Analysis of Algorithms
CS 477/677

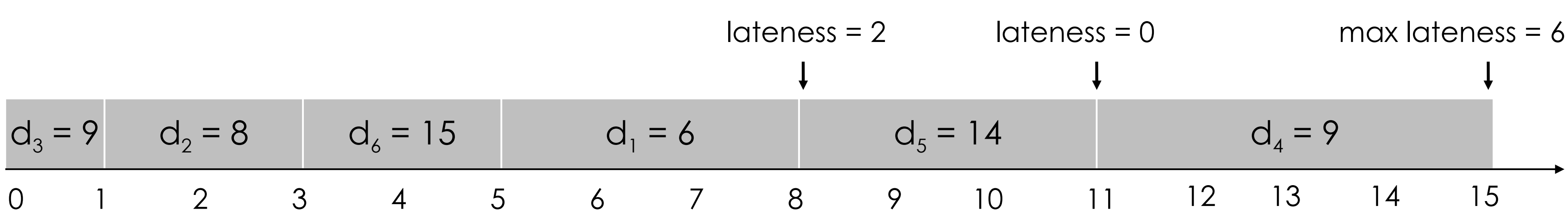
Instructor: Monica Nicolescu

Lecture 22

Scheduling to Minimizing Lateness

- Single resource processes one job at a time
- Job j requires t_j units of processing time, is due at time d_j
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$
- Lateness: $\ell_j = \max\{0, f_j - d_j\}$
- Goal: schedule all jobs to minimize **maximum** lateness $L = \max \ell_j$
- Example:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Greedy Algorithms

- Greedy strategy: consider jobs in some order
  - **[Shortest processing time first]** Consider jobs in ascending order of processing time t_j

counterexample

	1	2
t_j	1	10
d_j	100	10

Choosing t_1 first: $l_2 = 1$
Choosing t_2 first: $l_2 = l_1 = 0$

  - **[Smallest slack]** Consider jobs in ascending order of slack $d_j - t_j$

counterexample

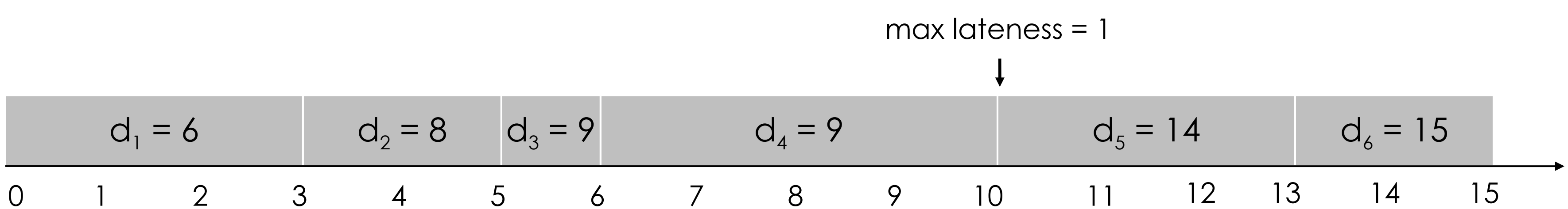
	1	2
t_j	1	10
d_j	2	10

Choosing t_2 first: $l_1 = 9$
Choosing t_1 first: $l_1 = 0$ and $l_2 = 1$

Greedy Algorithm

- Greedy choice: earliest deadline first

```
Sort n jobs by deadline so that d_1 < d_2 <… < d_n

t = 0
for j = 1 to n
   Assign job j to interval [t, t + t_j]
   s_j = t, f_j = t + t_j
   t = t + t_j
output intervals [s_j, f_j]
```



Minimizing Lateness: No Idle Time

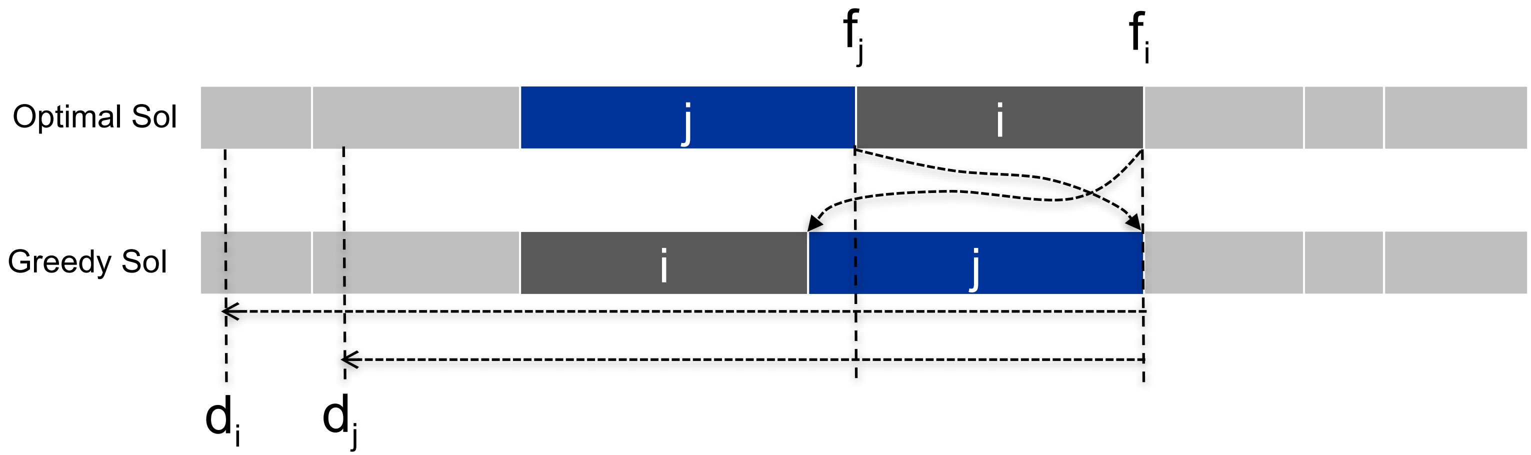
- Observation: The greedy schedule has no idle time
- Observation: There exists an optimal schedule with no **idle time**

Minimizing Lateness: Inversions

- An inversion in schedule S is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i

- Observation: greedy schedule has no inversions

Greedy Choice Property



- Optimal solution: $d_i < d_j$ but j scheduled before i
- Greedy solution: i scheduled before j
  - Job i finishes sooner, no increase in latency

Lateness(Job j)$_{GREEDY}$ = $f_i - d_j$

$\leq$ ➔ No increase in latency

Lateness(Job i)$_{OPT}$ = $f_i - d_i$

Greedy Analysis Strategies

- Exchange argument
  - Gradually transform any solution to the one found by the greedy algorithm without hurting its quality
- Structural
  - Discover a simple "structural" bound asserting that every possible solution must have a certain value, then show that your algorithm always achieves this bound
- Greedy algorithm stays ahead
  - Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's

Coin Changing

- Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins
- Ex: 34¢

- Ex: $2.89

Greedy Algorithm

- Greedy strategy: at each iteration, add coin of the largest value that does not take us past the amount to be paid

```
Sort coins denominations by value: c_1 < c_2 < … < c_n.

  coins selected
S = {}
while (x > 0) {
   let k be largest integer such that c_k <= x
   if (k = 0)
      return "no solution found"
   x = x - c_k
   S = S  U  {k}
}
return S
```

Greedy Choice Property

- Algorithm is optimal for U.S. coinage: 1, 5, 10, 25, 100

  Change = D * 100 + Q * 25 + D * 10 + N * 5 + P

  - Consider optimal way to change $c_k \le x < c_{k+1}$: greedy takes coin k
  - We claim that any optimal solution must also take coin k
  - If not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to add up to x
  - Problem reduces to coin-changing x - c_k cents, which, by induction, is optimally solved by greedy algorithm

Greedy Choice Property

- Algorithm is optimal for U.S. coinage: 1, 5, 10, 25, 100

Change = Dl * 100 + Q * 25 + D * 10 + N * 5 + P

- Optimal solution: Dl Q D N P
- Greedy solution: Dl' Q' D' N' P'

1. Value < 5
   - Both optimal and greedy use the same # of coins
2. 10 (D) > Value > 5 (N)
   - Greedy uses one N and then pennies after that
   - If OPT does not use N, then it should use pennies for the entire amount => could replace 5 P for 1 N

Greedy Choice Property

Change = Dl * 100 + Q * 25 + D * 10 + N * 5 + P

- Optimal solution: Dl Q D N P
- Greedy solution: Dl’ Q’ D’ N’ P’

3. 25 (Q) > Value > 10 (D)
   - Greedy uses dimes (D’s)
   - If OPT does not use D’s, it needs to use either 2 coins (2 N), or 6 coins (1 N and 5 P) or 10 coins (10 P) to cover 10 cents
   - Could replace those with 1 D for a better solution

Greedy Choice Property

Change = Dl * 100 + Q * 25 + D * 10 + N * 5 + P

- Optimal solution: Dl Q D N P
- Greedy solution: Dl' Q' D' N' P'

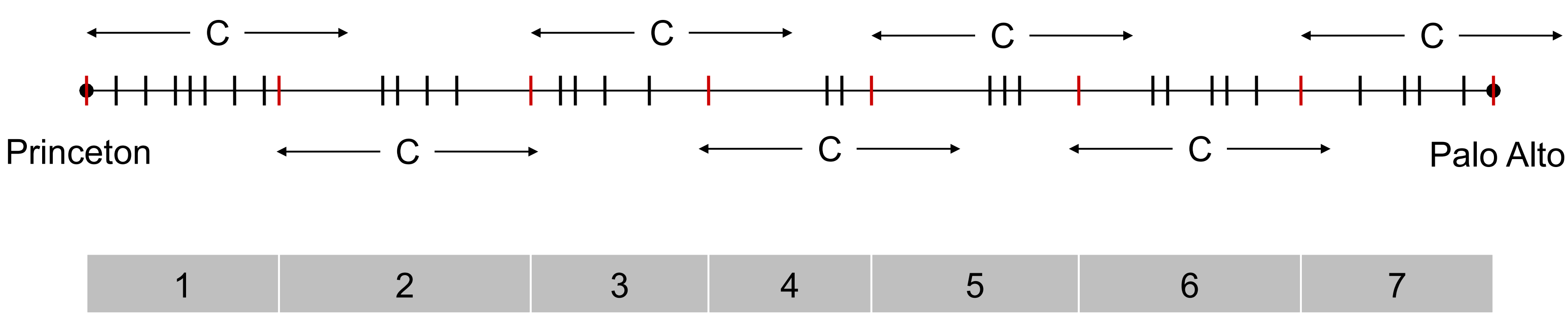
4. 100 (Dl) > Value > 25 (Q)
   - Greedy picks at least one quarter (Q), OPT does not
   - If OPT has no Ds: take all the Ns and Ps and replace 25 cents into one quarter (Q)
   - If OPT has 2 or fewer dimes: it uses at least 3 coins to cover one quarter, so we can replace 25 cents with 1 Q
   - If OPT has 3 or more dimes (e.g., 40 cents: with 4 Ds): take the first 3 Ds and replace them with 1 Q and 1 N

Coin-Changing
US Postal Denominations

- Observation: greedy algorithm is sub-optimal for US postal denominations:
  - \$.01, .02, .03, .04, .05, .10, .20, .32, .40, .44, .50, .64, .65, .75, .79, .80, .85, .98
  - \$1, \$1.05, \$2, \$4.95, \$5, \$5.15, \$18.30, \$18.95
- Counterexample: 160¢
  - Greedy: 105, 50, 5
  - Optimal: 80, 80

Selecting Breakpoints

- Road trip from Princeton to Palo Alto along fixed route
- Refueling stations at certain points along the way (red marks)
- Fuel capacity = C
- Goal:
  - makes as few refueling stops as possible
- Greedy strategy:
  - go as far as you can before refueling



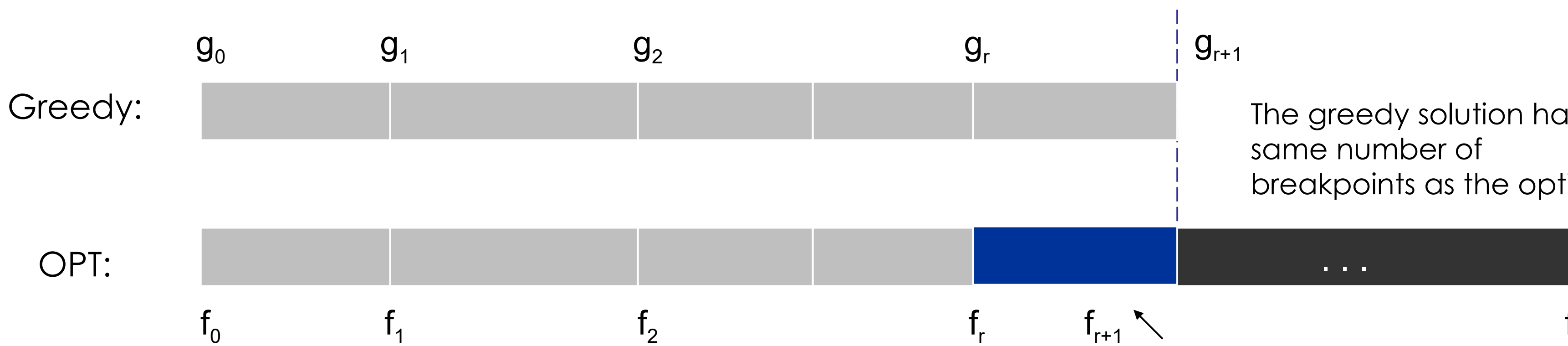
Greedy Algorithm

```
Sort breakpoints so that: 0 = b0 < b1 < b2 < ... < bn = L

S = {0}          ←  breakpoints selected
x = 0            ←  current location

while (x < bn)
   let p be largest integer such that bp <= x + C
   if (bp = x)
      return "no solution"
   x = bp
   S = S  U {p}
return S
```

- Implementation: O(n log n)
  - Use binary search to select each breakpoint p

Greedy Choice Property

- Let $0 = g_0 < g_1 < \ . \ . \ . < g_p = L$ denote set of breakpoints chosen by the greedy
- Let $0 = f_0 < f_1 < \ . \ . \ . < f_q = L$ denote set of breakpoints in an optimal solution with $f_0 = g_0, f_1 = g_1, \ . \ . \ . \ . \ , f_r = g_r$
- Note: $g_{r+1} > f_{r+1}$ by greedy choice of algorithm



The greedy solution has the same number of breakpoints as the optimal

why doesn't optimal solution drive a little further?

Problem – Buying Licenses

- Your company needs to buy licenses for **n** pieces of software
- Licenses can be bought only one per month
- Each license currently sells for \$100, but becomes more expensive each month
  - The price increases by a factor $r_j > 1$ each month
  - License j will cost $100 * r_j^t$ if bought t months from now
  - $r_i < r_j$ for license $i < j$
- In which order should the company buy the licenses, to minimize the amount of money spent?

Solution

- Greedy choice:
  - Buy licenses in decreasing order of rate r_j
  - $r_1 > r_2 > r_3 \ldots$
- Proof of greedy choice property
  - Optimal solution: …. $r_i\ r_j$….. $r_i < r_j$
  - Greedy solution: …. $r_j\ r_i$…..
  - Cost by optimal solution: $100 * r_i^t + 100 * r_j^{t+1}$
  - Cost by greedy solution: $100 * r_j^t + 100 * r_i^{t+1}$

$CG - CO = 100 * (r_j^t + r_i^{t+1} - r_i^t - r_j^{t+1}) < 0$

$r_i^{t+1} - r_i^t < r_j^{t+1} - r_j^t$

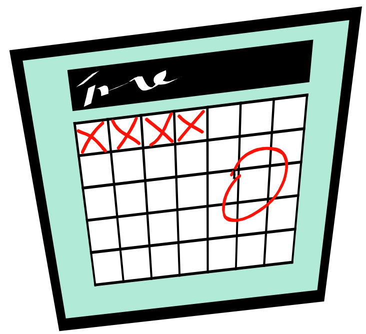
$r_i^t(r_i - 1) < r_j^t(r_j - 1)$

OK! (because $r_i < r_j$)

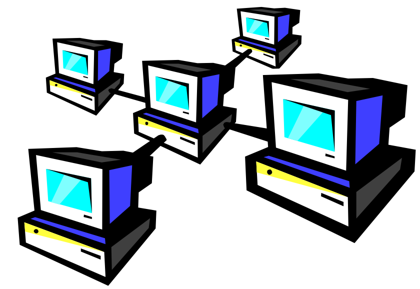
Graphs

- Applications that involve not only a set of items, but also the connections between them

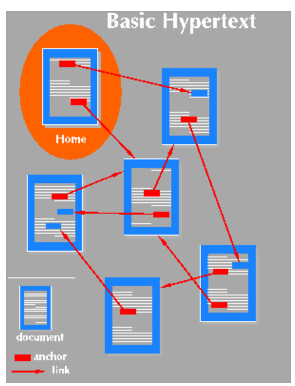
Maps



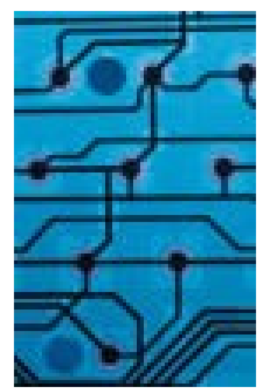
Schedules



Computer networks



Hypertext



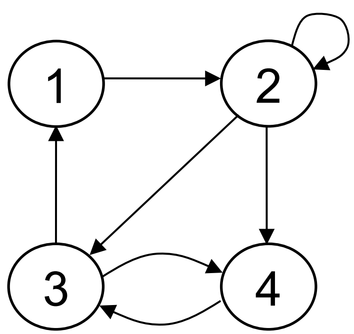
Circuits

Graphs - Background

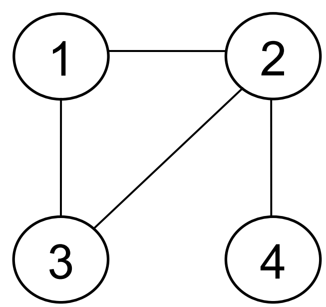
Graphs = a set of nodes (vertices) with edges (links) between them.

Notations:

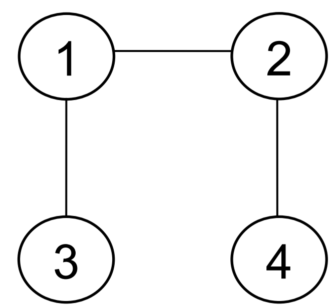
- G = (V, E) - graph
- V = set of vertices (size of V = n)
- E = set of edges (size of E = m)



Directed graph



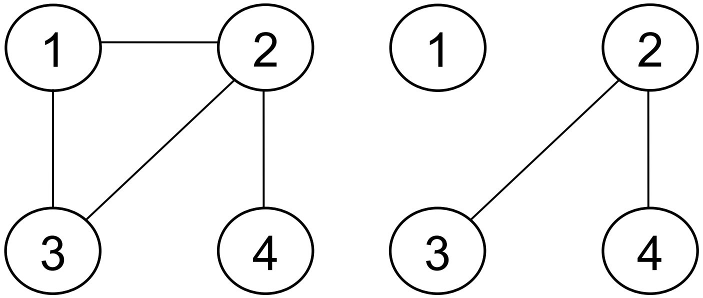
Undirected graph



Acyclic graph

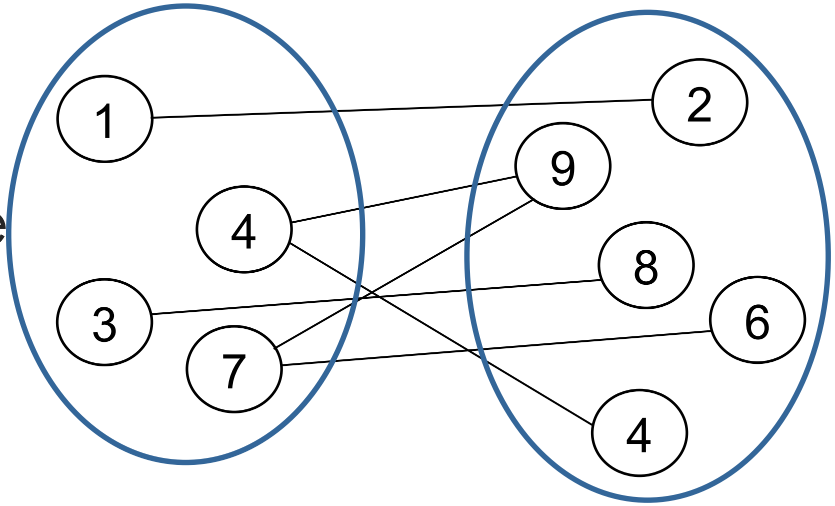
Other Types of Graphs

- A graph is **connected** if there is a path between every two vertices



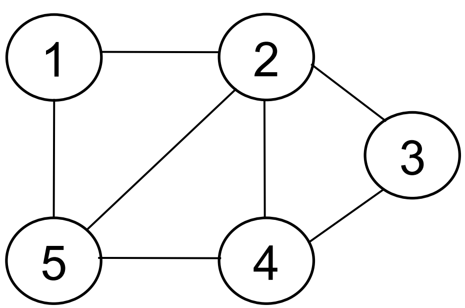
Connected Not connected

- A **bipartite graph** is an undirected graph G = (V, E) in which $V = V_1 + V_2$ and there are edges only between vertices in V_1 and V_2

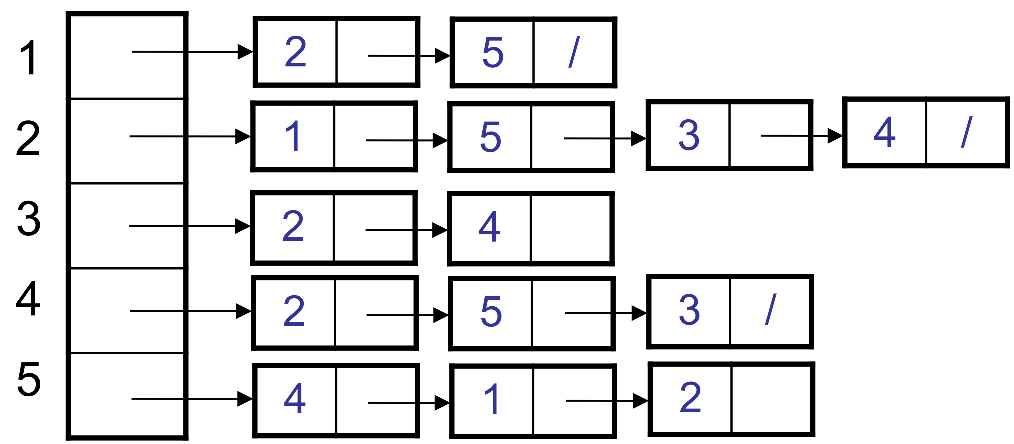


Graph Representation

- **Adjacency list representation** of G = (V, E)
  - An array of n lists, one for each vertex in V
  - Each list **Adj[u]** contains all the vertices v such that there is an edge between **u** and **v**
    - **Adj[u]** contains the vertices adjacent to **u** (in arbitrary order)
  - Can be used for both directed and undirected graphs

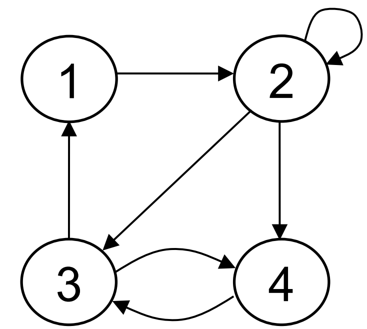


Undirected graph

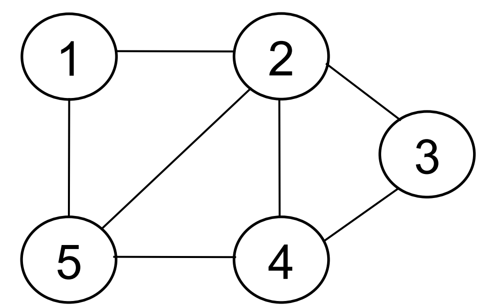


Properties of Adjacency List Representation

- Sum of the lengths of all the adjacency lists
  - Directed graph: size of E (m)
    - Edge (u, v) appears only once in u's list
  - Undirected graph: 2* size of E (2E)
    - u and v appear in each other's adjacency lists: edge (u, v) appears twice



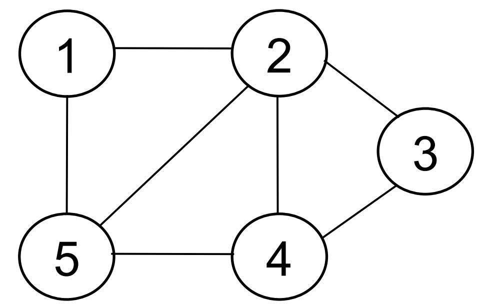
Directed graph



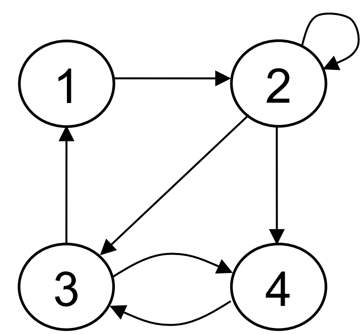
Undirected graph

Properties of Adjacency List Representation

- Memory required
  - $\Theta(m+n)$
- Preferred when
  - the graph is sparse: $m << n^2$
- Disadvantage
  - no quick way to determine whether there is an edge between node u and v
- Time to list all vertices adjacent to u:
  - $\Theta(\text{degree}(u))$
- Time to determine if (u, v) exists:
  - $O(\text{degree}(u))$



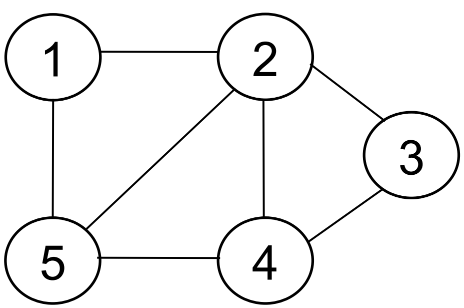
Undirected graph



Directed graph

Graph Representation

- **Adjacency matrix representation** of G = (V, E)
  - Assume vertices are numbered 1, 2, … n
  - The representation consists of a matrix A_{nxn}
  - $a_{ij} = \begin{cases} 1 & \text{if (i, j) belongs to E} \\ 0 & \text{otherwise} \end{cases}$



Undirected graph

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

For undirected graphs matrix A is symmetric:

$a_{ij} = a_{ji}$

$A = A^T$

Properties of Adjacency Matrix Representation

- Memory required
  - $\Theta(n^2)$, independent on the number of edges in G
- Preferred when
  - The graph is dense: m is close to n^2
  - We need to quickly determine if there is an edge between two vertices
- Time to list all vertices adjacent to u:
  - $\Theta(n)$
- Time to determine if (u, v) belongs to E:
  - $\Theta(1)$

Weighted Graphs

- **Weighted graphs** = graphs for which each edge has an associated weight **w(u, v)**

  w: *E* -> R, weight function

- Storing the weights of a graph
  - Adjacency list:
    - Store **w(u,v)** along with vertex **v** in **u**'s adjacency list
  - Adjacency matrix:
    - Store **w(u, v)** at location **(u, v)** in the matrix

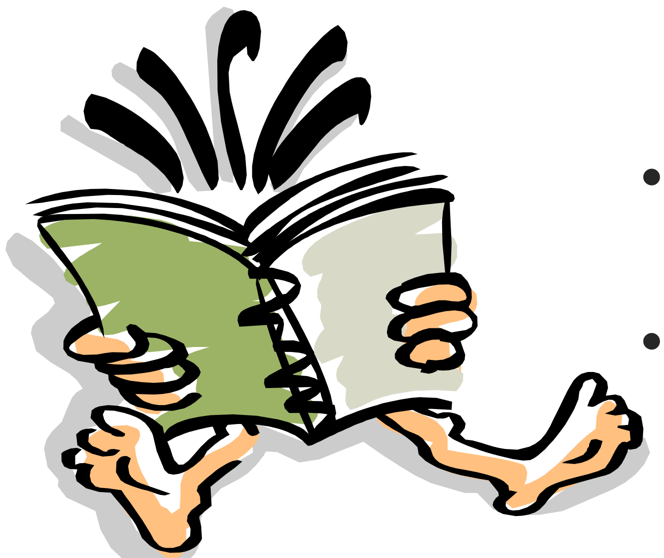
Searching in a Graph

- **Graph searching** = systematically follow the edges of the graph so as to visit the vertices of the graph
- Two basic graph searching algorithms:
  - Breadth-first search
  - Depth-first search
- The difference between them is in the order in which they explore the unvisited edges of the graph
- Graph algorithms are typically elaborations of the basic graph-searching algorithms

Breadth-First Search (BFS)

- **Input**:
  - A graph $G = (V, E)$ (directed or undirected)
  - A **source** vertex s from V
- **Goal**:
  - Explore the edges of G to "discover" every vertex reachable from s, taking the ones closest to s first
- **Output**:
  - $d[v]$ = distance (smallest # of edges) from s to v, for all v from V
  - A "breadth-first tree" rooted at s that contains all reachable vertices

Readings



- For this lecture
  - Chapter 15
- Coming next
  - Chapter 20