Why Study Algorithms

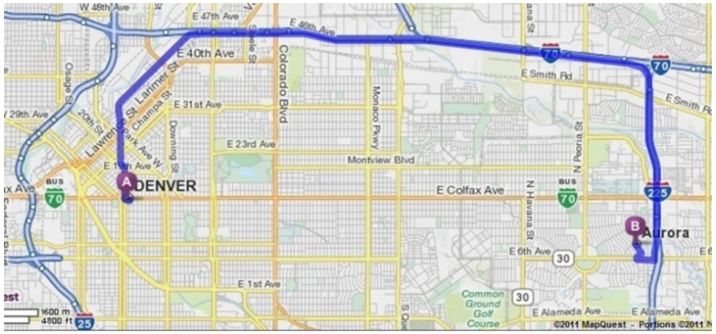
Introduction

- Data science is a hybrid discipline at the intersection of CS, statistics, and, often, a specific domain of study
- Big data
  - Advances in instrumentation, automation, and connectivity result in the generation of massive amounts of high-dim data
  - 2.5 quintillion bytes per day
  - 90% of all data has been generated in the last two years
  - Opportunistically collected data
- Data are not information!
  - Algorithms are central to the process of converting data into information
  - Big data ⇒ need efficient algorithms

Why Study Algorithmics?

- Core technology of computing in general and data science in particular
- Algorithms affect significantly the global behavior of every software system
  - *Moore's law:* # transistors/circuit doubles every two years
    - What is better, switch from a $\Theta(n^2)$ algorithm to a $\Theta(n)$ algorithm, or buy a faster computer?
- Depends on and impacts other technologies
  - Hardware with high clock rates, pipelining, cache, parallel computing (e.g., GPGPU)
  - Local and wide area networking $\Rightarrow$ distributed data structures and programming, Hadoop, MapReduce

Real-World Applications



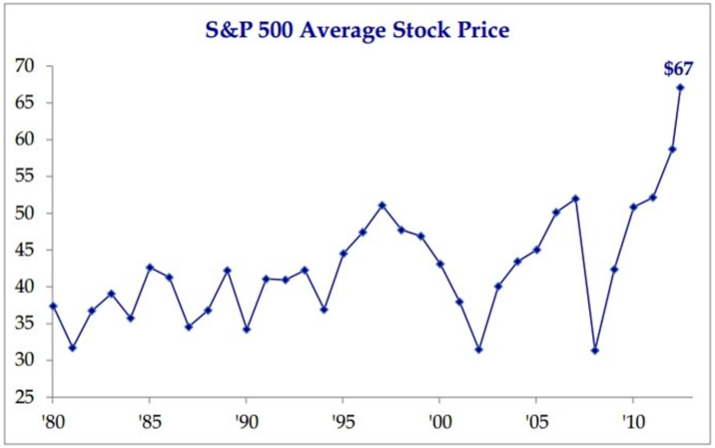
- Finding optimal paths
  - Internet routing
  - Shortest path in road network
  - Route inspection
  - Vehicle routing: UPS, FedEx
- Searching and indexing
  - Web search: Google, Yahoo
  - Human genome: 100,000 genes, three billion base pairs
  - Preference management systems: Netflix, Amazon
  - Determining likely authorship of a document

Real-World Applications

- Data compression
  - Lossless (e.g., text) or lossy (e.g., music, video)
- Privacy and security
  - RSA encrypting requires the generation of very large primes
  - Factoring: find a nontrivial factor of a large integer
- Programming tools
  - Is **`main.java`** a syntactically correct Java program?
  - Debugging
    - Does **`main.java`** go into an infinite loop on a given input?
- Optimization: find the cheapest/best way to do things
  - Airline schedules
  - Network design: AT&T, Sprint, bus transit systems

Real-World Applications

- *Data reduction:* Reduce a massive data set to a much smaller one with minimal loss of information.
  - **Example:** assess risk of developing diabetes
  - Input: the medical history of six million people; each input record consists of a large number of variables
  - Output: a small number of "canonical" profiles based on a small number of variables
- *Model fitting:* Fit a function to predict a value of interest.
  - **Example:** given the price history of a stock, predict its value t time units from now



Class Goals

- Appreciate the importance of efficient algorithms when processing big data sets.
- Familiarize yourself with a few important algorithms, data structures, and algorithm design techniques.
- Learn to analyze and predict various algorithm performance characteristics.
- Obtain a better understanding of the limits of computation (intractability, computability).

END

What Is an Algorithm?

Algorithms: Definition

- Abstraction of a computer program
- *Finite* and *effective* procedure that takes one or more values as *input* and, in *finite time,* produces one or more values as *output*
  - Finite sequence of instructions expressed in the language of a processing agent
- Must solve a *general problem,* specified by:
  - Set of infinitely many possible input instances
  - Properties that the output must satisfy for a given input

Example: sorting, GCD, shortest paths in a graph

Issues

- What models/languages do we use to describe algorithms?
  - Python, Java, C++, pseudo-code, English?
- What characterizes good algorithms?
  - Correctness
  - Efficiency (cost models)
- How do we design good algorithms?
  - Algorithm design paradigms and data structures
- What sort of problems can be solved by algorithms and at what cost?
  - Computability and intractability

Properties of Good Algorithms

- Correctness
  - An algorithm must implement the correct input to output transformation for *all* input instances
  - Not enough to try your algorithm on a few instances
- Efficiency
  - Time
  - Memory
- Simplicity
  - Other things equal, prefer an algorithm that is easier to implement
- Generality

Algorithmic Approach

1. Understand the problem.
   - What data are available, and what is the desired output?
   - Special cases and assumptions
2. Formulate the problem formally.
   - Construct a ***mathematical model***
   - Input and output format
3. Design an algorithm.
   - Algorithm design techniques
   - Data structures
   - Approximation algorithms and heuristics
4. Implement the algorithm as a program.

Pseudocode

- ***Pseudocode*** is a high-level (and often less formal) representation of a program. While it uses constructs typical of a programming language, pseudocode is intended to be read by humans, not computers.
- An ***algorithm*** is a solution to a problem expressed in pseudocode. It consists of a finite sequence of effective steps and a flow of control policy that determines when each step is executed. It must terminate.

Example:. To what degree is friendship transitive?

DegreeOfTransitivity(V, E):

Precondition: Input $G(V, E)$ is an undirected graph

1. Initialize $q = 0$ and $t = 0$
2. **foreach** $(x, y, z) \in V \times V \times V$ such that $\{x, y\} \in E, \{y, z\} \in E$ **do**
3. $\quad t = t + 1$
4. $\qquad$ **if** $\{x, z\} \in E$, **then**
5. $\qquad\quad q = q + 1$
6. **return** q/t

Pseudocode Guidelines

1. Aim for clarity and precision. A competent programmer should be able to implement the algorithm in any language without understanding why it works.
2. Avoid the urge to describe repeated operations informally.
3. Use the constructs of programming languages (loops, conditionals, etc.) to reflect the structure of the algorithm.
4. Use indentation carefully and consistently.
5. Use mnemonic names (except for idioms such as loop indices). Never use pronouns.
6. Write individual steps using English, standard mathematical notation, or a combination.
7. Avoid math notation if English is clearer (e.g., insert x into *S*).
8. Use one statement or structuring element per line.
9. Font should enhance structure and functionality.

END

Efficiency

Efficiency

- What do we mean by efficiency?
  - Minimum use of resources: time and space required
  - Time/space are machine, language, implementation dependent
- Program efficiency is important, especially for big data, but how do we measure it?
  - Need a methodology that applies broadly to any algorithm
  - Expressed as a function of input size (denoted by n)
    - Defined per problem class: length of list (sorting, searching), number of bits in single input number (primality testing), number of nodes and edges in a graph (transitivity)
  - Would like to choose among different algorithms *before* implementation ⇒ need a computation model independent of implementation

Example: Matrix Multiplication

- *Input:* matrices *A*, *B* of sizes $q \times p$ and $p \times r$, respectively
- *Output:* matrix *C* of size $q \times r$, where *C* = *A* × *B*

$$C_{ij} = \sum_{k=1}^{p} A_{ik} \cdot B_{kj}$$

Example

$$\begin{bmatrix} 2 & -3 & 3 \\ -2 & 6 & 5 \\ 4 & 7 & 8 \end{bmatrix} \times \begin{bmatrix} -1 & 9 & 1 \\ 0 & 6 & 5 \\ 3 & 4 & 7 \end{bmatrix} = \begin{bmatrix} 7 & 12 & 8 \\ 17 & 38 & 63 \\ 20 & 110 & 95 \end{bmatrix}$$

Matrix Multiplication Algorithm

```
MatrixMul(A, B, p)
   ▷ Multiply two p × p matrices A and B
   for i ← 0 to p − 1
        do for j ← 0 to p − 1
              do sum ← 0
                 for k ← 0 to p − 1
                      do sum ← sum + A[i, k] · B[k, j]
                 C[i, j] ← sum
   return C
```

p^3

- How many operations are performed?
  - What is the input size? $n = p^2$
  - # of operations is a function of *n* $\Theta\left(n\sqrt{n}\right)$

Brute Force Approach

- MatrixMul is an example of a ***brute force*** solution, referring to the fact that the algorithm uses a straightforward approach, directly based on the problem statement and definitions of the concepts involved.
- It usually results in algorithms that are easy to implement but not very efficient.
- Optimization problems can often be solved by brute force by performing an *exhaustive search* in a space of candidate solutions.

END

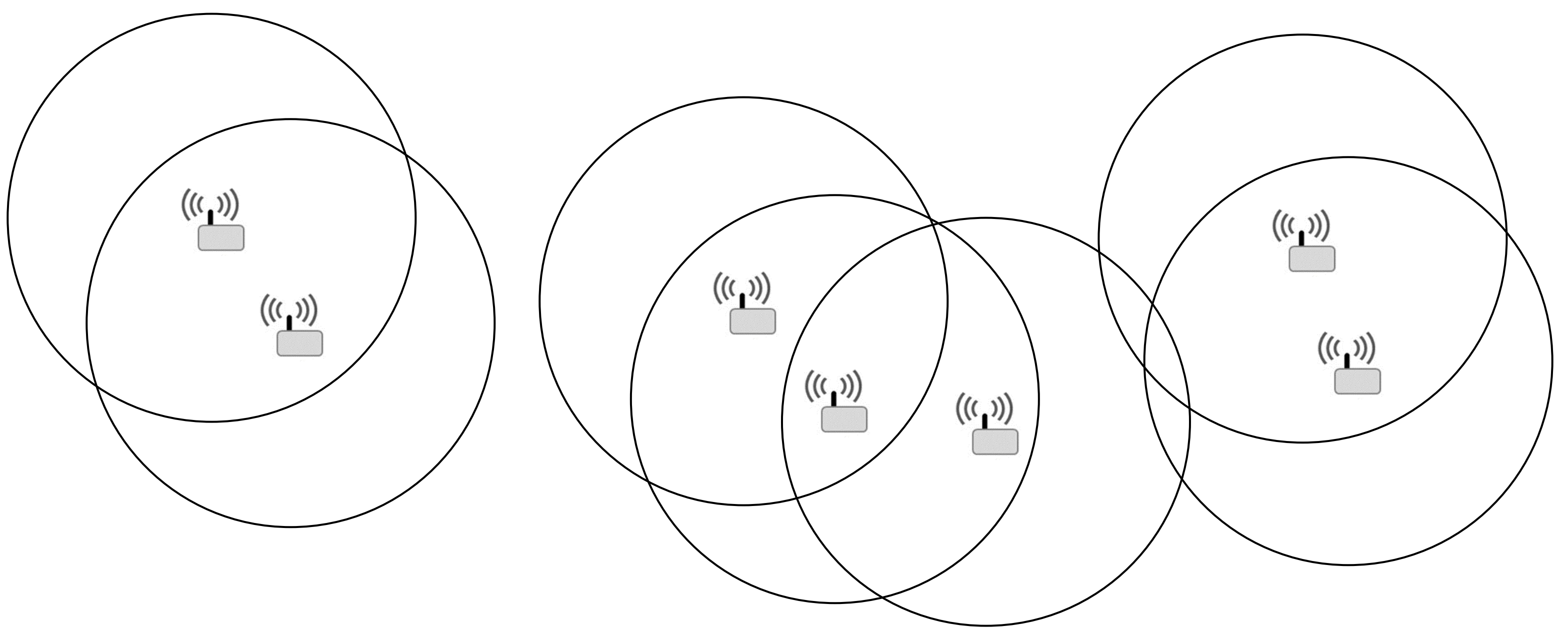
Models

Models

- The most important skill of an algorithm designer is, arguably, that of *modeling,* the art of abstracting away the messy details of a real-world application into a mathematical structure suitable for algorithmic solution.
- With many fundamental algorithms implemented in Python libraries, you still need to model your problem properly before choosing the right algorithms to use.
- Real-world problems deal with real-world entities, such as people, Web pages, accounts, and so on.
- An algorithmic solution, on the other hand, deals with properly defined *abstract structures,* such as graphs, sets, maps, and so on.

Example 1

- Consider a wireless sensor network consisting of n sensors, each capable of transmitting within distance r. Determine which pairs of sensors can communicate with each other, possibly indirectly by message relaying.



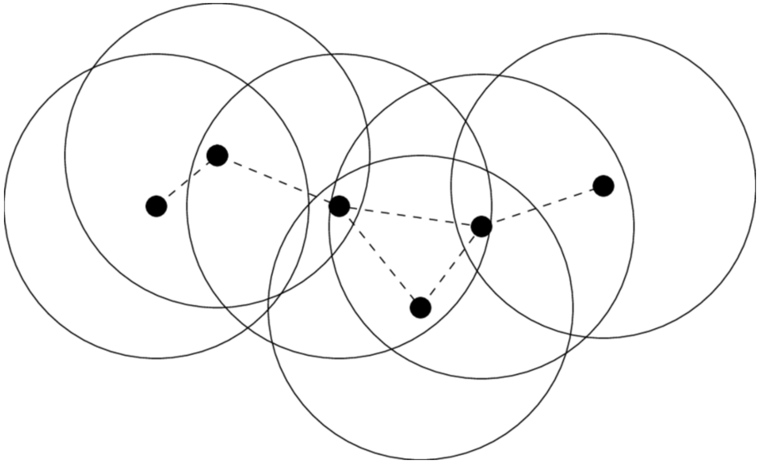
Example 2

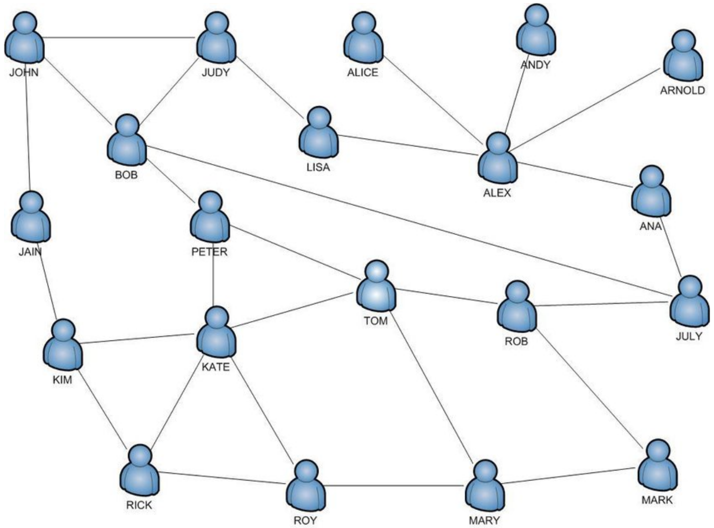
- Consider a system of interactions, such as the friendship relation among a set of people.
- We want to determine to what extent the interactions exhibit transitivity. Recall that a relation R is transitive for all x, y, z: If xRy and yRz, then xRz (xRy means that x interacts with y).
- Given Facebook data, we wish to investigate how likely it is that x is a friend of z whenever x is a friend of y and y is a friend of z.

Graph Models

Graph Models

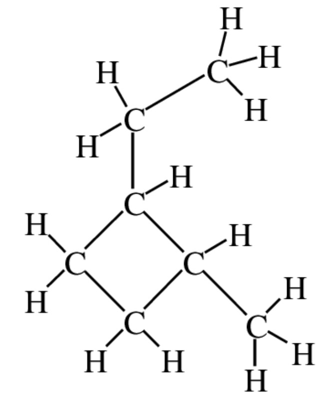
- What do the previous examples have in common?
- *Both* can be modeled using a mathematical abstraction called a ***graph.***
    - Create a dot for each sensor and a link between two sensors if they each lie within each others' range.
    - Create a dot for each person and a link between two people if they are mutual friends.

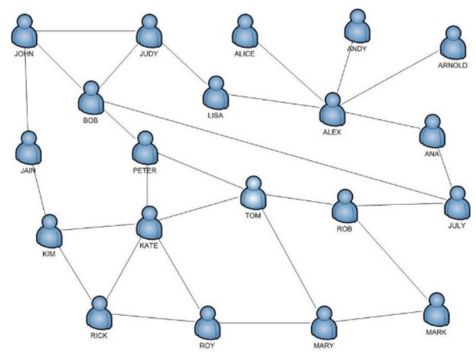


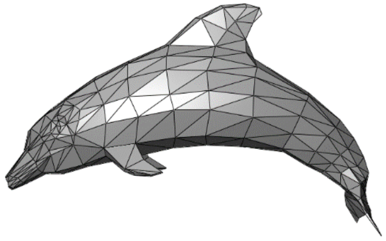


Graphs: Review

- Many problems can be abstracted by graphs
- A graph is a pair of sets
  - A nonempty finite set *V* of elements
  - A finite set *E* of links connecting pairs of elements
- Numerous applications
  - Mathematical structures such as relations, permutations, trees, functions
  - Physical/biological structures such as molecules
  - World Wide Web
  - Subway and road maps
  - Social, phone, computer networks
  - Terrains and objects as polygonal meshes
  - Many more







Definitions

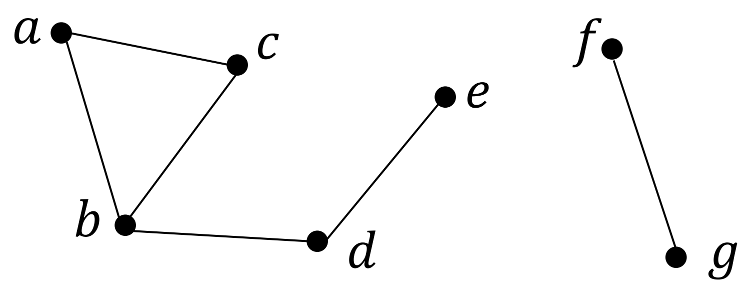
- A ***graph*** G is an ordered pair (V, E), where V is a finite set and E is a finite multiset of pairs from $V \times V$. The members of V are the ***vertices*** of G and those of *E* are the ***edges*** of G.
- In an ***undirected graph,*** the edges have no orientation. Hence, each edge is a two-element multiset of vertices.
    - Since edge $(u, v) = (v, u)$, we can simply write $\{u, v\}$.
- In a ***directed graph,*** edges are ordered pairs. The pair (u, v) denotes the edge from *u* to *v*.
    - Replacing each (u, v) by $\{u, v\}$ gives the ***symmetrization*** of G.
- An edge of the form (v, v) is called a ***loop.***
- Two edges with the same end vertices are said to be ***parallel.***
- A **simple graph** is one with no loops or parallel edges; a graph that allows parallel edges is called a ***multigraph.***

Definitions

- If $e = (u, v) \in E$ and G is directed, then v is the ***head*** *of* e and u is the ***tail*** *of* e.
- If $e = (u, v) \in E$, then u and v are ***adjacent*** and e is ***incident*** with u and with v.
- If G is undirected, the number of edges incident with v is the ***degree*** *of* v and denoted $\deg_G(v)$. If G is directed, we distinguish between ***in-degree*** (number of incident heads) and ***out-degree*** (number of incident tails) and denote these using $\mathrm{in}_G(v)$ and $\mathrm{out}_G(v)$, respectively.
- A ***weighted graph*** $G(V, E, w)$ incudes a mapping $w: E \to \mathbb{R}$ that assigns a real number (cost, distance, etc.) to each edge.
- In the sequel, the term "graph" by itself, refers to an undirected graph.

Simple Undirected Graphs

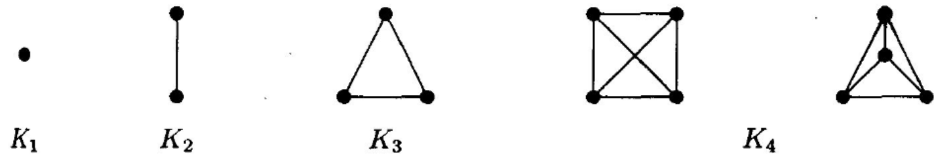
- A *simple undirected graph* G consists of a nonempty set V of vertices and a set E of two-element subsets of V.
- They are used to model relationships that are *symmetric,* such as being married, speaking the same language, overlapping in time, and so on.



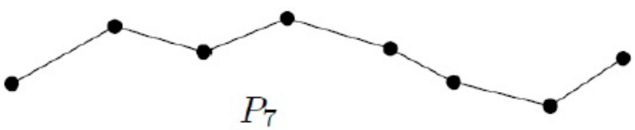
$V = \{a, b, c, d, e, f, g\}$ $\quad$ $E = \{\{a, b\}, \{b, c\}, \{a, c\}, \{b, d\}, \{d, e\}, \{f, g\}\}$

Common Graphs Types

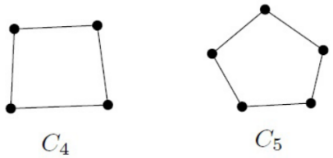
- The *complete graph or clique* K_n has vertex set $\{1, \ldots, n\}$ and an edge between every pair of vertices—that is, $E = \binom{V}{2}$.



- The *path* P_n has vertex set $\{0,1, \ldots, n\}$ and edge set $E = \{\{i-1, i\}, i = 1, \ldots, n\}$.

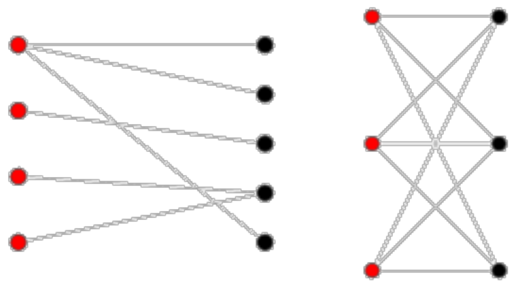


- The *cycle* C_n has vertex set $\{1, \ldots, n\}$ and edge set $E = \{\{1,2\}, \ldots, \{n-1, n\}, \{1, n\}\}$.



Common Graph Types

- A *bipartite graph* $G(U \cup V, E)$ is a graph whose vertices can be partitioned into sets U and V such that every edge is incident with a vertex in U and a vertex in V.

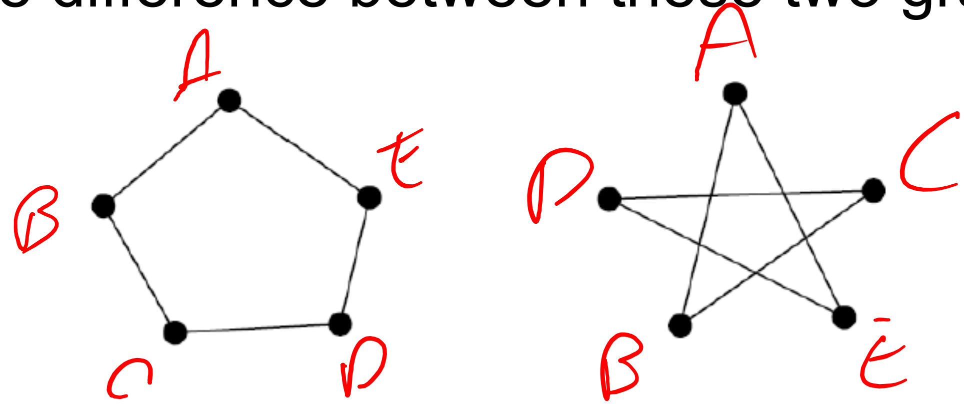


- The complete bipartite graph $K_{n,m}$ has vertex set $U \cup V$, $U = \{u_1, \dots, u_n\}, V = \{v_1, \dots, v_m\}$ and edges $E = U \times V$.

$K_{1,1}$ $K_{1,2}$ $K_{1,3}$ $K_{2,3}$ $K_{3,3}$

Computer Representation

- What is the difference between these two graphs?



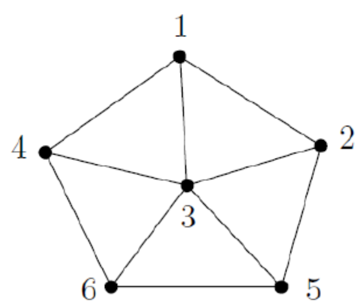
- A graph is not a picture! With the right vertex labeling, both pictures represent the graph $\big(\{1,2,3,4,5\}, \{\{1,2\},\{2,3\},\{3,4\},\{4,5\},\{1,5\}\}\big)$.
- There are two common ways to represent a graph $G(V,E)$ in a program:
  1. Adjacency matrix
  2. Adjacency list

Adjacency Matrix

- **Definition:** Let $G = (V, E)$ be a graph with n vertices. Label the vertices $\{v_1, \dots, v_n\}$. The ***adjacency matrix*** of G, with respect to the given numbering, is a list A of length n, where $A[i]$ is a $n \times n$ matrix $A(G) = (a_{ij})$ defined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E \\ 0 & \text{otherwise} \end{cases}$$

- ***Example***

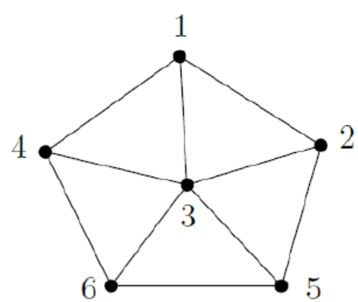


$$\begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ \\ \\ \\ \\ \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

- Requires $\Theta(n^2)$ space $\Rightarrow$ inefficient when $m \ll n(n-1)/2$

Adjacency List

- **Definition:** Let $G = (V, E)$ be a graph with n vertices. Label the vertices $\{v_1, \ldots, v_n\}$. The ***adjacency list*** of G, with respect to the given numbering, is a list A of length n, where $A[i]$ is a list of the vertices of G adjacent to v_i.
- ***Example***

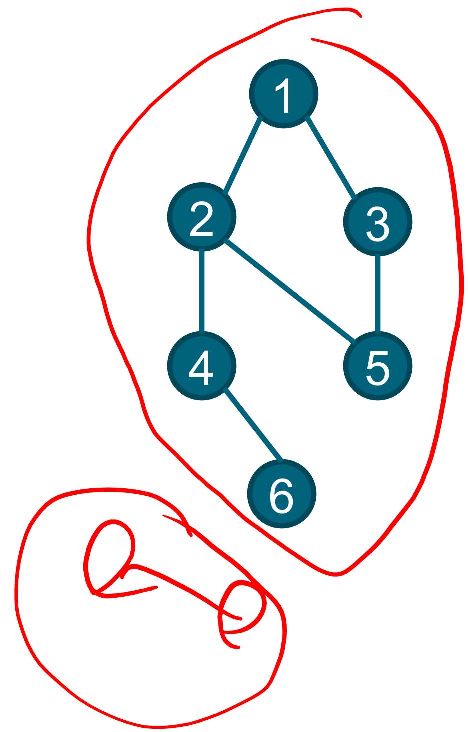


1	$\langle 2, 3, 4 \rangle$
2	$\langle 1, 3, 5 \rangle$
3	$\langle 1, 2, 4, 5, 6 \rangle$
4	$\langle 1, 3, 6 \rangle$
5	$\langle 2, 3, 6 \rangle$
6	$\langle 3, 4, 5 \rangle$

- Requires only $\Theta(n + m)$ space. Why? How about performance? How would you implement in Python?

Paths and Connectedness

- Let $G = (V, E)$ denote an undirected graph and let $u, v \in V$. There is a path of length k between $u = u_0$ and $v = u_k$ if there are $k - 1$ nodes $u_1, \ldots, u_{k-1}$ such that $\{u_{i-1}, u_i\} \in E$ for $i = 1, \ldots, k$.
- A simple path is one with no repeated nodes.
- The ***distance*** between nodes u and w, denoted d_{uw}, is the smallest k such that there is a path of length k between u and w.
- Let $u \rightsquigarrow v$ if there is a path from u to v. Graph $G = (V, E)$ is ***connected*** if $\forall u, v \in V \; u \rightsquigarrow v$.
- $\rightsquigarrow$ is an equivalence relation. The equivalence classes of $\rightsquigarrow$ are the ***connected components*** of G.



END

Acknowledgements

- Mapquest.com
- wsj.com (Wall Street Journal)
- Filipowski, Tomasz & Kazienko, Przemyslaw & Brodka, Piotr & Kajdanowicz, Tomasz. (2012). Web-based knowledge exchange through social links in the workplace. Behaviour & Information Technology. 31. 779-790
- Janezic Dusanka & Milicevic, Ante & Nikolic, Sonja & Trinajstic, Nenad. (2007). Graph Theoretical Matrices in Chemistry.
- “An Invitation to Discrete Mathematics”, Jiri Matousek and Jaroslav Nesetril, Oxford University Press, 2008.