Foundations

- **Design and Analysis of Algorithms, an Example (Ch 1).**
- **Basics of Algorithm Analysis, Computational Models, Computational Tractability, Asymptotic Order of Growth (Ch 2).**
- **Graphs, Undirected graphs, Directed graphs, Formulate Problems by Graphs (Ch 3).**

The lecture notes/slides are adapted from those associated with the text book by J. Kleinberg and E. Tardos.

Design and Analysis of Algorithms

- **Algorithms are a foundation of computer science**
- **Design and analysis of algorithms**
  - **Identify a problem**
  - **Formulating the problem with certain mathematical precision**
  - **Design an algorithm to solve the problem**
  - **Analyze the algorithm**

    Prove the correctness of algorithm

    Analyze the efficiency of algorithm

Identify a problem, an example

Stable Matching Problem

- **A group of employers are hiring; a group of students are applying to employers for jobs.**
- **Each employer has a preference order on the applicants; each student has a preference order on employers.**
- **Based on preferences, employers give offers to some applicants; applicants choose one offer to accept.**
- **Design a scheme to assign each applicant to an employer so that employer and applicant pairs are stable.**

- **Unstable employer-applicant pairs**
  - **Applicant A had accepted an offer from employer E_1,**
    later employer E_2 gave A an offer,
    A prefers E_2 to E_1 and retracts the acceptance from E_1.
  - **Employer E had given an offer to applicant A_1,**
    later E received an application from applicant A_2,
    E thinks A_2 is better than A_1 and retracts the offer to A_1.
- **Stable employer-applicant assignment, there is no unstable employer-applicant pair**

- **Stable employer-applicant assignment**

  Given the preferences of employers and applicants, assign applicants to employers so that for each employer E and each applicant A who is not assigned to E, one of the following two conditions holds:

  1. **E prefers every applicant it made an offer to A; or**
  2. **A prefers her current employer to E.**

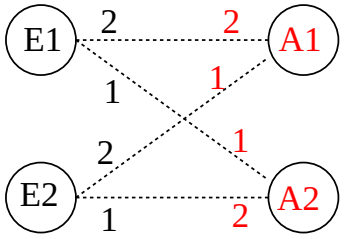
- **Stable matching, one of the following two conditions holds:**
  1. E **prefers every applicant it made an offer to** A**; or**
  2. A **prefers her current employer to** E**.**
- **Example, two employers** E_1, E_2 **and two applicants** A_1, A_2

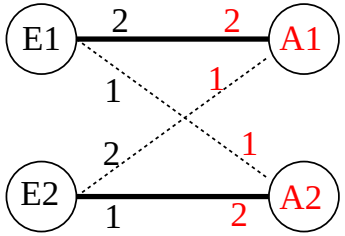
  E_1 **prefers** A_1 **to** A_2 **and** E_2 **prefers** A_1 **to** A_2

  A_1 **prefers** E_1 **to** E_2 **and** A_2 **prefers** E_2 **to** E_1

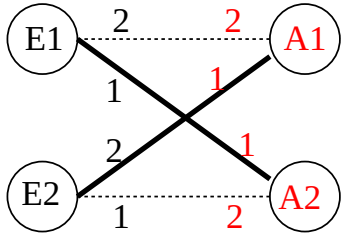
  $\{(E_1, A_1), (E_2, A_2)\}$ **is a stable assignment, Conditions 2 is satisfied**

  $\{(E_1, A_2), (E_2, A_1)\}$ **is a not stable assignment, neither of Conditions 1&2 is satisified,** (E_1, A_1) **does not satisfy Condition 1 nor 2**





Stable matching
S={(M1,A1),(M2,A2)}



Unstable matching
S={(M1,A2),(M2,A1)}

- **Stable matching, one of the following two conditions holds:**
  1. E **prefers every applicant it made an offer to** A**; or**
  2. A **prefers her current employer to** E**.**
- **More stable assignment examples**

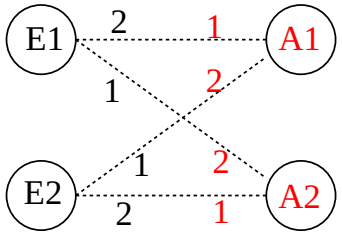
  two employers E_1, E_2 **and two applicants** A_1, A_2

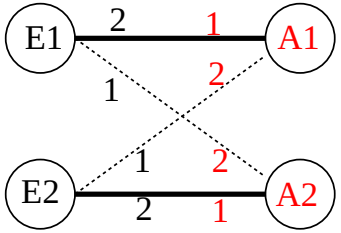
  E_1 **prefers** A_1 **to** A_2 **and** E_2 **prefers** A_2 **to** A_1

  A_1 **prefers** E_2 **to** E_1 **and** A_2 **prefers** E_1 **to** E_2

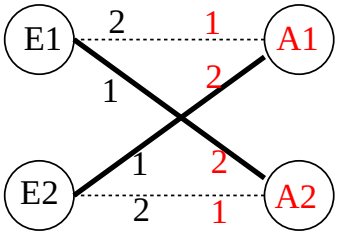
  $\{(E_1, A_1), (E_2, A_2)\}$ **is a stable assignment, as Condition 1 is satisfied**

  $\{(E_1, A_2), (E_2, A_1)\}$ **is a stable assignment, as Condition 2 is satisfied**





Stable matching
S={(M1,A1),(M2,A2)}



Stable matching
S'={(E1,A2),(E2,A1)}

Formulate the problem

- **Simplify the problem**

  Each student applies to every employer, each employer accepts one applicant.

- **Notation for employer and applicant pairs**

  Let $M = \{m_1, .., m_n\}$, $W = \{w_1, .., w_n\}$ **and**
  $M \times W = \{(m, w) | m \in M, w \in W\}$.

  $S \subseteq M \times W$ **is a matching if each element of** M **appears in at most one pair of** S **and each element of** W **appears in at most one pair of** S.

  S **is a perfect matching if each element of** M **appears in exactly one pair of** S **and each element of** W **appears in exactly one pair of** S.

  Example: $M = \{m, m'\}, W = \{w, w'\}, S_1 = \{(m, w)\}$ **is a matching,**
  $S_2 = \{(m, w), (m, w')\}$ **is not a matching as** m **appears in more than 1 pair in** S_2,
  $S_3 = \{(m, w), (m', w')\}$ **is a perfect matching.**

- **Preferences and ranks**

  Each $m \in M$ **has a distinct rank** $f_m(w)$ **on every** $w \in W$

  each $w \in W$ **has a distinct rank** $g_w(m)$ **on every** $m \in M$

  Example: $M = \{m, m'\}$ **and** $W = \{w, w'\}$

  - $f_m(w) = 2$ **and** $f_m(w') = 1$, m **prefers** w **to** w'
  - $f_{m'}(w) = 2$ **and** $f_{m'}(w') = 1$, m' **prefers** w **to** w'
  - $g_w(m) = 2$ **and** $g_w(m') = 1$, w **prefers** m **to** m'
  - $g_{w'}(m) = 1$ **and** $g_{w'}(m') = 2$, w' **prefers** m' **to** m
  - **In general, bijection** $f_m : W \to \{1, 2, .., n\}$, **bijection** $g_w : M \to \{1, 2, .., n\}$.

    If m **prefers** w **to** w' **then** $f_m(w) > f_m(w')$.

    If w **prefers** m **to** m' **then** $g_w(m) > g_w(m')$.

- **Instability**

  Given a matching $S \subseteq M \times W$, $(m, w) \in (M \times W) \setminus S$ is unstable w.r.t. (with respect to) S if

  1. $(m, w') \in S$, w **does not appear in** S **and** $f_m(w) > f_m(w')$ **or**
  2. $(m', w) \in S$, m **does not appear in** S **and** $g_w(m) > g_w(m')$ **or**
  3. $(m, w') \in S, (m', w) \in S$, $f_m(w) > f_m(w')$ **and** $g_w(m) > g_w(m')$.

- **Stable matching**

  A matching S is stable if

  1. S **is a perfect matching; and**
  2. **there is no instability w.r.t.** S**: for every** $(m, w'), (m', w)$ **in** S $f_m(w') > f_m(w)$ **(Condition 1) or** $g_w(m') > g_w(m)$ **(Condition 2).**

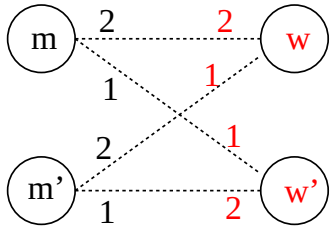
- **Stable employ-applicant assignment: find a stable matching**

- **Questions**

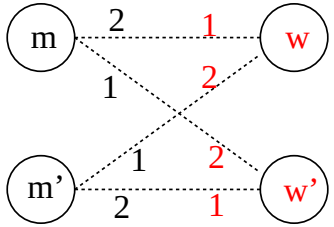
  Does there exist a stable matching for every set of preference lists?

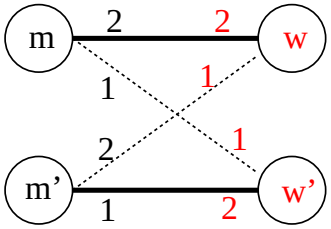
  Given a set of preference lists, if a stable matching can be computed efficiently?

Stable matching examples

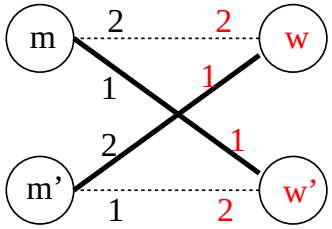


$f_m(w)$ $g_w(m)$

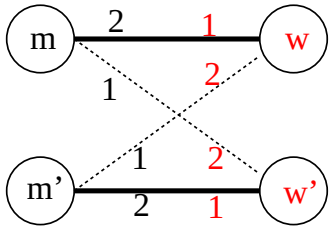




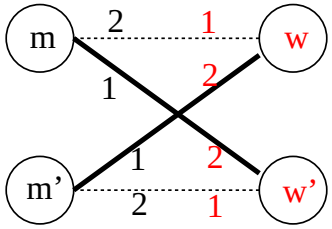
Stable matching
S={(m,w),(m',w')}



Unstable matching
S={(m,w'),(m',w)}
$f_m(w) > f_m(w')$, $g_w(m) > g_w(m')$



Stable matching
S={(m,w),(m',w')}



Stable matching
S'={(m,w'),(m',w)}

Design algorithm

G-S Algorithm [Gale and Shapley 1962] for Stable Matching

1. **Initially, every element is unpaired.**
2. **For every unpaired** m**, let** $W_m = \{w | w \in W$ **and** m **has not tried to pair**$\}$ **and let** $w \in W_m$ **with the largest** $f_m(w)$**;**

   If w **is unpaired then put** (m, w) **into a pair;**

   Otherwise (there is a pair (m', w)**),**

   if $g_w(m) > g_w(m')$ **then remove pair** (m', w) **and put** (m, w) **a pair,**

   otherwise mark w **as that** m **has tried to pair.**
3. **Finally, algorithm stops when every one is paired.**

Initially, $S := \emptyset$; $W_m = W$ for every $m \in M$; /* every one is unpaired */
while $\exists$ unpaired $m \in M$ **do**
    Choose an unpaired m; Let $w \in W_m$ with the largest $f_m(w)$;
    if w is unpaired **then**
        $S := S \cup \{(m, w)\}$
    else
        Let (m', w) be the pair in S;
        if $g_w(m) > g_w(m')$ **then**
            remove (m', w) from S; $S := S \cup \{(m, w)\}$
        else
            Remove w from W_m;
        end if
    end if
end while
Return S;

Example, $M = \{m, m'\}$ **and** $W = \{w, w'\}$,

$f_m(w) = 2$ **and** $f_m(w') = 1$, $f_{m'}(w) = 2$ **and** $f_{m'}(w') = 1$,

$g_w(m) = 2$ **and** $g_w(m') = 1$, $g_{w'}(m) = 1$ **and** $g_{w'}(m') = 2$

Initially, $S = \emptyset$, $W_m = \{w, w'\}$, $W_{m'} = \{w, w'\}$

Choose m', w **has the largest** $f_{m'}(w) = 2$ **in** $W_{m'}$, w **is unpaired,** $S = \{(m', w)\}$

Choose m, w **has the largest** $f_m(w) = 2$ **in** W_m, w **is paired with** m',

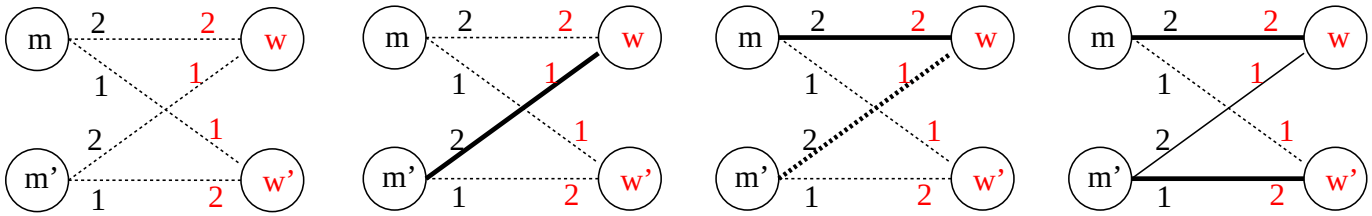
$g_w(m) = 2 > g_w(m') = 1$, $S = (S \setminus \{(m', w)\}) \cup \{(m, w)\} = \{(m, w)\}$

Choose m', w **has the largest** $f_{m'}(w) = 2$ **in** $W_{m'}$, w **is paired with** m,

$g_w(m') = 1 \not> g_w(m) = 2$, $W_{m'} = \{w, w'\} \setminus \{w\} = \{w'\}$

Choose m', m' **has the largest** $f_{m'}(w') = 1$ **in** $W_{m'}$, w' **is unpaired,**

$S = S \cup \{(m', w')\} = \{(m, w), (m', w')\}$



$f_m(w)$ $g_w(m)$

W_m ={w,w'} $W_{m'}$ ={w,w'}

Choose (m',w), S={m',w}

Choose (m,w), w is paired but w prefers m, S={m,w}
Choose (m',w), w is paired but w prefers m, $W_{m'}$ ={w'}

Choose (m',w'), S={(m,w),(m',w')}

Analyze algorithm

- **Properties of G-S algorithm**

  (1) Once a $w \in W$ is paired, w remains paired.

  (2) For pairs (m_1, w) and (m_2, w) with (m_1, w) created earlier than (m_2, w), $g_w(m_1) < g_w(m_2)$.

- **G-S Algorithm terminates after at most n^2 iterations of the while loop.**

  Proof. Let $P(t)$ be the set of pairs (m, w) s.t. m has tried to pair by the end of iteration t of the while loop. Then $|P(t+1)| > |P(t)|$. Since there are n^2 pairs in $M \times W$. □

- **$|P(t)|$ is a progress measure which specifies that each step of the algorithm makes the algorithm closer to termination.**

Theorem. [Gale-Shapley 1962] G-S Algorithm returns a stable matching at the termination.

Proof. Since S is a matching and $|M| = |W| = n$, by Property (1), if there is an unpaired m then there is an unpaired w s.t. m has not tried to pair with yet. So, every m will pair with a w and the algorithm returns a perfect matching S at the termination.

Assume for contradiction that there is an instability w.r.t. S. Then there is a pair $(m, w) \in (M \times W \setminus S)$ s.t. $(m, w'), (m', w) \in S$, $f_m(w) > f_m(w')$ and $g_w(m) > g_w(m')$. Then m had tried to pair with w before m paired with w'. Since $(m, w) \notin S$, by Property (2), $g_w(m) < g_w(m')$, a contradiction. So, S is stable. □

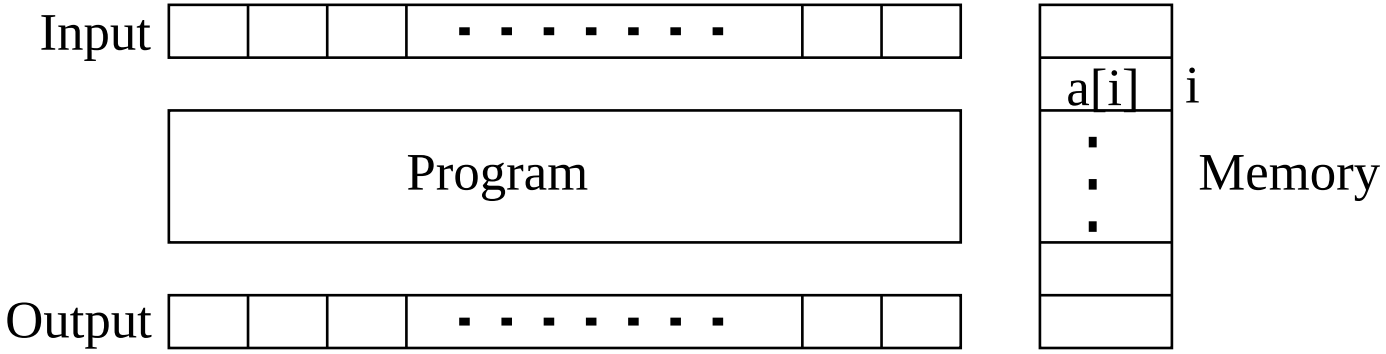
Discussion

- **The number of iterations of the while loop can be close to n^2.**
- **There may be multiple stable matchings for an input instance.**
- **G-S Algorithm may have different orders of m to make pairs in different executions.**
- **For a same input instance, different executions of the algorithm give the same stable matching S^* which has the following properties:**
  - **For $(m, w^*) \in S^*$ and (m, w) in any stable matching S, $f_m(w^*) \geq f_m(w)$.**
  - **For $(m^*, w) \in S^*$ and (m, w) in any stable matching S, $g_w(m^*) \leq g_w(m)$.**

Computation Models

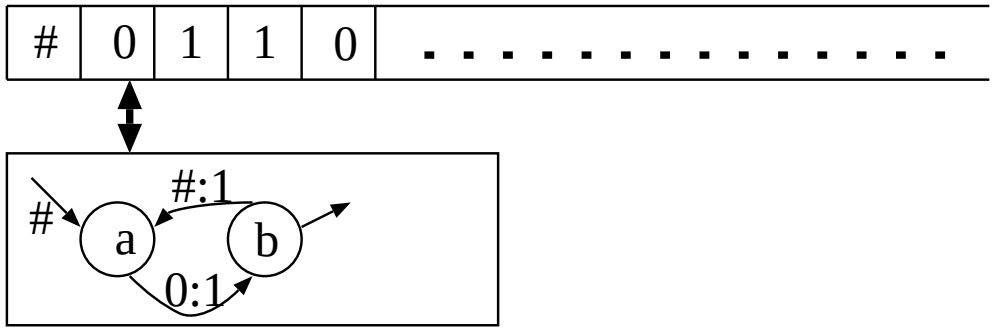
Word RAM (Random Access Memory)

- **Random access memory, input, output and program**
- **Each memory/input/output cell stores a w-bit integer.**
- **Primitive operations, arithmetic/logic operations, read/write memory, array indexing, following a pointer, conditional branch, ...**
- **Running time, number of primitive operations (constant time per operation).**
- **Memory size, number of memory cells used.**
- **More refined models may be needed (e.g., for n-bit integers).**



Turing Machines

- **A tape divided into cells, each cell has a symbol from a finite set.**
- **A set of finite machine states.**
- **A set of finite instructions which define that given a machine state q_i and a symbol a_j, the next step action of the machine.**
- **When every next step action is unique, it is deterministic Turing machine.**
- **When some of the next step actions is a set of actions and the machine can take any one action in the set, it is nondeterministic Turing machine.**
- **Running time, number of steps.**
- **Memory size, number of tape cells used.**



Computational Tractability

- **An algorithm is a poly-time (polynomial running time) algorithm if there are constants $c > 0$ and $d > 0$ s.t. for every input of size n, the running time of the algorithm is bounded above by cn^d primitive computational steps.**
- **A problem admits a poly-time algorithm if there is an algorithm which, given any input of size n of the problem, gives an exact solution to the input instance in polynomial time in n.**
- **A problem is computational tractable if it admits a poly-time algorithm.**
- **A poly-time algorithm is called an efficient algorithm.**
- **A poly-time algorithm has the desirable scaling property: When the input size doubles, the running time increases by at most a constant factor.**

Measures for running time

- **Worst-case running time**

  Largest running time of an algorithm for any input of size n.

- **Average-case running time**

  Running time of an algorithm averaged for all possible inputs of size n.

  The notion of ”averaged” assumes a probability distribution over the inputs.

- **Expected running time**

  Running time of a randomized algorithm.

- **Amortized running time**

  Worst case running time of any sequence of n operations.

Asymptotic Order of Growth

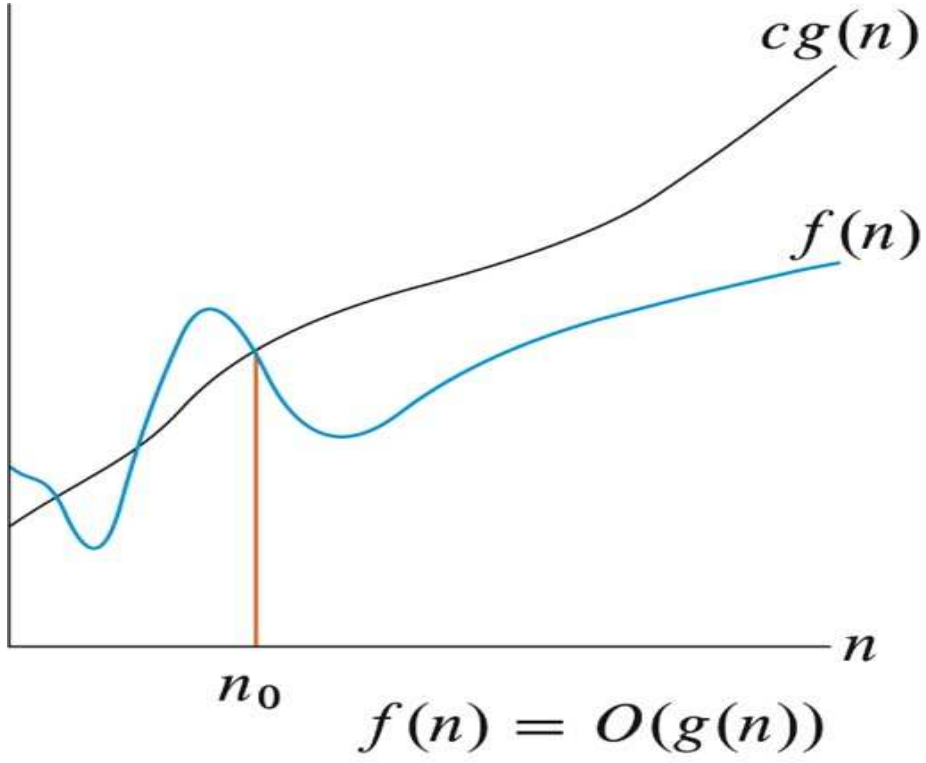
- **A way to describe how the value of a function increases in the limit.**
- **Foucs on the dominating term for the growth, igonor constant factors and low-order terms.**
- **A way to express running times of algorithms.**
- **A way to compare the growth rates of functions.**
- **Notations for comparing order of growth**

$$
\begin{aligned}
f(n) \text{ is } O(g(n)) &\approx a \leq b \\
f(n) \text{ is } \Omega(g(n)) &\approx a \geq b \\
f(n) \text{ is } \Theta(g(n)) &\approx a = b \\
f(n) \text{ is } o(g(n)) &\approx a < b \\
f(n) \text{ is } \omega(g(n)) &\approx a > b
\end{aligned}
$$

- **Big-Oh notation, $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.**

  For upper bounds on time/space complexity.

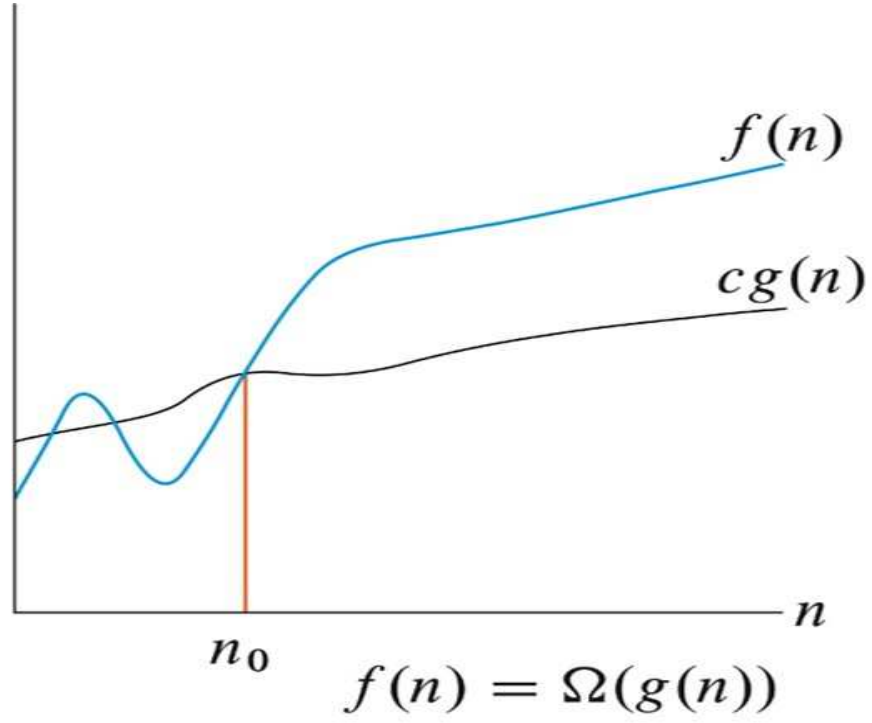
  $f(n) = 5n^2 + 7n + 4$, **$f(n)$ is $O(n^2)$; $f(n)$ not $O(n^k)$ for any $0 \leq k < 2$ nor $O(n \log n)$.**



- **Big-Omega notation, $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. $f(n) \geq c \cdot g(n) \geq 0$ for all $n \geq n_0$.**

  For lower bounds on time/space complexity.

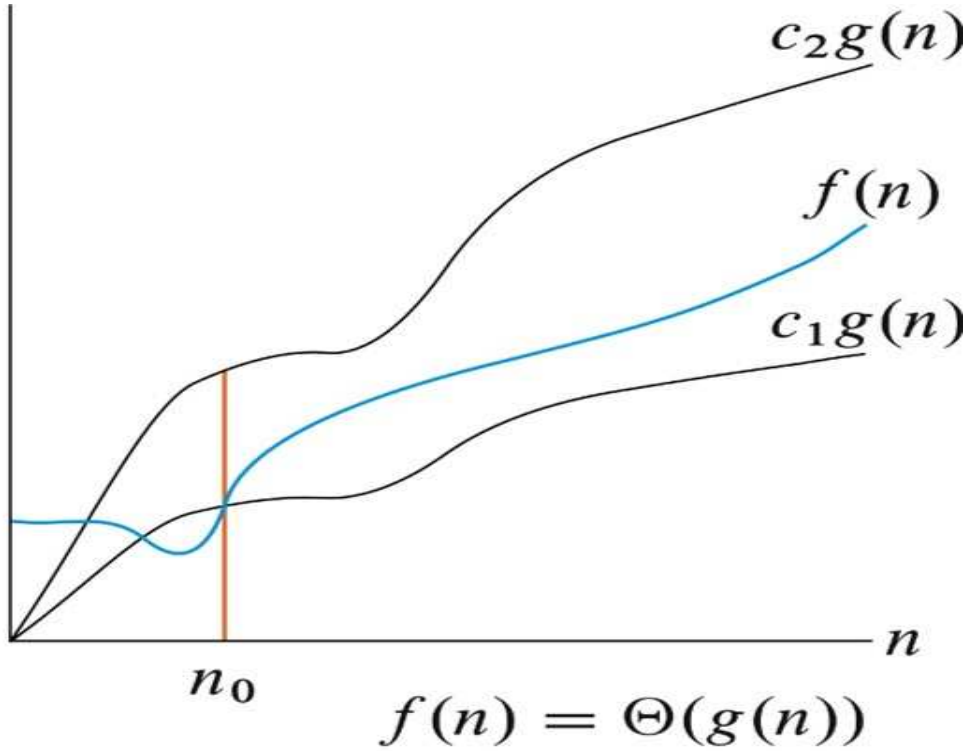
  $f(n) = 5n^2 + 7n + 4$, $f(n)$ is $\Omega(n^k)$ for any $k \leq 2$ and $\Omega(n \log n)$; $f(n)$ is not $\Omega(n^k)$ for any $k > 2$.



- **Big-Theta notation,** $f(n)$ **is** $\Theta(g(n))$ **if there exist constants** $c_1, c_2 > 0$ **and** $n_0 \geq 0$ **s.t.** $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ **for all** $n \geq n_0$**;**

  $f(n)$ **is** $\Theta(g(n))$ **iff** $f(n)$ **is** $O(g(n))$ **and** $\Omega(g(n))$**. For tight (up to a constant factor) upper and lower bounds on time/space complexity.**

  $f(n) = n^2 + 7n + 4$**,** $f(n)$ **is** $\Theta(n^2)$**;** $f(n)$ **is not** $\Theta(n^k)$ **for** $k \neq 2$**.**



- **Little-oh notation,** $f(n)$ **is** $o(g(n))$ **if for any constant** $c > 0$**, there exists a constant** $n_0 > 0$ **s.t.** $0 \leq f(n) < cg(n)$ **for all** $n \geq n_0$ **(or** $\lim_{n\to\infty} f(n)/g(n) = 0$**).**

  $f(n) = n^2 + 7n + 4$ **is** $o(n^k)$ **for** $k > 2$**.**

- **Little-omega notation,** $f(n)$ **is** $\omega(g(n))$ **if for any constant** $c > 0$**, there exists a constant** $n_0 > 0$ **s.t.** $0 \leq cg(n) < f(n)$ **for all** $n \geq n_0$ **(or** $\lim_{n\to\infty} f(n)/g(n) = \infty$**).**

  $f(n) = n^2 + 7n + 4$ **is** $\omega(n^k)$ **for** $k < 2$**.**

More on Big-Oh notation

- $f(n)$ **is** $O(f(n))$**.**
- **If** $f(n)$ **is** $O(g(n))$ **and** $c > 0$ **is a constant then** $c \cdot f(n)$ **is** $O(g(n))$**.**
- **If** $f(n)$ **is** $O(f_1(n))$ **and** $g(n)$ **is** $O(g_1(n))$**, then** $f(n) \cdot g(n)$ **is** $O(f_1(n) \cdot g_1(n))$**, and** $f(n) + g(n)$ **is** $O(\max\{f_1(n), g_1(n)\})$**.**
- **If** $f(n)$ **is** $O(f_1(n))$ **and** $f_1(n)$ **is** $O(f_2(n))$ **then** $f(n)$ **is** $O(f_2(n))$**.**
- **Big-Oh with multiple variables,** $f(m, n)$ **is** $O(g(m, n))$ **if there exist constants** $c > 0$**,** $m_0 \geq 0$ **and** $n_0 \geq 0$ **s.t.** $f(m, n) \leq c \cdot g(m, n)$ **for all** $m \geq m_0$ **and** $n \geq n_0$**.**

Name	Running time	Example
Constant time	$O(1)$	**primitive operations**
Inverse Ackermann time	$O(\alpha(n))$	**some operations in disjoint set data structure**
Iterated logarithmic time	$O(\log^* n)$	
Log-logarithmic time	$O(\log \log n)$	**some operations in a priority queue**
Logarithmic time	$O(\log n)$	**binary search in a sorted array**
Poly-logarithmic time	$\mathrm{Poly}(\log n)$	**AKS primality test on integer** n
Fractioner power time	$O(n^c), 0 < c < 1$	**factorizing integer** n
Linear time	$O(n)$	**merge two sorted lists into one sorted list**
Linearithimic time	$O(n \log n)$	**merge sort**
Quadratic time	$O(n^2)$	**addition of two** $n \times n$ **matrices**
Cubic time	$O(n^3)$	**conventional multiplication of two** $n \times n$ **matrices**
Polynomial time	$\mathrm{Poly}(n), \mathrm{Poly}(\log N)$	**AKS primality test on integer** N **of** n **bits**
Sub-exponential time	$O(2^{n^\epsilon}), 0 < \epsilon < 1$	**find maximum independent set in planar graph**
Exponential time	$O(2^{\mathrm{Poly}(n)})$	**factorizing integer** N **of** n **bits (**$N = \Omega(2^n)$**)**
Factorial time	$O(n!)$	**enumerate all permutations of** n **elements**

AKS primality test: M. Agrawal, N. Kayal and N. Saxena [2002, IIT Kanpur] gave an algorithm which, given integer N of n bits, decide if N is a prime number in $O(n^k)$ ($O((\log N)^k)$) time, k is a constant.

Factorizing integer: given integer N of n bits, find the prime factors of N. It is open whether this problem admits an algorithm with poly-time in n or NP-hard. Naive algorithm takes $O(\sqrt{N}) = O(2^{n/2})$ time, heuristic of $O(N^{1/4}) = O(2^{n/4})$ time is known.

If factorizing integer can be solved in poly-time in n (the number of bits of the integer) then many public-key encryption schemes including RSA can be cracked easily.

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

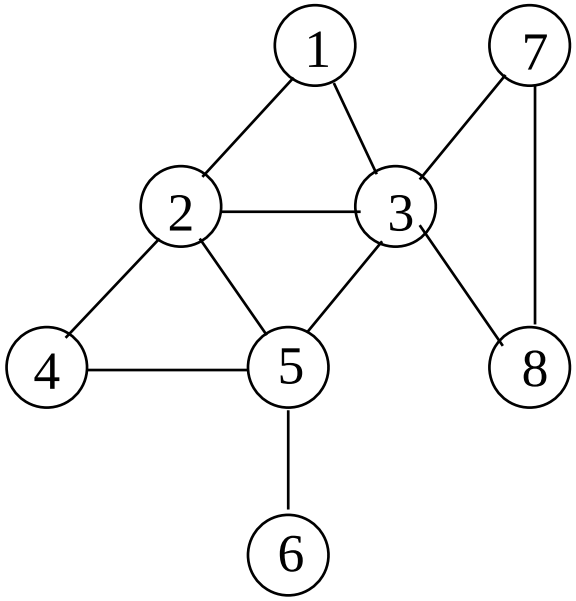
Running time of G-S Algorithm

Input M**, rank lists of** M**,** W **and rank lists of** W**;** /* $O(n^2)$ */
Initially, $S := \emptyset$**;** $W_m = W$ **for every** $m \in M$**;** /* $O(n)$ */
while $\exists$ **unpaired** $m \in M$ **do** /* **check condition,** $O(1)$**; while loop iterates** $O(n^2)$ **times** */
  Choose an unpaired m**; Let** $w \in W_m$ **with the largest** $f_m(w)$**;** /* $O(1)$ */
  if w **is unpaired then** /* $O(1)$ */
    $S := S \cup \{(m, w)\}$ /* $O(1)$ */
  else
    Let (m', w) **be the pair in** S**;** /* $O(1)$ */
    if $r_w(m) > r_w(m')$ **then** /* $O(1)$ */
      remove (m', w) **from** S**;** $S := S \cup \{(m, w)\}$ /* $O(1)$ */
    else
      Remove w **from** W_m**;** /* $O(1)$ */
    end if
  end if
end while
Return S**;** /* $O(n)$ */
Running time of G-S Alg is $O(n^2)$**.**

Graphs

Undirected graphs

- **Graph $G(V, E)$, V is a set of nodes (vertices), E is a set of edges, edge $\{u, v\}$ connects nodes u and v and called an edge between u and v.**
- **Graph size parameters, $n = |V|, m = |E|$.**
- **If there are more than one edge between u and v, these edges are called multiple edges. An edge $\{u, u\}$ connects node u itself is called a self loop.**
- **Simple graph, does not have multiple edge nor self loop.**
- **For an edge $e = \{u, v\}$, u, v are the end nodes of e, u is adjacent to v (v is adjacent to u), u and v are neighbors, e is incident to u and v.**
- **Node degree $\deg(u)$ of node u, number of edges incident to u.**
- **Graph H is a subgraph of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.**



V={1,2,3,4,5,6,7,8}

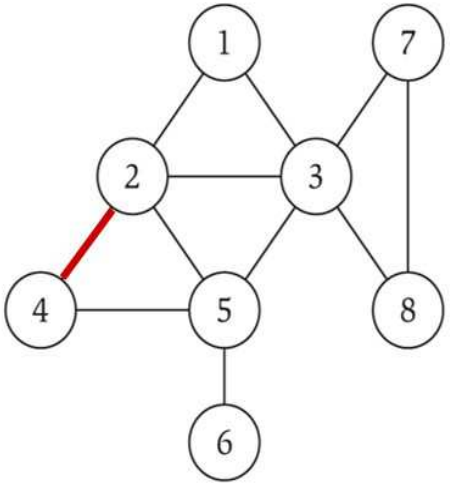
E={ {1,2},{1,3},{2,3},{2,4},{2,5}
{3,5},{3,7},{3,8},{4,5},{5,6},{7,8} }

n=8, m=11

A simple undirected graph

Graph representation: Adjacency matrix

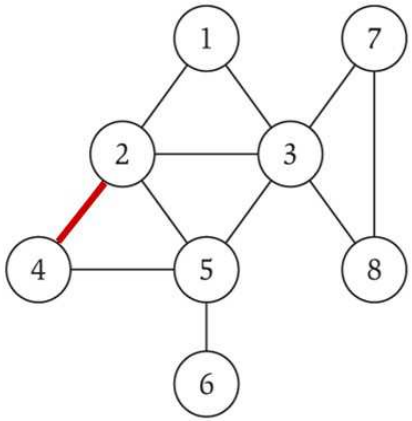
- **For an edge $\{u, v\}$ in G, u is adjacent to v and v is adjacent to u; edge $\{u, v\}$ is incident to u and v.**
- **Adjacency matrix, an $n \times n$ array A with $A[u, v] = 1$ if u is adjacent to v.**
- **Space, $O(n^2)$.**
- **Time, $O(1)$ to check if $\{u, v\}$ is an edge; $O(n^2)$ time to identify all edges.**

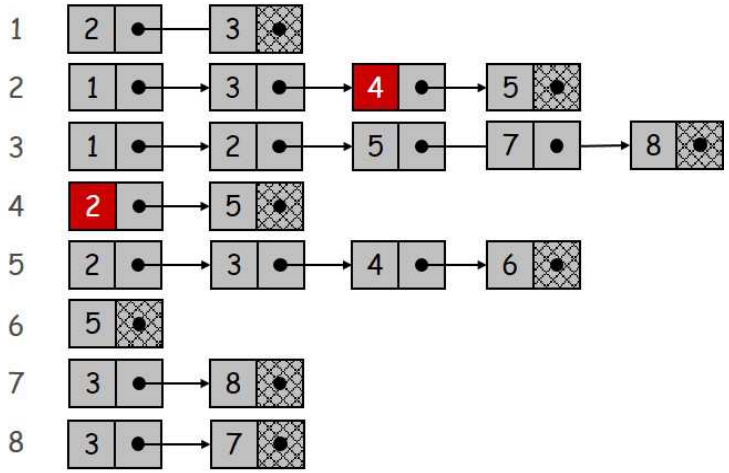


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Graph representation: Adjacency list

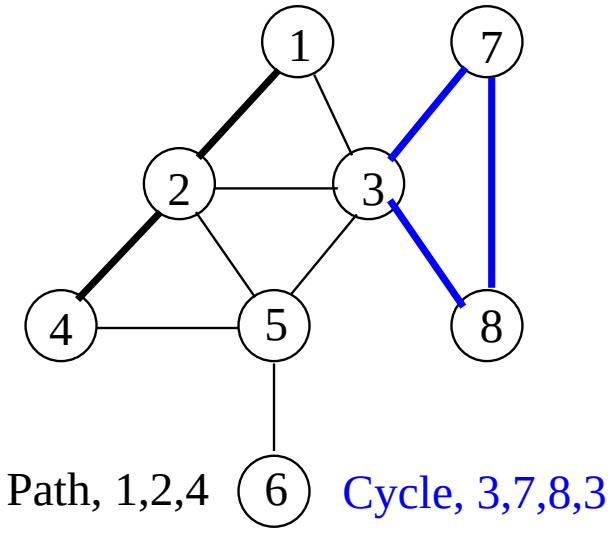
- **Adjacency Lists: for every node u of G, all nodes adjacent to u are put in a list.**
- **Space, $O(m+n)$.**
- **Time, $O(\deg(u))$ to check if $\{u,v\}$ is an edge; $O(m+n)$ to identify all edges.**





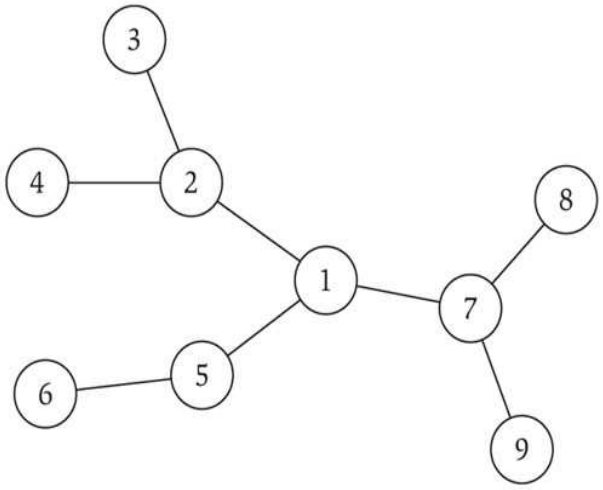
Path and cycle

- **Path in G, a sequence of nodes $v_1, v_2, .., v_k$ s.t. each $\{v_{i-1}, v_i\}$, $1 < i \leq k$, is an edge of G.**
- **Simple path, a path with all nodes distinct.**
- **Cycle, a path $v_1, v_2, .., v_k$ with $v_1 = v_k$ and $v_1, .., v_{k-1}$ distinct.**
- **For graph G, a cycle is defined having at least three edges.**
- **Length of path (cycle), number of edges in the path (cycle).**

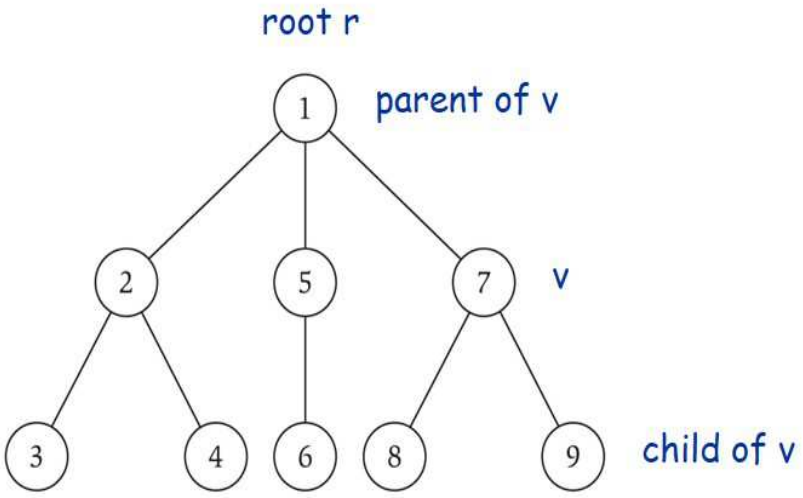


Trees

- **A graph G is connected if for any nodes u, v in G, there is a path between u and v.**
- **G is a tree if G is connected and does not have a cycle.**
- **Rooted tree, a tree with one node identified as the root.**



a tree

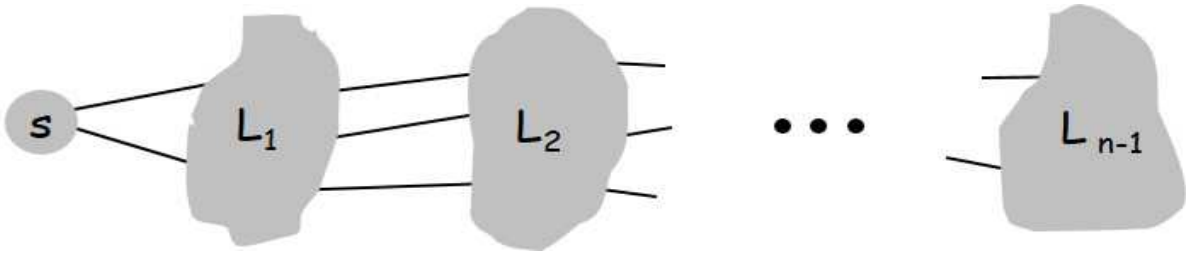


the same tree, rooted at 1

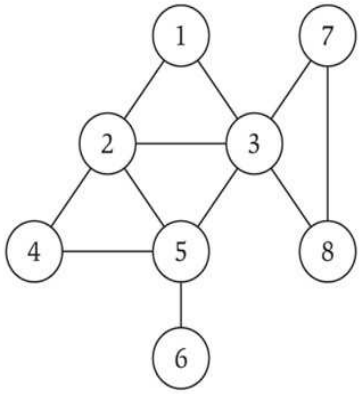
Graph connectivity and traversal

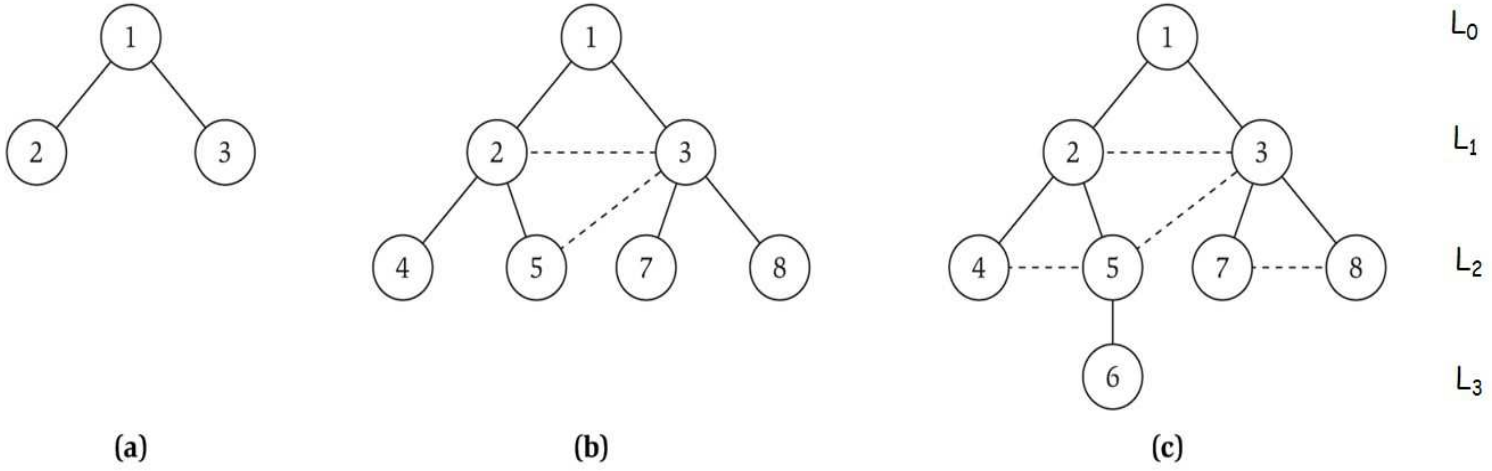
- $s-t$ **connectivity, given nodes** s **and** t**, is there a path between** s **and** t**.**
- **Graph traversal, breadth first search (BFS) and depth first search (DFS).**
- **BFS from a node** s**:**
  - $L_0 = \{s\}$**.**
  - $L_1 = \{$**all nodes adjacent to** $s\}$**.**
  - **For** $i > 1$**,**

    $L_i = \{$**nodes not in** $L_0 \cup L_1 \cup .. \cup L_{i-1}$ **and adjacent to a node in** $L_{i-1}\}$**.**
- **For each** i**,** $L_i = \{$**nodes with the minimum path length exactly** i **from** $s\}$**.**



BFS example

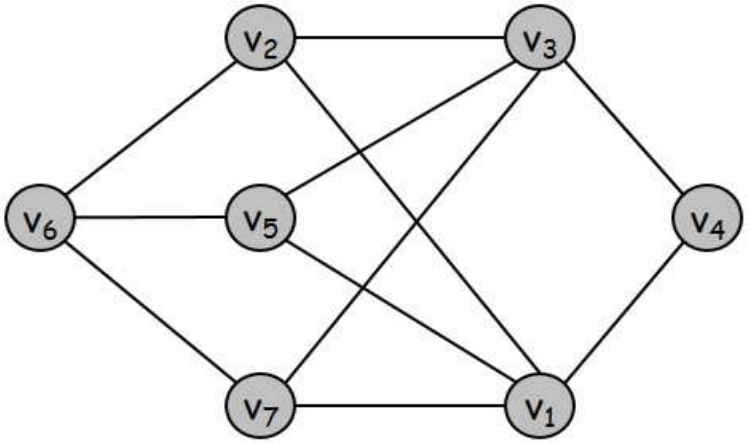




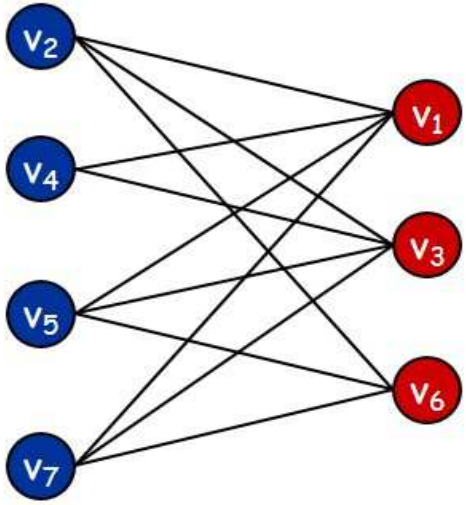
19

Bipartite graphs

- **Bipartite G, nodes of G can be partitioned into two subsets V_1 and V_2 s.t. for every edge e of G, one end node of e is in V_1 and the other is in V_2.**
- **G is bipartite iff G does not have an odd-length cycle.**



a bipartite graph G



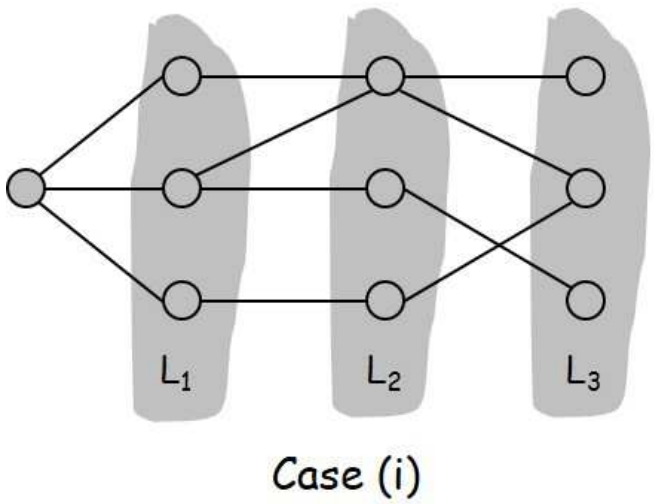
another drawing of G

Test bipartiteness

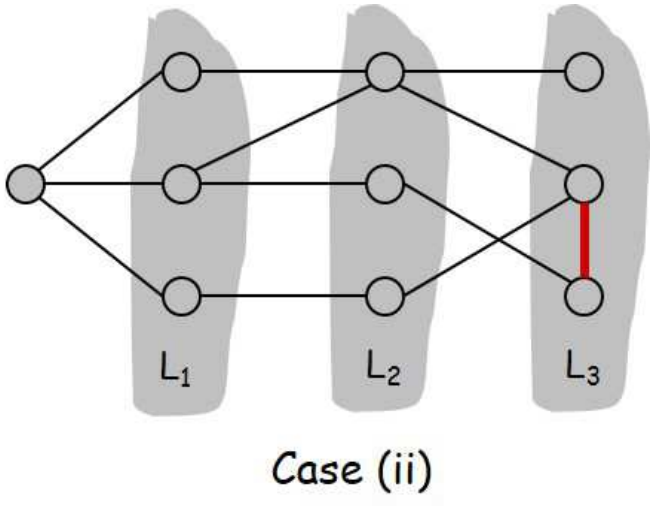
- **Use BFS on G to compute $L_0, .., L_k$.**

  (i) If no edge $\{u, v\}$ for $u, v \in L_i$ for any i, G is bipartite.

  (ii) If there is an edge $\{u, v\}$ for u, v in some L_i, G is not bipartite.



Case (i)



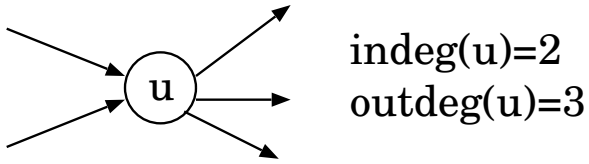
Case (ii)

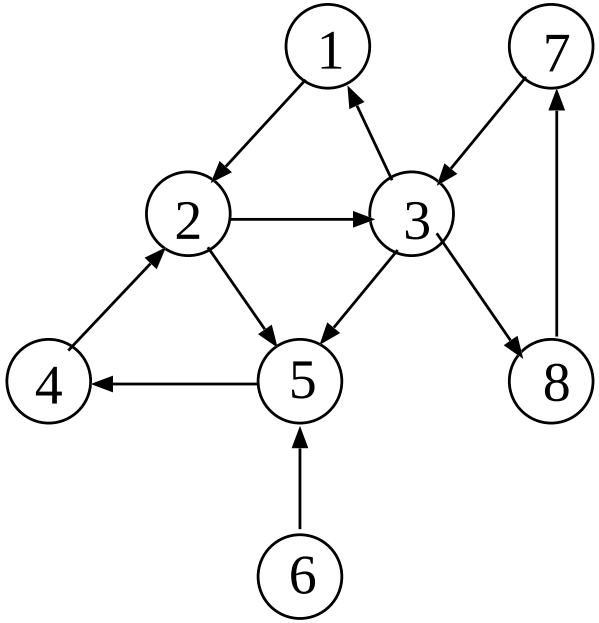
If G is bipartite, $V_1 = \cup_{\text{even } i} L_i$ and $V_2 = \cup_{\text{odd } i} L_i$.

Directed Graphs

- **Digraph $G(V, E)$, V is a set of nodes (vertices), E is a set of arcs (directed edges), arc (u, v) is from node u to node v.**
- **Graph size parameters, $n = |V|$, $m = |E|$.**
- **If there are more than one arc from u to v, these arcs are called multiple arcs. An arc (u, u) from u to itself is called a self loop.**
- **Simple digraph, does not have multiple arcs nor self loop.**
- **Indegree $\mathrm{indeg}(u)$ of node u, number of arcs to u;**

  outdegree $\mathrm{outdeg}(u)$ of node u, number of arcs from u.





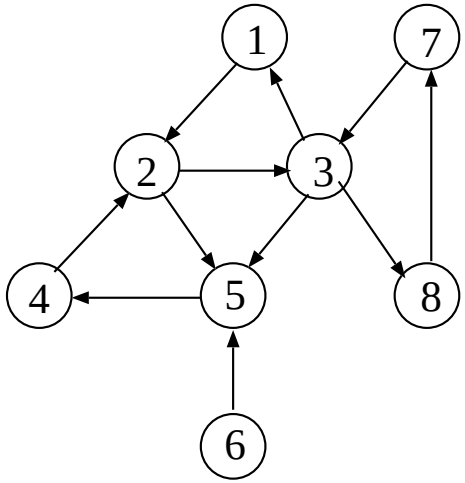
V={1,2,3,4,5,6,7,8}

A={ (1,2),(3,1),(2,3),(4,2),(2,5)
(3,5),(7,3),(3,8),(5,4),(6,5),(8,7) }

n=8, m=11

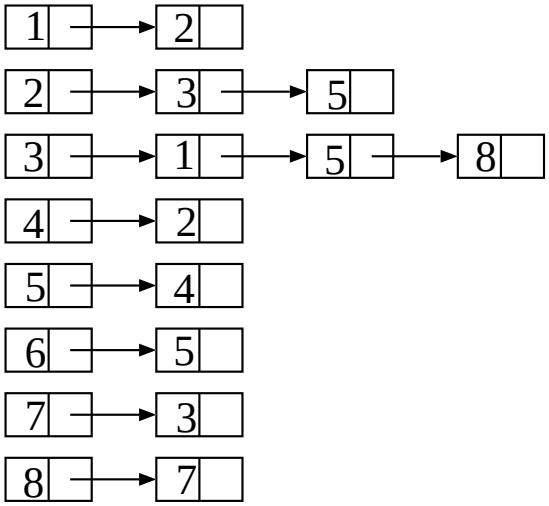
A simple directed graph

Adjacency matrix and list for digraphs



	1	2	3	4	5	6	7	8
1	0	1	0	0	0	0	0	0
2	0	0	1	0	1	0	0	0
3	1	0	0	0	1	0	0	1
4	0	1	0	0	0	0	0	0
5	0	0	0	1	0	0	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	0
8	0	0	0	0	0	0	1	0

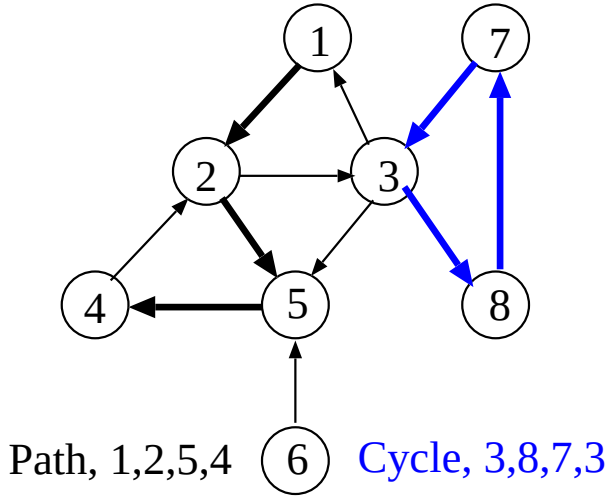
Adjacency matrix



Adjacency list

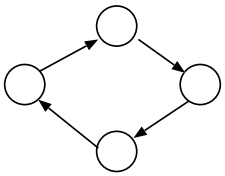
Directed path and cycle

- **Path in digraph G, a sequence of nodes $v_1, v_2, .., v_k$ s.t. each (v_{i-1}, v_i), $1 < i \leq k$, is an arc of G.**
- **Simple path, a path with all nodes distinct.**
- **Cycle, a path $v_1, v_2, .., v_k$ with $v_1 = v_k$ and $v_1, .., v_{k-1}$ distinct.**
- **Length of path (cycle), number of edges in the path (cycle).**

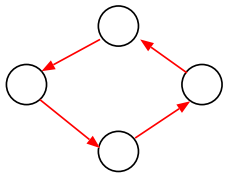


Digraph search

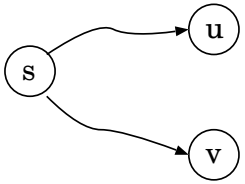
- **Node t is reachable from node s in digraph G if there is a path from s to t.**
- **Nodes s and t are mutual reachable if each of them is reachable from the other.**
- **Directed reachability, find all nodes reachable from s.**
- **G is strongly connected if every pair of nodes is mutually reachable.**
- **If G is strongly connected can be checked in $O(m+n)$ time: Run BFS from a node s of G; reverse the orientation of every arc of G to get graph $\bar{G}$; run BFS from s of $\bar{G}$; return true iff all nodes are reachable from s in both BFS executions.**



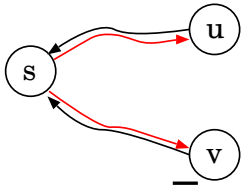
G



$\overline{G}$



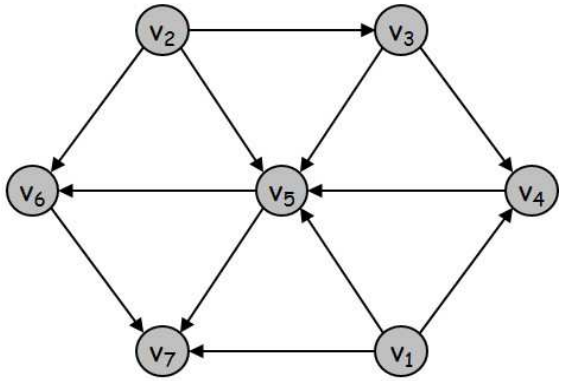
1st BFS on G



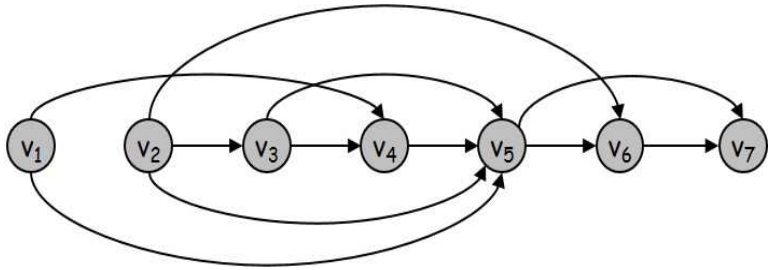
2nd BFS on $\overline{G}$, a s-->u path in $\overline{G}$ implies a u-->s path in G

Directed acyclic graph (DAG)

- **A DAG is a directed graph with no cycle.**
- **A topological order of directed graph G is an ordering of nodes of G as $v_1, .., v_n$ s.t. for every arc (v_i, v_j) of G, $i < j$.**
- **G has a topological order iff G is a DAG.**
- **Given a DAG G, a topological order of G can be found in $O(m + n)$ time.**



a DAG



A topological order

Topological-Order(G) **/*input DAG** G**, output topological order of** G

find a node v **of** G **with** $\text{indeg}(v) = 0$**;**

order v **first; delete** v **from** G

Topological-Order$(G \setminus \{v\})$ **and append this order after** v**;**

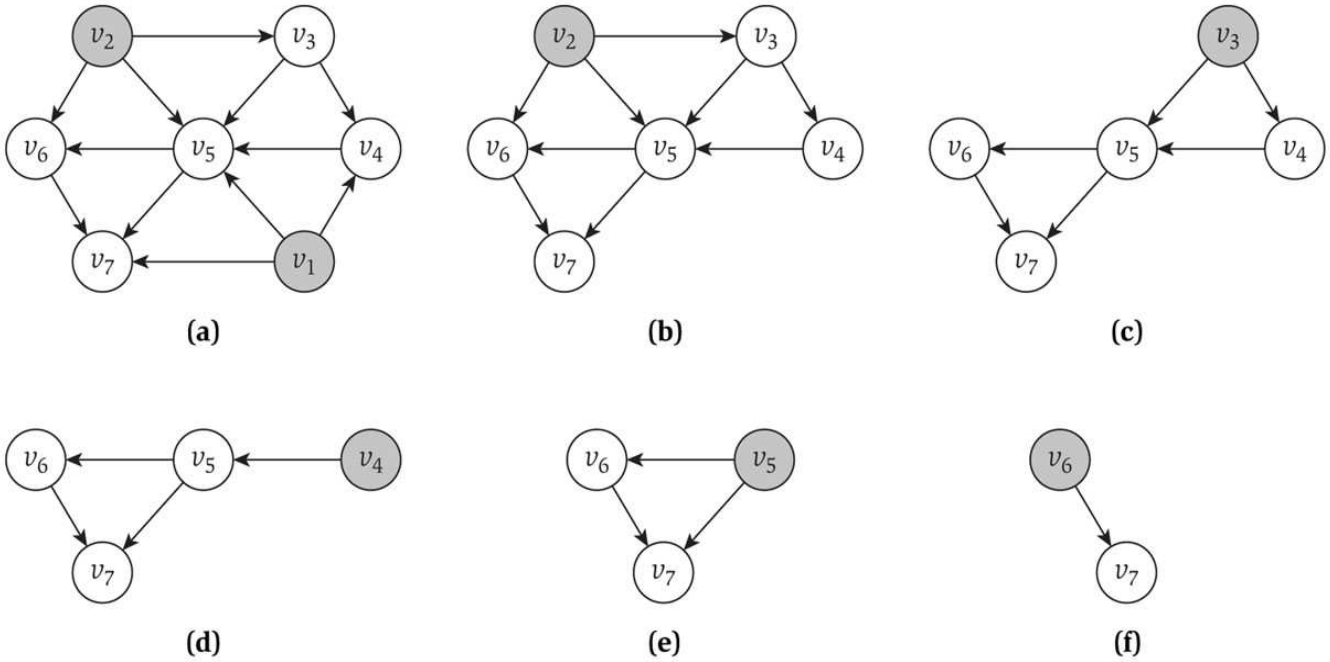


Figure 3.8 Starting from the graph in Figure 3.7, nodes are deleted one by one so as to be added to a topological ordering. The shaded nodes are those with no incoming edges; note that there is always at least one such edge at every stage of the algorithm's execution.

Formulate Problems by Graphs

- **Given a graph G, a matching of G is a subset M of edges in G s.t. no edges of M share a common end node.**

  The stable matching problem can be modelled as a matching problem in bipartite graphs. This problem can be solved in polynomial time.

- **Assume you have n friends, some of your friend may dislike some others, and you want to invite as many of your friends as possible s.t. no two friends invited dislike each other.**

  We can formulate this problem as a problem in graph G: Each node G denotes a friend and there is an edge between two nodes if they dislike each other. An independent set S of G is a subset of nodes s.t. there is no edge between any two nodes in S. The maximum independent set problem is to find an independent set of maximum size. This problem is NP-complete.

- **Consider a two players game problem: Two companies A and B want to open similar stores (e.g., provide fast food) in n zones. Each zone has a value of potential profit for having a store. There are zone regulations that each zone can have at most one store and if some zone has a store, any zone adjacent to the zone can not have any store. A and B alternate in choosing a zone to open a store, starting from A. If each company can reach a predefined profit target?**
- **The problem is known as Competitive Facility Location problem and can be formulated as a problem in a graph G: A node of G denotes a zone and is assigned a positive weight (profit). There is an edge between two nodes if the two zones are adjacent. A and B alternate in choosing a node of G, starting from A, to find an independent set of G to maximize the total profit for itself. This problem is PSPACE-complete.**

Zone 1	Zone 2	Zone 3	Zone 4	Zone 5	Zone 6
profit	profit	profit	profit	profit	profit
10	5	6	9	7	8