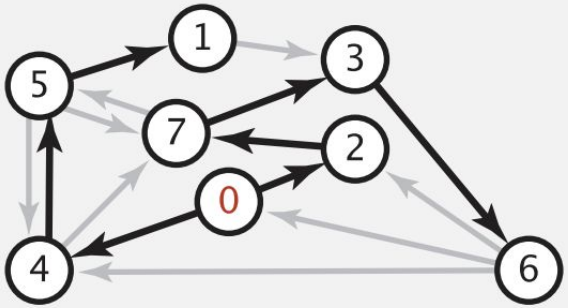
Vertices are visited according to different algorithms depending on edge weights and whether the graph has cycles:

- Dijkstra’s algorithm - no negative weights
- Bellman-Ford algorithm - no negative cycles
- Topological sorting - no directed cycles

Shortest Path Tree (SPT) data structure

Data for **shortest path tree -** path from vertex s to every other vertex - can be represented with two vertex-indexed arrays.

- distTo[v] is length of shortest path from s to v
- edgeTo[v] is last edge on shortest path from s to v



shortest-paths tree from 0

	edgeTo[]	distTo[]
0	null	0
1	5->1 0.32	1.05
2	0->2 0.26	0.26
3	7->3 0.37	0.97
4	0->4 0.38	0.38
5	4->5 0.35	0.73
6	3->6 0.52	1.49
7	2->7 0.34	0.60

parent-link representation

Single-source Shortest Path, cont.

- Distance to source vertex set to 0
- Distance from source vertex is initialized as **infinity** for all other vertices
- As each vertex is visited, it's distance from the source vertex is **relaxed** (updated based on path used to reach it, if new distance would be lower)
- Each **relaxation** decreases distance to source vertex for some v

Dijkstra's Algorithm

Computes a SPT in any edge-weighted digraph with nonnegative weights

- Consider vertices in increasing order of distance from source (s)
- Add vertex to the shortest-path tree and relax all edges pointing from that vertex
- Each edge is relaxed exactly once
- Algorithm stops when no relaxation happens (distance from source is not decreased for any vertex)

Essentially same as **Prim's Algorithm**. Differs by choosing as next vertex the one closest to the source (via a directed path)

Dijkstra's Algorithm - performance

Performance depends on choice of priority queue implementation

Implementation	insert	Delete min	Decrease-key	Total
Unordered array	1	V	1	V^2
Binary heap	log V	log V	log V	E log V
d-way heap	$\log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap	1	log V	1	E + V log V

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations
- Fibonacci heap best in theory, but not worth implementing

Bellman-Ford Algorithm - Negative Cycles

A **negative cycle** is a directed cycle whose sum of edge weights is negative

computes SPT in any edge-weighted digraph with no negative cycles in time proportional to E × V in worst case

Can be optimized to not relax vertices whose value didn’t change in the previous iteration

Topological Sort

A Topological sort algorithm computes the SPT in any edge-weighted **directed acyclical graph (DAG)** in time proportional to E + V. Works even with negative weights

- Consider vertices in topological order
- Start from source vertex
- Relax all edges pointing from that vertex

Useful for finding longest path (e.g. **parallel job scheduling** or **critical path**):

- Negate all weights
- Find shortest paths
- Negate weights in result

Cost Summary

Algorithm	Restriction	Typical case	Worst case	Extra space
Topological sort	No directed cycles	E + V	E + V	V
Dijkstra's Algorithm (binary heap)	No negative weights	E log V	E log V	V
Bellman-Ford	No negative cycles	E V	E V	V
Bellman-Ford (queue-based)		E + V	E V	V