Week 9 - Spanning Trees & Shortest Paths

AD 325 - Brenden West

Contents

Learning Outcomes

- Spanning trees
- Finding minimum spanning trees (MST)
- Finding shortest path in a graph
- Data structures for shortest paths

Reading & Videos

- Carrano & Henry: Chapter 29 (Paths)
- https://www.coursera.org/learn/algorithms-part2/home/week/2

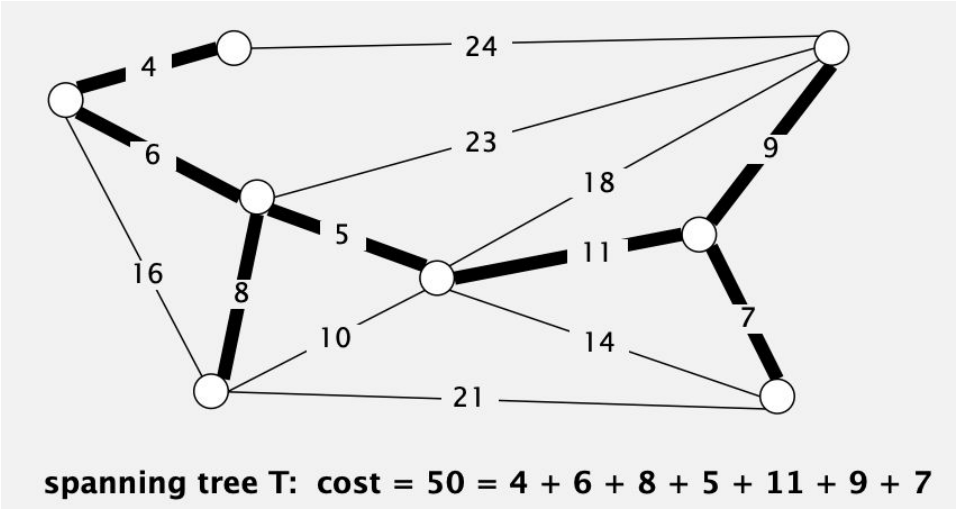
Reference

- https://algs4.cs.princeton.edu/43mst/
- https://algs4.cs.princeton.edu/44sp/
- https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#minimumSpanningTree
- https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#shortestPath

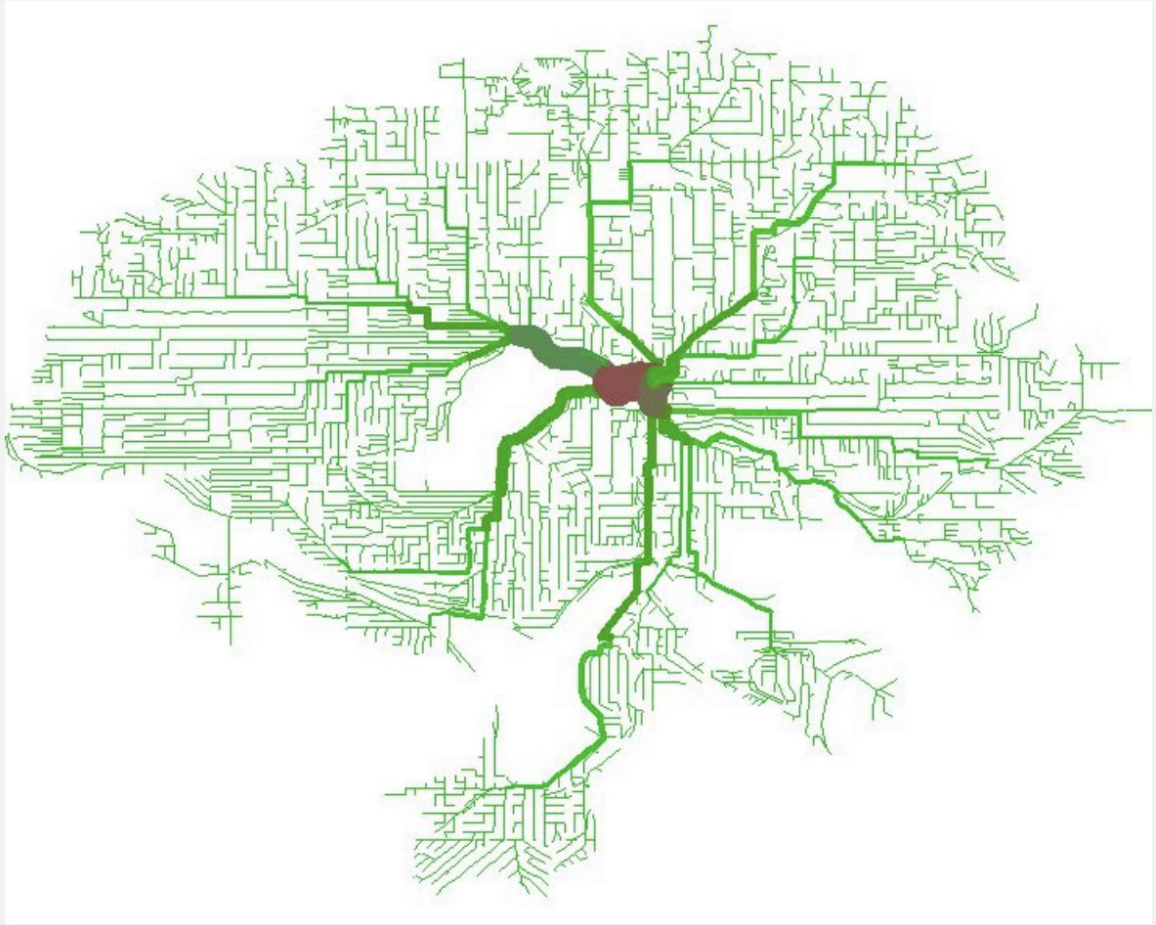
Overview

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.



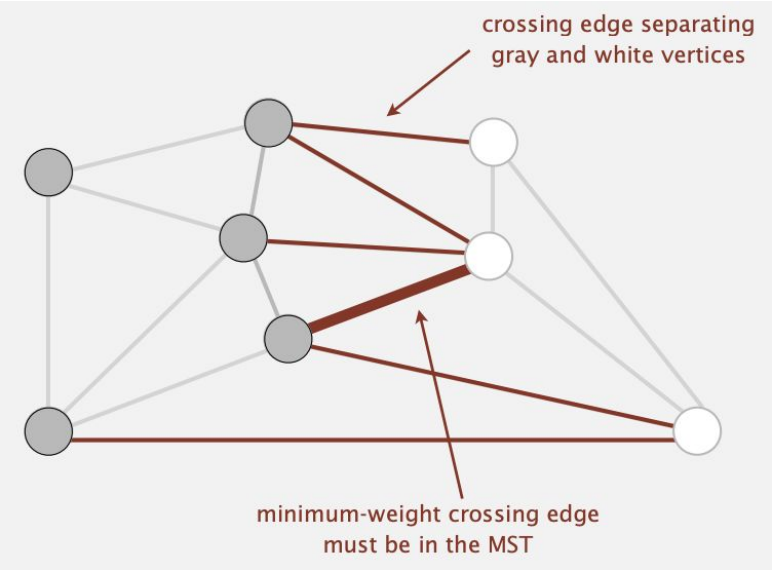
MST of bicycle routes in North Seattle



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)

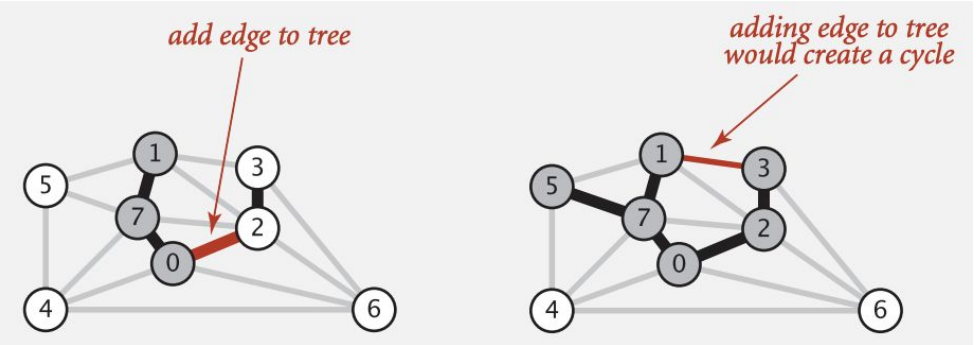


Simplifying Assumptions

- 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.

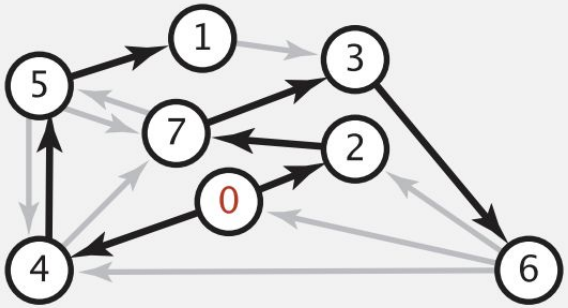
Vertices are visited according to different algorithms depending on edge weights and whether the graph has cycles:

- 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