DSC 40B

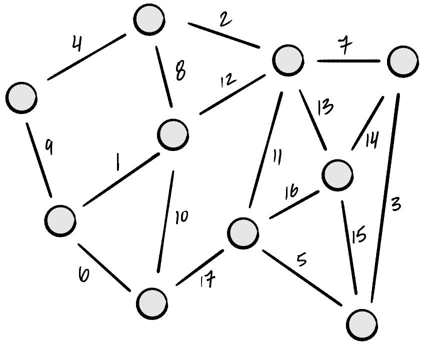
Theoretical Foundations II

Kruskal's Algorithm

Last Time: Minimum Spanning Tree

- The **minimum spanning tree** problem is as follows:
  - GIVEN: A weighted, undirected graph $G = (V, E, \omega)$.
  - COMPUTE: a spanning tree of G with minimum cost (i.e., minimum total edge weight).

Example



Last Time: Building MSTs

- How do we build a MST efficiently?

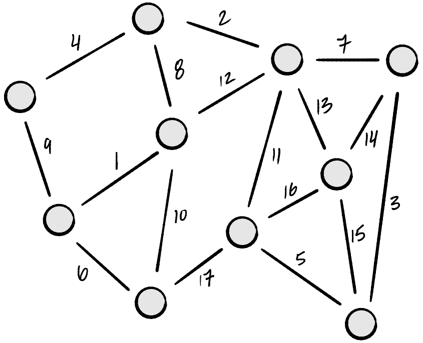
- We'll adopt a **greedy** approach.
  - Build a tree edge-by-edge.
  - At every step, doing what looks best at the moment.

- This strategy isn't guaranteed to work in all of life's situations, but it works for building MSTs.

Two Greedy Approaches

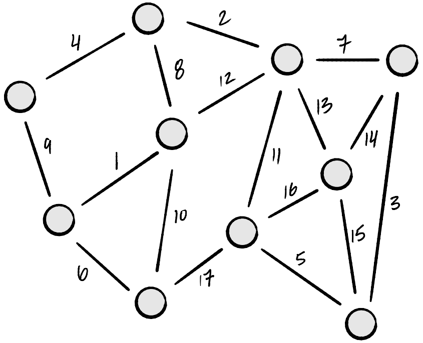
- We'll look at two greedy algorithms:
    - Last Time: Prim's Algorithm
    - Today: Kruskal's Algorithm
- Differ in the order in which edges are added to tree.
- Also differ in time complexity.

Prim's Algorithm, Informally



- Start by picking any node to add to "tree", *T*.
- While *T* is not a spanning tree, greedily add **lightest** edge from a node in *T* to a node not in *T*.
    - "lightest" = edge of smallest weight

Kruskal's Algorithm, Informally



- Start with empty forest: $T = (V, E_{\text{mst}})$, where $E_{\text{mst}} = \emptyset$.
- Loop through edges in increasing order of weight.
    - If edge does not create a cycle in T, add it to T.
    - If T is a spanning tree, break.

Being Greedy

- Prim: add the **node** with smallest estimated cost and update neighbors.
    - Works locally, “grows” a connected tree.

- Kruskal: add the **edge** with smallest weight.
    - As long as it doesn’t make a cycle.
    - Edge can be anywhere in graph.

Kruskal’s Algorithm (Pseudocode)

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        # if u and v are not already connected
        if ...:
            mst.add_edge(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

Checking for Connectivity

- Each iteration: check if u and v are connected in $T = (V, E_{\text{mst}})$.

- We can do a DFS/BFS on each iteration.
  - $\Theta(V + E_{\text{mst}}) = \Theta(V)$ each time.
  - **Expensive**!

- Remember:
  - If you're computing something once, use a fast algorithm.
  - If you're computing it repeatedly, consider a **data structure**.

Disjoint Set Forests

- Represent a collection of disjoint sets.

$$\{\{1,5,6\},\{2,3\},\{0\},\{4\}\}$$

- `.union(x, y)`: Union the sets containing x and y.

- `.in_same_set(x, y)`: Return `True`/`False` if x and y are in the same set.[1]

[1] Usually implemented as a `.find(x)` method returning representative of set containing `x`.

Example

```
>>> # create a DSF with {{0}, {1}, {2}, {3}, {4}, {5}}
>>> dsf = DisjointSetForest([0, 1, 2, 3, 4, 5])
>>> dsf.union(0, 3)
>>> dsf.union(1, 4)
>>> dsf.union(3, 1)
>>> dsf.union(2, 5)
>>> # dsf now represents {{0, 1, 3, 4}, {2, 5}}
>>> dsf.in_same_set(0, 3)
True
>>> dsf.in_same_set(0, 2)
False
```

Disjoint Set Forests

- Operations take $\Theta(\alpha(n))$ time, where n is number of objects in collection.
- $\alpha(n)$ is the **inverse Ackermann function**.
- It grows very, **very** slowly.
- Essentially constant time.

Disjoint Set Forests

- Can be used to keep track of CCs of a **dynamic graph**.

- Nodes of CCs are disjoint sets.
  - Add an edge (u, v): `.union(u, v)`
  - Check if u and v are connected: `.in_same_set(u, v)`

- To check if u, v are already connected:
  - BFS/DFS: $\Theta(V)$ each time.
  - DSF: $\Theta(\alpha(V))$ each time (essentially $\Theta(1)$).

Kruskal’s Algorithm

```python
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

Time Complexity

```python
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

Time Complexity

- Assume graph is connected. Then $E = \Omega(V)$.

- Kruskal's takes $\Theta(E \log E) = \Theta(E \log V)$ time.
  - Dominated by sorting the edges.

- Note: if graph disconnected, Kruskal's produces a **minimum spanning forest**.

DSC 40B

Theoretical Foundations II

Kruskal v. Prim

Kruskal v. Prim

- Both algorithms for computing MSTs.
- Which is “better”?
- There’s no clear winner.

Time Complexity

- Prim:
    - Binary heap: $\Theta(V \log V + E \log V)$
    - Fibonacci heap: $\Theta(V \log V + E)$
- Kruskal: $\Theta(E \log V)$
- If the graph is dense, $E = \Theta(V^2)$, and Prim's with Fibonacci heap "wins".
    - $\Theta(V^2)$ versus $\Theta(V^2 \log V)$.

Not so fast...

- Fibonacci heaps are hard to implement, high overhead.
- Prim’s will be faster for very large dense graphs.
- But Kruskal’s may be faster for smaller dense graphs.
- The right choice depends on your application.

Main Idea

Asymptotic time complexity isn’t everything. For small inputs, the “inefficient” algorithm may beat the “efficient” one. There’s also ease of implementation to consider.

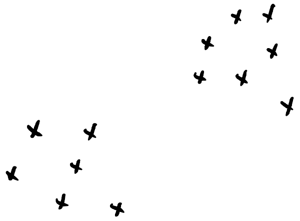
DSC 40B

Theoretical Foundations II

MSTs and Clustering

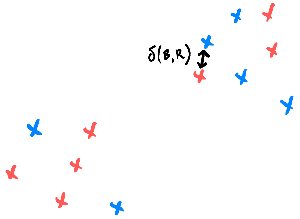
Clustering

Goal: identify the groups in data. Example:

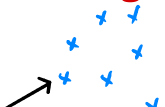


Clustering, Formalized

- We frame as an optimization problem.
  - GIVEN: n data points.
  - GOAL: assign color to each point (red or blue) to maximize the distance between the closest pair of red and blue points.



Bad Clustering

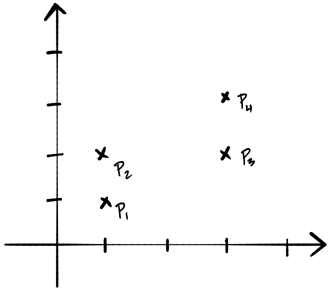


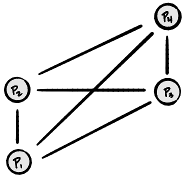
Brute Force Solution

- Try all possible assignments; return best.
- If there are n data points, there are $\Theta(2^n)$ assignments.
- Exponential time; very slow. Practical only for ~ 50 data points.
- Instead, we will turn it into a graph problem.

Distance Graphs

- Given n data points, $p_1, p_2, \dots, p_n$, create complete graph with $V = \{p_1, \dots, p_n\}$.
- Set weight of edge $(p_i, p_j) = \text{dist}(p_i, p_j)$.
- The result is a weighted, undirected **distance graph**.



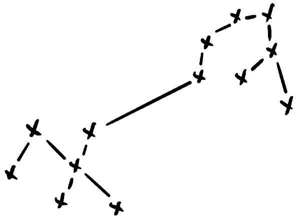


Main Idea

We can always think of a set of points in a (metric) space as a weighted distance graph. This is a **very** important idea, because it allows us to use our graph algorithms!

Clustering with MSTs

- Given n data points and a number of clusters, k:
  - Create distance graph G.
  - Run Kruskal's Algorithm on G until there are only k components.



  - The resulting connected components are the **clusters**.
  - This is known as **single-linkage clustering**.

Single-Linkage Clustering

- Time complexity of single-linkage is determined by Kruskal's Algorithm: $\Theta(E \log E)$.

- Since distance graph is complete, $E = \Theta(V^2)$, and so

$$\Theta(E \log E) = \Theta(V^2 \log V) = \Theta(n^2 \log n)$$

- Practically, can cluster $\sim 10,000$ points.

Summary

- We started the quarter with a brute force solution.
    - Took $\Theta(2^n)$ time, only feasible for a few dozen points.

- We've now reframed the problem using graph theory.
    - Now only $\Theta(n^2 \log n)$ time!
    - Feasible for tens of thousands of points.

Why Algorithms?

- Data scientists use computers as tools.
- But solving a problem isn’t just about coding it up.
- You need to know how to analyze your code and use the right algorithms and data structures to make your solution efficient.