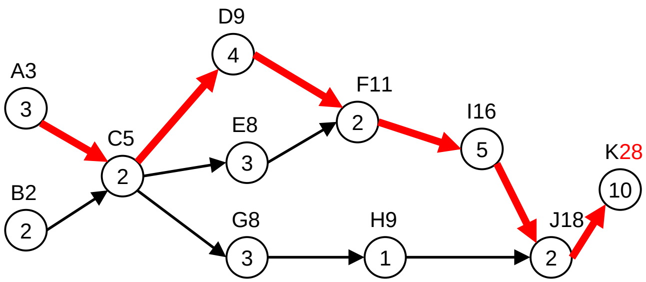
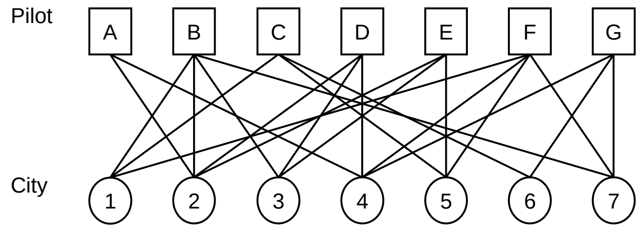
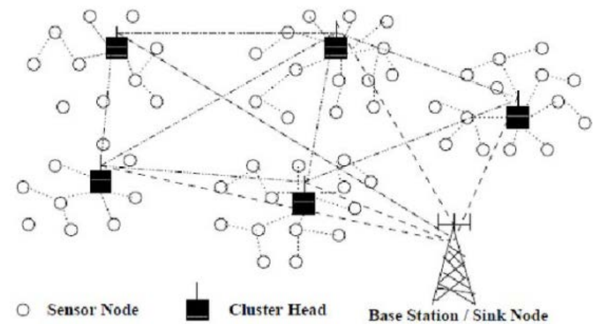
Graphs

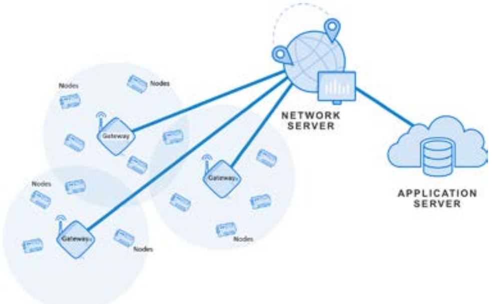
Contents:

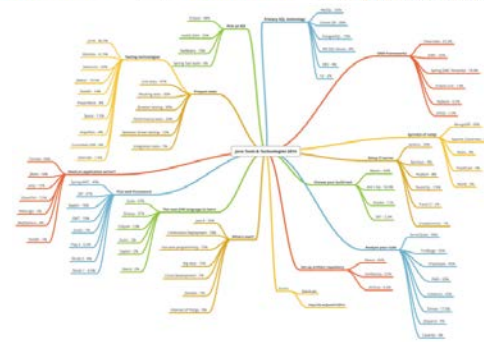
Examples of Graphs





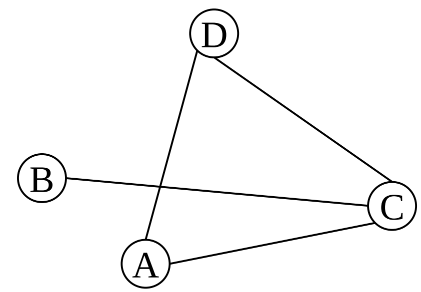






Graphs and their representations

- A graph is a nonempty finite set V along with a set E of 2-element subsets of V
  - V: vertices (vertex)   (or vertex)
  - E: edges (undirected)   (or Arcs: directed)
- A graph can be described either by the use of sets, or diagrams
- Edge e={u,v} means
  - e joins vertices u and v
  - u and v are adjacent
  - e is incident with u (and vice versa)



- Degree of v: # edges incident with v

Subgraph & Complete graph

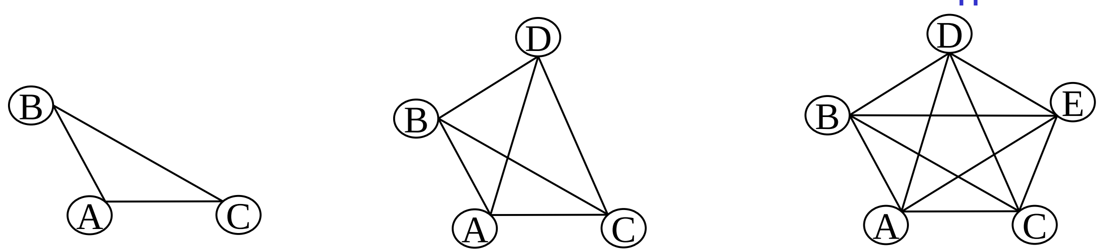
Thm 4.1

In a graph, the sum of the degrees of the vertices equals twice the # of edges

Labeled graph: G=(V,E) with n vertices labeled $v_1, v_2, \ldots v_n$

Subgraph $G_1=(V_1,E_1)$ of G=(V,E), if $\varnothing \neq V_1 \subseteq V$, and $E_1 \subseteq E$ where each edge in E_1 joins vertices in V_1.

Complete graph K_n: a graph of n vertices, each vertex is joined to every other vertex, i.e. each vertex in K_n has degree n-1



K_3 K_4 K_5

Adjacency Matrix & Adjacency List

Adjacency matrix of G: A(G), an n x n matrix where
(i,j) entry is 1, if there is an edge from v_i to v_j
(i,j) entry is 0, if there is NO edge from v_i to v_j

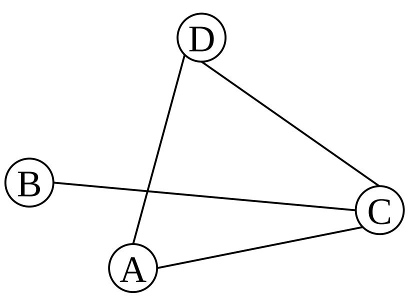
- easy to represent G, but need n^2 storage, inefficient for sparse graph

Thm 4.2
The sum of the entries in row i of A(G) is the degree of the vertex v_i

Adjacency List of G:

- List each vertex followed by the vertices adjacent to it
- Corresponds to the nonzero entries in A(G) ➔ 2m storage

e.g. n=|V|=4, m=|E|=4



$$A(G) = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Adjacency List:
A: C,D
B: C
C: A,B,D
D: A,C

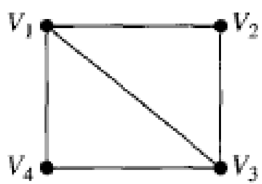
Exercises 4.1 (Adjacency)

Q.18: draw the graph with V={1,2,…,10},
E={(x,y): x, y in V, x≠y, x divides y or vice versa}

Q.22: Show that there are an even # of vertices with odd degree in any graph

Q.24: Can there be a graph with n=8, m=29?

Q28:



Adj M:

Adj L:

Q36: Qualified Adj M?

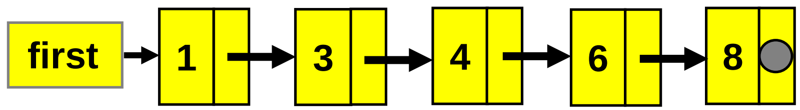
$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Efficiency: Array vs. Linked List

An array of size 10 that contains only 5 items

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]
1	0	1	1	0	1	0	1	0	0

A singly linked list of 5 items

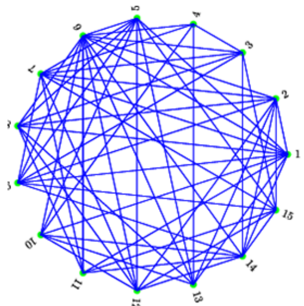


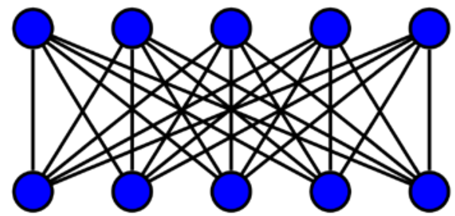
Major operations on k items	Array (size n)	Linked List (size k)
• Query: existence of 1 item		
• Query: output the k^{th} value		
• Insertion / Deletion of 1 item		
• Modify the value of the k^{th} item		

Dense vs. Sparse Graphs

Dense Matrix: # nonzeros ≈ $O(n^2)$ e.g., K_n , wireless network, IoT

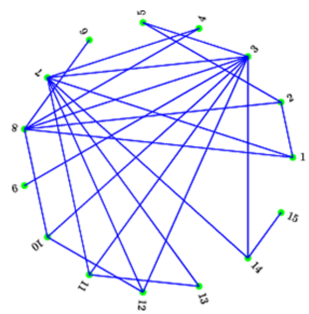
1	2	31	2	9	7	34	22	11	5
11	92	4	3	2	2	3	3	2	1
3	9	13	8	21	17	4	2	1	4
8	32	1	2	34	18	7	78	10	7
9	22	3	9	8	71	12	22	17	3
13	21	21	9	2	47	1	81	21	9
21	12	53	12	91	24	81	8	91	2
61	8	33	82	19	87	16	3	1	55
54	4	78	24	18	11	4	2	99	5
13	22	32	42	9	15	9	22	1	21

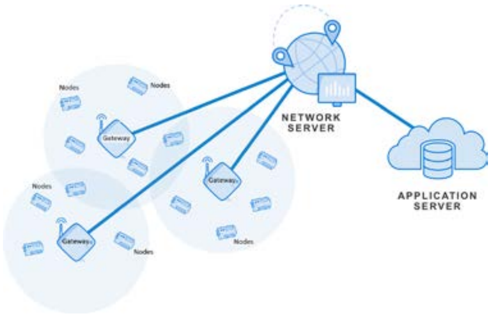




Sparse Matrix: # nonzeros ≈ $O(n)$ e.g., street map, wired network, IoT

1	.	3	.	9	.	3	.	.	.
11	.	4	.	.	.	.	.	2	1
.	.	1	.	.	.	4	.	1	.
8	.	.	.	3	1	.	.	.	.
.	.	.	9	.	.	1	.	17	.
13	21	.	9	2	47	1	81	21	9
.	.	.	.	.	.	.	.	.	.
.	.	.	.	19	8	16	.	.	55
54	4	.	.	.	11	.	.	.	.
.	.	2	.	.	.	.	22	.	21





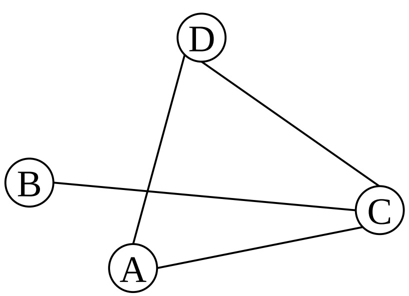
Real-world applications: mostly **sparse** graphs
e.g., social network, 6-degree separation theory,
transportation network, logistics network

Isomorphism

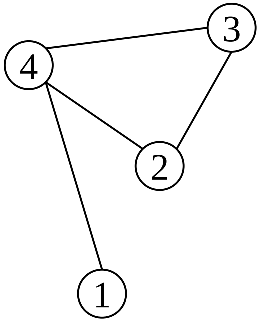
- G_1 is isomorphic to G_2: there is a one-to-one correspondence *f* between the vertices of G_1 and G_2 such that vertices *u* and *w* are *adjacent* in G_1 iff the vertices *f(u)* and *f(w)* are *adjacent* in G_2
- Another way to say, G_1 and G_2 are isomorphic, denoted $G_1 \cong G_2$
- Such a function is called an isomorphism of G_1 with G_2

Thm 4.3

Let *f* be an isomorphism of G_1 with G_2 . For any vertex v in G_1, the degrees of v and *f*(v) are equal



G_1

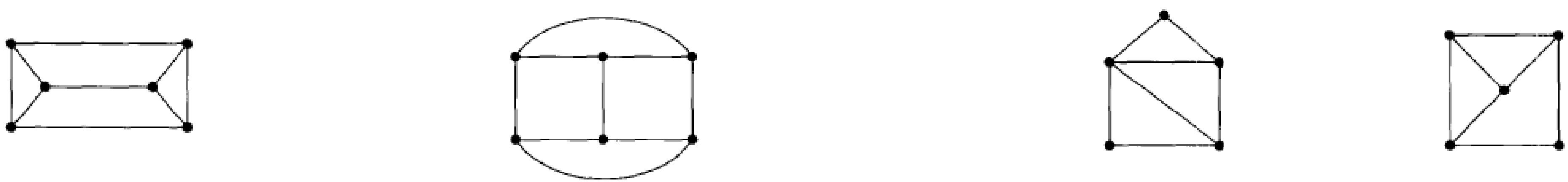


G_2

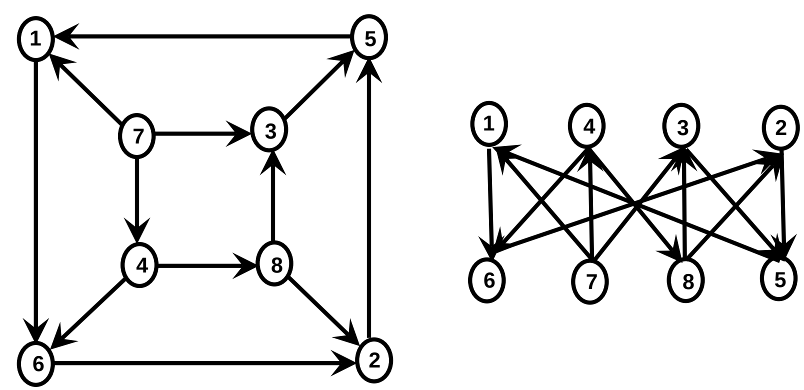
Exercises 4.1 (Isomorphism)

Q42: isomorphic?

Q43 isomorphic?



- Isomorphism is an equivalence relation
  - Symmetric, reflexive, transitive
- Isomorphism is one-to-one & onto

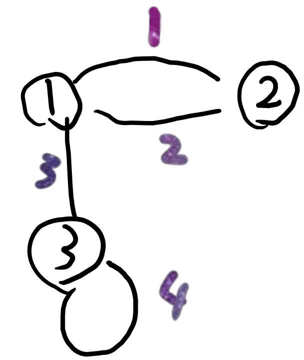


Multigraph & Path

- A multigraph (pseudograph) consists of a nonempty finite set of vertices and a set of edges, where we allow an edge to join a vertex to itself (loop), and several edges joining the same pair of vertices (parallel edges).
- Loop:
  –
- Parallel edge:
- A simple graph is a graph that contains NO &
- u-v path (path from u to v) with length **k**:
  sequence of k+1 vertices $v_1 v_2 \ldots v_{k+1}$, where v_1=u, v_{k+1}=v
  sequence of k edges $e_1 e_2 \ldots e_k$
  – These k+1 vertices & k edges may appear more than once (i.e. a path may pass the same vertex or edge twice)
    [such definition may be different in other books]
- u-v simple path:

Data Structure for a Multigraph

- Adjacency Matrix ?
  - need the 3rd dimension ?



- Adjacency List

In C/C++, define **struct** for node & arc
Array of **n** nodes & **m** arcs

Simple Path, Cycle

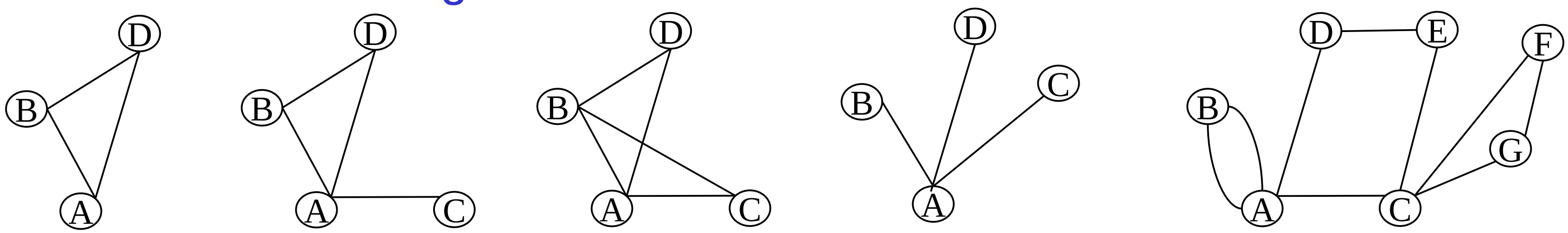
Thm 4.4 Every u-v path contains a u-v simple path

- A multigraph is connected if there is a path between every 2 vertices
  - In a connected multigraph, each vertex is reachable from every other vertex
  - An disconnected graph may contain isolated vertices which has zero degree
- A circuit is a path from a vertex to itself without repeated intermediate edges
- A cycle is a circuit without repeated intermediate vertices.

Euler Circuit & Euler Path

Euler path: a path in a multigraph G that includes exactly once all the edges of G and has different origin and destination

Euler circuit: a path in a multigraph G that includes exactly once all the edges of G and has the same origin and destination



EC: ABDA   EP: ABDAC   EP: ABDACB   EP: ? EC: ?   EC: ABA DEC FGCA

Thm 4.5 For a connected multigraph G (with |V|≧2)

(a) each vertex in G has even degree iff G contains an Euler circuit
each vertex in G has even degree except for 2 distinct vertices iff G contains an Euler path

(b) Euler path starts at one and ends at the other of the 2 vertices of odd degree

- This theorem gives a O(n) algorithm to determine whether a given graph contains Euler circuits, or Euler paths.

Proof of Thm 4.5 (1/2)

(a) **deg(i)** is **even** for each i in V ⇔ There exists an **Euler Circuit**

(b): an **Euler Path** starts and end at the only 2 vertices of **odd degree**

🡐 trivial,

Proof of Thm 4.5 (2/2)

Details on the Euler Circuit Algorithm ➔

choose a starting vertex u, if there is a loop on u, pass it, then leave u along an edge e_1 (u,u_1) (it must exist since there are more than 2 vertices and the graph is connected), in u_1, if there is a loop on u_1, pass it, then leave u_1 along another edge e_2 different from e_1 (e_2 must exist since each vertex has even degree) to vertex u_2.
Repeat these steps until finally we return u. (this has to happen since # edge is finite and the graph is connected) Then we get a circuit C_1 from u to u.

If C_1 covers all the edges, then we get an Euler circuit, done.

Otherwise, we delete all the edges on C_1 from G, then delete all the induced isolated vertices from G, and obtain the remaining subgraph G_1.

On G_1, we choose a vertex v (it exists due to connected graph), and do the same steps as above to form a circuit C_2 back to v.

Now, combine C_1 & C_2, we get a larger circuit C’ starts from u along C_1 to v, then along C_2 back to v, and along C_1 back to u. if C’ covers all the edges, we are done. Otherwise, we repeat the same procedures until finally all the edges are covered. The algorithm will stop since the graph is finite, at which time, we obtain an Euler circuit by combining all the smaller circuits.

Euler Circuit Algorithm

Given: connected multigraph G in which every vertex has even degree

```
begin
    initialization: Set E={all edges of G}
      select a vertex u, let C be the path consisting of just u    <-- Pass loop first
    while E is not empty do
          choose a vertex v in C that is incident with some edge in E
          set P to be the path consisting of just v
          set w=v
          while there is an edge e in E that is incident with w
                remove e from E                                      } Constructing
                replace w with the other vertex on e                 } a circuit P
                append edge e and vertex w to path P                 }
          end while
          replace any one occurrence of v in C with path P   <-- Merging P with C
    end while
    output C: Euler circuit
end
```

Elementary operation → remove e from E

Complexity: O(|E|)

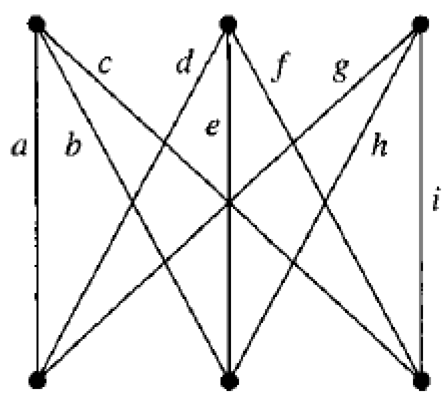
Exercise on Euler circuits (1/2)

Ex1: when does the complete graph K_n contains an Euler circuit?

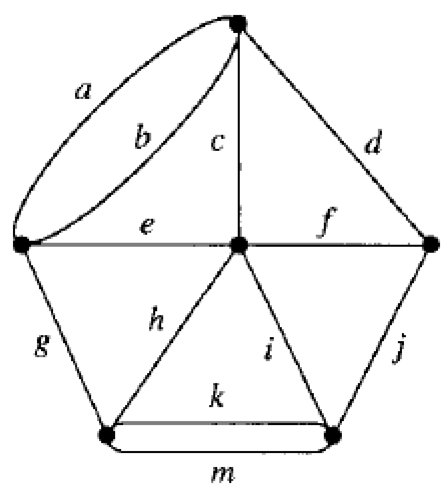
Ex2: give an algorithm to determine whether a given connected graph G contains

Exercise 4.2- determine whether or not there is an E.C. or E.P., draw them if yes.

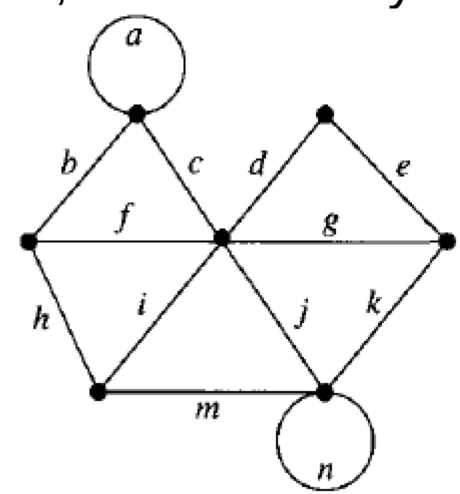
Q19,



Q21,



Q23



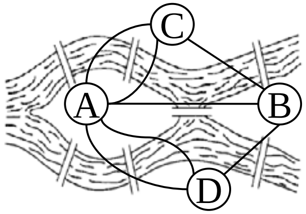
Exercise on Euler circuits (2/2)

Q **31.** Could the citizens of Königsberg find an acceptable route by building a new bridge? If so, how?

32. Could the citizens of Königsberg find an acceptable route by building two new bridges? If so, how?

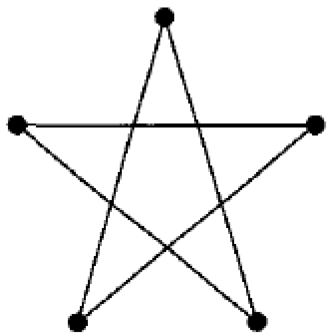
33. Could the citizens of Königsberg find an acceptable route by tearing down a bridge? If so, how?

34. Could the citizens of Königsberg find an acceptable route by tearing down two bridges? If so, how?

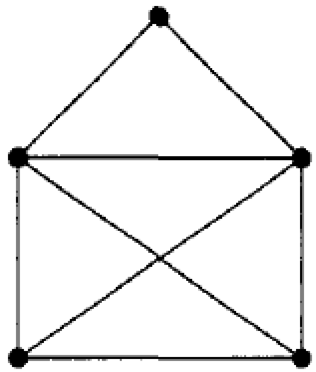


An old childhood game asks children to trace a figure with a pencil without either lifting the pencil from the figure or tracing a line more than once. Determine if this can be done for the figures in Exercises 35–38, assuming that you must begin and end at the same point.

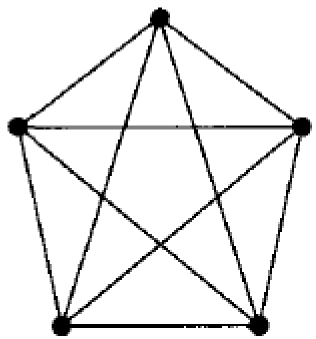
35.



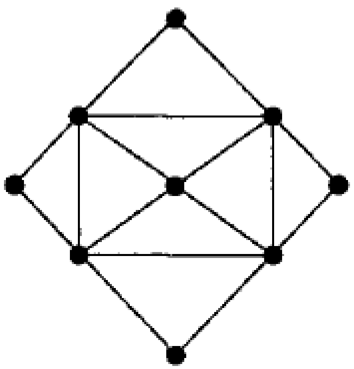
36.



37.

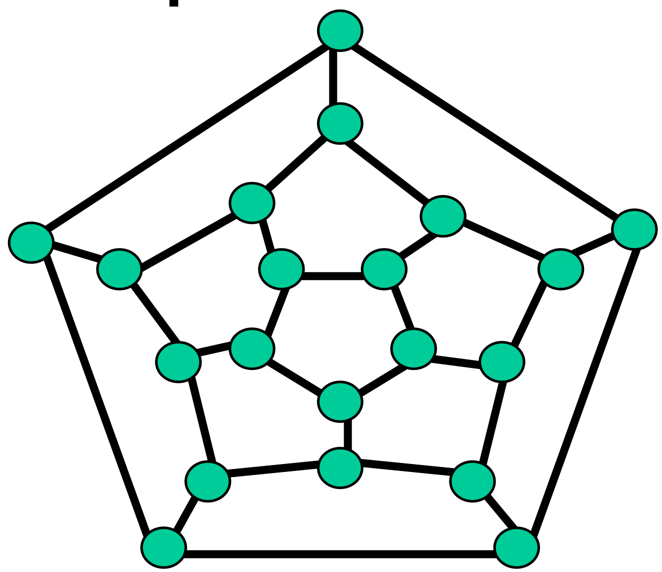


38.



Hamilton's Around the World Game

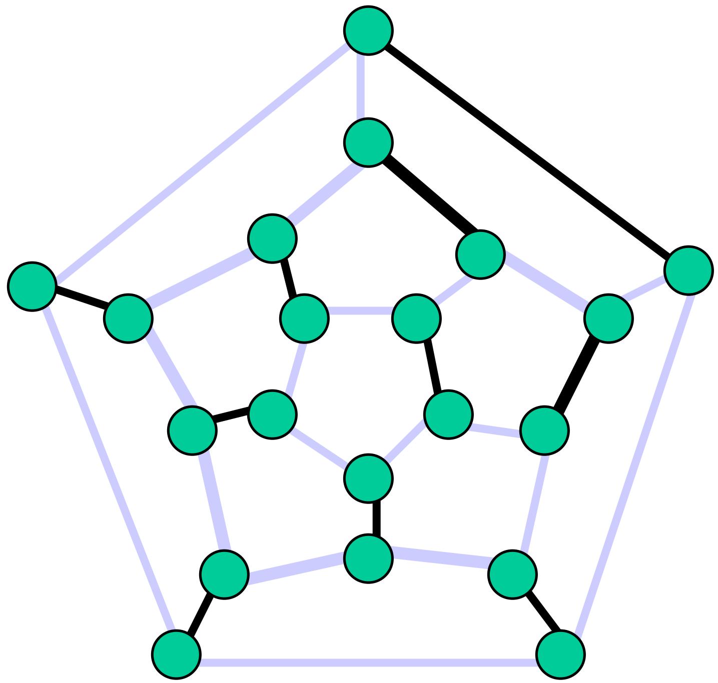
In 1857, the Irish mathematician, Sir William Rowan Hamilton invented a puzzle that he hoped would be very popular.



The objective was to make a hamiltonian cycle.

The game was not a commercial success, especially the 3D version. But the mathematics of hamiltonian cycles is very popular today.

Hamilton's Around the World Game



This problem can be generalized to be the *traveling salesman problem.*

Hamiltonian Cycles & Paths

Hamiltonian path: a path in a multigraph G that includes exactly once all the vertices of G and has different origin and destination

Hamiltonian cycle: a cycle in a multigraph G that includes exactly once all the vertices of G

- It is a major unsolved problem to determine necessary & sufficient conditions for a graph to have either Hamiltonian cycle or path.
  - A special case of the Traveling Salesman Problem (TSP)
  - A complete graph must be Hamiltonian (i.e. contains Hamiltonian cycle)

Thm 4.6 (Ore's Thm) suppose G is a simple graph with $n \geqq 3$ vertices if for each pair of nonadjacent vertices u and v we have $\deg(u)+\deg(v) \geqq n$, then G has a Hamiltonian cycle

Show Ore's Theorem

Try to show by contradiction, Suppose such G has NO H.C. to show $deg(x)+deg(y)<n$

G is a subgraph of K_n (but G is not K_n [why?]).

We can recursively add edges between 2 nonadj. vertices of G until it becomes another graph H, where adding one more edge joining 2 nonadj. vertices in H will create a H.C.

Let x,y be 2 nonadj. vertices in H ➔ x,y are nonadj in G [why?]

If we add a new edge (x,y) on H ➔ we will create a H.C. (by assumption)

i.e., there exists a H.P. from x to y in H, say, $x=v_1-v_2-v_3-\ldots-v_{i-1}-v_i-v_{i+1}-\ldots-v_{n-1}-v_n=y$

Suppose deg(x)=r

Claim: if there exists an edge (x,v_i), then there exists NO edge (v_{i-1},y)

Pf: if there exists edge (v_{i-1},y), then H has a H.C. Contradiction.

The claim is true for i=2,3,…,n-1

Since deg(x)=r, thus there exists r vertices in $\{v_1,v_2,\ldots,v_{n-2}\}$ NOT adjacent to y.

Thus $deg(y)\leqq(n-2)-r$ which means $deg(x)+deg(y)\leqq n-2<n$ in H (and in G, too)

Thus, if $deg(x)+deg(y)\geqq n$ in G ➔ such G has H.C.

Exercise on Hamiltonian Cycles (1/2)

Dirac's Thm

suppose G is a simple graph with n≧3 vertices, and every vertex has degree at least n/2 then G contains a Hamiltonian cycle

Some criteria to check whether G has a H.C. or H.P.

(!Note! If the following 4 criteria all fail, it's still possible to have H.C. or H.P.)

if $\deg(x)+\deg(y) \ge n-1$ for each distinct $x,y \in V$, then G has a Hamiltonian path

if $\deg(x)+\deg(y) \ge n$ for each distinct $x,y \in V$, then G has a Hamiltonian cyclce

if $\deg(x) \ge (n-1)/2$ for each $x \in V$, then G has a Hamiltonian path

if $\deg(x) \ge n/2$ for each $x \in V$, then G has a Hamiltonian cycle

Ex1: give examples of simple graphs that (a) have E.C. & H.C. (b) have E.C. but NO H.C. (c) has H.C. but no E.C.

Ex2: For a K_n, how many distinct (a) H.P. (b) H.C. ?

Exercise on Hamiltonian Cycles (2/2)

Q.52 A **bipartite graph** is a graph in which the vertices can be divided into two disjoint nonempty sets A and B such that no two vertices in A are adjacent and no two vertices in B are adjacent. The **complete bipartite graph** $\mathcal{K}_{m,n}$ is a bipartite graph in which the sets A and B contain m and n vertices, respectively, and every vertex in A is adjacent to every vertex in B. The graph $\mathcal{K}_{2,3}$ is given below. How many edges does $\mathcal{K}_{m,n}$ have?

Q.53 For which m and n does $\mathcal{K}_{m,n}$ have an Euler circuit?

Q.54 For which m and n does $\mathcal{K}_{m,n}$ have a Hamiltonian cycle?

Q: How to use Ore's or Dirac's Thm to detect H.C./H.P.? Complexity of your methods?

Shortest Paths and Distance

Shortest path: a path of minimal length between vertices s and t

Distance from s to t: smallest possible # of edges in a path from s to t

!Note! This definition only applies for unweighted graph

Weighted graph: a graph where each edge has a "weight" as its length.

Weight of a path: sum of weights of the edges in the path

- To trace a path, we can use predecessor (pred) or successor (succ)

If a path passes vertex i then consecutively to vertex j, we say

pred(j)=i , succ(i)=j

If we record the pred(i) (or succ(i)) for each vertex i in any path-related algorithm, we then will be able to trace the entire path

- For the first vertex in the path, say, s, we define pred(s)=NULL (or "-")
- For the last vertex in the path, say, t, we define succ(t)=NULL (or "-")

e.g. pred(1)=3, pred(2)=4, pred(3)=2, pred(4)=NULL

4 — 2 — 3 — 1

e.g. succ(4)=2, succ(2)=3, succ(3)=1, succ(1)=NULL

Shortest Paths (SP) on a Unit Graph

Unit graph: a graph where all edges have the same length

1-1 SP on a unit graph: a s-t path with the minimum number of edges

e.g., to measure the closeness of 2 persons in a social network

Search algorithm:

INPUT: A network G, and starting vertex s

OUTPUT: the set of reachable vertices from s: {j : there is a path from s to j in G}

Settings:

For vertex i:

–distance label: *d[i]* is the distance (# edges) between s & i

–Predecessor: *pred[i]* is the predecessor of i on the path from s to I

–A vertex is either *marked (visited)* or *unmarked (unvisited)*

Initially only s is marked.

If a vertex is marked ➔ it is reachable from s

–An edge (i,j) in A is *admissible* if vertex i is marked but j is unmarked

visited unvisited

Breadth-First Search

To find out all the vertices that are reachable from a specific vertex (source), say, s

M represents infinity (∞)

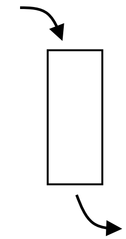
begin

initialization: Set d[v]=M for each v in V; LIST= ∅

d[s]=0; pred[s]=NULL;

add s into LIST;

LIST is a queue (FIFO)



while LIST is not empty **do**

remove a vertex i from LIST; ← **Select i** **Vertex Selection [VS]**

while there exists an admissible edge (i,j)

d[j]=d[i]+1; pred[j]=i;

add j into LIST; mark (i,j) as scanned;

Scan i **Distance Updating [DU]**

end while

end while

for each vertex i **do**

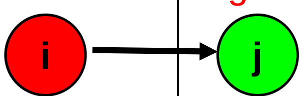
tracepath(s,i);

end for

end

BFS Animation

Admissible edge/arc



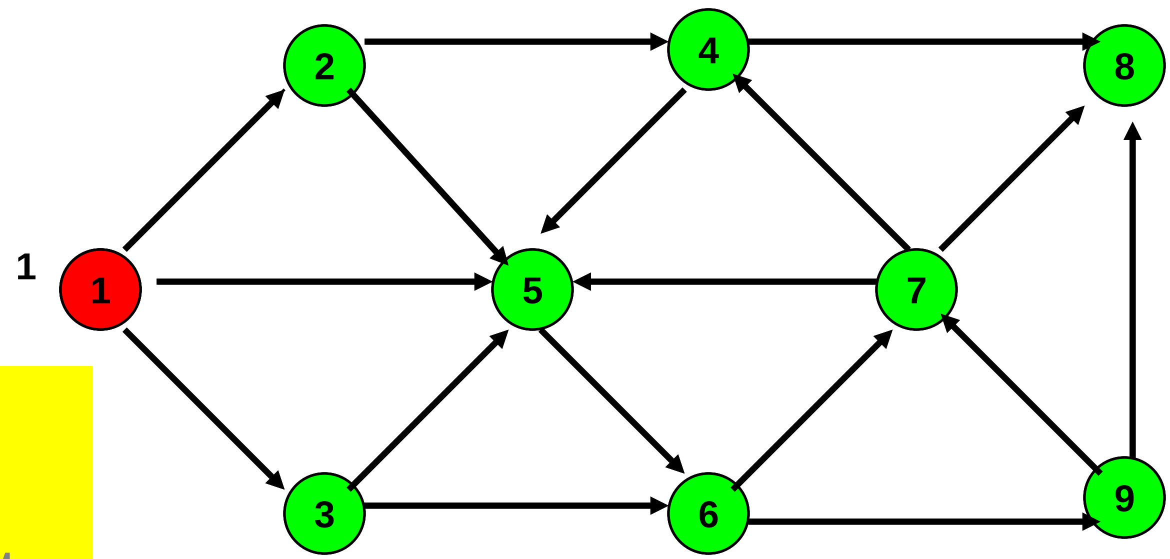
visited

unvisited

d[j]=M

Initialize

Breadth first search animation



pred(1) = 0

next := 1
order(1) = next

LIST:= {1}

LIST

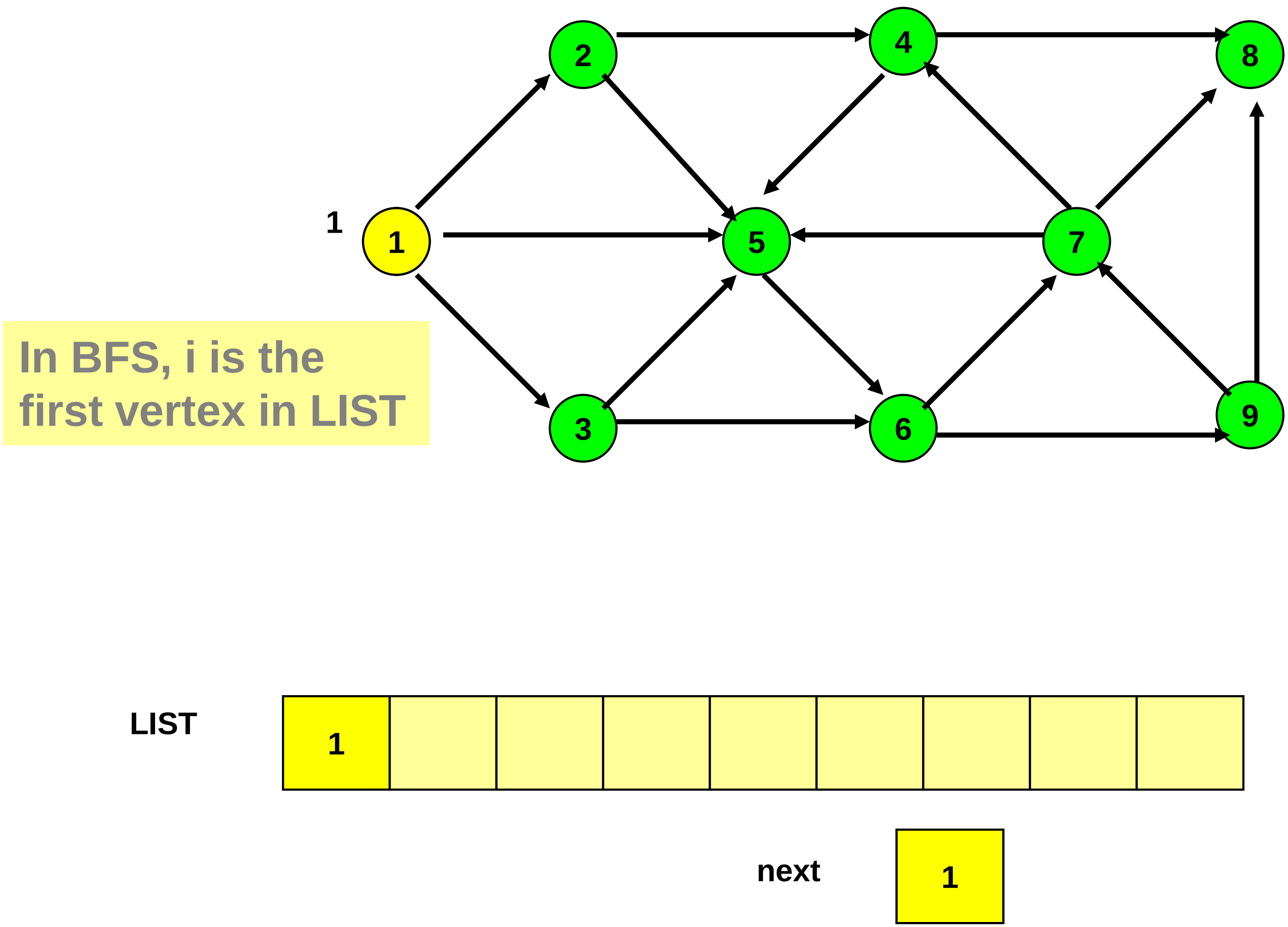
1								

next

1

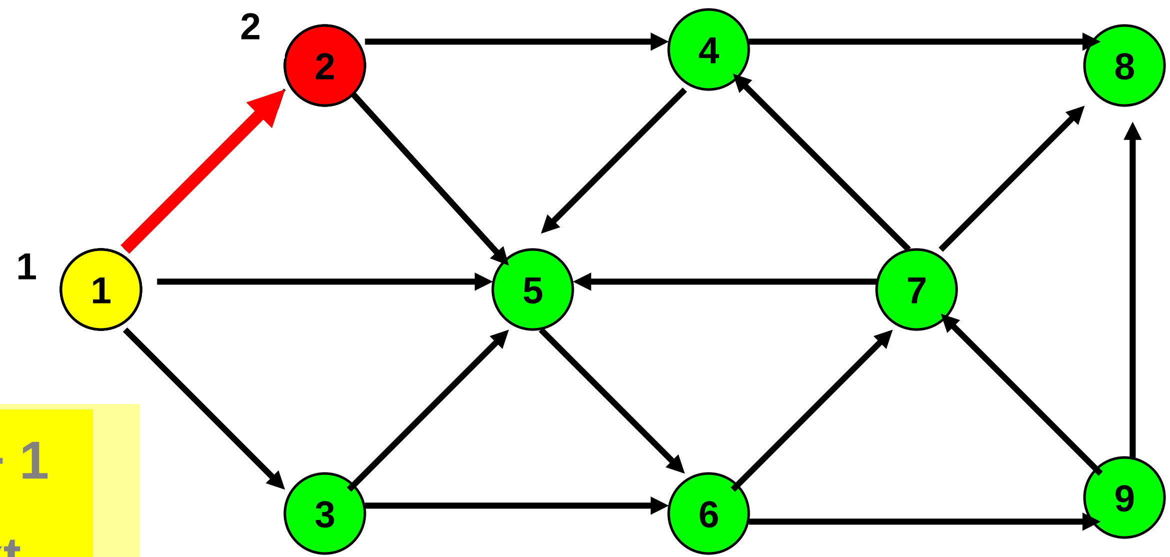
Select a vertex i in LIST [VS]

Breadth first search animation



If vertex i is incident to an admissible edge … [DU]

Breadth first search animation



next := next + 1

order(j) := next

add j to LIST

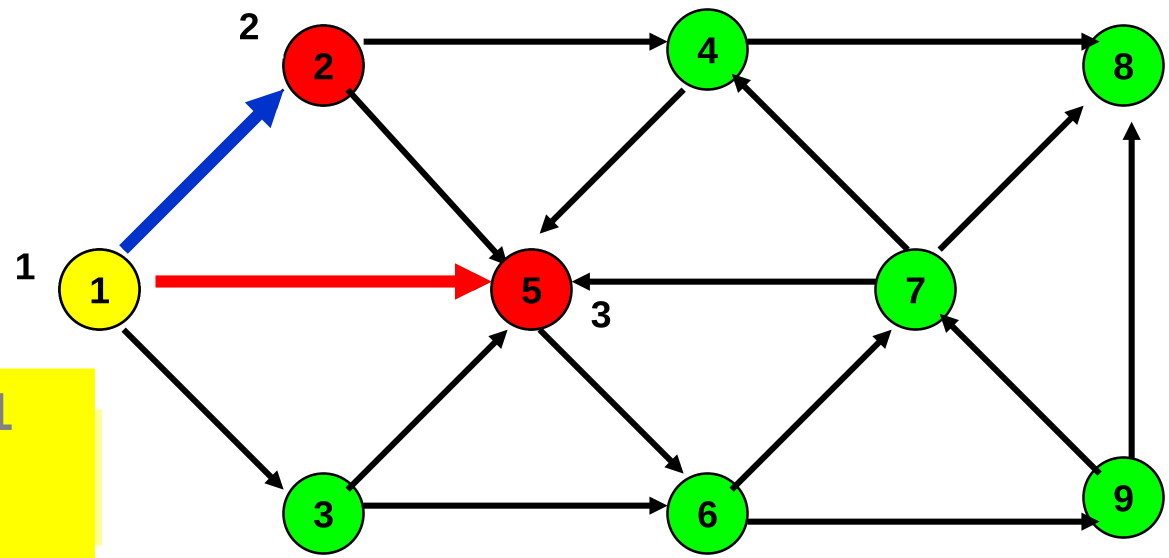
LIST

1	2							

next: 2

If vertex i is incident to an admissible edge… [DU]

Breadth first search animation



next := next + 1

order(j) := next

add j to LIST

LIST

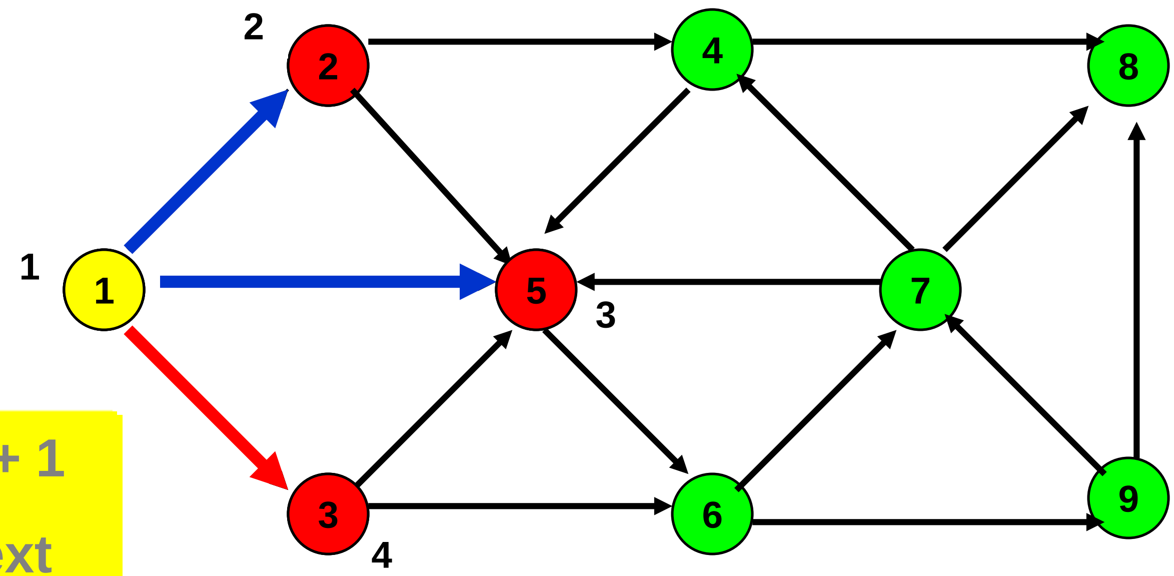
1	2	5						

next

3

If vertex i is incident to an admissible edge … [DU]

Breadth first search animation



next := next + 1

order(j) := next

add j to LIST

LIST

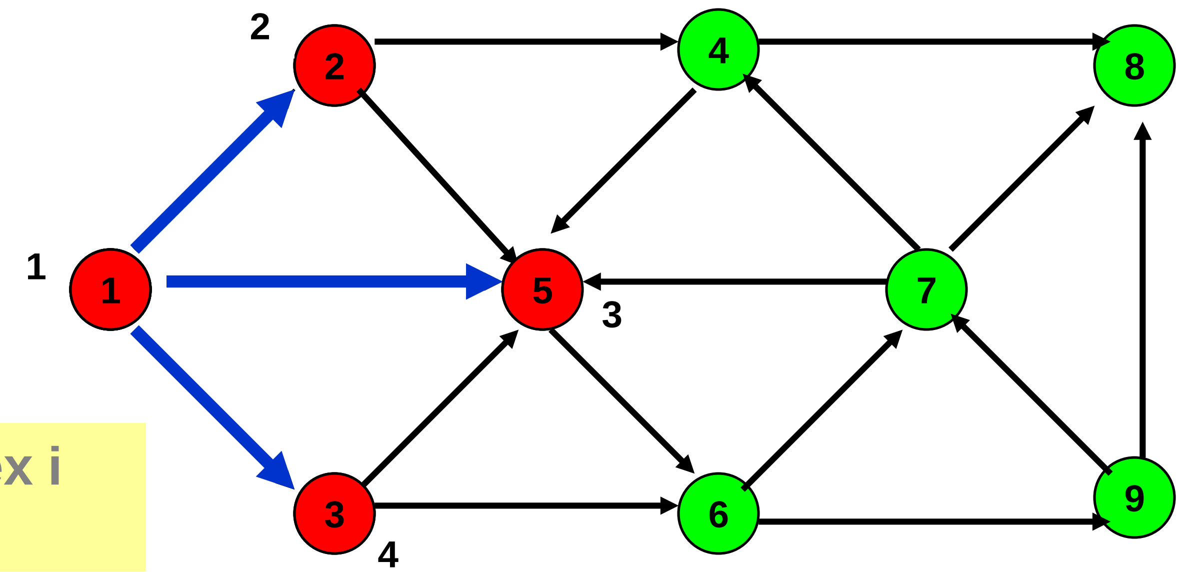
1	2	5	3					

next

4

If vertex i is not incident to an admissible edge [DU]

Breadth first search animation



Delete vertex i from LIST

LIST

2	5	3					

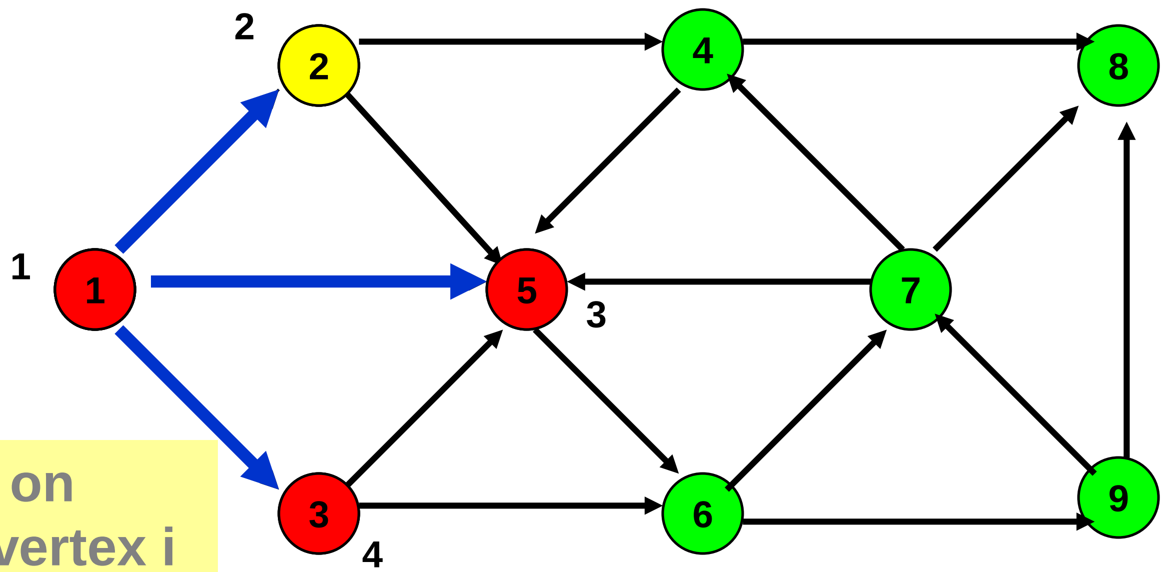
next

4

Select Vertex i

[VS]

Breadth first search animation



The first vertex on LIST becomes vertex i

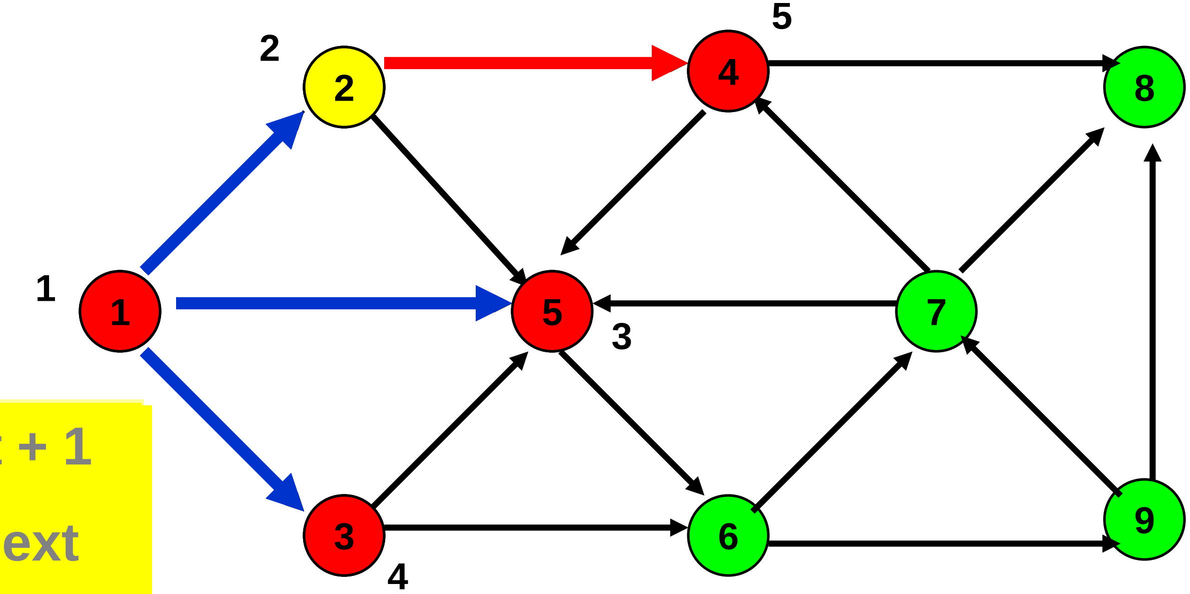
LIST

2	5	3					

next 4

If vertex i is incident to an admissible edge … [DU]

Breadth first search animation



next := next + 1

order(j) := next

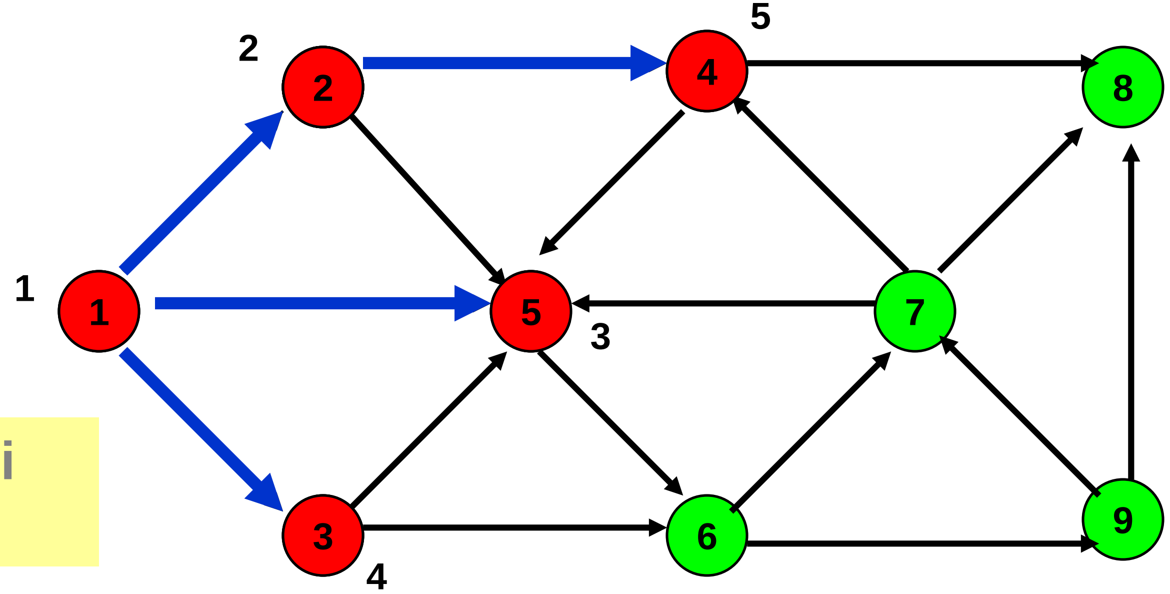
add j to LIST

LIST							
2	5	3	4				

next: 5

If vertex i is not incident to an admissible edge [DU]

Breadth first search animation



Delete vertex i from LIST

LIST

5	3	4				

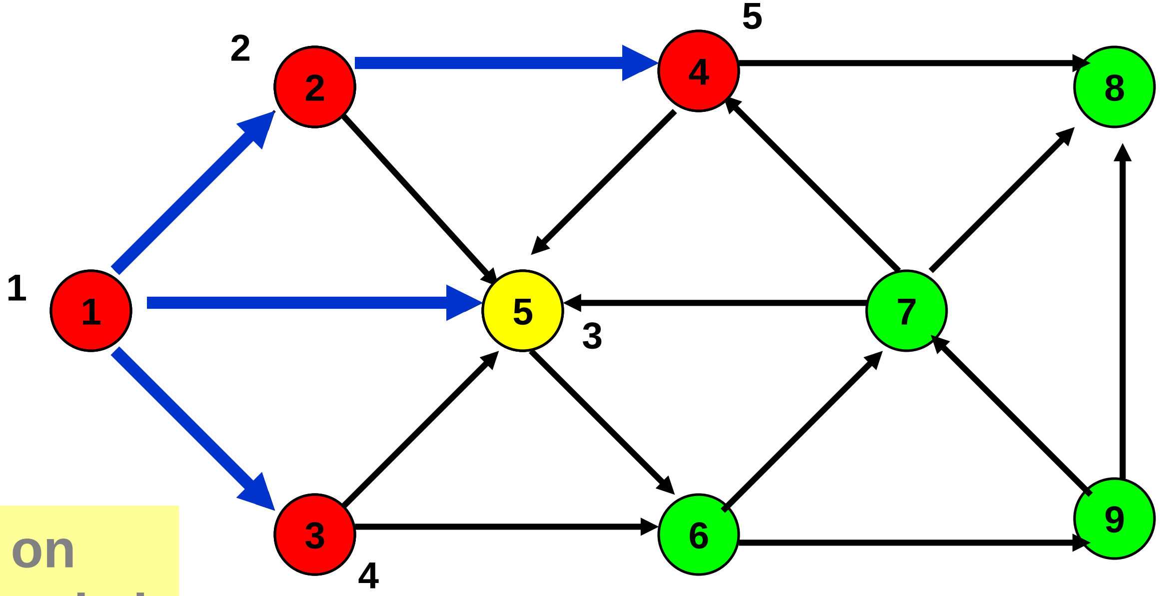
next

5

Select a vertex

[VS]

Breadth first search animation



The first vertex on LIST becomes node i

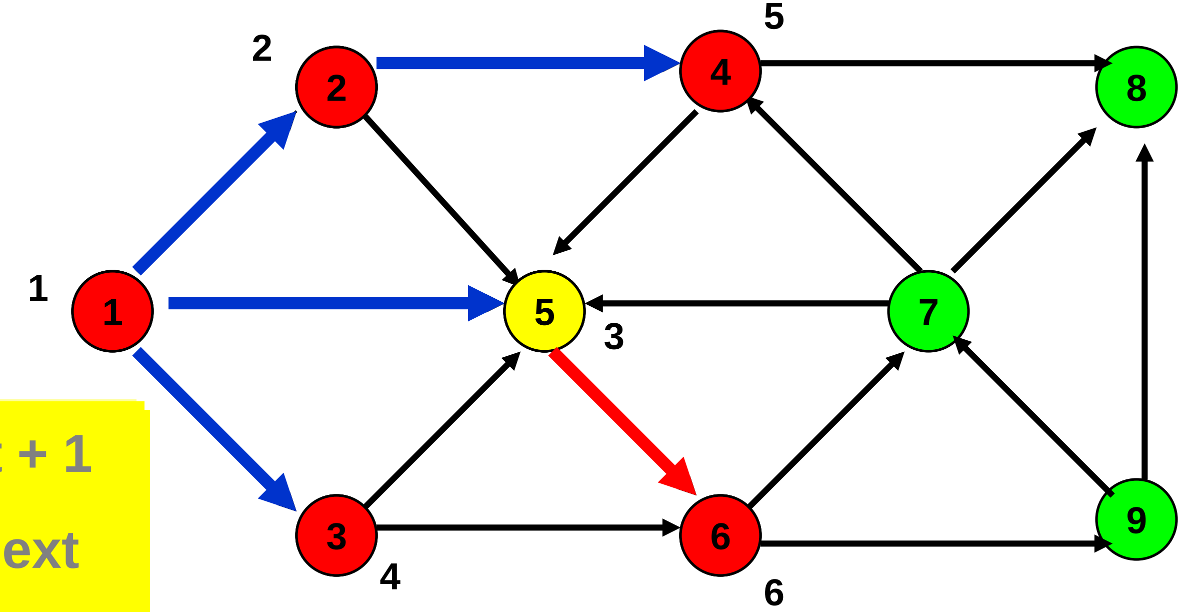
LIST

5	3	4				

next 5

If vertex i is incident to an admissible edge …[DU]

Breadth first search animation



next := next + 1

order(j) := next

add j to LIST

LIST

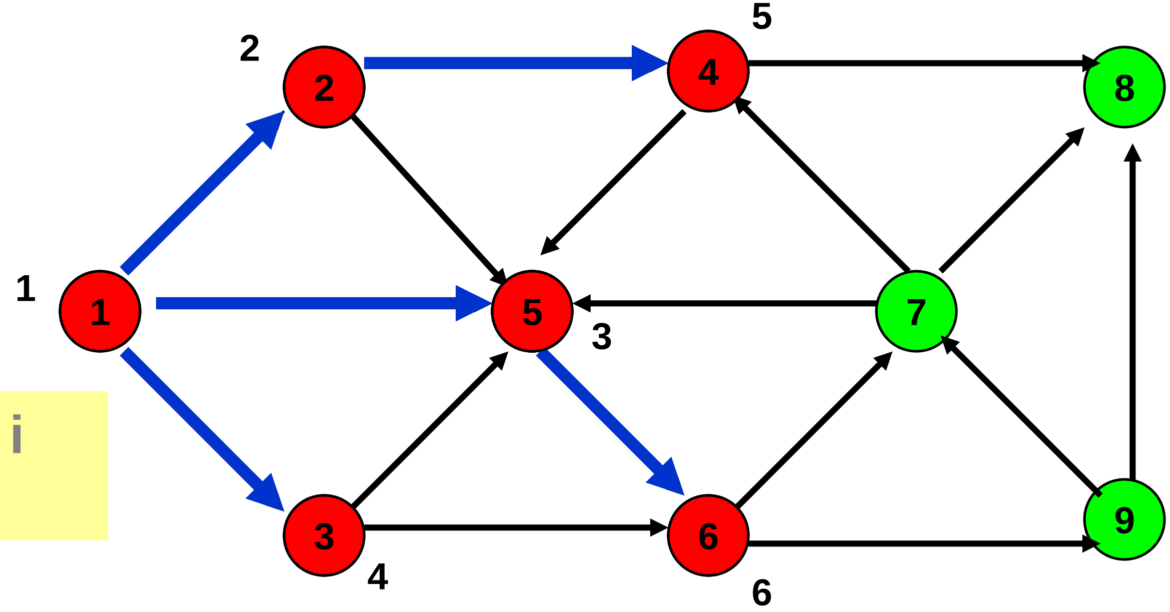
5	3	4	6			

next

6

If vertex i is not incident to an admissible edge [DU]

Breadth first search animation



Delete vertex i from LIST

LIST

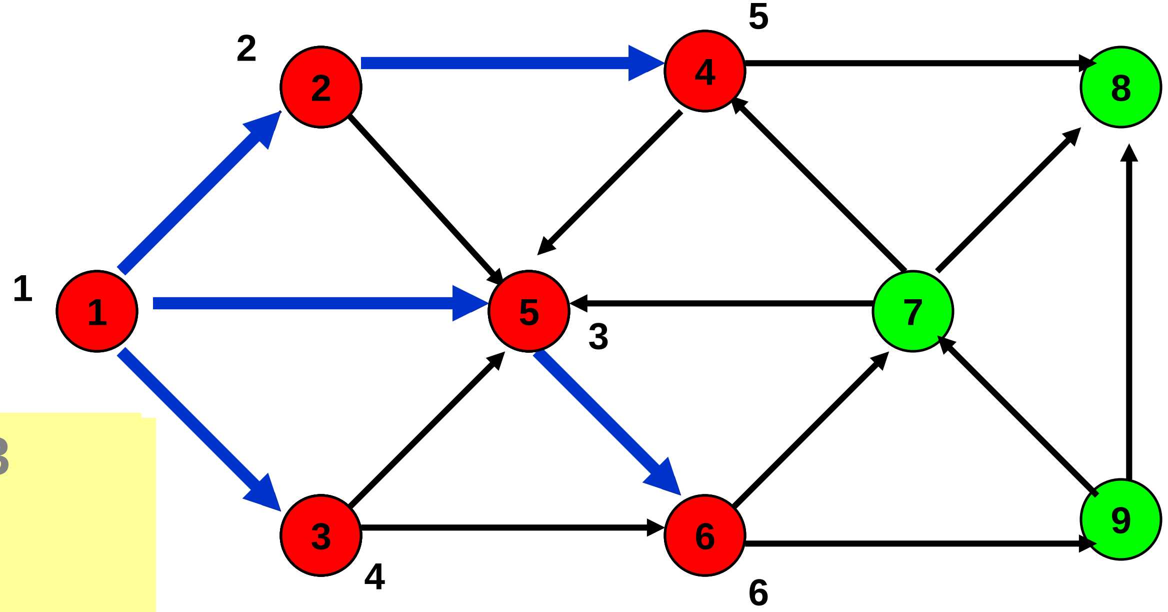
3	4	6			

next 6

Select vertex 3

[VS]

Breadth first search animation



delete vertex 3 from LIST

LIST

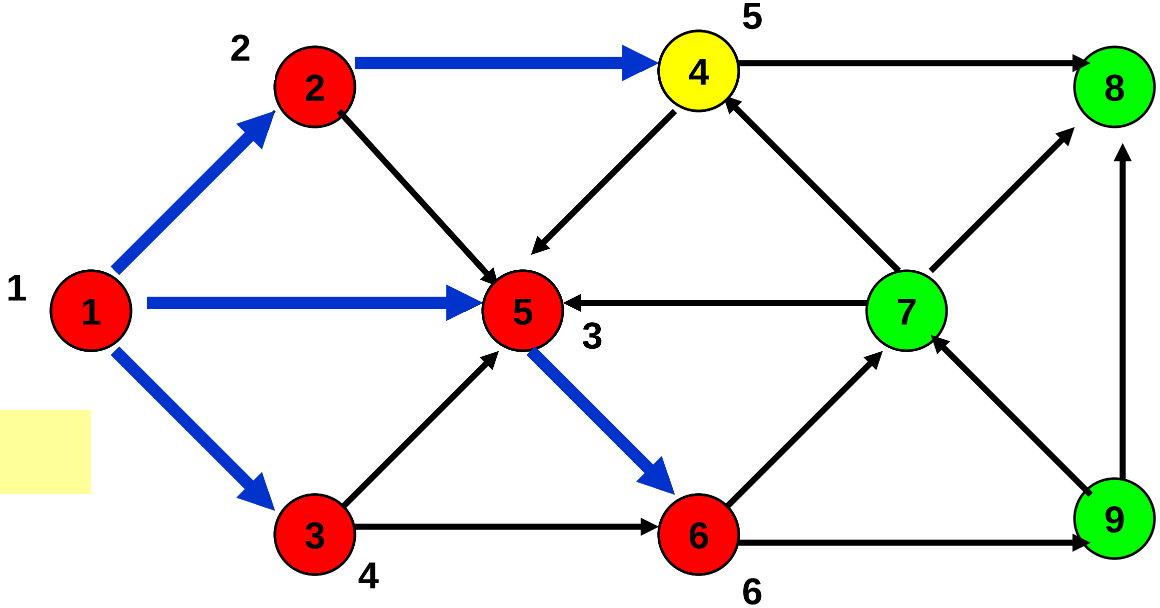
4	6			

next 6

Select a vertex

[VS]

Breadth first search animation



i : = 4

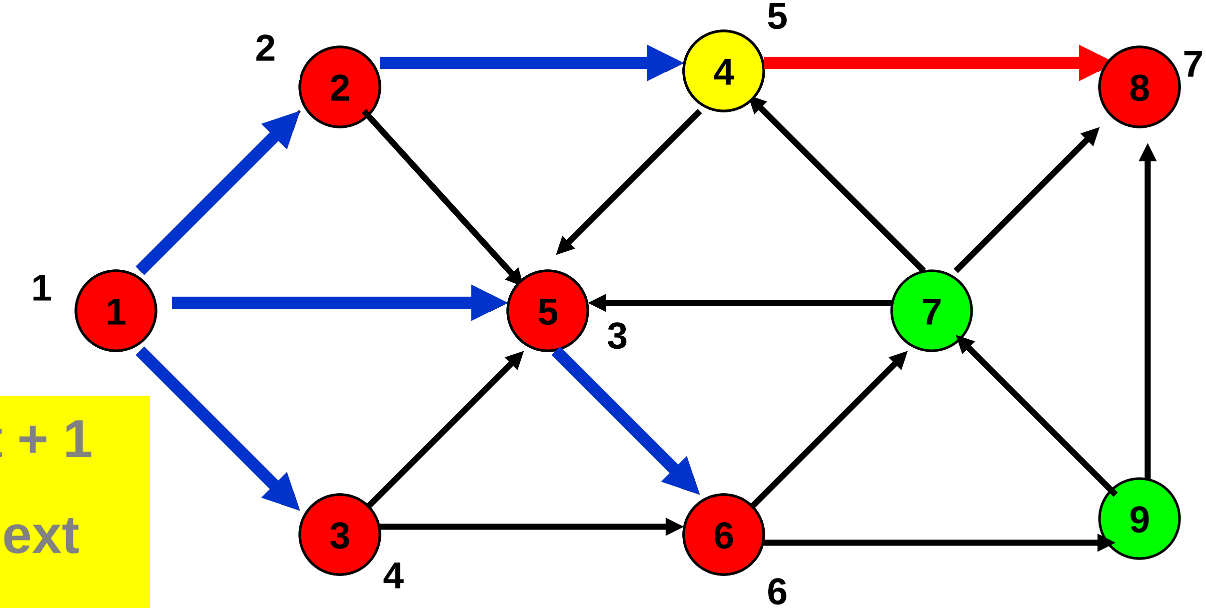
LIST

4	6

next 6

If vertex i is incident to an admissible edge …[DU]

Breadth first search animation



next := next + 1

order(j) := next

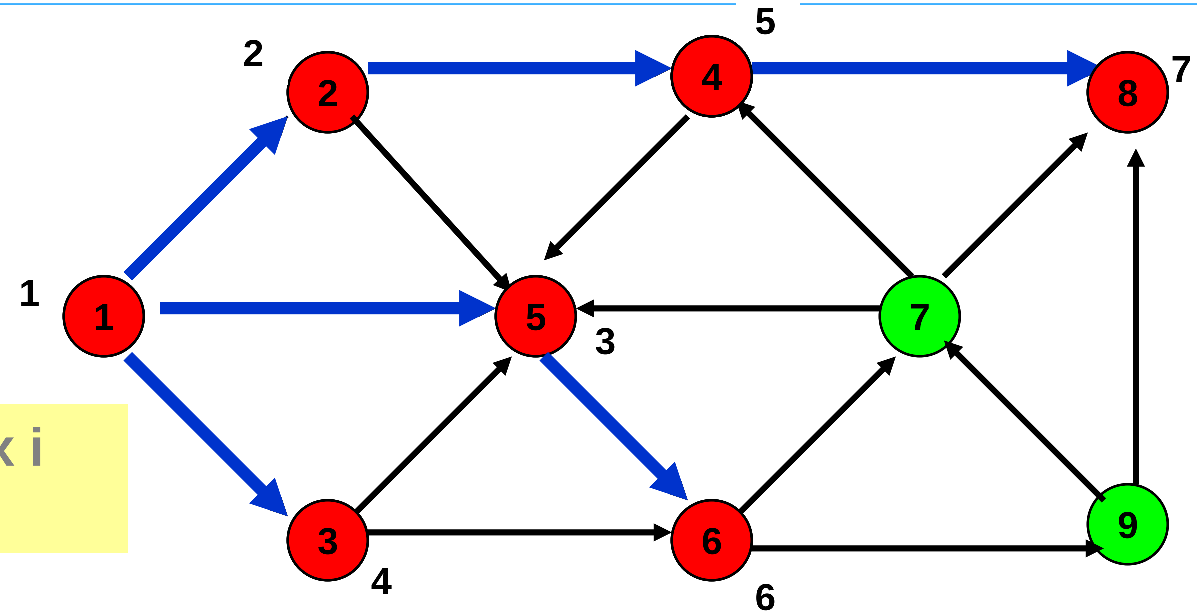
add j to LIST

LIST					
	4	6	8		

next	
	7

If vertex i is not incident to an admissible edge [DU]

Breadth first search animation



Delete vertex i from LIST

LIST

6	8		

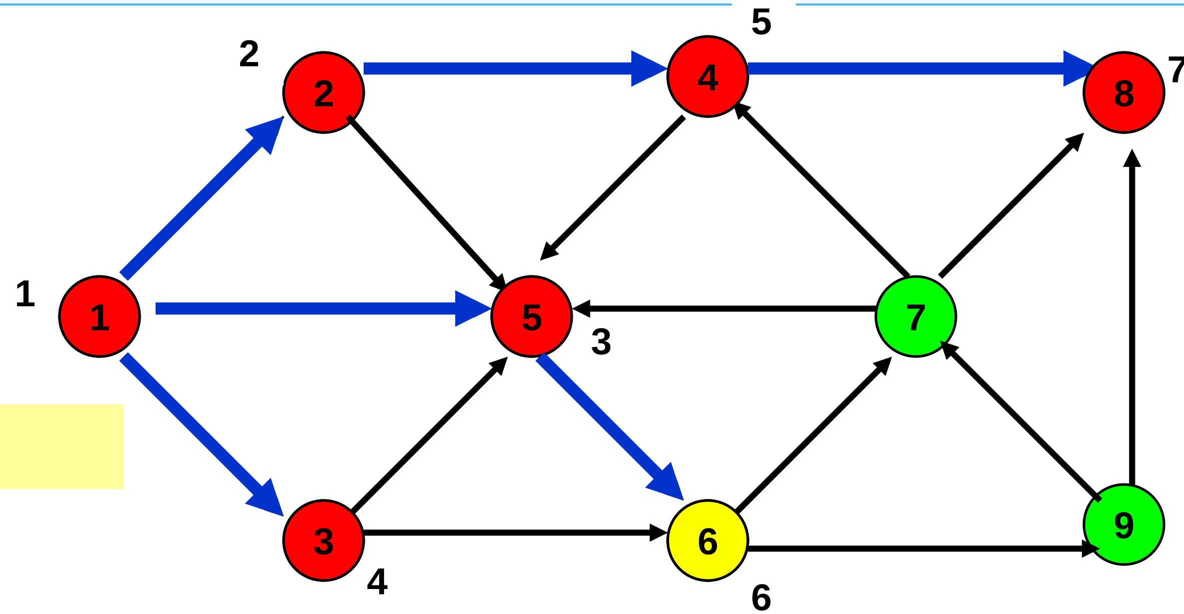
next

7

Select vertex i

[VS]

Breadth first search animation



i := 6

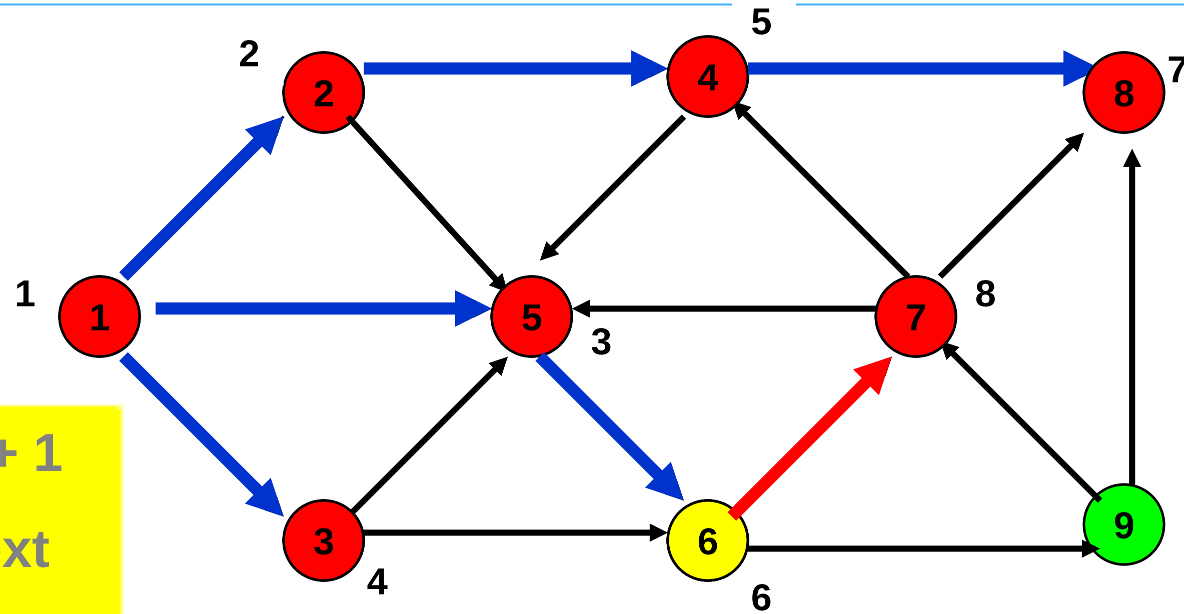
LIST

6	8		

next 7

If vertex i is incident to an admissible edge …[DU]

Breadth first search animation



next := next + 1

order(j) := next

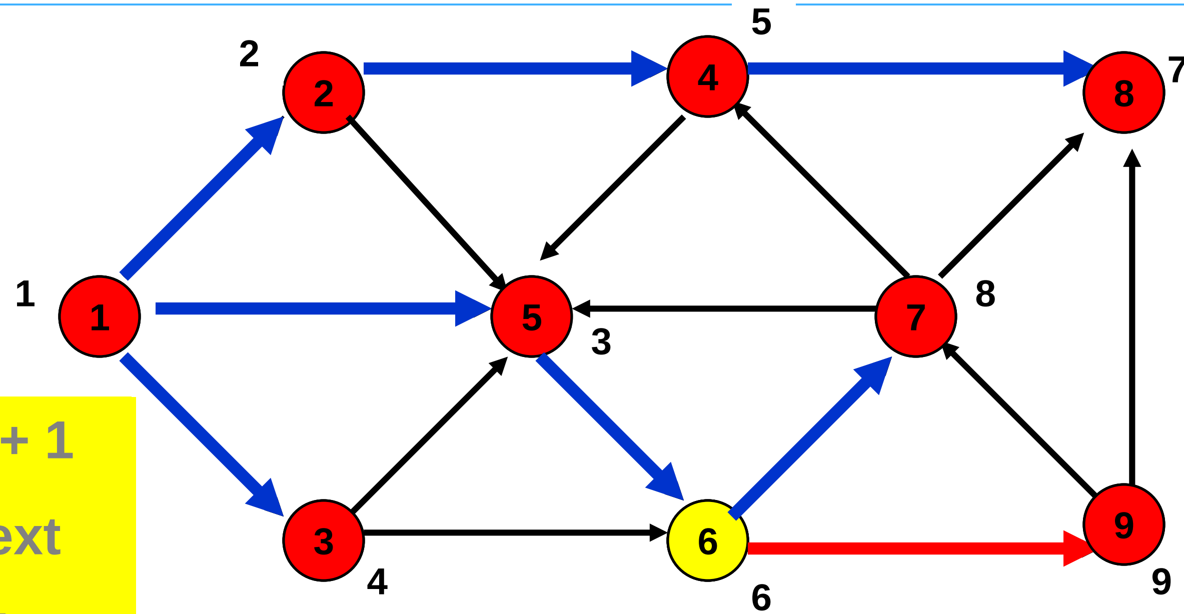
add j to LIST

LIST				
	6	8	7	

next 8

If vertex i is incident to an admissible edge …[DU]

Breadth first search animation



next := next + 1

order(j) := next

add j to LIST

LIST

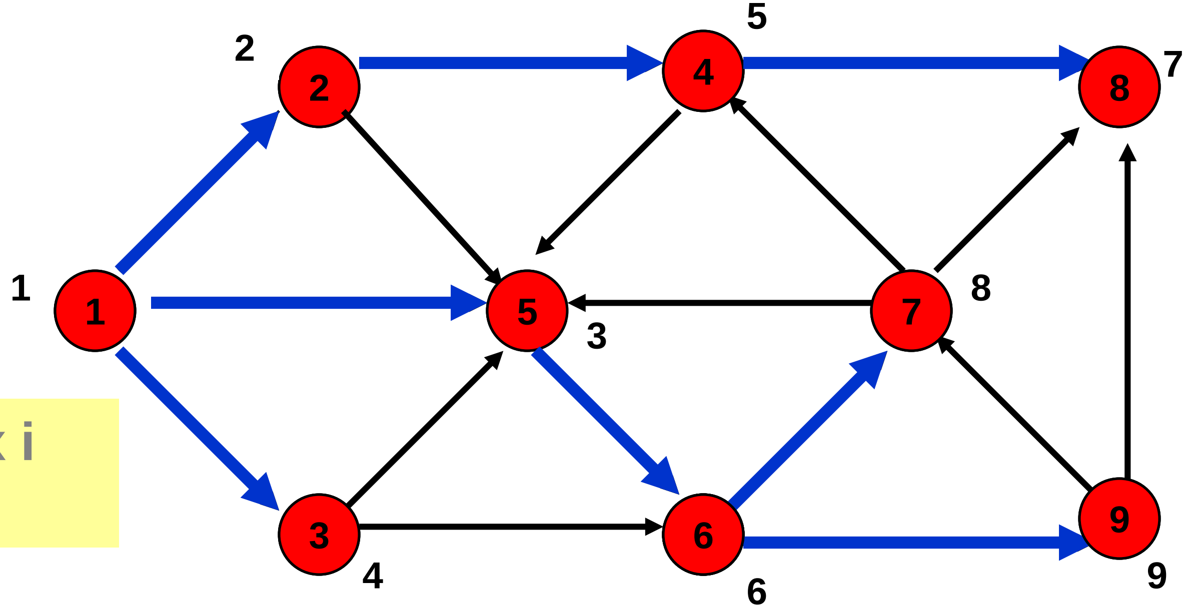
6	8	7	9

next

9

If vertex i is not incident to an admissible edge [DU]

Breadth first search animation



Delete vertex i from LIST

LIST

8	7	9

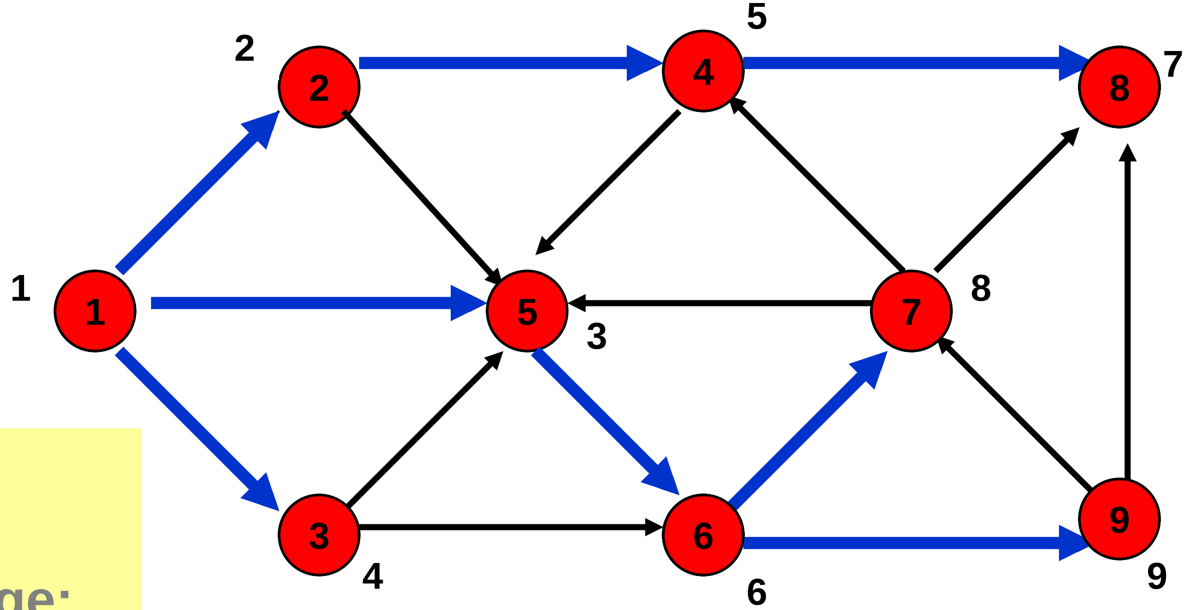
next

9

Select vertex 8

[VS]

Breadth first search animation



vertex 8 is not incident to an admissible edge; delete it from LIST

LIST

7	9

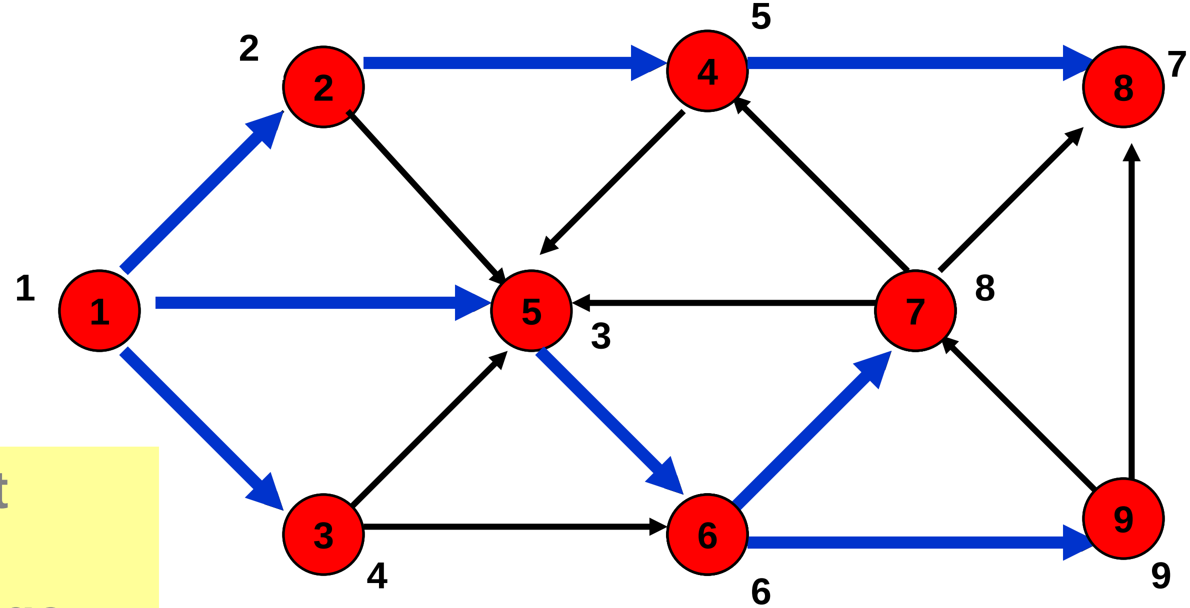
next

9

Select vertex 7

[VS]

Breadth first search animation



vertex 7 is not incident to an admissible edge; delete it from LIST

LIST

9

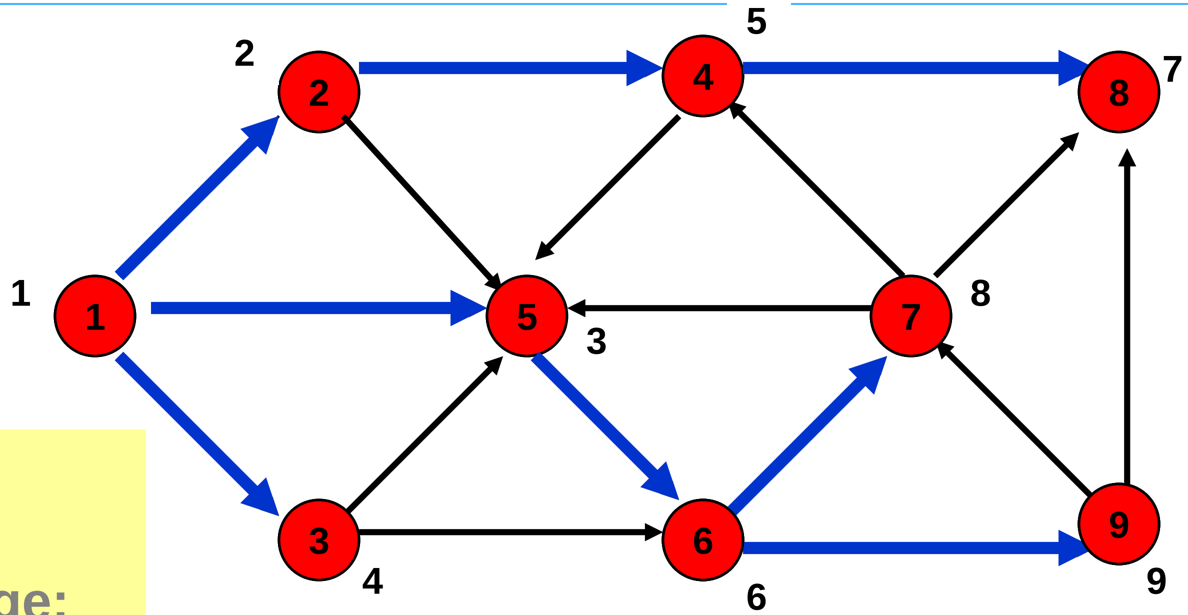
next

9

Select vertex 9

[VS]

Breadth first search animation



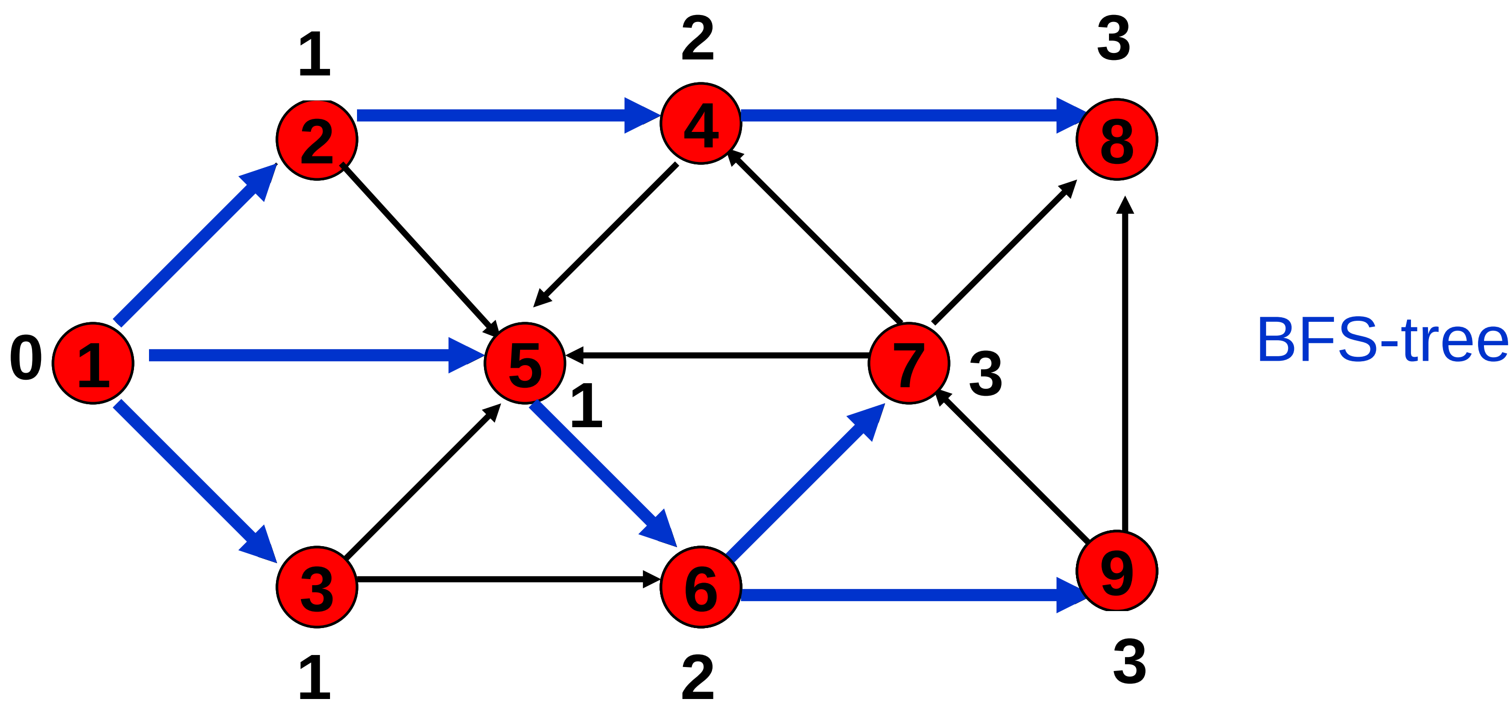
vertex 9 is not incident to an admissible edge; delete it from LIST

LIST

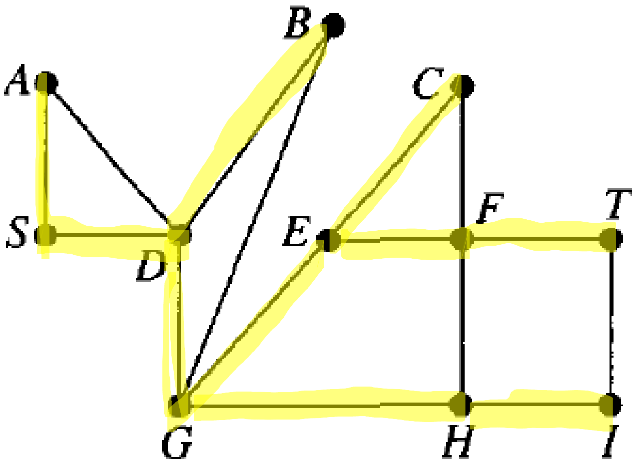
next **9**

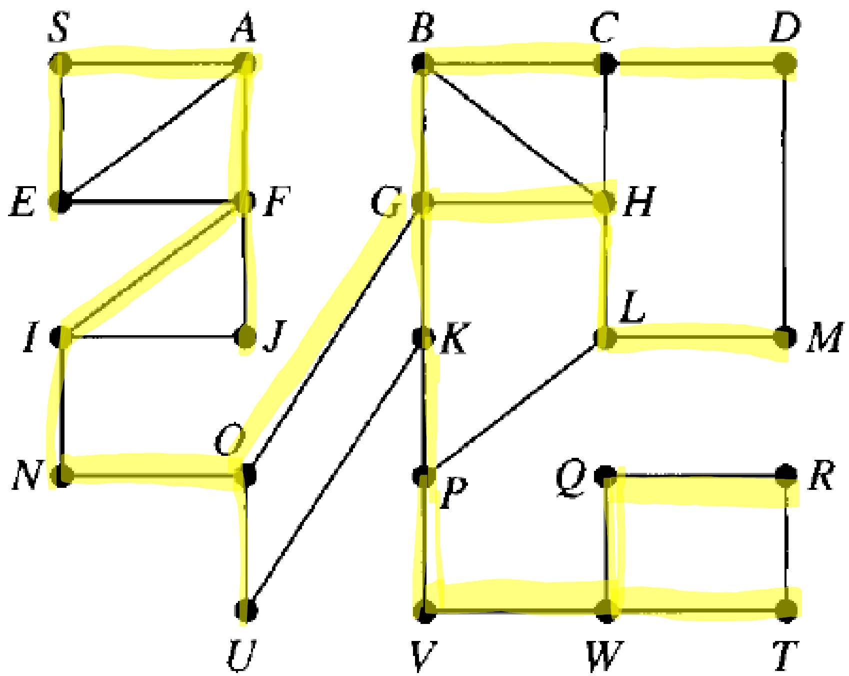
More on Breadth-First Search

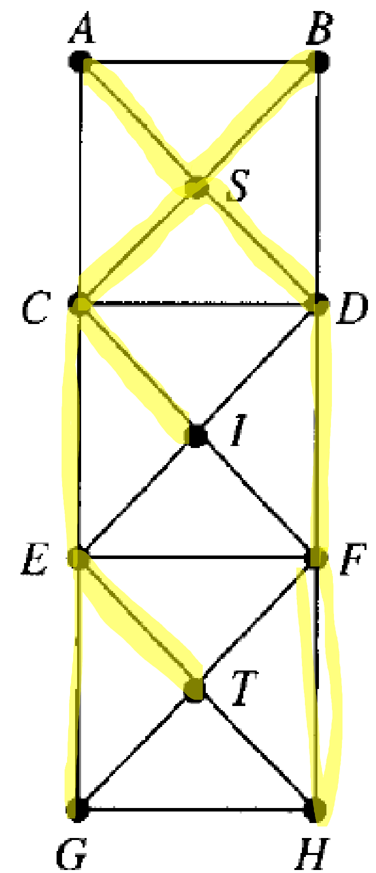
Theorem. The breadth first search tree is the "**shortest path tree**", that is, the path from s to j in the tree has the **fewest possible number of edges**.

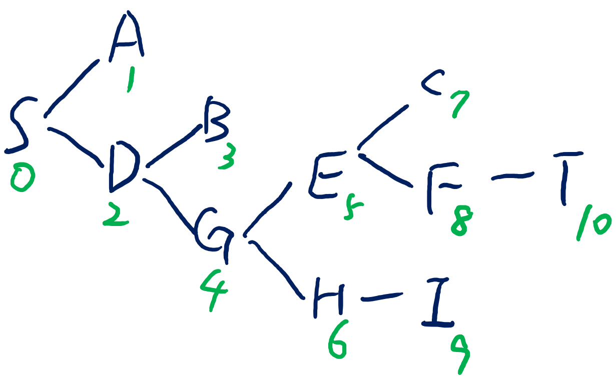


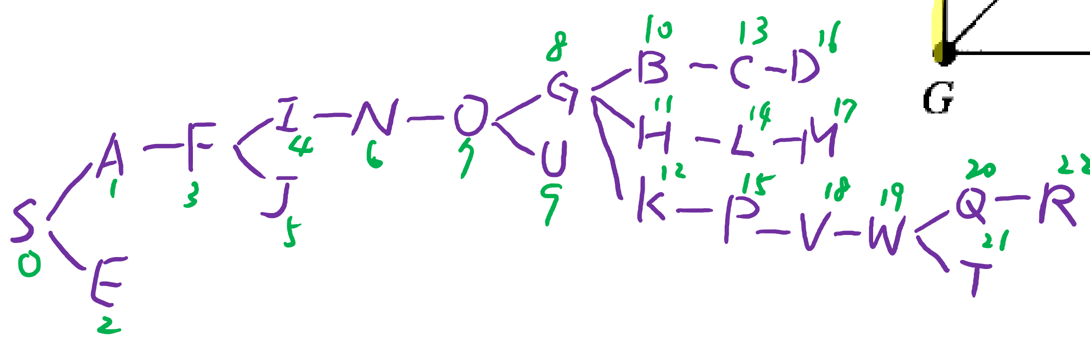
Exercise of BFS



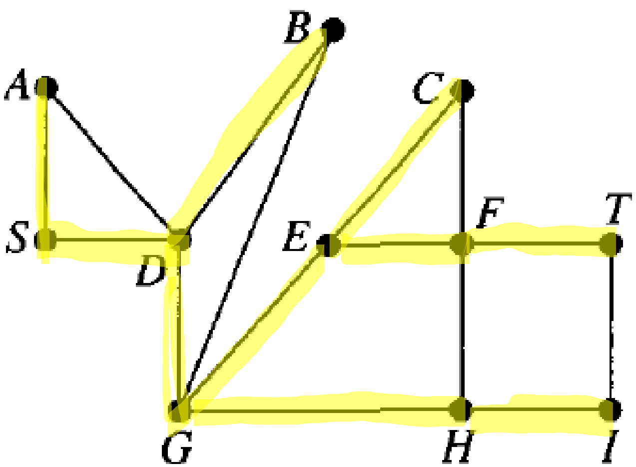


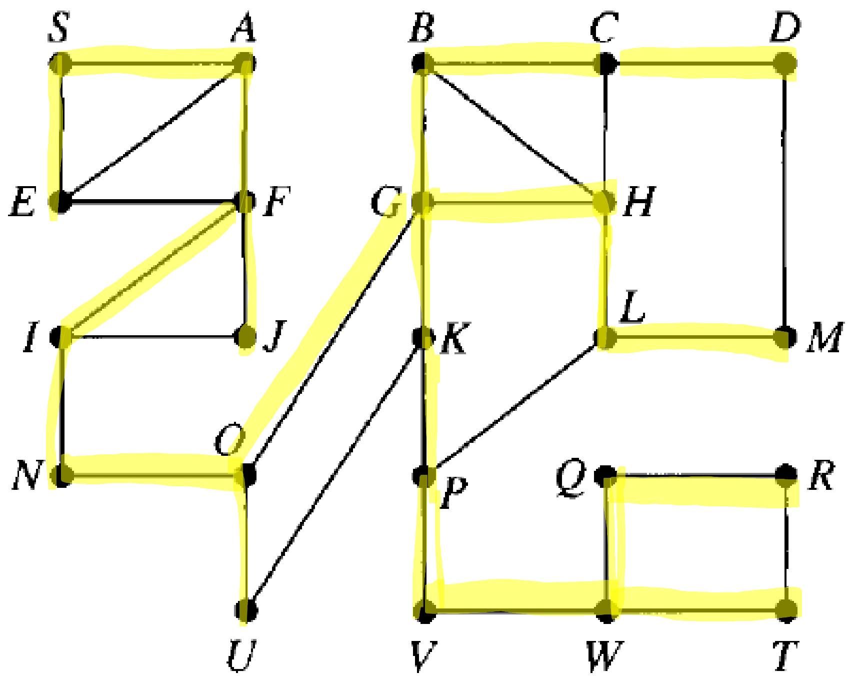


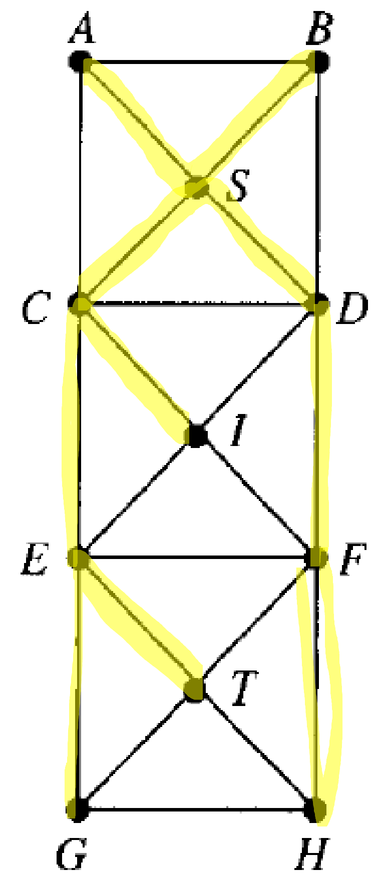




Exercise of BFS







Complexity of Breadth-First Search

Scan vertex i:

scan all the edges (i,j) connecting vertex i

Scan edge (i,j): check whether d[j]=M or not

if yes then update distance label of vertex j

d[j]=d[i]+1; pred[j]=i;

add j into LIST;

Two major operations:

Vertex Selection: each vertex is selected to be scanned exactly once

i.e. total # of vertex selection operations=|V|

Distance Updating: each edge is scanned exactly twice

i.e. total # of edge scanning operations=2|E|

during these 2|E| scanning, at most |E| of them perform distance updating

Complexity: O(|V|+|E|)=O(|E|)

usually O(|E|)=O(|V|^2) but O(|E|) is a better approximation

More about Breadth-First Search

How to select a vertex from LIST?

Queue: FIFO

How to trace a path from vertex s to i?

```
begin
    output i;
    while pred[i] ≠ s do
        i=pred[i];
        output i
    end while
    output s
end
```

This prints out the path from s to i in reverse order

Complexity : O(|V|)

Why the d[j] represents the shortest distance from s to j?

math. induction

Shortest Path on Weighted Graphs

Distance from s to t on a weighted graph = Weight of a path

Negative cycle:

a cycle that has a total weight which is negative

If there is a path from s to t that contains a negative cycle
then there exists NO shortest path from s to t

If there exists a shortest path from s to t
then we can assume such a path is simple path

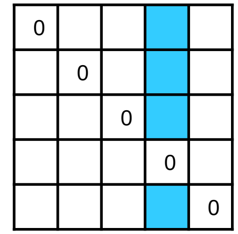
(Thm 4.4 every u-v path contains a u-v simple path)

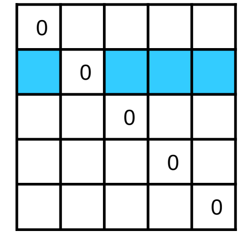
Shortest path problems:
1-1, 1-ALL, ALL-1, ALL-ALL, SOME-SOME

Nonnegative edge length vs. negative edge length

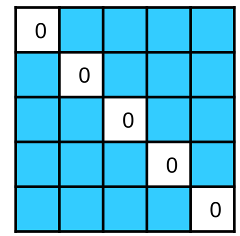
Shortest Path Problems

- Single Source Shortest Path (SSSP)
  - Nonnegative Arc Cost (LS)
  - General Arc Cost (LC)

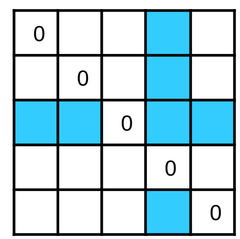


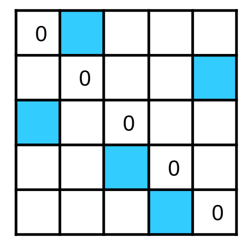


- All Pairs Shortest Paths (APSP)
  - Combinatorial Type Algorithms (repeated SSSP)
  - Algebraic Type Algorithms (FW)
  - LP Type Algorithms (Network simplex)



- Multiple Pairs Shortest Paths (MPSP)





LP form for SSSP Problems

Primal formulation

$$\min \sum_{(i,j)\in E} c_{ij} x_{ij}$$

$$s.t.$$

$$\sum_{(i,j)\in E} x_{ij} - \sum_{(j,i)\in E} x_{ji} = b_i \quad \forall i \in V$$

$$x_{ij} \geq 0 \quad \forall (i,j) \in E$$

Dual formulation

$$\max \sum_{i\in V} \pi_i b_i$$

$$s.t.$$

$$\pi_i - \pi_j \leq c_{ij} \quad \forall (i,j) \in E$$

$$\pi : \text{free}$$

1-1: from s to t

$$b_s = +1,\ b_t = \text{-}1,\ b_i = 0\ \forall i \in V \setminus \{s,t\}$$

1-ALL: from s to all other vertices

$$b_s = +(n-1),\ b_i = \text{-}1,\ \forall i \in V \setminus \{s\}$$

ALL-1: from all vertices except t to t

$$b_t = -(n-1),\ b_i = +1,\ \forall i \in V \setminus \{t\}$$

reduced cost: $c_{ij}^{\pi} = c_{ij} - \pi_i + \pi_j$

optimal condition:

$$c_{ij}^{\pi} \geq 0 \quad \forall (i,j) \in E$$

Dijkstra's Algorithm

Only valid for graphs with nonnegative edge length

1-ALL: from s to all other vertices

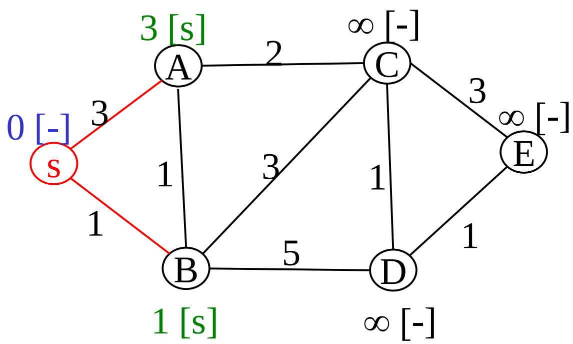
begin
  initialization: $S=\emptyset$; $S^c=V$;
    $d[i]=\infty$, $pred[i]$=NULL for each i in V
    $d[s]=0$, $pred[s]$=NULL;
  while $|S|<n$ **do**
    choose i from S^c such that $d[i]=\min\{d[j]: j \text{ in } S^c\}$;
    $S=S\cup\{i\}$; $S^c= S^c\backslash\{i\}$;
    for each (i,j) in $E[i]$ **do**
      if $d[j]>d[i]+c_{ij}$ **then**
        $d[j]=d[i]+c_{ij}$; $pred[j]=i$;
    end for
  end while
end

Outgoing edge list

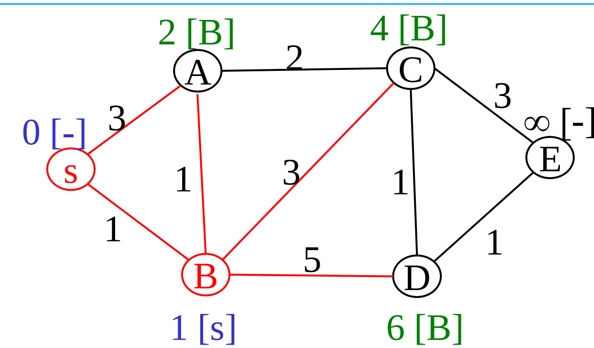
Vertex Selection
select a vertex i to scan
then vertex i becomes
permanently labeled

Scan edge (i,j)
Distance Updating
(triple comparison)

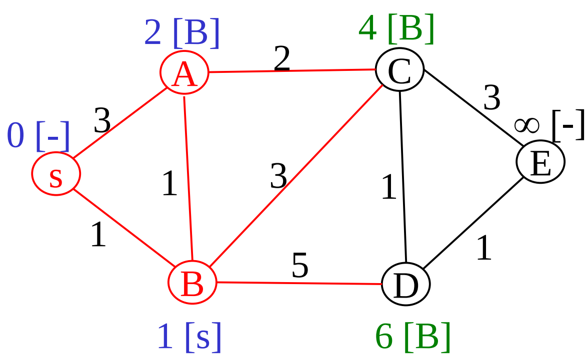
Example of Dijkstra's Algorithm



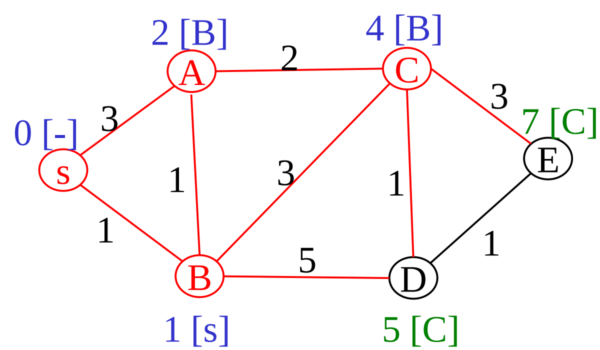
Choose s
S={s}
S^c={A,B,C,D,E}
Scan (s,A), (s,B)



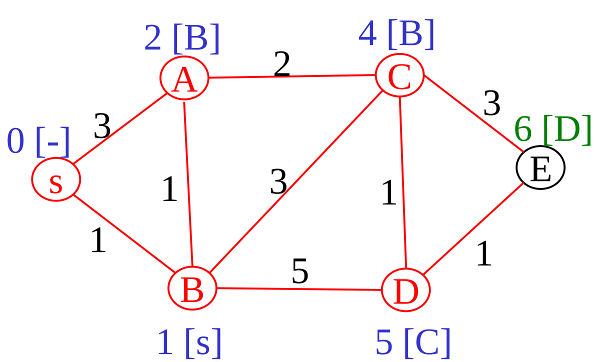
Choose B
S={s,B}
S^c={A,C,D,E}
Scan (B,A),(B,C),(B,D)



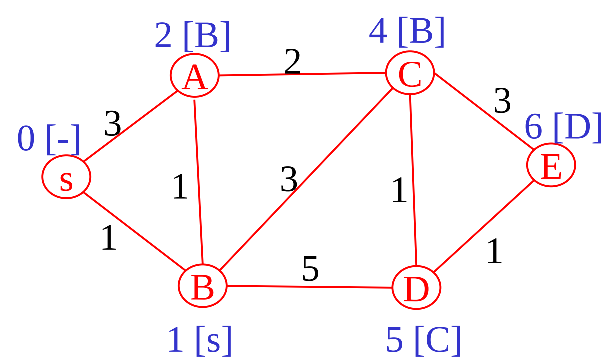
Choose A
S={s,B,A}
S^c={C,D,E}
Scan (A,C)



Choose C
S={s,B,A,C}
S^c={D,E}
Scan (C,D),(C,E)



Choose D
S={s,B,A,C,D}
S^c={E}
Scan (D,E)



Choose D
S={s,B,A,C,D,E}
S^c=∅
DONE!!

Binary Min-Heap

Priority queue: any data structure that supports the operations of search min, insert, and delete min (or max)

Heap: a min heap is a complete binary tree with the property that the value at each vertex is ≦ the values at its children

Suppose there are n elements in the heap:

- Search min: O(1) (i.e. the toppest vertex)
- Insert an element: O(log n)
  put the new element to the last vertex, then compare with its ancestors, swap parents/children when necessary
- Delete min: O(log n)
  delete the toppest vertex, put the last vertex to be the toppest, then compare with its descendants, swap parents/children when necessary

More about Binary Heap

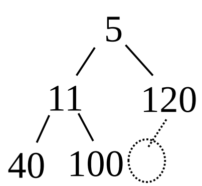
Why is the depth of a binary tree “log n” for a tree of n vertices?

Since we keep “balanced” binary tree, suppose there are n vertices in the binary tree,

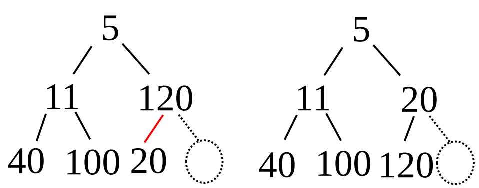
$$n = 1+2+4+8+\ldots+2^{\lfloor \log n \rfloor -1} + (n+1-2^{\lfloor \log n \rfloor -1}), \text{ where } 2^{\lfloor \log n \rfloor} < (n+1-2^{\lfloor \log n \rfloor}) \le 2^{\lfloor \log n \rfloor}$$

e.g. n=12, $\log n = 3.\cdots$, i.e. n=12=1+2+2^2+5, where $2^2 < 5 = 12+1-2^3 \le 2^3$

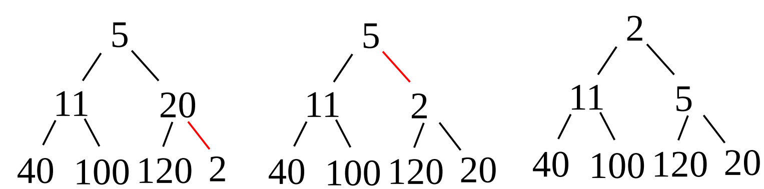
Graphical illustration of inserting & deleting an element in min-heap



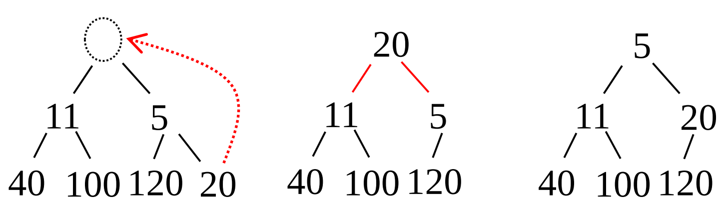
Original min-heap



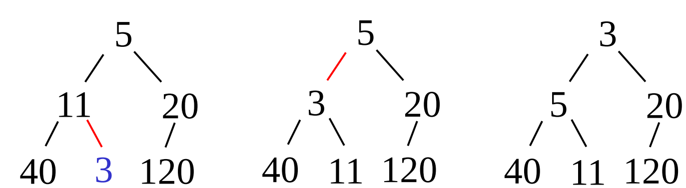
Insert a new element 20



Insert a new element 2



Delete an element 2



Change element 100 to be 3

Complexity of Dijkstra's Algorithm

Scan vertex i:

scan all the outgoing edges (i,j) connecting vertex i (i.e. edges in E[i])
after i is scanned, we say i is permanently labeled
Once a vertex is permanently labeled, its distance label is set and won't be changed.

Scan edge (i,j): check whether $d[j]>d[i]+c_{ij}$ or not

if yes then update distance label of vertex j
$d[j]= d[i]+c_{ij}$; $pred[j]=i$;

Two major operations:

Vertex Selection: each vertex is selected to be scanned exactly once
however, effort to choose the vertex with minimal distance label:
by linear search: $|V|+|V-1|+|V-2|+\ldots+1= O(|V|^2)$
by binary heap: $O(|V|\log|V|)$

Distance Updating: each outgoing edge is scanned exactly once
i.e. total # of edge scanning operations= $|E|$
using binary heap: after each updating, need to resort the data structure, each sorting takes $O(\log|V|)$, so totally takes $O(|E|\log|V|)$

Complexity: $O(|V|^2+|E|)$, or $O((|V|+|E|)\log|V|)$

There are many other variants of Dijkstra's algorithm. Most of them use different techniques to do the vertex selection.

Other Implementations of Dijkstra’s algorithm

Variants of Dijkstra’s algorithm:

Dial’s bucket implementation (for integral edge length)

nC+1 unit-length buckets to store temporarily labeled vertices ($C=\max\{c_{ij}\}$)

bucket k stores vertices with distance label k (using doubly linked list)

idea: 1. choose the 1st nonempty bucket;

2. scan all vertices in the bucket one by one;

3. redistribute vertices into new buckets if their distance labels are decreased;

e.g. In iteration 5, d[E]=9 (i.e. put vertex E in the bucket 9), if d[E] becomes 7

in iteration 7, we have to move vertex E from bucket 9 to bucket 7

4. remove any scanned vertices;

5. then choose the next nonempty bucket, repeat steps 1~4 until all buckets are checked

Bucket									….
Distance label	0	1	2	3	4	5	6	7	….

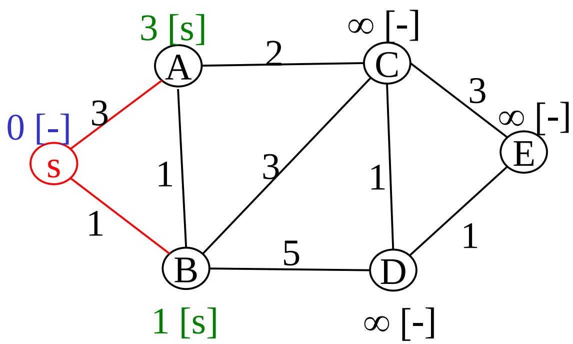
Complexity: O(m+nC)

Radix Heap implementation

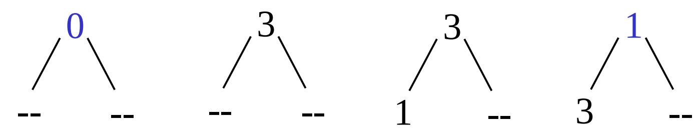
more complicated data structure (not covered in this course)

Binary Heap & Dial's implementation-1

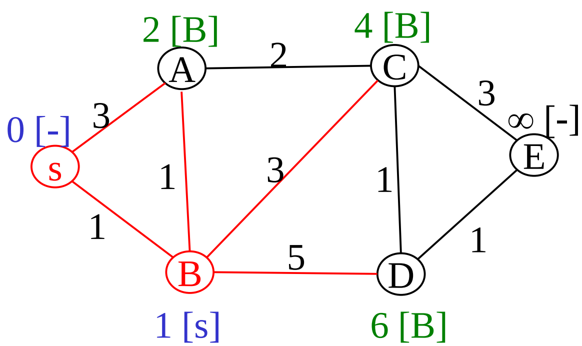
Binary min-heap: parent.key ≦ child.key
key: distance label



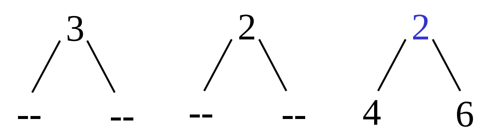
Choose s
S={s}
S^c={A,B,C,D,E}
Scan (s,A), (s,B)



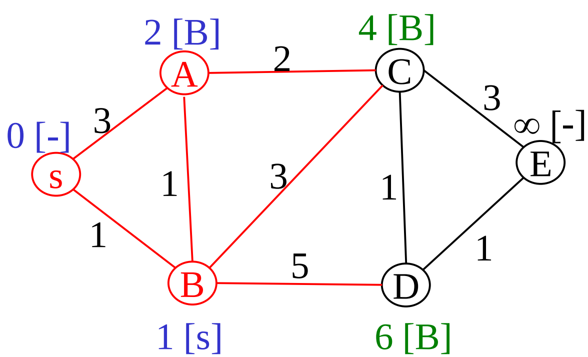
Bucket	s	B		A				
Distance label	0	1	2	3	4	5	6	7



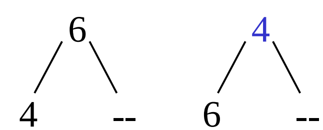
Choose B
S={s,B}
S^c={A,C,D,E}
Scan (B,A),(B,C),(B,D)



Bucket			A		C		D	
Distance label	0	1	2	3	4	5	6	7

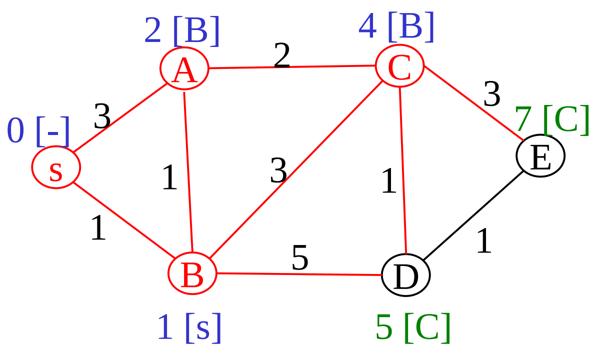


Choose A
S={s,B,A}
S^c={C,D,E}
Scan (A,C)



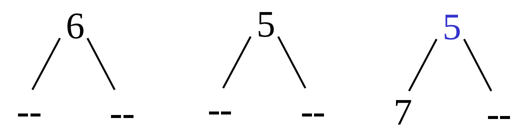
Bucket					C		D	
Distance label	0	1	2	3	4	5	6	7

Binary Heap & Dial's implementation-2

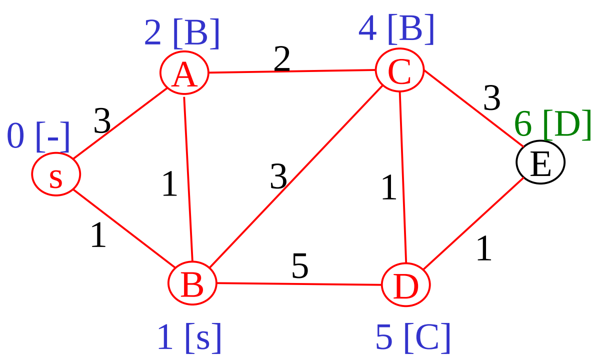


Choose C
S={s,B,A,C}
S^c={D,E}
Scan (C,D),(C,E)

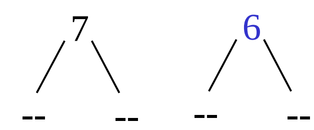
Binary min-heap: parent.key ≦ child.key



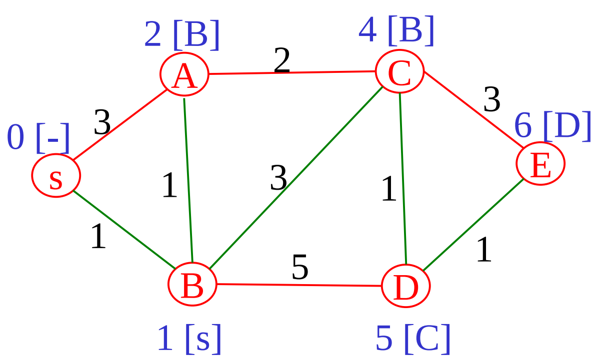
Bucket						D		E
Distance label	0	1	2	3	4	5	6	7



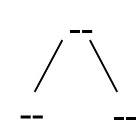
Choose D
S={s,B,A,C,D}
S^c={E}
Scan (D,E)



Bucket							E	
Distance label	0	1	2	3	4	5	6	7



Choose D
S={s,B,A,C,D,E}
S^c=∅
DONE!!



Bucket								
Distance label	0	1	2	3	4	5	6	7

Trace Paths
s-B-A
s-B
s-B-C
s-B-C-D
s-B-C-D-E

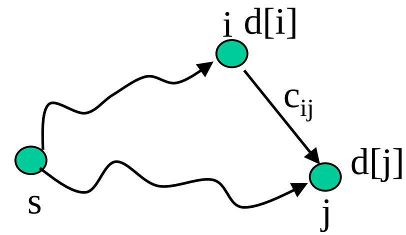
Triple Comparisons

Distance Updating:

when we scan vertex i, we check each outgoing edges (i,j) as follows:

if d[j]>d[i]+c_{ij} **then**

d[j]=d[i]+c_{ij}; pred[j]=i;



such an operation is called a "triple comparison"

Remember in the dual form of SSSP

reduced cost: $c_{ij}^{\pi} = c_{ij} - \pi_i + \pi_j$

optimal condition:

$$c_{ij}^{\pi} \geq 0 \quad \forall (i,j) \in E$$

By defining d[i]=-π_i, the above become

reduced cost: $c_{ij}^{\pi} = c_{ij} + d[i] - d[j]$

optimal condition:

$$c_{ij}^{\pi} \geq 0 \quad \forall (i,j) \in E \quad \Leftrightarrow \quad d[j] \leq d[i] + c_{ij} \quad \forall (i,j) \in E$$

Thus, triple comparison is an operation to achieve the LP optimal condition.

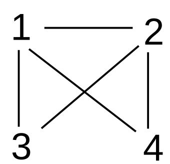
- All Shortest Path algorithms perform sequences of triple comparisons.

Number of Paths

How many paths of p edges between 2 specific vertices?

Thm 4.7 Given a graph G with vertices V, edges E
the # paths of p edges from vertex i to vertex j is the $(i,j)^{th}$ entry of A^p, where A is the adjacency matrix

Ex:



$$A=\begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&0\\1&1&0&0\end{bmatrix}, A^2=\begin{bmatrix}3&2&1&1\\2&3&1&1\\1&1&2&2\\1&1&2&2\end{bmatrix}, A^3=\begin{bmatrix}4&5&5&5\\5&4&5&5\\5&5&2&2\\5&5&2&2\end{bmatrix}$$

Paths of 3 edges from 3 to 4: 3-1-2-4, 3-2-1-4

$$A^3_{3,4} = A^1_{3,1}A^2_{1,4} + A^1_{3,2}A^2_{2,4} = A^1_{3,1}(A^1_{1,2}A^1_{2,4}) + A^1_{3,2}(A^1_{2,1}A^1_{1,4})$$

Paths of 3 edges from 4 to 2: 4-1-3-2, 4-1-4-2, 4-2-1-2, 4-2-3-2, 4-2-4-2

$$A^3_{4,2} = A^1_{4,1}A^2_{1,2} + A^1_{4,2}A^2_{2,2} = A^1_{4,1}(A^1_{1,3}A^1_{3,2} + A^1_{1,4}A^1_{4,2})$$
$$+ A^1_{4,2}(A^1_{2,1}A^1_{1,2} + A^1_{2,3}A^1_{3,2} + A^1_{2,4}A^1_{4,2})$$

Path Algebra

Path algebra: an ordered semiring $(S,\oplus,\otimes,e,\varnothing,\leq)$ with $S=\mathbb{R}\cup\{\infty\}$ with 2 binary operations defined as

	Generalized addition	Generalized multiplication	Unit element	Null element
Original sense	a ⊕ b	a ⊗ b	e	0
Path algebra	min{a,b}	a+b	0	∞

Measure matrix: C, where C_{ij} represents the length of edge (i,j)

C_{ii} =0 for all i; C_{ij}=∞ , if there exists no edge (i,j)

$$C^2 = C \otimes C,$$

$$C_{i,j}^2 = (C_{i,1} \otimes C_{1,j}) \oplus (C_{i,2} \otimes C_{2,j}) \oplus \ldots \oplus (C_{i,|V|-1} \otimes C_{|V|-1,j}) \oplus (C_{i,|V|} \otimes C_{|V|,j})$$

$$= \min\{C_{i,1}+C_{1,j}, C_{i,2}+C_{2,j}, \ldots, C_{i,|V|-1}+C_{|V|-1,j}, C_{i,|V|}+C_{|V|,j}\}$$

|V| addition
|V|-1 comparison

$$= \text{length of a shortest path from i to j passing at most 2 edges}$$

Compute APSP by $C^{|V|-1}$

$C_{i,j}^{k}$ = length of a shortest path from i to j passing at most k edges

Every simple path between 2 vertices at most contains |V|-1 edges

Thus

$$C_{i,j}^{|V|-1} = \text{length of a shortest path from i to j}$$

By computing matrix $C^{|V|-1}$, we can obtain ALL-ALL shortest paths

How to efficiently compute $C^{|V|-1}$?

$$C \to C^2 \to C^4 \to \ldots \to C^{2^{\lceil \log(|V|-1) \rceil}}$$

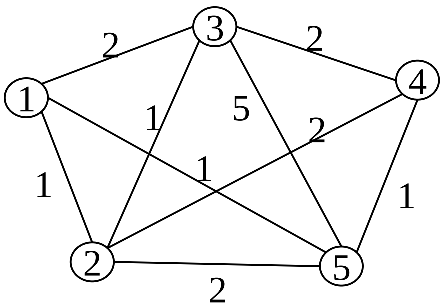
Totally requires $\lceil \log(|V|-1) \rceil$ times of matrix multiplication

Each matrix multiplication takes $O(|V|^3)$ ← $O(|V|^2 * |V|)$ since there are $|V|^2$ entries, each takes $O(|V|)$

Thus totally takes $O(|V|^3 \log|V|)$ time

There are many fast-matrix-multiplication techniques that can be used to improve this complexity, but such algorithms are complicated.

Example of Matrix Multiplication



$$C^{2^{\lceil \log(5-1) \rceil}} = C^{2^2} = C^4$$

$$C = \begin{bmatrix} 0 & 1 & 2 & \infty & 1 \\ 1 & 0 & 1 & 2 & 2 \\ 2 & 1 & 0 & 2 & 5 \\ \infty & 2 & 2 & 0 & 1 \\ 1 & 2 & 5 & 1 & 0 \end{bmatrix}, C^2 = \begin{bmatrix} 0 & 1 & 2 & 2 & 1 \\ 1 & 0 & 1 & 2 & 2 \\ 2 & 1 & 0 & 2 & 3 \\ 2 & 2 & 2 & 0 & 1 \\ 1 & 2 & 3 & 1 & 0 \end{bmatrix}, C^4 = \begin{bmatrix} 0 & 1 & 2 & 2 & 1 \\ 1 & 0 & 1 & 2 & 2 \\ 2 & 1 & 0 & 2 & 3 \\ 2 & 2 & 2 & 0 & 1 \\ 1 & 2 & 3 & 1 & 0 \end{bmatrix}$$

$$C^2_{1,2} = \min\{0+1, 1+0, 2+1, \infty+2, 1+2\} = 1$$

$$C^2_{1,4} = \min\{0+\infty, 1+2, 2+2, \infty+0, 1+1\} = 2$$

$$C^2_{3,5} = \min\{2+1, 1+2, 0+5, 2+1, 5+0\} = 3$$

Floyd-Warshall Algorithm

Let D be a n×n distance matrix, whose entry D_{ij} is initially set to be C
final D_{ij} denotes the length of the shortest path from i to j
PRED be a n×n predecessor matrix, whose entry $PRED_{ij}$ is initially set to be i
final $PRED_{ij}$ denotes the predecessor of j on the path from i to j

Triple comparison : $d_{ij}=\min\{d_{ij}, d_{ik}+d_{kj}\}$

Claim:

if we perform triple comparisons for successive k=1,2,…,|V|
then the final d_{ij} equals the length of the shortest path from i to j

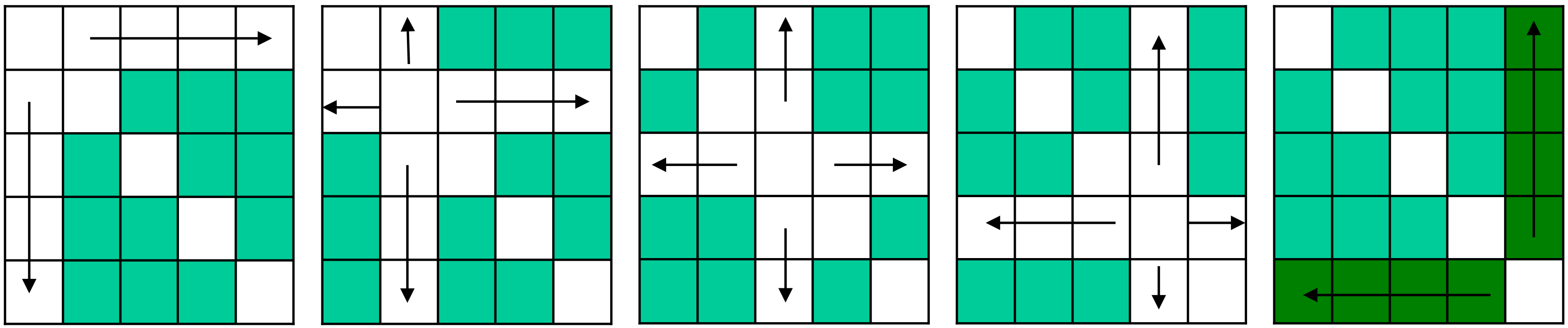
Floyd-Warshall Algorithm is based on the above claim

begin
  initialization: D:=C; $PRED_{ij}$:=i for each (i,j);
  for k=1 to |V| **do**
    for i=1 to |V| **do**
      for j=1 to |V| **do**
        if $d_{ij}>d_{ik}+d_{kj}$ **then**
          $d_{ij}=d_{ik}+d_{kj}$; $PRED_{ij}=PRED_{kj}$;
end

Properties of Floyd-Warshall Algorithm

Let D_{ij}^{k} denote the entry D_{ij} after the k^{th} iteration of Floyd-Warshall algorithm

D_{ij}^{k} is the length of shortest path from i to j with intermediate vertices in {1,2,…,k-1,k}

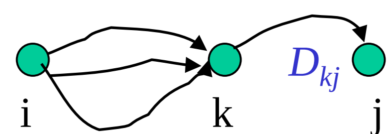


Totally $O(|V|^3)$ operations, most efficient for complete graph

Detect negative cycle: if $D_{ii}^{k}<0$, there is a neg. cycle with intermediate nodes in {1,…,k}

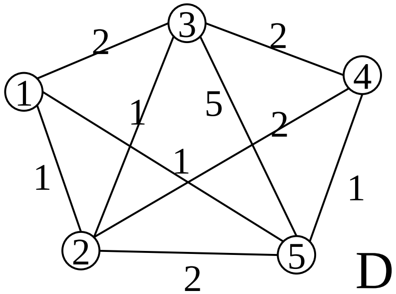
Dynamic Programming:

Bellman's Equation: $$D_{ij} = \begin{cases} \min\limits_{k \neq i, j} \left(C_{ik} + D_{kj} \right) & , i \neq j \\ 0 & , i = j \end{cases} \quad \Leftrightarrow \quad D = C \otimes D \oplus I_n$$



Floyd-Warshall algorithm ⇔ Gauss Jordan method

Example of APSP problems



D_k: distance matrix after the k^{th} iteration
$PRED_k$: predecessor matrix after the k^{th} iteration

After the $|V|-1^{th}$ iteration, the last column & row are already optimal

D^1

0	1	2	∞	1
1	0	1	2	2
2	1	0	2	3
∞	2	2	0	1
1	2	3	1	0

D^2

0	1	2	3	1
1	0	1	2	2
2	1	0	2	3
3	2	2	0	1
1	2	3	1	0

D^3

0	1	2	3	1
1	0	1	2	2
2	1	0	2	3
3	2	2	0	1
1	2	3	1	0

D^4

0	1	2	3	1
1	0	1	2	2
2	1	0	2	3
3	2	2	0	1
1	2	3	1	0

D^5

0	1	2	2	1
1	0	1	2	2
2	1	0	2	3
2	2	2	0	1
1	2	3	1	0

$PRED^1$

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

$PRED^2$

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

$PRED^3$

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

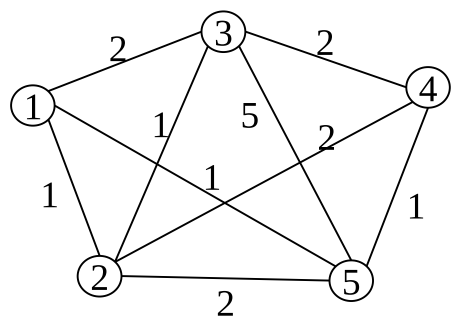
$PRED^4$

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

$PRED^5$

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

Trace Shortest Paths by PRED matrix



$PRED^5$

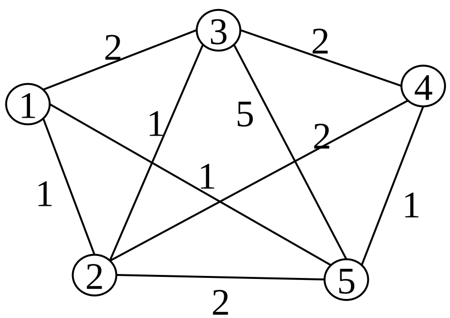
1	1	1	5	1
2	2	2	2	2
3	3	3	3	1
5	4	4	4	4
5	5	1	5	5

Trace_path(PRED,i,j)
begin
    k=j;
    output k;
    while $PRED_{ik} \neq i$ **do**
        $k=PRED_{ik}$;
        output k
    end while
    output i
end

trace path 1→4:   i=1,j=4
k=j=4
output 4
Since $PRED_{14}=5 \neq 1$
    $k= PRED_{14}=5$
    output 5
Since $PRED_{15}=1 = 1$
output 1
Thus we obtain 4←5←1 as the shortest path

Coloring a Graph

How many colors do we need to color each vertex such that its adjacent vertices have different color?

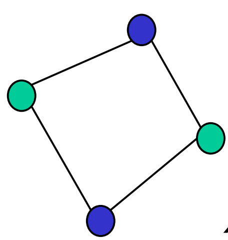


|V| vertices ➔ we can use |V| or more colors

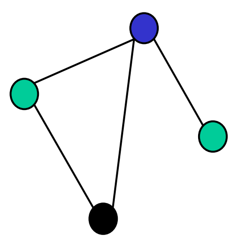
- Can we use fewer colors?
- How to determine whether k colors are enough or not?
- Can we do it efficiently?

- If we can color a graph with k colors, we say the graph is k-colorable
- The smallest k for which the graph is k-colorable is called the chromatic number of the graph, denoted by $\chi(G)$

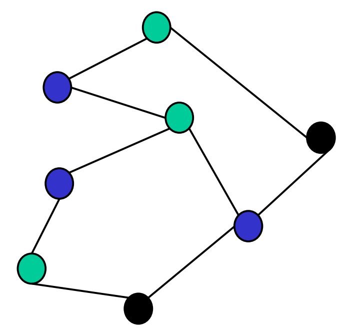
E.g.



$\chi = 2$



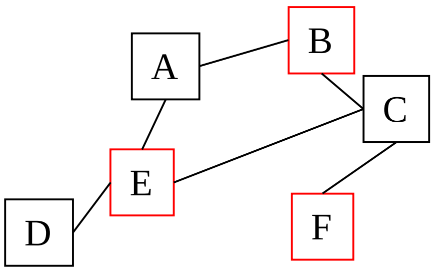
$\chi = 3$



$\chi = 3$

Colors on Even cycle, Odd cycle

Ex: Jim has 6 children: {A,B,C,D,E,F}. Among these 6 kids,
C fights with B,F,E; E fights (besides C) with A,D; A fights with B
How many rooms do Jim at least need to put all 6 kids so that there will be no fights inside each room?



At least 2 colors
➔ 2 rooms: {A,C,D}, {B,E,F}

Observation:

- If G is k-colorable, then G is (k+1)-colorable
- Isolated vertices are 1-colorable
- Bipartite graphs are 2-colorable
- A complete graph K_n has chromatic number n
- A cycle of even # of vertices (even cycle) can be colored with 2 colors
- An odd cycle can be colored with 3 colors
  ➔ if a graph contains odd cycles, then it can not be 2-colorable

2-colorable Graphs

Thm 4.8

A graph G is 2-colorable iff it contains no odd cycle

Pf:➔: if a graph contains odd cycles, then it can not be 2-colorable (known already)

🡐: without loss of generality (w.l.o.g.), we assume G is connected (i.e. the discussion can be applied to any component of G).

Suppose G has no odd cycle. Choose any vertex s, apply BFS algorithm starting from s. Each vertex j will have a distance label d[j]. Let's color all vertices with even distance label by red color, and blue color, otherwise. Now we want to show any adjacent vertices have different color by contradiction.

By the BFS algorithm, we know that for any adjacent vertices i and j, |d[i]-d[j]| = 0 or 1. Suppose there exists an edge between 2 vertices, say, i & j, of the same color, which means i & j are adjacent and d[i]=d[j]. Using the predecessor info by the BFS algorithm, we can trace both the path backwards from i to s, and from j to s to identify the first common "ancestor" vertex W. Thus we identify a cycle from i to W to j with edges= (d[j]-d[W])+(d[i]-d[W])+1=2(d[j]-d[W])+1, which is odd.

This means, if color[i]=color[j], we will have an odd cycle →←

Therefore, using BFS algorithm to do the 2-coloring on a graph G that has no odd cycle, any adjacent vertices will have different color.

Algorithms to find Chromatic number

There exists NO efficient algorithm to find the chromatic number for general graphs, except for the 2-colorable graphs.

The proof of Thm 4.8 gives a polynomial-time algorithm (i.e. BFS) to

- Find a 2-coloring, if it exists
- Find an odd cycle, if the 2-coloring does not exist

For general graphs, there are theorems to give upper bounds on $\chi(G)$

Thm 4.9 (by Brooks)
if every vertex of G has degree at most d, then G is (d+1)-colorable

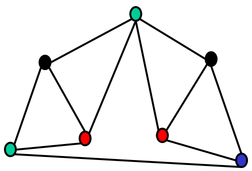
Pf: Trivial if G has fewer than or equal to (d+1) vertices.
Suppose the thm holds for a graph G of k (>d+1) vertices. In a graph G with k+1 vertices, we can pick any vertex v, and construct a new graph G' by deleting v and all its adjacent edges (at most d such edges) from G. By induction hypothesis, G' is (d+1)-colorable. Thus we can use d+1 colors to color all of v's neighbors. Since v has at most d neighbors, we can use d colors to color all of v's neighbors, and color v by the remaining color. Therefore, G is still (d+1)-colorable. This completes the induction.

More about Coloring

Observation:

if G contains a K_p subgraph, then G is not (p-1)-colorable

Q: if G does not contain a K_p subgraph, then G is (p-1)-colorable?



False!

The graph in the left does not contain a K_4, but it is NOT 3-colorable

- Chromatic number is graph isomorphism invariant

Coloring maps: there are countries on a planar map, adjacent countries must be colored by different colors. How many colors are enough?

country ⇔ vertex,
country i is adjacent to country j ⇔ draw an edge (i,j)

Four-Color Thm:

A graph without crossover edges

if G is a planar graph, then $\chi(G) \leq 4$ (i.e. at most 4)

Conjectured by Francis Guthrie in 1852, but verified by Kenneth Appel & Wolfgang Haken in 1976

Exercise on Graph Coloring

Ex1: suppose G is a graph with 3 vertices. How many ways are there to assign 3 colors to the vertices? 3^3

Ex2: show that if a graph with n vertices has chromatic number n, then the graph has n(n-1)/2 edges

If not a K_n, remove any 2 nonadjacent vertices, the remaining graph is n-2 colorable
(since a graph of k nodes should be k-colorable);
then add these 2 vertices back with a new color, which make the chromatic number=n-1, -><-

Directed Graphs

Directed graph (digraph): contains a finite nonempty set V and a set E of ordered pairs of distinct elements of V.

Directed edge (i,j): an edge from vertex i to vertex j (not the other way)

Outdegree of vertex v: outdeg(v), # directed edges outgoing from v

Indegree of vertex v: indeg(v), # directed edges incoming to v

Thm 4.10 in a directed graph G=(V,E)

$$\sum_{v\in V} indeg(v) = \sum_{v\in V} outdeg(v) = |E|$$

Adjacency matrix A(G) & Adjacency lists are defined as before

Thm 4.11 Let $A(G)_{ij}$ denotes the (i,j)th entry of A(G)

$$\sum_{j=1}^{|V|} A(G)_{ij} = outdeg(i) \quad \forall i \in V; \qquad \sum_{i=1}^{|V|} A(G)_{ij} = indeg(j) \quad \forall j \in V$$

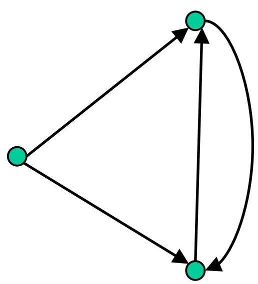
Directed Multigraphs

Directed Multigraph: a Multigraph whose edges are directed

Thm 4.12

Every u-v directed path contains a u-v simple directed path

A directed multigraph is strongly connected if there exists a directed path from any vertex to any other vertex



Connected, but NOT strongly connected

Thm 4.13 For a connected directed multigraph G (with $|V|\geqq 2$)

(a) each vertex v in G has outdeg(v)=indeg(v) iff
G contains a directed Euler circuit

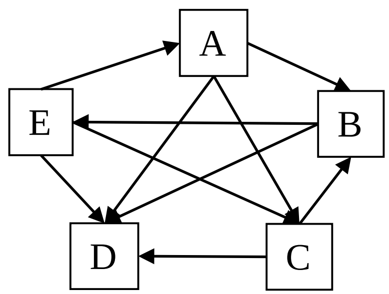
(b) each vertex in G has outdeg(v)=indeg(v) except for 2 distinct vertices s & t
where outdeg(s)=indeg(s)+1, indeg(t)=outdeg(t)+1 iff
G contains an directed Euler path from s to t

Tournament

Tournament: a digraph T where for every 2 distinct vertices u and v of T, exactly one of (u,v), (v,u) is an edge

Score of v in a tournament: s(v)=outdeg(v)

Score sequence of T: list of outdegrees in nonincreasing order



Score: s(A)=3,s(B)=2,s(C)=2,s(D)=0,s(E)=3
Score sequence: 3,3,2,2,0

Round-robin competition in sport
A beats BCD, B beats DE, C beats BD, E beats ACD

find a ranking ➔ find a Hamiltonian path
EACBD is a directed Hamiltonian path

Thm Every tourrnament has a directed Hamiltonian path

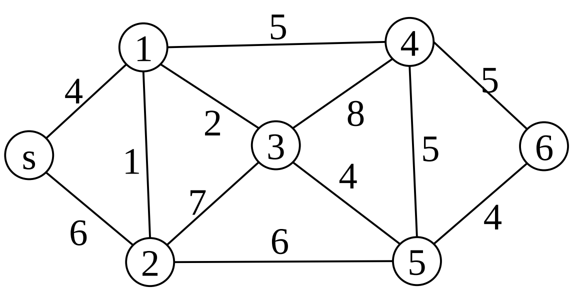
A tournament is transitive iff whenever (u,v) & (v,w) exists, then (u,w) also exists

Thm the following are equivalent
(1) T has a unique directed Hamiltonian path
(2) T is transitive
(3) Every player in T has a different score

Exercises

p.211 example 4.40
p.213 ex.29,30,31,32
p.214 ex.32,34
p.215 ex.55
p.218 ex.76

Now, do the following new problem set:



Given the graph as left, use Dijkstra's algorithm to compute all the shortest paths starting from vertex s, also trace all these shortest paths by the predecessors.