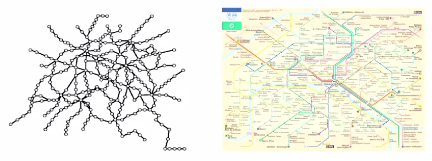
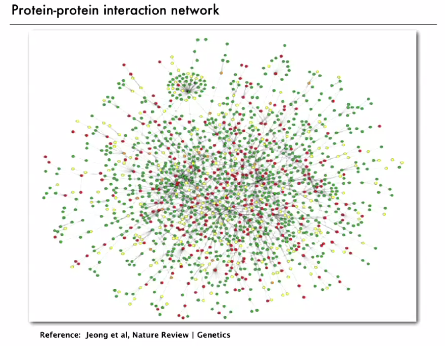
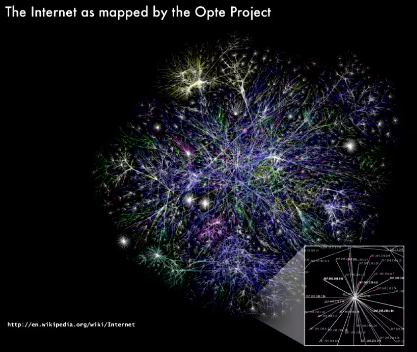
Undirected Graphs

Graph: A set of vertices connected pairwise by edges.

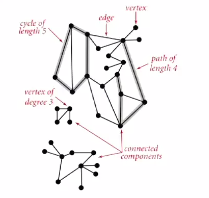


Why study graph algorithms?

* Thousands of practical applications
* Hundreds of graph algorithms known
* Interesting and broadly useful abstraction
* Challenging branch of computer science and discrete math  
    
  



Graph terminology



**Path**: sequence of vertices connected by edges  
**Cycle**: path whose first and last vertices are the same

Two vertices are connected if there is a path between them

**Connected components**: subsets of the graph where each pair of vertices is connected

Graph-processing problems

**Path**: Is there a path between *s* and *t*?  
**Shortest path**: what is the shortest path between *s* and *t*?

**Cycle:** Is there a cycle in the graph?  
**Euler tour:** Is there a cycle that uses each edge exactly once?  
**Hamilton tour:** Is there a cycle that uses each vertex exactly once?

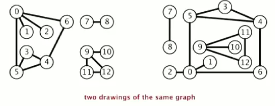
**Connectivity:** Is there a way to connect all of the vertices?  
**MST (minimal spanning tree):** What is the best way to connect all of the vertices?  
**Biconnectivity:** Is there a vertex whose removal disconnects the graph?

**Plurality:** Can you draw the graph on the plane with no crossing edges?  
**Graph isomorphism:** Do two adjacency lists represent the same graph?

Challenge: which problems are easy? Difficult? Intractable?

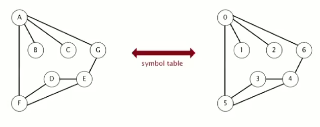
**Graph API**

Graph drawing: provides intuition about the structure of the graph, but intuition can be misleading.



Visual representation

* This lecture: use integers between 0 and *V –* 1 (can use vertex-indexed arrays)
* Applications: convert between names and integers with symbol table



There can be **anomalies**



E.g. we may draw edges while in reality there are multiple edges, we may also have a self loop

**Graph API**

Public class Graph

Graph(int *V*) : create an empty graph with V vertices

Graph(In in): create an empty graph from an input stream

Void addEdge(int v, int w) add an edge v-w

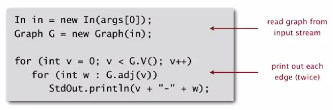
Iterable<Integer> adj(int v) : vertices adjacent to v

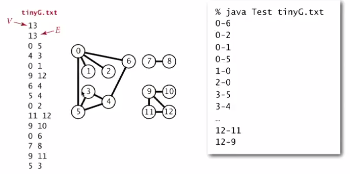
Int V() : number of vertices

Int E() : number of edges

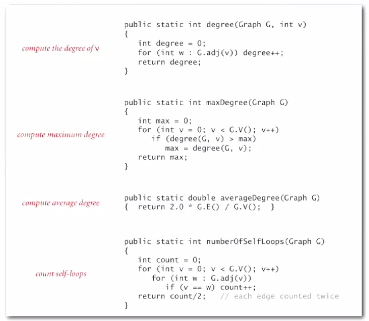
String toString : string representation

Client program below prints out every edge of graph:



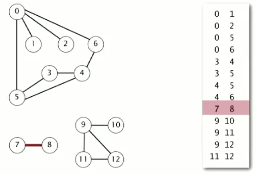


Some common static methods a client may use with graphs



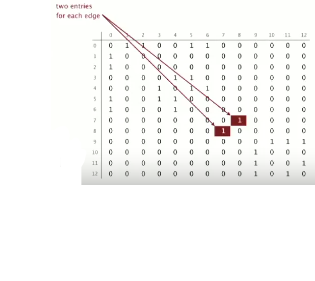
Set-of-edges representation

Maintain a list of the edges (linked list or array) -> but inefficient implementations



Adjacency matric representation

Maintain a 2D *v-*by-*v* Boolean array;  
for each edge v-w in graph: adj[v][w] == adj[w][v] == true

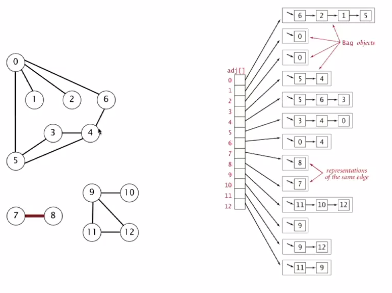


This has limited purposes and is not widely used.

Why? For a huge graph, such as billions of vertices, you would need that number squared (1,000,000,0002) entries in the array, which would be too big for your computer most likely.

Adjacency-list graph representation

Maintain a vertex-indexed array of lists, where for every vertex we maintain a list of vertices that are adjacent to that.

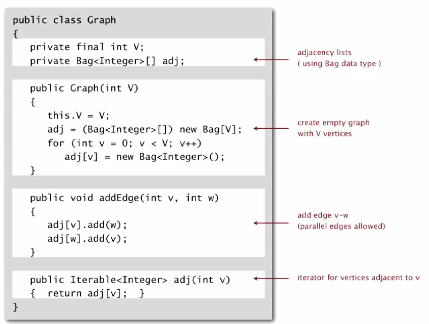


In lower level implementations we may use something like a linked list. However, we will use a bag (which is implemented with a linked list)… and we don’t have to think of the linked list when talking about graphs.

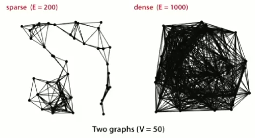
We iterate through in time proportional to entries. Space is proportional to the number of entries.

THIS MEANS WE CAN PROCESS HUGE GRAPHS.

Adjacency list graph implementation:



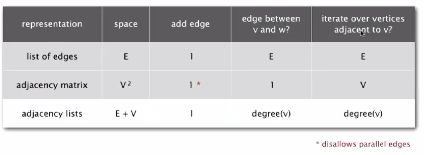
In practice: use adjacency-lists representation.

* Algorithms based on iterating over vertices adjacent to *v*
* Real-world graphs tend to be sparse (huge number of vertices, small average vertex degree) 

Algorithms are based on iterating over vertices adjacent to *v*; this takes time proportional to number of such vertices. Also, in the real world, graphs have many vertices but small vertex degree.

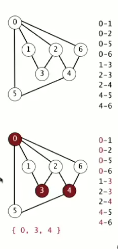
1. We can afford to represent graphs (space proportional to number of edges)
2. We can afford to process because time is proportional to the number of edges

Performance



**Graph Challenges**

* Problem: is a graph bipartite? (Bipartite means: can divide vertices into two subsets with the property that every edge connects a vertex in one subset to a vertex in another)



DIFFICULTY: A diligent algorithms student could create a simple DFS solution to solve this problem.

* Problem: find a cycle

DIFFICULTY: Any programmer could do with DFS

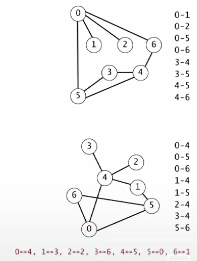
* Problem: Bridges of Konigsberg -🡪 Is there a (general) cycle that uses each edge exactly once?  
  *Euler tour*Answer: A connected graph is Eulerian iff all vertices have even degrees

DIFFICULTY: A diligent algorithms student could program this, but it would be a challenge to complete and to debug.

* Problem: Find a cycle that visits every vertex exactly once (the traveling salesperson)

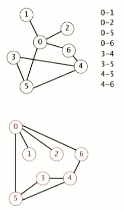
DIFFICULTY: This is intractable. The Hamiltonian cycle problem (NP complete problem). Nobody knows an efficient solution to this problem for large graphs.

* Problem: Are two graphs identical except for vertex names? (graph isomorphism problem)



DIFFICULTY: graph isomorphism is a longstanding problem that nobody yet knows how to solve- or even classify (e.g. easy or impossible)!

* Problem: Can you lay out a graph in the plane without crossing edges?



DIFFICULTY: You need to hire an expert for this. There is a linear-time DFS-planarity algorithm that was discovered by Tarjan in the 1970s… but it’s too difficult for most practitioners.