Graph Algorithms

Discrete Mathematics and Optimization
Bioinformatics

Outline

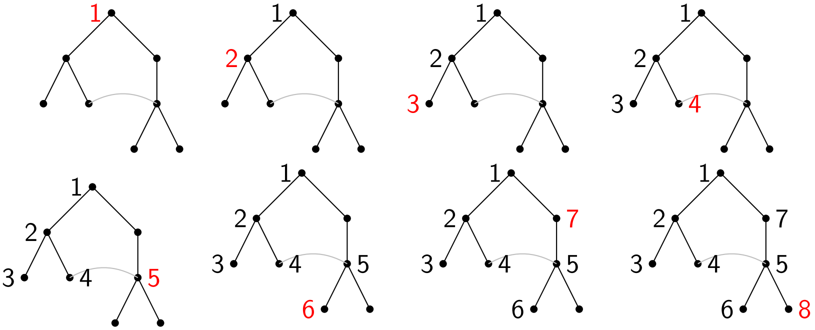
- Depth First Search
- Breadth First Search
- Application: Check bipartiteness
- Minimum Spanning Tree: Kruskal algorithm
- Minimum Spanning Tree. Prim algorithm
- Application: Clustering

Depth First Search

Exploring graphs

DFS Algorithm

- Input: A graph G (by adjacency lists) and a root vertex r
- Output: An ordering of the vertices in the connected component of G containing r .
- $DFS(Ex, r)$: Explore first indepth before backtracking

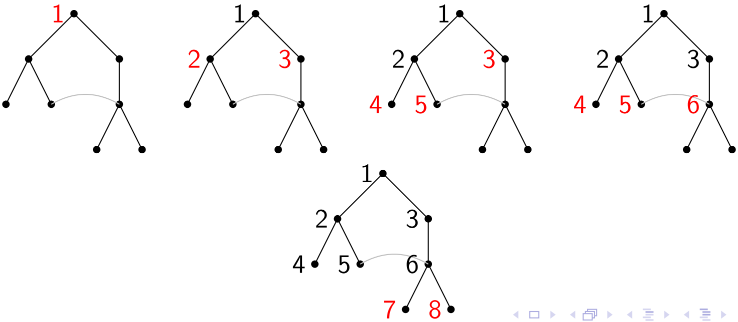


Breadth First Search

Exploring graphs

BFS Algorithm

- Input: A graph G (by adjacency lists) and a root vertex r
- Output: An ordering of the vertices in the connected component of G containing r .
  - Start at initial vertex (root) r.
  - Visit all vertices adjacent to the root r (put them on a queue).
  - Repeatedly visit the neighbours of visited vertices.
  - Stop when there are no new vertices to visit.



Breadth First Search

Exploring graphs

BFS Algorithm

- Input: A graph G (by adjacency lists) and a root vertex r
- Output: An ordering of the vertices in the connected component of G containing r .
  - Start at initial vertex (root) r.
  - Visit all vertices adjacent to the root r (put them on a queue).
  - Repeatedly visit the neighbours of visited vertices.
  - Stop when there are no new vertices to visit.

Set Explored[r] = true and Explored[v] = false for $v \neq r$
$L[0] = s$ (vertices at distance 0 from r)
$i = 0$ (layer counter)
Set $T = \emptyset$ (initial tree)
While $L[i]$ is not empty
    Initialize an empty list $L[i+1]$
    For each vertex u in $L[i]$ Consider each edge (u, v)
        If $Explored[v] = false$ then
            Set $Explored[v] = true$ Add (u, v) to T Add v to $L[i+1]$
        Endif
    Endfor

Breadth First Search

Exploring graphs

BFS Algorithm

- Input: A graph G (by adjacency lists) and a root vertex r
- Output: An ordering of the vertices in the connected component of G containing r .
  - Start at initial vertex (root) r.
  - Visit all vertices adjacent to the root r (put them on a queue).
  - Repeatedly visit the neighbours of visited vertices.
  - Stop when there are no new vertices to visit.

- Time complexity $O(|V| + |E|)$.
- It provides a Layer decomposition $V = L_0 \cup L_1 \cup \cdots \cup L_t$ of the connected component containing r.
- It can provide the (number of) connected components of G: restart the algorithm with a new root if not all vertices have been explored.
- Can be applied to find the shortest path distance from the root to every other node in the graph.

Breadth First Search

Check if a graph is bipartite

A graph is bipartite if there is a bipartition $V = A \cup B$ such that every edge has precisely one end point in A and one endpoint in B.

- Input: A connected graph G
- Output: Decide if G is bipartite (decision algorithm)
  - Choose a vertex r.
  - Run BFS algorithm from r.
  - Build $A = L_0 \cup L_2 \cup \cdots$ and $B = L_1 \cup L_3 \cup \cdots$.
  - Check if there are edges in A or in B. If not, output G is bipartite.

Theorem

G is a bipartite graph if and only if it contains no odd cycles.

Breadth First Search

Check if a graph is bipartite

A graph is bipartite if there is a bipartition $V = A \cup B$ such that every edge has precisely one end point in A and one endpoint in B.

Theorem

G is a bipartite graph if and only if it contains no odd cycles.

- If G is bipartite then all of its subgraphs are bipartite
- An odd cycle is nonbipartite: a bipartite graph can not contain odd cycles
- If G does not contain odd cycles then $A = L_0 \cup L_2 \cup \cdots$ and $B = L_1 \cup L_3 \cup \cdots$ form a bipartition of G.

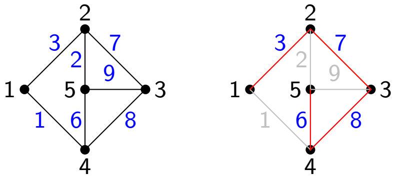
Minimum Spanning Tree

Weighted Graph $G = (V, E)$

- V set of nodes or vertices
- E set of edges (pairs of nodes) (can be directed, multiple, loops,...)
- $w : E \to \mathbb{R}^+$: each edge has a weight.

The weight of a subgraph $H \subset G$ is

$$w(H) = \sum_{e \in H} w(e).$$



$w(H) = 24$

MST Problem

Given an edge–weighted graph $G = (V, E, w)$ find a spanning tree of G with minimum weight.

Kruskal Algorithm

- Input: An edge–weighted graph G.
- Ouput: A forest of minimum weight (in terms of its edges)
  - Sort the edges in non–decreasing weights $\{e_1, \ldots, e_m\}$
  - Initialize $F_0 = \{\emptyset\}$.
  - **f**or i from 1 to n **d**o

    if $F_{i-1} \cup e_i$ is acyclic then $F_i = F_{i-1} \cup e_i$ **else** $F_i = F_{i-1}$.
  - Return F_m.

Kruskal Algorithm

- Input: An edge–weighted graph G.
- Ouput: A forest of minimum weight (in terms of its edges)
  - Sort the edges in non–decreasing weights $\{e_1, \ldots, e_m\}$
  - Initialize $F_0 = \{\emptyset\}$.
  - **f**or i from 1 to n **d**o

    if $F_{i-1} \cup e_i$ is acyclic then $F_i = F_{i-1} \cup e_i$ **e**lse $F_i = F_{i-1}$.
  - Return F_m.

- Correctedness of the algorithm
- Complexity of the algorithm
- Implementation

Kruskal Algorithm

- Input: An edge–weighted graph G.
- Ouput: A forest of minimum weight (in terms of its edges)
  - Sort the edges in non–decreasing weights $\{e_1, \ldots, e_m\}$
  - Initialize $F_0 = \{\emptyset\}$.
  - **f**or i from 1 to n **d**o

    if $F_{i-1} \cup e_i$ is acyclic then $F_i = F_{i-1} \cup e_i$ **e**lse $F_i = F_{i-1}$.
  - Return F_m.

Correctedness:

- The output is a forest:
  - acyclic by construction
  - The output is a minimum weight spanning forest: $E(F_i)$ contains the edges of an MST T for each i.

Kruskal Algorithm

- Input: An edge–weighted graph G.
- Ouput: A forest of minimum weight (in terms of its edges)
  - Sort the edges in non–decreasing weights $\{e_1, \ldots, e_m\}$
  - Initialize $F_0 = \{\emptyset\}$.
  - **f**or i from 1 to n **d**o

    if $F_{i-1} \cup e_i$ is acyclic then $F_i = F_{i-1} \cup e_i$ **e**lse $F_i = F_{i-1}$.
  - Return F_m.

Kruskal Algorithm

- Input: An edge–weighted graph G.
- Ouput: A forest of minimum weight (in terms of its edges)
  - Sort the edges in non–decreasing weights $\{e_1, \dots, e_m\}$
  - Initialize $F_0 = \{\emptyset\}$.
  - **f**or i from 1 to n **d**o

    if $F_{i-1} \cup e_i$ is acyclic then $F_i = F_{i-1} \cup e_i$ **else** $F_i = F_{i-1}$.
  - Return F_m.

Complexity:

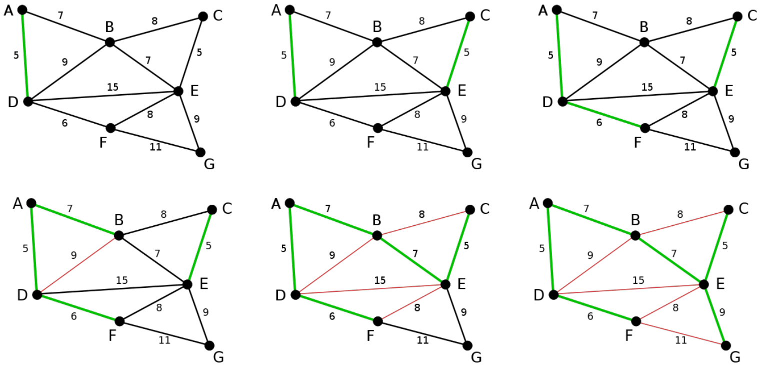
- $O(m \log m)$ to sort the edges.
- Check acyclicity at each step: keep track of the connected components (n at the begining). Each added edge has one only vertex in an existing connected component: check in constant time: $O(m)$.

Complexity is $O(m \log m) = O(m \log n)$.

Kruskal Algorithm

- Input: An edge–weighted graph G.
- Ouput: A forest of minimum weight (in terms of its edges)
  - Sort the edges in non–decreasing weights $\{e_1, \ldots, e_m\}$
  - Initialize $F_0 = \{\emptyset\}$.
  - **for** i from 1 to n **do**

    if $F_{i-1} \cup e_i$ is acyclic then $F_i = F_{i-1} \cup e_i$ **else** $F_i = F_{i-1}$.
  - Return F_m.



Kruskal Algorithm

- Input: An edge–weighted graph G.
- Ouput: A forest of minimum weight (in terms of its edges)
  - Sort the edges in non–decreasing weights $\{e_1, \ldots, e_m\}$
  - Initialize $F_0 = \{\emptyset\}$.
  - **f**or i from 1 to n **d**o

    if $F_{i-1} \cup e_i$ is acyclic then $F_i = F_{i-1} \cup e_i$ **e**lse $F_i = F_{i-1}$.
  - Return F_m.

Implementation

Disjoint Sets data structure

- makeset(x): creates a singleton set containing just x
- find(x): returns the identifier of the set containing x
- union(x,y): merges the sets containing x and y.

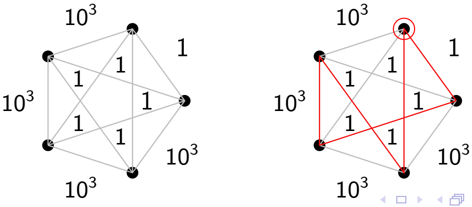
See kruskal.py

Kruskal Algorithm

- Input: An edge–weighted graph G.
- Ouput: A forest of minimum weight (in terms of its edges)
  - Sort the edges in non–decreasing weights $\{e_1, \ldots, e_m\}$
  - Initialize $F_0 = \{\emptyset\}$.
  - **for** i from 1 to n **do**
    if $F_{i-1} \cup e_i$ is acyclic then $F_i = F_{i-1} \cup e_i$ **else** $F_i = F_{i-1}$.
  - Return F_m.

Kruskal MST algorithm is an example of a Greedy algorithm: at each step optimize locally your next move.

Greedy algorithms not always give the desired result.
Hamiltonian Path with minimum weight (TSP)



Kruskal Algorithm

- Input: An edge–weighted graph G.
- Ouput: A forest of minimum weight (in terms of its edges)
  - Sort the edges in non–decreasing weights $\{e_1, \ldots, e_m\}$
  - Initialize $F_0 = \{\emptyset\}$.
  - **for** i from 1 to n **do**

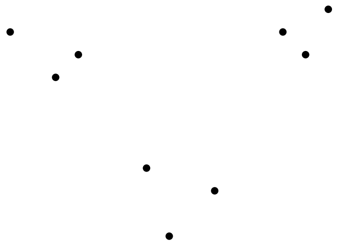
    if $F_{i-1} \cup e_i$ is acyclic then $F_i = F_{i-1} \cup e_i$ **else** $F_i = F_{i-1}$.
  - Return F_m.

Joseph Kruskal (1928-2010)

American mathematician, statistician and computer scientist
Worked in Bell Labs (1959–1995)

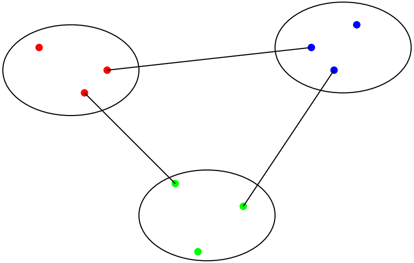
Application of Kruskal algorithm to Clustering

- Input: A set of n items and the distance (u, v) between each pair.
- Output: Divide the items into k groups so that the minimum distance between the groups is maximized



Application of Kruskal algorithm to Clustering

- Input: A set of n items and the distance (u, v) between each pair.
- Output: Divide the items into k groups so that the minimum distance between the groups is maximized



Application of Kruskal algorithm to Clustering

- Input: A set of n items and the distance (u, v) between each pair.
- Output: Divide the items into k groups so that the minimum distance between the groups is maximized

- Build the weighted graph with distances as weights
- Run Kruskal algorithm keeping track of the number of connected components at each step.
- Stop when the number of connected components is k
- Return the connected components

Prim Algorithm

- Input: An edge–weighted connected graph G.
- Ouput: A tree of minimum weight
  - Start at a vertex x_0 (root)
  - Initialize $F = (\{x_0\}, \emptyset)$.
  - While $F \neq V(G)$
    - ⋆ Find the edge $e = xy$ with minimum weight with $x \in F$ and $y \notin F$
    - ⋆ $F = (V(F) \cup \{y\}, E(F) \cup \{xy\})$
  - Return F.

Prim Algorithm

- Input: An edge–weighted connected graph G.
- Ouput: A tree of minimum weight
  - Start at a vertex x_0 (root)
  - Initialize $F = (\{x_0\}, \emptyset)$.
  - While $F \neq V(G)$
    - Find the edge $e = xy$ with minimum weight with $x \in F$ and $y \notin F$
    - $F = (V(F) \cup \{y\}, E(F) \cup \{xy\})$
  - Return F.

- Correctedness of the algorithm: similar to Kruskal
- Complexity of the algorithm: $O(n \log n)$
- Implementation: somewhat simpler

Prim Algorithm

- Input: An edge–weighted connected graph G.
- Ouput: A tree of minimum weight
  - Start at a vertex x_0 (root)
  - Initialize $F = (\{x_0\}, \emptyset)$.
  - While $F \neq V(G)$
    - ⋆ Find the edge $e = xy$ with minimum weight with $x \in F$ and $y \notin F$
    - ⋆ $F = (V(F) \cup \{y\}, E(F) \cup \{xy\})$
  - Return F.