Graph Theory

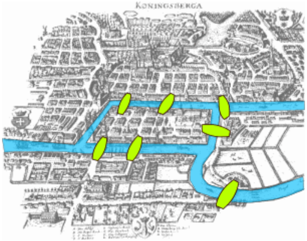
Dov Kruger

Department of Electrical and Computer Engineering
Stevens Institute of Technology

April 19, 2023

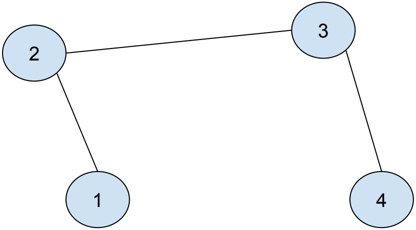
Graph Theory

- Invented by Leonard Euler 1735
- A set V of vertices
- E (edges connecting pairs of vertices)
- First Problem: Seven Bridges of Konigsberg


KONINGSBERGA

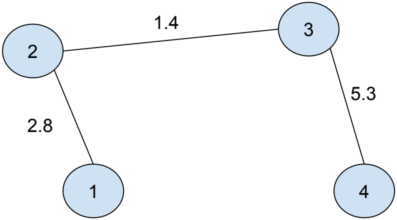
A Graph has a set V of vertices, and E edges connecting pairs of vertices

Edges are by default not directional (travel can go in both directions)

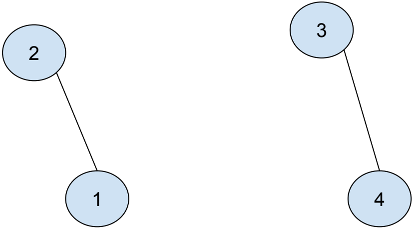


Weights

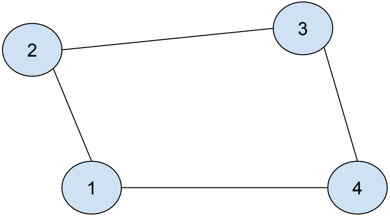
Edges may also have weights representing the cost to traverse an edge



A graph is said to be connected iff from each vertex it is possible to reach all other vertices

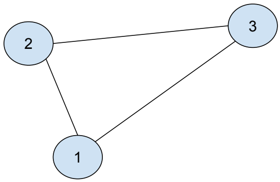


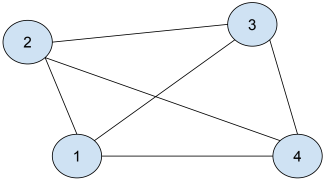
A graph is biconnected if removing any single edge leaves it still connected



A complete graph has all edges connecting every vertex to every other

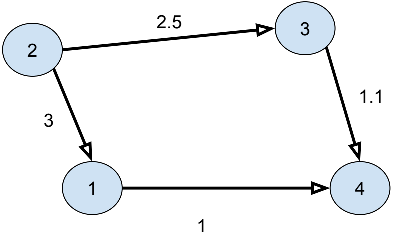
What is the number of edges to make a complete graph as a function of V?



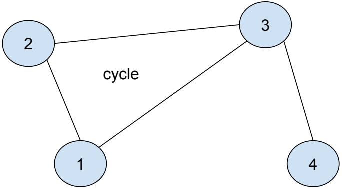


Directed Graph (DiGraph)

A Directed Graph has edges that can only be traversed in one direction



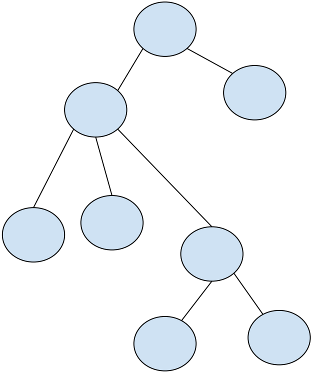
A cycle in a graph is a loop



A Tree is a graph

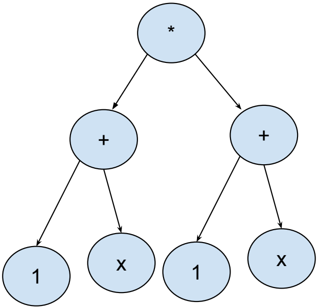
- With a designated root
- Once branches diverge they never rejoin
- In other words, a tree has no cycles

Notice that any node could be the root

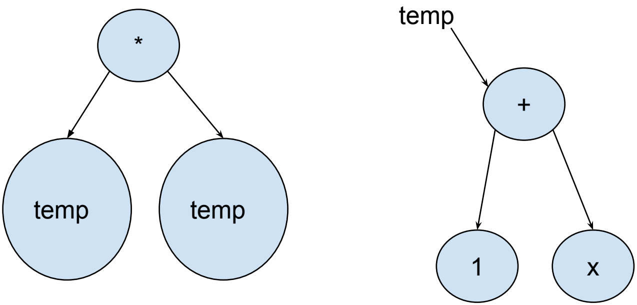


Directed Acyclic Graph (DAG)

A DAG is like a tree except the branches can rejoin. There are no cycles. Common in expressions where common terms repeat
This diagram shows x used twice

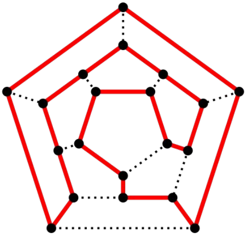


DAG Referring to Common Subexpression

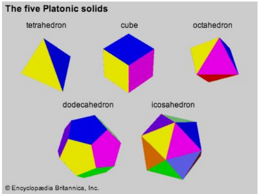


A Hamiltonian path

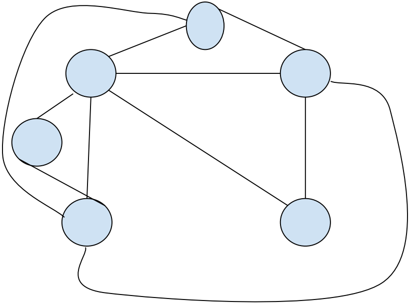
- Begins at one vertex
- Visits every vertex
- Returns to the original vertex
- All without visiting any edge twice



Hamilton proved that the vertices of platonic solids can be traversed with a Hamiltonian path

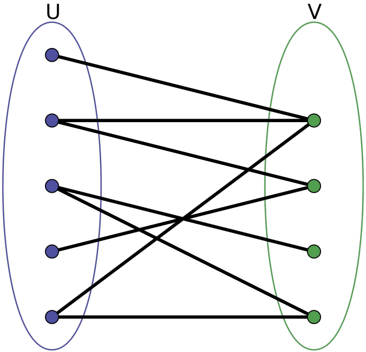


A planar graph can be drawn on a 2D surface (like paper) without any lines crossing



Bipartite Graph

A bipartite graph can be broken down into two regions, each with a corresponding value in the other



Graph Representations

There are three good graph representations, and one bad one

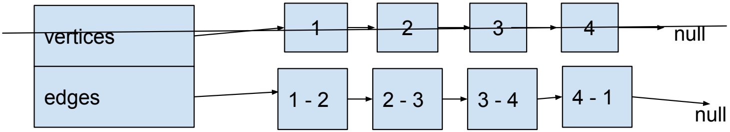
The bad one will be shown first so you know what not to use

- Global List of Edge (bad)
- Per-vertex Edge List
- Matrix
- Compressed Sparse Row (CSR)

Global Edge List

Having a single list of all the edges is bad because $E = O(V^2)$

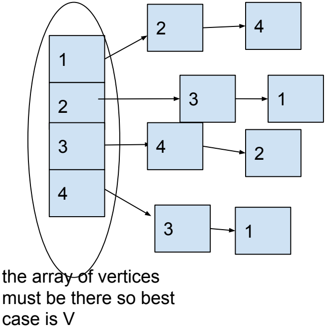
- Too much data to look through (slow)
- Other representations will have the number of edges from any vertex $= O(V)$
- By splitting edges into categories (which vertex they come from) speed can be drastically improved
- Notice, even this bad implementation does not need a list of vertices



Per-Vertex Edge List

For each vertex, there is a list of edges

- The max edges per vertex is $V - 1$ therefore $O(V)$
- The min edges per vertex is 0 therefore (1)
- Pro: Finding if vertex i adjacent to j is (1)
- Con: but it is also $O(V)$

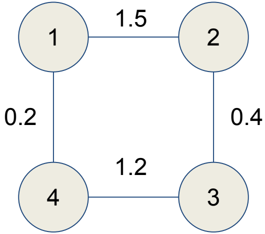


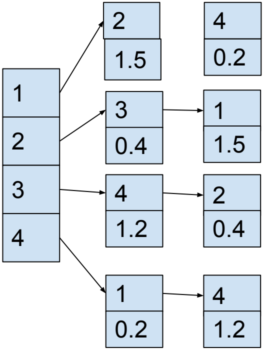
The Matrix Representation of a graph

- Pro: finding whether vertex i is adjacent to j is $O(1)$
- Con
  - Listing all vertices adjacent to i is $\theta(V)$
  - Space is $\theta(V^2)$

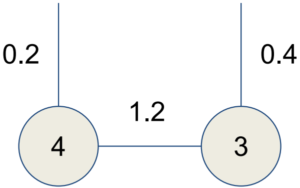
$$\begin{array}{cc} & \text{to} \\ \text{from} & \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

Per-Vertex Edge List with Weights





Matrix Representation with Weights

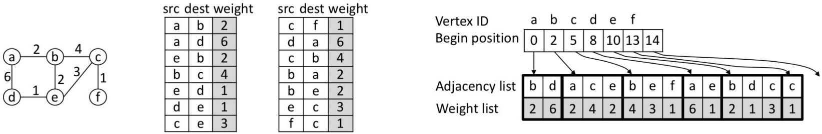


$$\begin{array}{cc} & \text{to} \\ \text{from} & \begin{bmatrix} \infty & 1.5 & \infty & 0.2 \\ 1.5 & \infty & 0.4 & \infty \\ \infty & 0.4 & \infty & 1.2 \\ 0.2 & \infty & 1.2 & \infty \end{bmatrix} \end{array}$$

CSR Representation

Compressed Sparse Row is optimized for GPUs

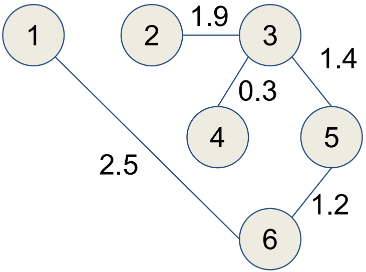
- Sequential access is much faster
- Two Lists: adjacencies and weights
- Third list is the starting position of each vertex within the lists



(a) Sample graph (b) Edge (tuple) list format (Source, destination, weight) (c) CSR format

Image courtesy Professor Hang Liu

Represent the graph in Edge list, matrix, and CSR Representations



Pseudocode for low-level Methods CSR

```
double isAdjacent(int from, int to) {
  int start = startPos[from]; // starting index into main arrays
  for (int i = start; i < startPos[from+1]; i++) {
    if (adjacent[i] == to)
      return weight[i];
  }
  return INFINITY;
}
```

DFS Recursive

Depth-First Search is an algorithm for

- Visiting all connected vertices in a graph
- In a definite order

Can be written easily recursively

- With slightly more work, iteratively

There is a significant advantage to avoiding recursion

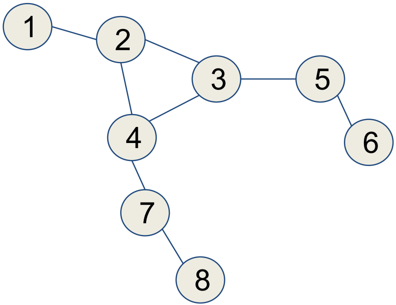
- Performance penalty for recursion
- Stack limitations on modern operating systems to defend against stack smashing attacks

DFS Visually

Start with a known vertex

For each neighbor, if it has never been visited

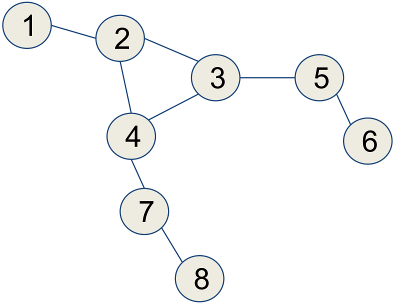
- Go there
- Mark it visited
- Example: start at 4: 4 2 1 3 5 6 7 8



DFS: Different Answers are Possible

DFS solution depends on the order of traversing neighbors For this discussion, always consider that neighbors are visited in ascending order

- Start with 1: 1 2 3 4 7 8 5 6
- Start with 5: 5 3 2 1 4 7 8 6



```
g.DFS(v)
  scheduled[*] ← false //O(V)
  todo.push(v)
  scheduled[v] ← true
  while (NOT stack.empty())
    v â  todo.pop()
    print v
    foreach n ← neighbor(v)
      if NOT  scheduled[n]
        todo.push(n)
        scheduled[n] ← true
      end
    end
  end
end
```

BFS is the same as DFS if solved iteratively

But instead of a stack, **use a queue**

```
g.BFS(v)
  visited[*] ← false
  queue.enqueue(v)
  visited[v] ← true
  while NOT queue.isEmpty() // O(V)
    v ← queue.dequeue()
        foreach n ← neighbor(v) // O(V) omega(1) i
          if NOT visited[n]
            queue.enqueue(n)
            visited[n] ← true
        end
    end
  end
end
```

Dijkstra

```
Graph.Dijkstra(Graph, source)
  for v in Graph.Vertices:
    dist[v] ← INFINITY
    prev[v] ← UNDEFINED
    add v to Q
    dist[source] ← 0
    while Q is not empty
      u ← vertex in Q with min dist[u]
      remove u from Q
      for each neighbor v of u still in Q
        alt ← dist[u] + Graph.Edges(u, v)
        if alt < dist[v]:
          dist[v] ← alt
          prev[v] ← u
```

Floyd-Warshall solves the minimum cost from all points to all points
Example: find the cheapest airline ticket between any two cities (EWR=Newark Airport, LHR=London Heathrow, PEK=Beijing, BOM=Mumbai)
Note: it costs nothing to go from anywhere to itself, you are already there

	EWR	LHR	PEK	BOM
EWR	0			
LHR		0		
PEK			0	
BOM				0

https://www.npr.org/2019/02/13/694352593/lufthansa-airlines-sues-customer-who-skipped-part-of-his-return-flight

```
dist ← |V| x |V| array, all infinity
for each edge (u, v)
  dist[u][v] ← w(u, v) // The weight of the edge (u
for each vertex v
  dist[v][v] ← 0 // going from v to itself costs n
for k ← 1 to V
  for i ← 1 to V
    for j ← 1 to V
      if dist[i][j] > dist[i][k] + dist[k][j]
        dist[i][j] ← dist[i][k] + dist[k][j]
      end if
```

```
dist ← V x V array, all infinity
next ← V x V array, all null
for each edge (u, v)
  dist[u][v] ← w(u, v) // The weight of the edge (u
  next[u][v] ← v
for each vertex v
  dist[v][v] ← 0 // going from v to itself costs n
for k ← 1 to V
  for i ← 1 to V
    for j ← 1 to V
      if dist[i][j] > dist[i][k] + dist[k][j]
        dist[i][j] ← dist[i][k] + dist[k][j]
        next[i][j] ← next[i][k]
      end if
```

A spanning tree is a minimal graph sufficient to connect all vertices
Used in designing efficient networks

- Power
- Internet
- Water
- Sewer

Conversely, can be used by the military

- Maximum disruption per munition

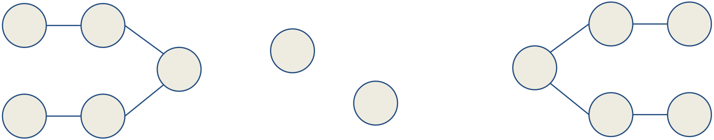
Unfortunately competing goals

- Minimizing cost
- Reliability

Redundancy is expensive

Most networks are trees at the customer side

Internally there can be some redundancy



Prim

Kruskal