알고리즘

02. Graph

2014/10/08

미디어소프트웨어학과
민경하

Contents

0. Prologue

1. Divide & conquer

2. Graph

3. Greedy algorithm

4. Dynamic programming

2. Graph

2.0 Introduction

2.1 Why graph?

2.2 Depth-first search in undirected graph

2.3 Depth-first search in directed graphs

2.4 Strongly connected components

2.5 Distances

2.6 Breadth-first search

2.7 Single source shortest path

2.8 All pairs shortest path
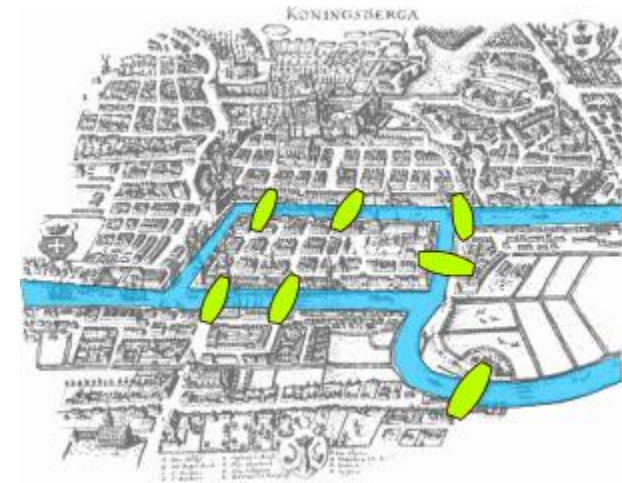
2.0 Introduction

- Graph
  - An abstract representation of a set of objects where some pairs of the objects are connected by links
  - The interconnected objects are called **vertices**, and the links that connect some pairs of vertices are called **edges**
  - Depicted in diagrammatic form as a set of dots for the vertices, joined by lines or curves for the edges

    G=(V, E)
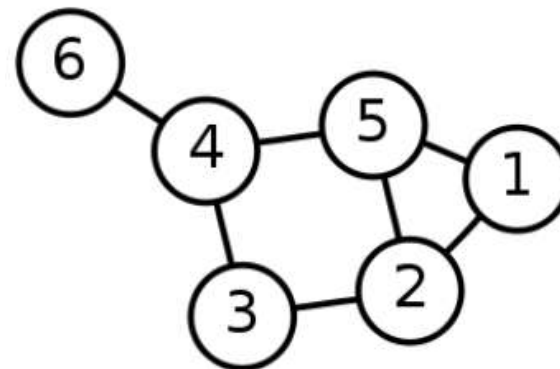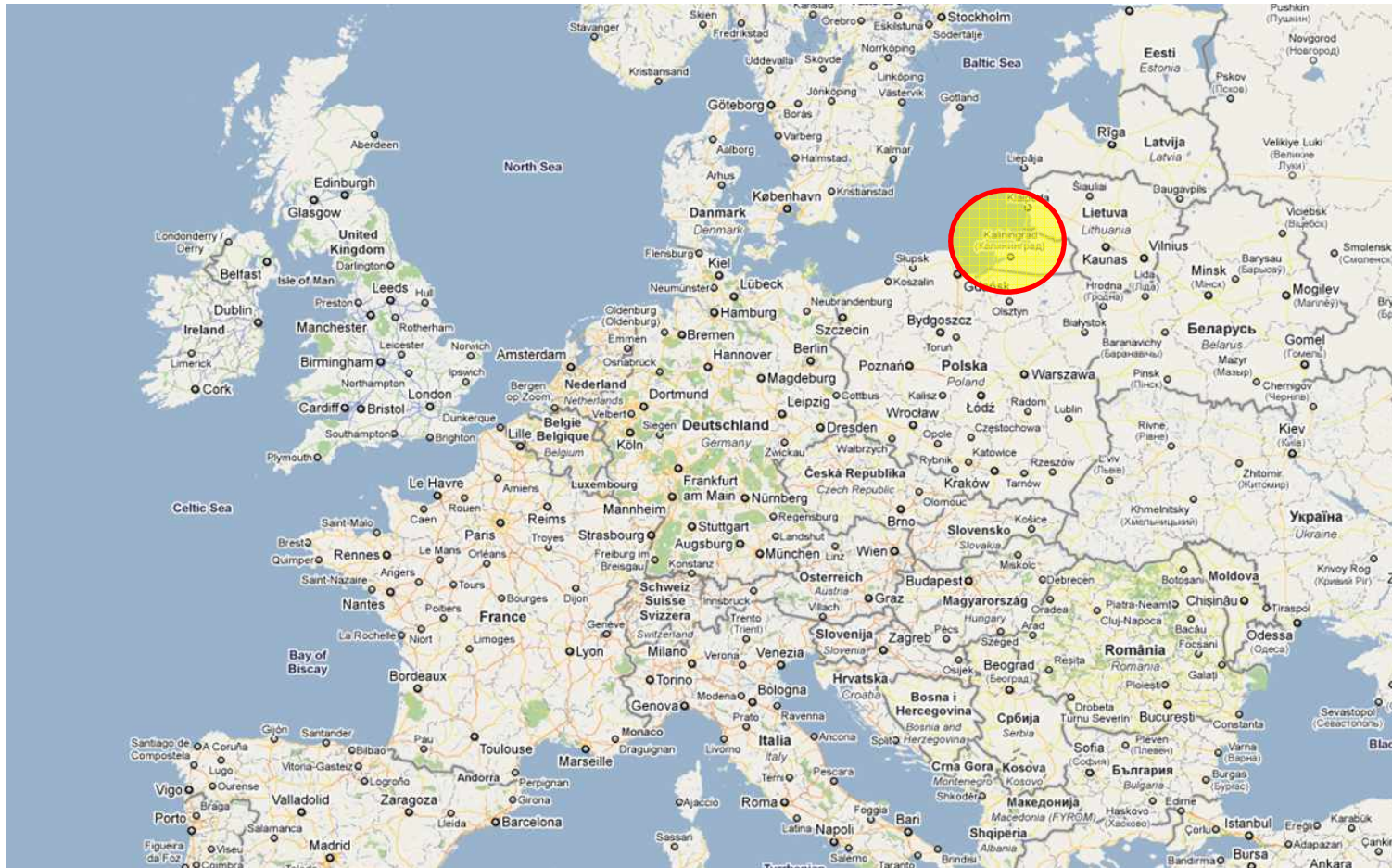
2.0 Introduction

- ## Konigsberg bridge problem
  - The city of Königsberg in Prussia was set on both sides of the Pregel River, and included two large islands which were connected to each other and the mainland by seven bridges.
  - To find a walk through the city that would cross each bridge once and only once

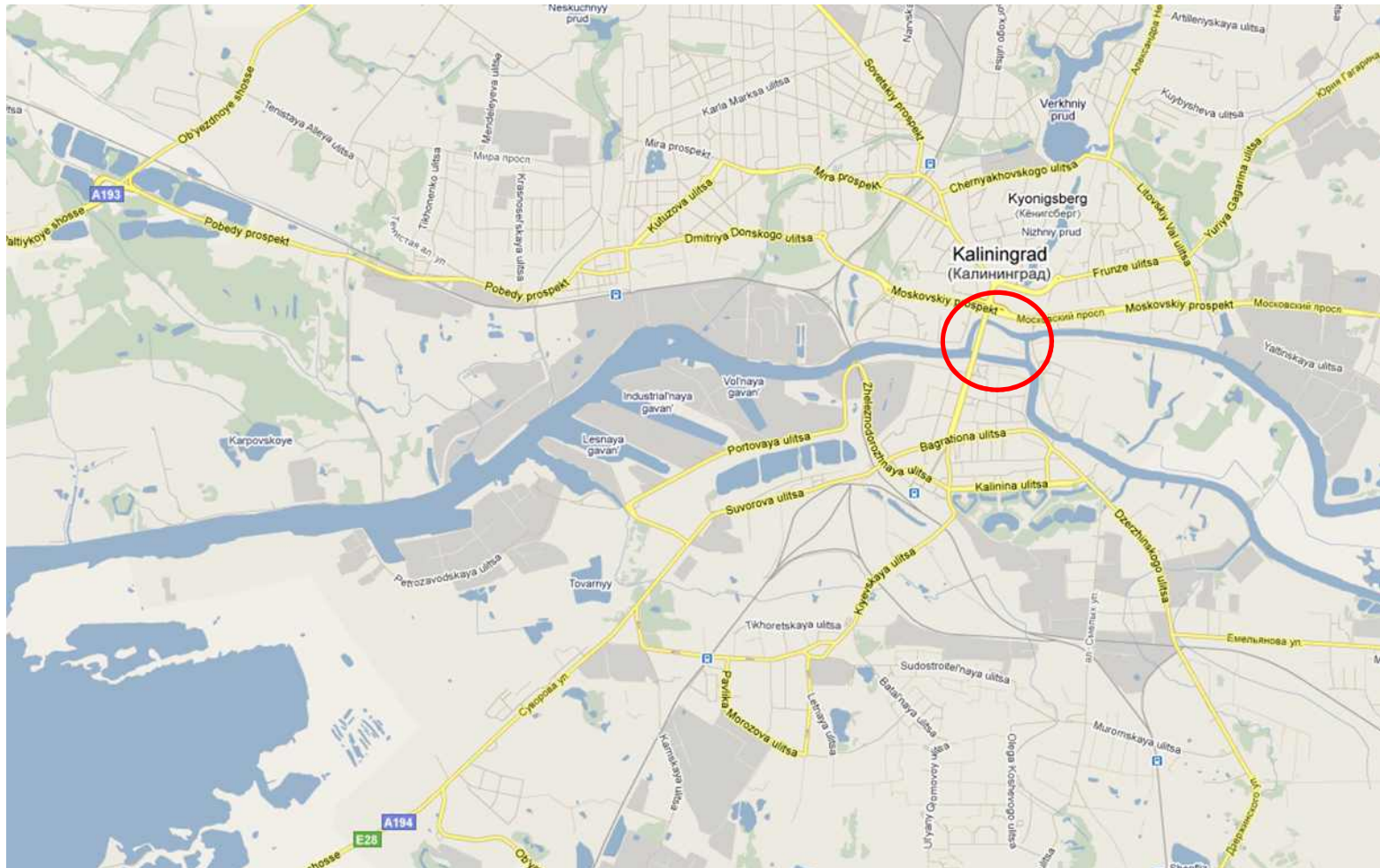 - Euler was invited to attack!!
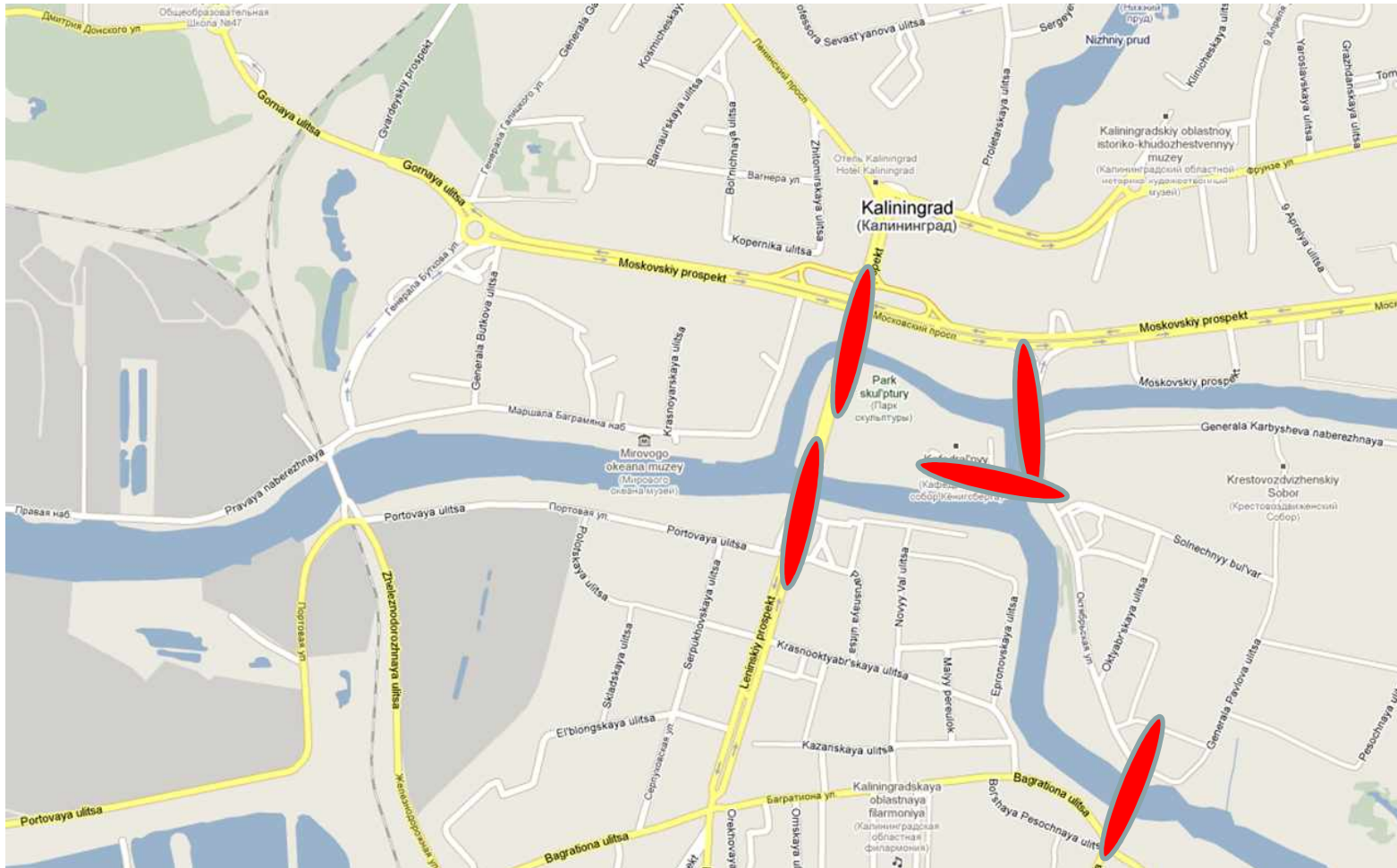
2.0 Introduction

- Konigsberg bridge problem

2.0 Introduction

- Konigsberg bridge problem

2.0 Introduction

- Konigsberg bridge problem

2.0 Introduction

- Euler's solution doesn't matter
  - First, Euler pointed out that the choice of route inside each land mass is irrelevant.
  - The only important feature of a route is the sequence of bridges crossed. This allowed him to reformulate the problem in abstract terms, eliminating all features except the list of land masses and the bridges connecting them.

2.0 Introduction

- Euler's solution doesn't matter
  - In modern terms, one replaces each land mass with an abstract "vertex" or node, and each bridge with an abstract connection, an "edge", which only serves to record which pair of vertices (land masses) is connected by that bridge.
  - The resulting mathematical structure is called a graph.
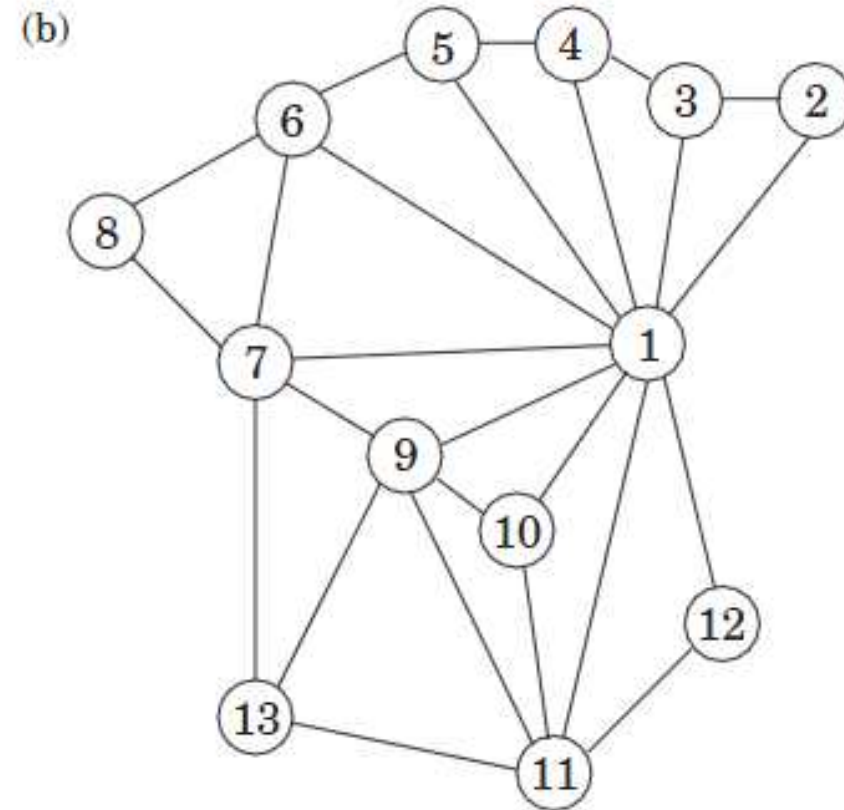
2.1 Why graph?

(1) The merit of a graph

- A wide range of problems can be expressed clearly and precisely by using graphs.

- Ex: Coloring a map

  - What is the minimum number of colors needed, with the obvious restriction that neighboring countries should have different colors?

- Obstacle: the map itself

- Sol:

  - Convert the map into a graph

    - Country → vertex
    - Neighbors → edge

2.1 Why graph?

(a) Map of South America
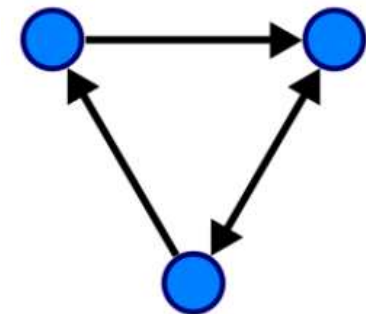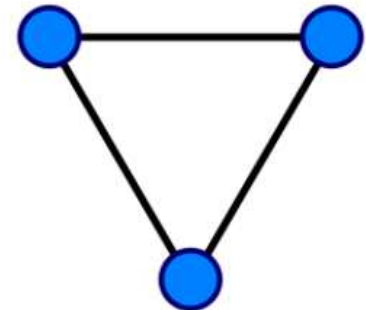
(b) Graph representation

2.1 Why graph?

(2) Definition of a graph

- specified by a set of vertices (also called *nodes*) V and by edges E between select pairs of vertices.
- G = (V, E)
- Ex: G in the previous page
  - V = {1, 2, …, 13}
  - E = { {1, 2}, {1, 3}, … }

(3) The type of a graph

- An undirected graph
  - {v, w} = {w, v}
- A directed graph
  - (v, w) ≠ (w, v)

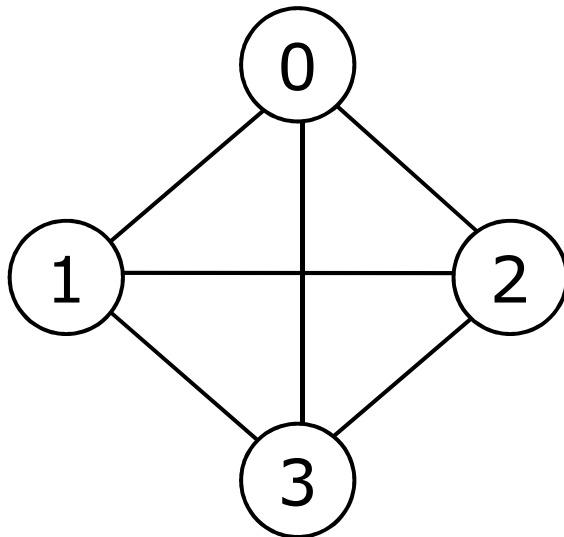## (4) Representation of a graph

(4.1) Adjacency matrix of G = (V, E)

- A two-dimensional n X n array: a[n][n]
- a[i][j] = 1, if $(v_i, v_j) \in E$
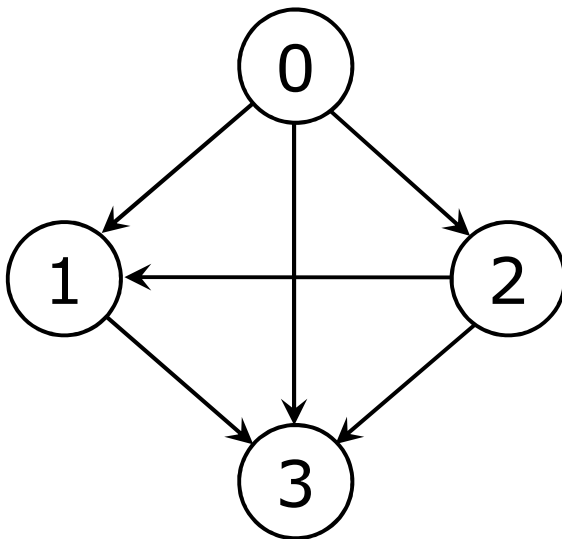- a[i][j] = 0, if $(v_i, v_j) \,!\!\in E$

(4) Representation of a graph

(4.1) Adjacency matrix of G = (V, E)

- A two-dimensional n X n array: a[n][n]
- a[i][j] = 1, if $<v_i, v_j> \in E$
- a[i][j] = 0, if $<v_i, v_j> !\in E$

(4) Representation of a graph

(4.2) Adjacency list of G = (V, E)

- adjLists[n]
- adjLists[i] is a pointer to the first node in the adjacency list for vertex i

2.1 Why graph?

(4) Representation of a graph
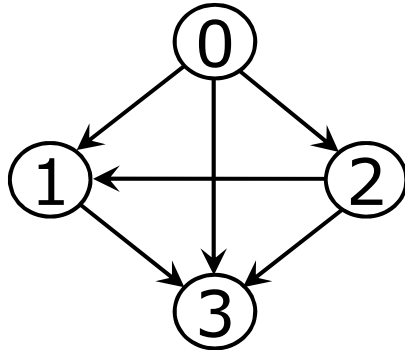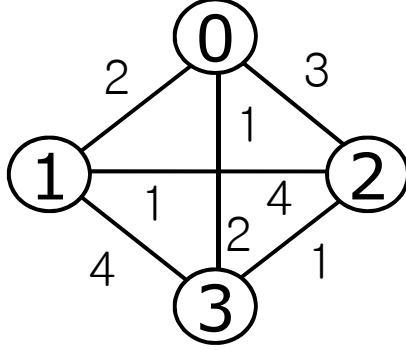
(4.2) Adjacency list of G = (V, E)

- adjLists[n]
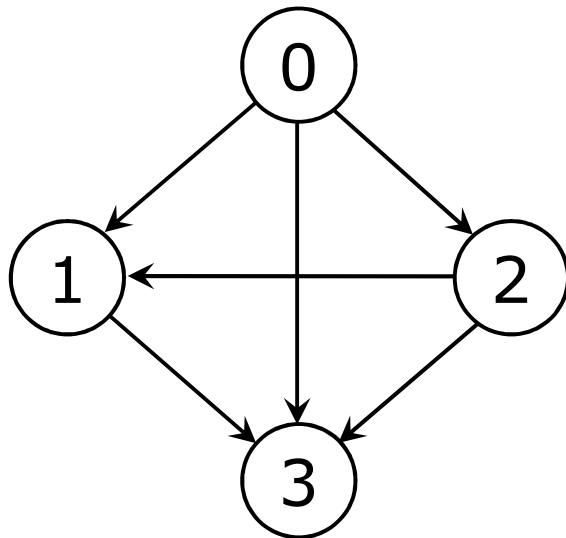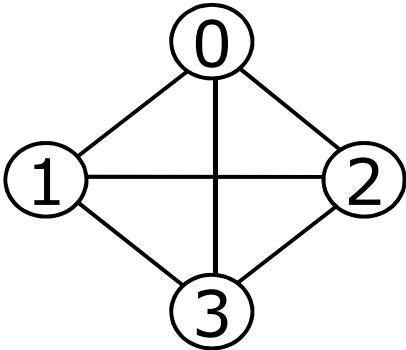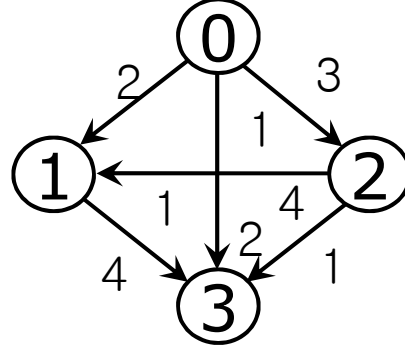- adjLists[i] is a pointer to the first node in the adjacency list for vertex i

2.1 Why graph?

(5) Classification of graph

2.1 Why graph?

(5) Classification of graph algorithms
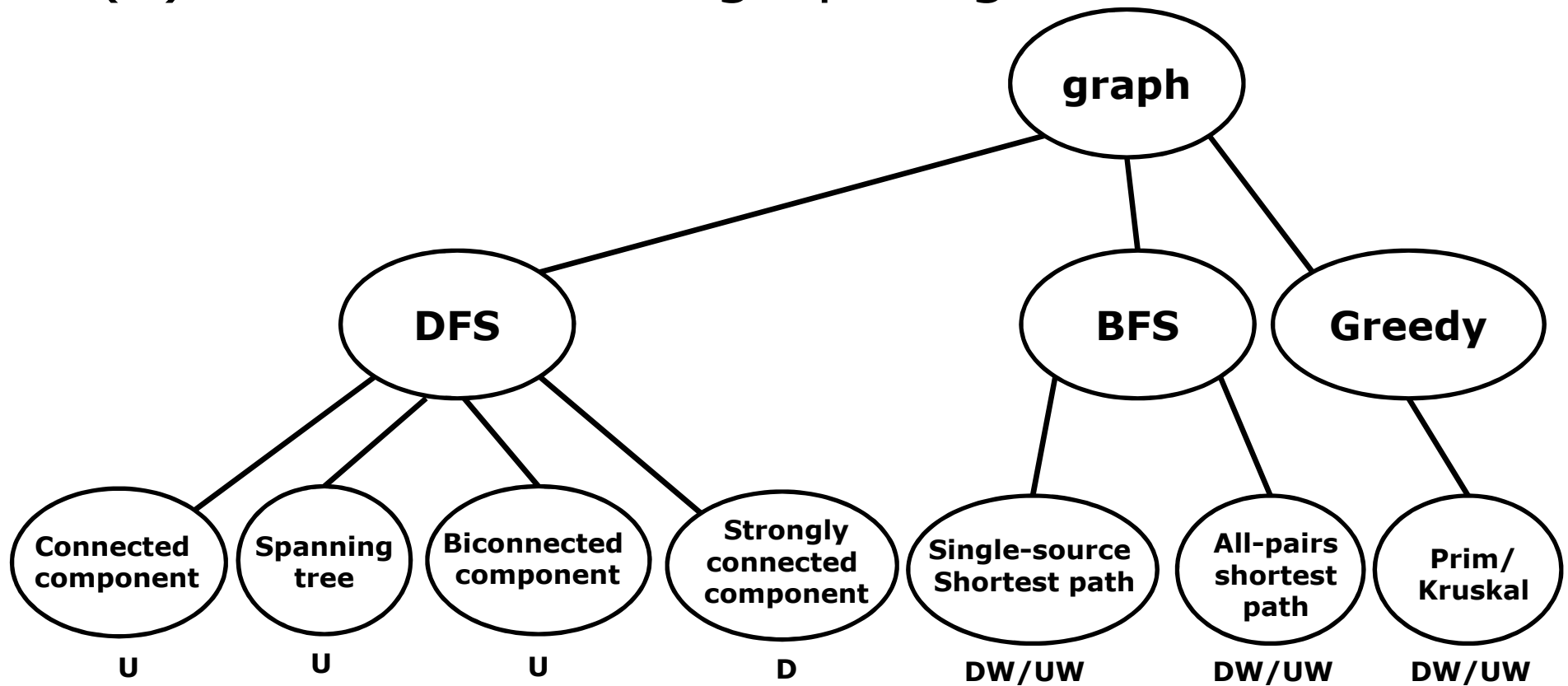
(1) Search of a graph

– *What parts of the graph are reachable from a given vertex?*
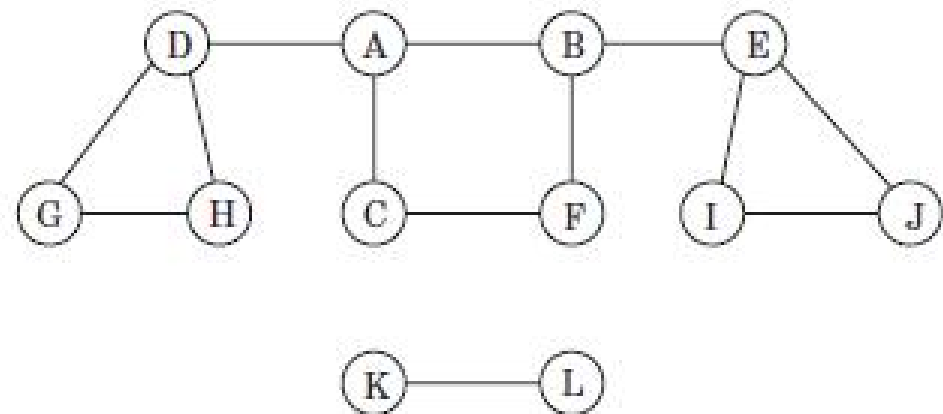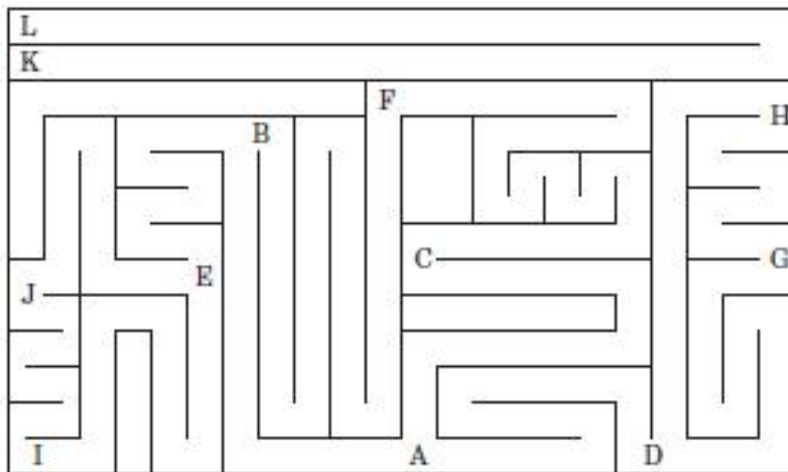
(1) Search of a graph

– Use a recursive call to search a graph

```
procedure explore(G,v)
Input:  G = (V, E) is a graph; v ∈ V
Output: visited(u) is set to true for all nodes u reachable
    from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
    if not visited(u): explore(u)
postvisit(v)
```

(2) Basic strategy of depth-first search

- – Visit connected nodes as far as possible
- – Use a stack
- – Implemented using a recursive call

```
procedure dfs(G)

for all v ∈ V:
    visited(v) = false

for all v ∈ V:
    if not visited(v): explore(v)
```

(3) Example of depth-first search (1)

(4) Previsit and postvisit ordering

- – At visiting vertices, mark
  - The first time we visit → previsit
  - The last time we departure → postvisit
- – For any nodes u and v, the two intervals [pre(u), post(u)] and [pre(v), post(v)] are either disjoint or one is contained within the other.

```
for each edge (v, u) ∈ E:
    if not visited(u): explore(u)
```

```
procedure previsit(v)
pre[v] = clock
clock = clock + 1

procedure postvisit(v)
post[v] = clock
clock = clock + 1
```
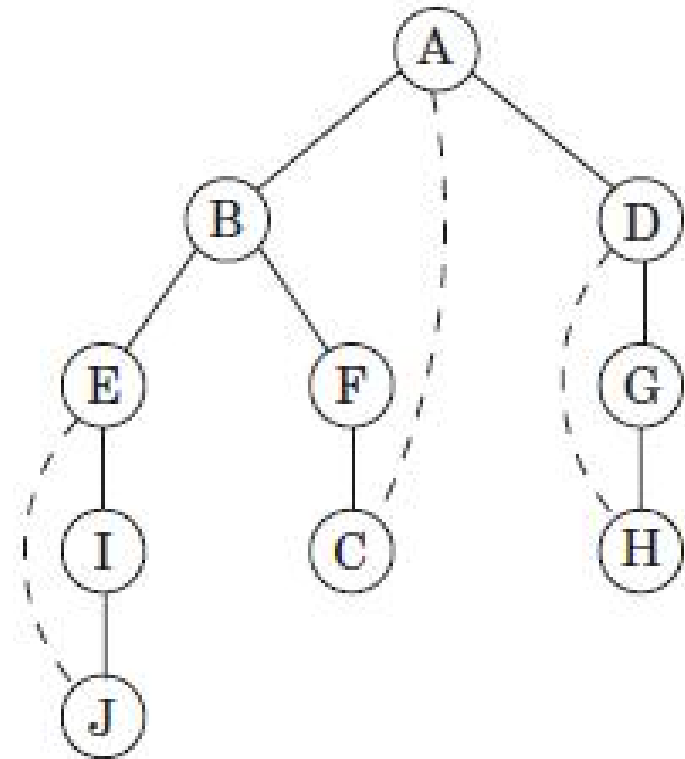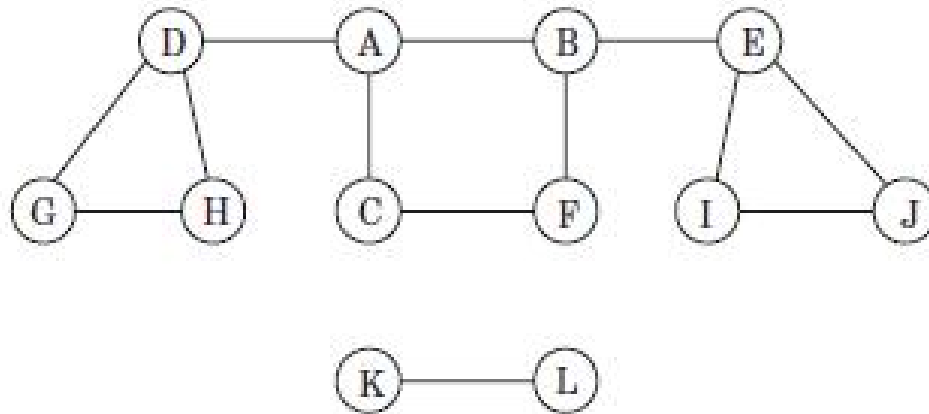
2.2 Depth-first search in undirected graphs

(4) Previsit and postvisit ordering

- Property
  - For any nodes u and v, the two intervals [pre[u], post[u]] and [pre[v], post[v]] are either disjoint or one is contained within the other.

(5) Example of depth-first search (2)
– with previsit & postvisit
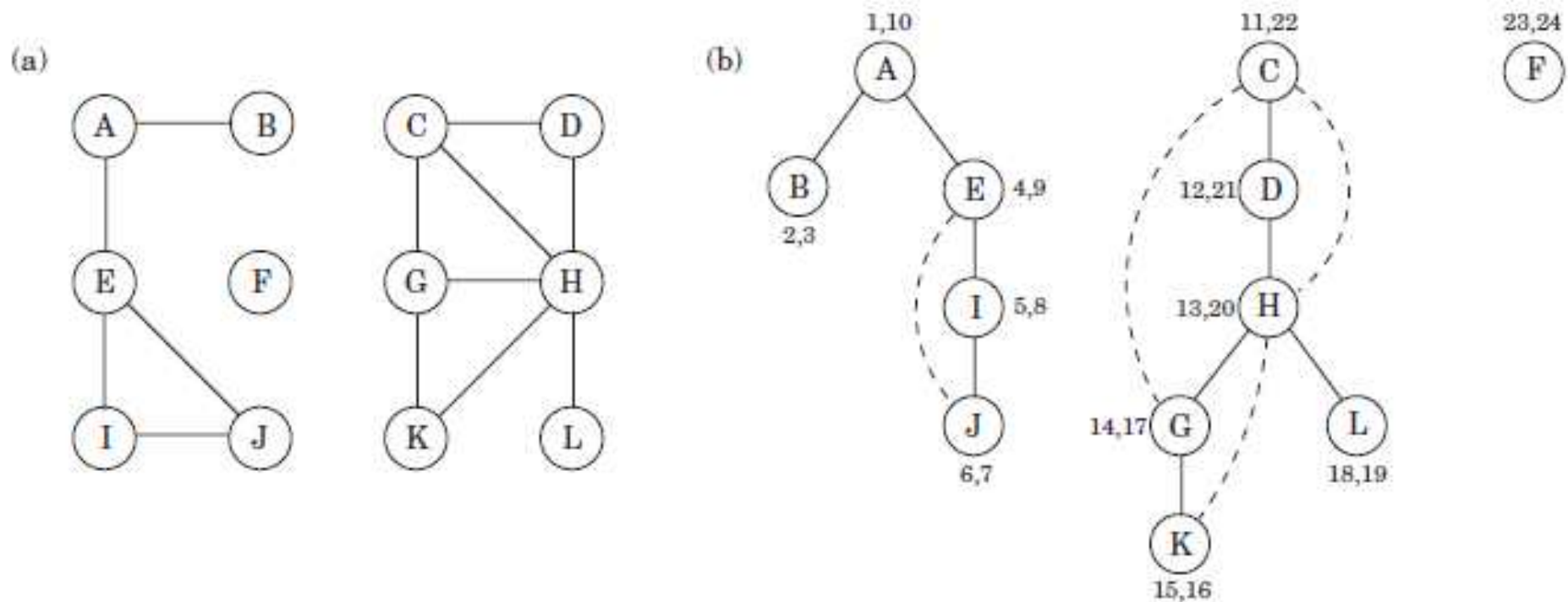
(6) Connected components

- Strategy
  - Use a search algorithm to explore all the vertices in a connected component
  - In calling `previsit (v)`,

  ```
  procedure previsit(v)
  ccnum[v] = cc
  ```

  - `cc`
    - initialized as 0
    - Incremented each time `explore ( )` is called

- Example of a depth-first search on digraph
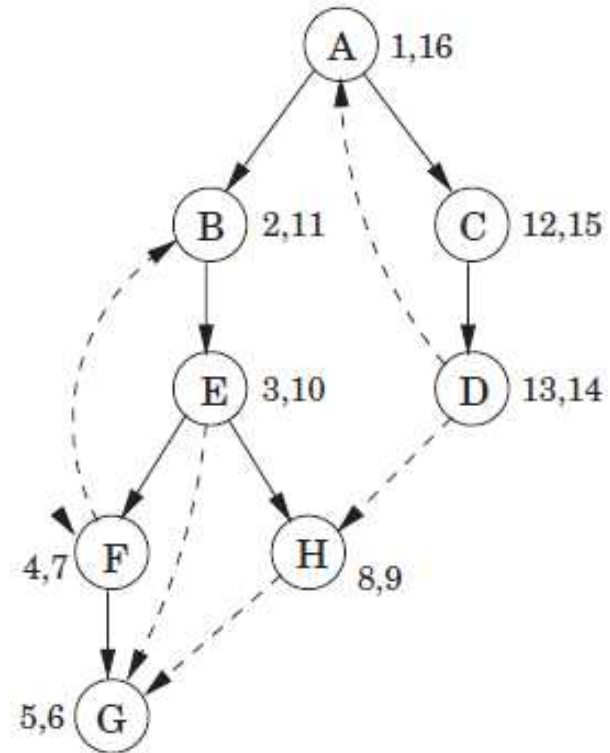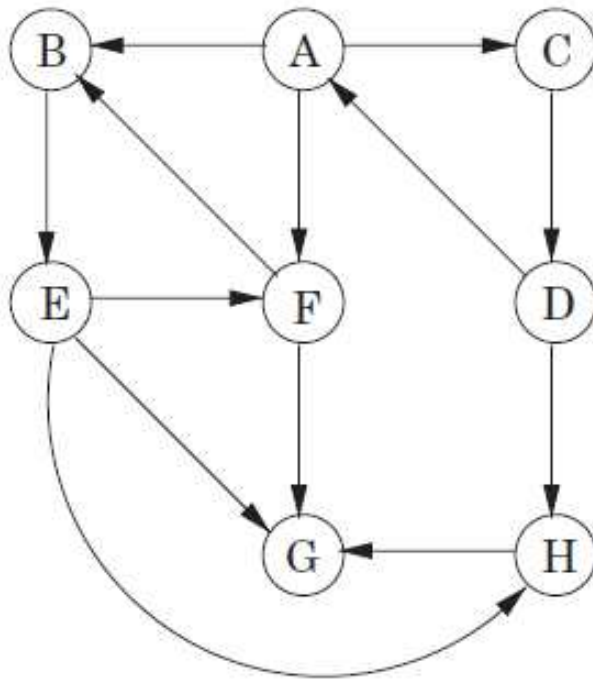
(1) Types of edges
– An example on digraph → DFS tree
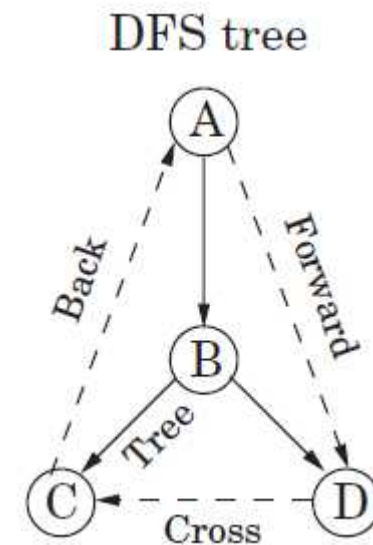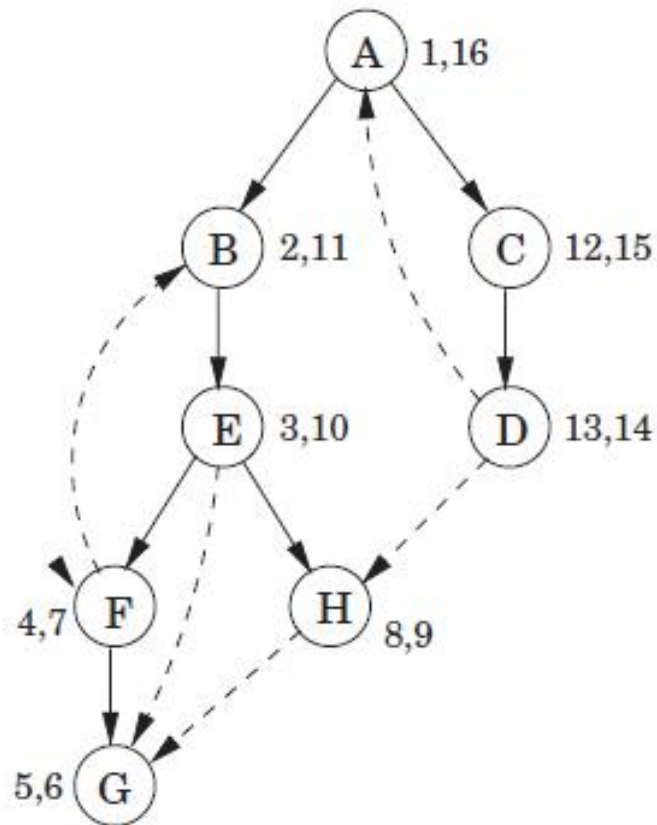
(1) Types of edges

– Four types of edges in DFS tree

• Tree edge

– Edges in the DFS tree

• Forward edge

– To a non-child descendant

• Backward edge

– To an ancestor

• Cross edge

– To neither descendant nor ancestor

DFS tree
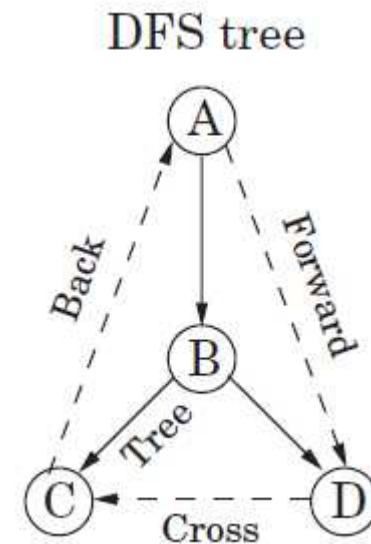
(1) Types of edges

- Relation with pre and post
  - For an edge <u, v>

[	[	]	]	Tree/Forward
u	v	v	u	
[	[	]	]	Backward
v	u	u	v	
[	]	[	]	Cross
v	v	u	u	

2.3 Depth-first search in directed graphs

(2) Directed Acyclic Graph (dag)
- Definitions (1)
  - Cycle
    - A circular path in a directed graph
    - u → v → w → …… → u
    - Cycles in this graph?
  - Acyclic graph
    - A graph without a cycle

(2) Directed Acyclic Graph (dag)

– Definitions (2)

- Source
    - A vertex that has only out-edges
- Sink
    - A vertex that has only in-edges

(2) Directed Acyclic Graph (dag)

– Definitions (3)

• Topological order

– Order the vertices one after the other in such a way that each edge goes from an earlier vertex to a later vertex

– Linearization

– How many linearizations in this graph?

(2) Directed Acyclic Graph (dag)

- Properties
  - A directed graph has a cycle if and only if its depth-first search reveals a back edge

  - In a dag, every edge leads to a vertex with a lower `post` number

  - Every dag has at least one source and at least one sink

2.4 Strongly connected components

(1) Connectivity for directed graph

- Connected
  - Two nodes u and v of a directed graph are connected if there is a path from u to v and vice versa
  - Example)
    - B & E are connected
    - G & H are connected
    - J & L are connected
    - E & G are not connected
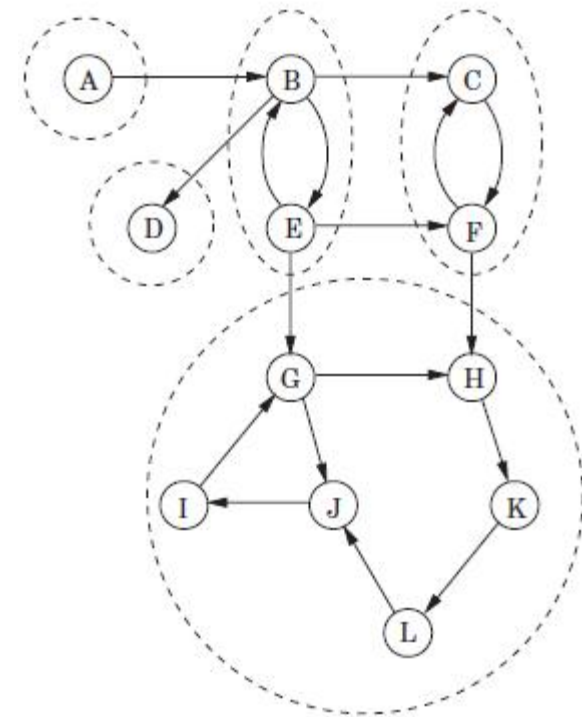
2.4 Strongly connected components

(1) Connectivity for directed graph

- Strongly connected component (SCC)
  - Partitioning of V into disjoint sets according to the definition of "connected"
  - Example)
    » B & E are SCC
    » A is SCC
    » G, H, I, J, K & L are SCC
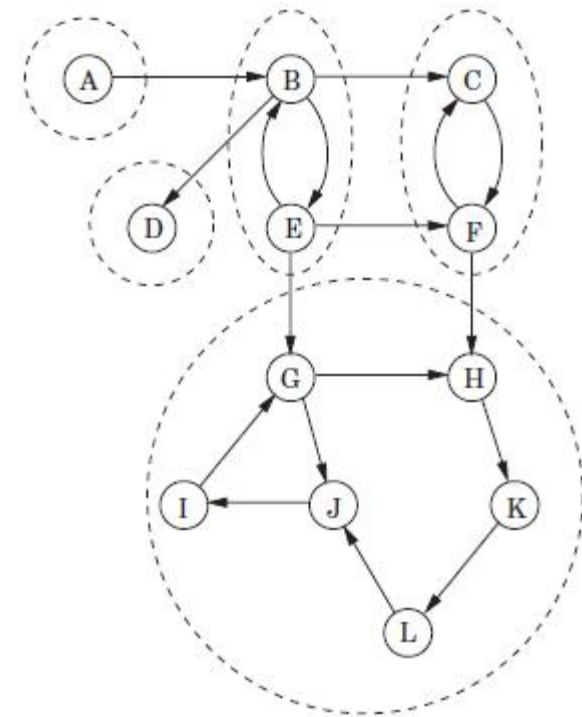
2.4 Strongly connected components

(1) Connectivity for directed graph

– Property

• Every directed graph is a dag of its strongly connected components
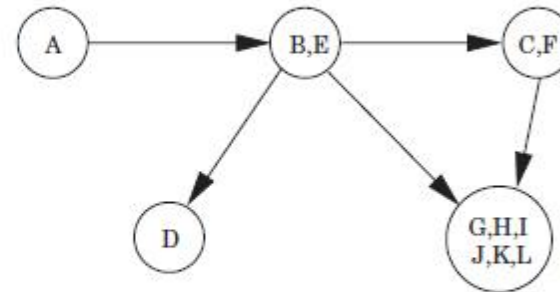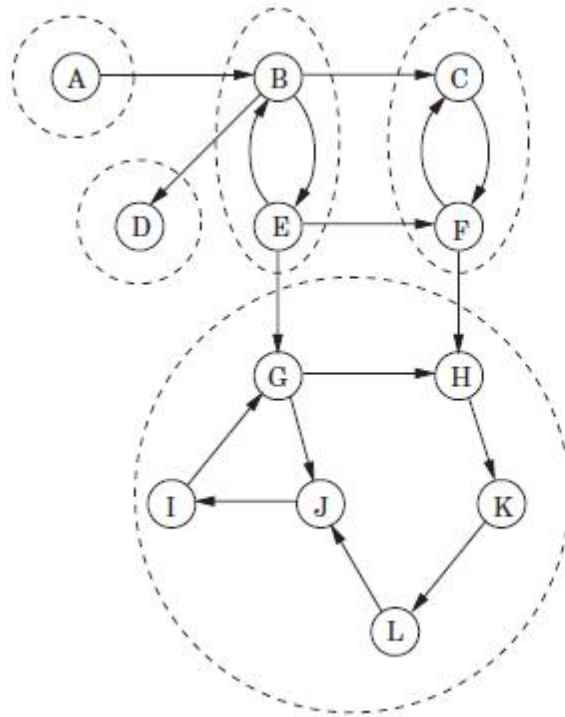
2.4 Strongly connected components

(2) Algorithm

- Property1
  - If the explore ( ) subroutine is started at a node u, then it will terminate precisely when all nodes reachable from u have been visited

- Property2
  - The node that receives the highest post number in a depth-first search must lie in a source strongly connected components

- Property3
  - If C and C' are strongly connected components, and there is an edge from a node in C to a node in C', then the highest post number in C is bigger than the highest post number in C'

2.4 Strongly connected components

(2) Algorithm

- Strategy
  - Find a sink strongly connected component and remove it
  - Repeat this until we have only one strongly connected component
- Problem
  - How can we find a sink strongly connected component?
    - Motivation
      - Use Property2
      - Define G^R from $G = (V, E)$
      - G^R has same V, but reverse E
      - Sink component in G = Source component in G^R

– Example
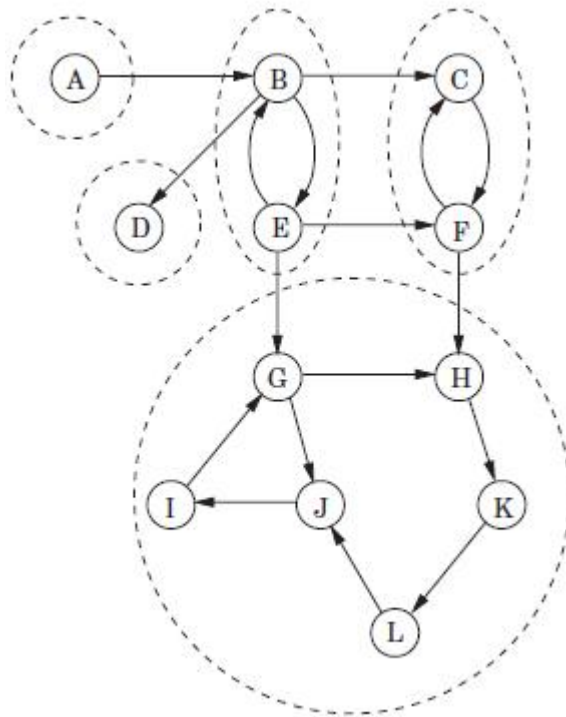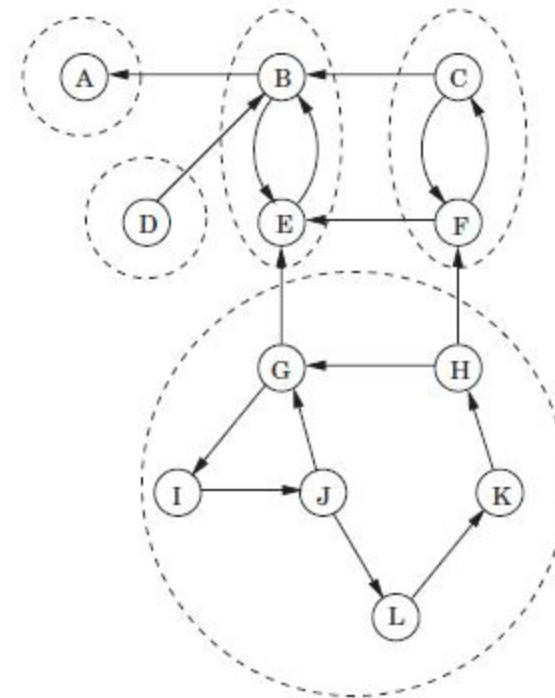
(2) Algorithm

- Tarzan-Gabow algorithm

- Steps

    - Compute G^R from G

    - Run depth-first search on G^R

    - Run the undirected connected components algorithm

        - Process the vertices in decreasing order of their post numbers from the previous step

(2) Algorithm
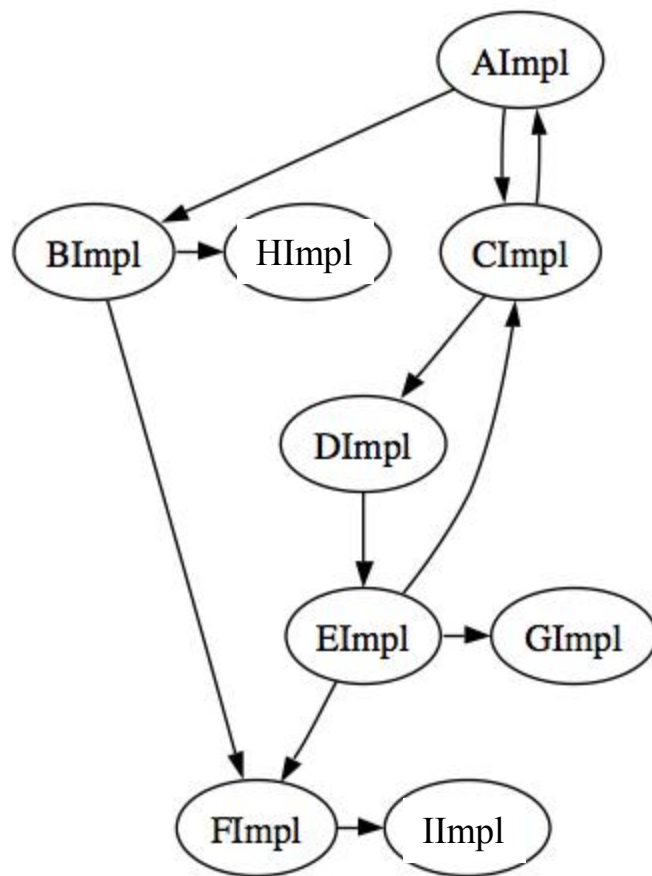
– Test

2.5 Distance

- Distance
  - In real world, the distances between nodes are not identical
- Weighted graph
  - Edges on a graph have different weights

2.5 Distance

- Distance between two nodes
  - The length of the shortest path between them
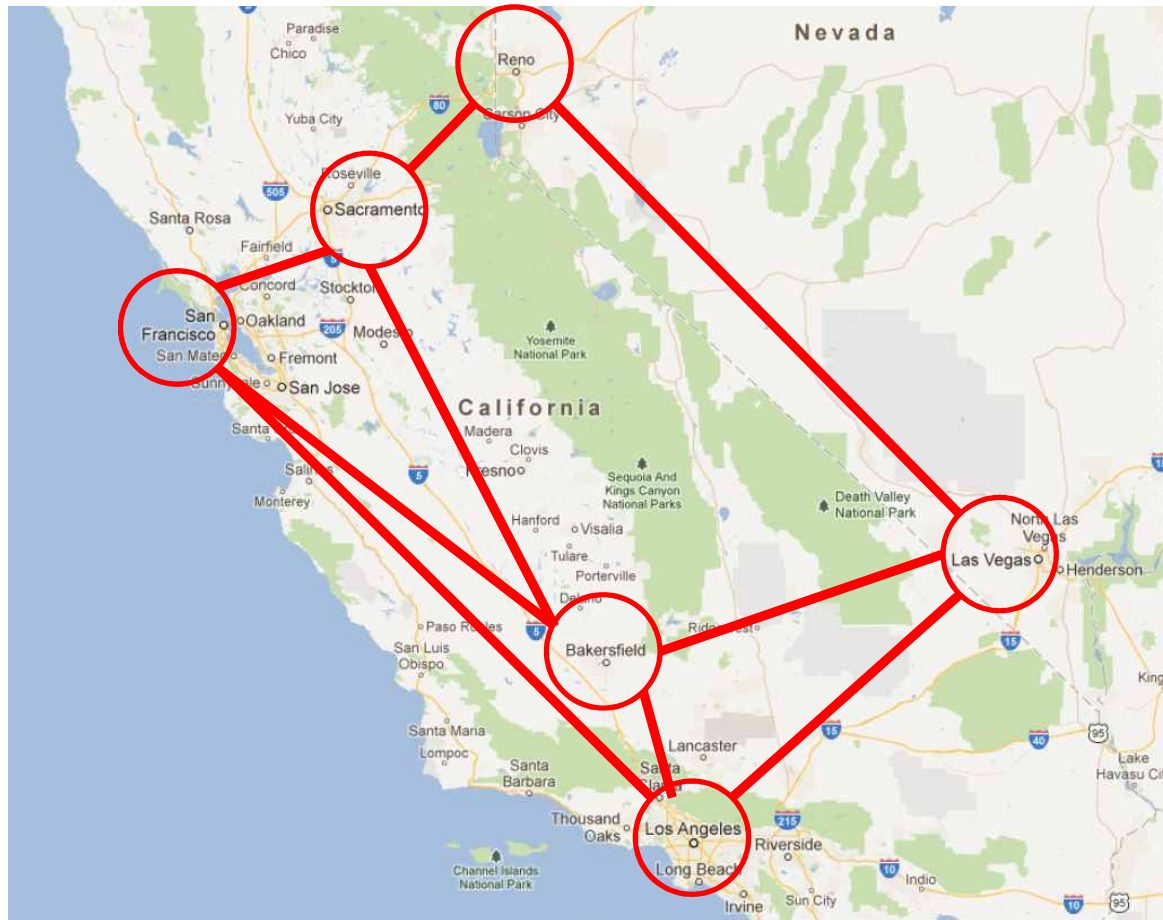
2.5 Distance

- Distance between two nodes
  - The length of the shortest path between them
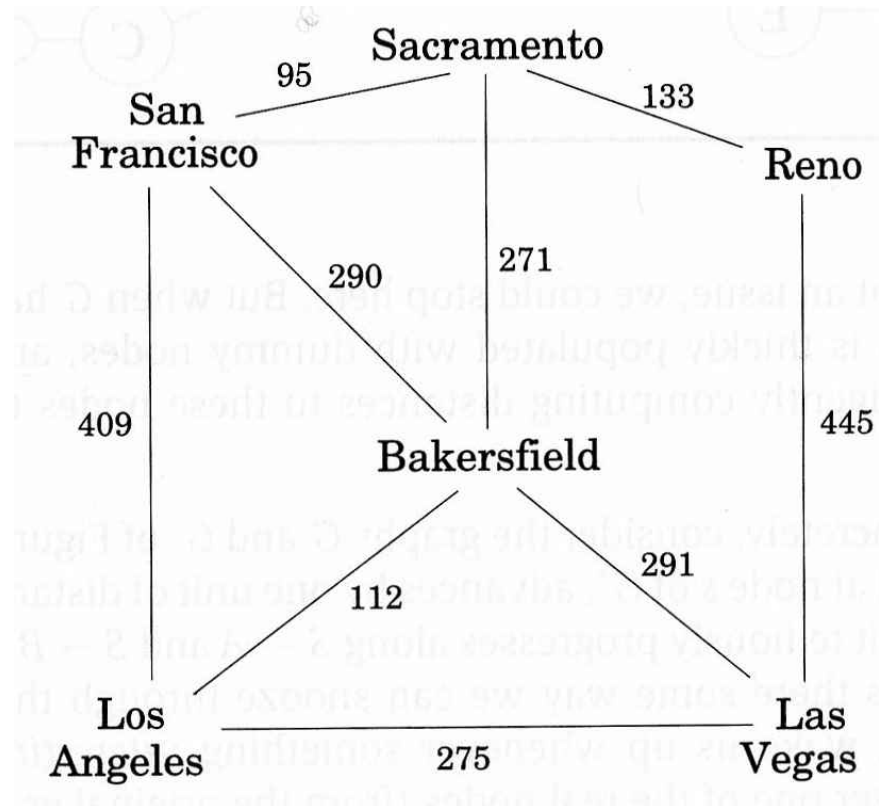
2.5 Distance

- Finding shortest path between two nodes in a graph
  - What is the shortest path from B to G?
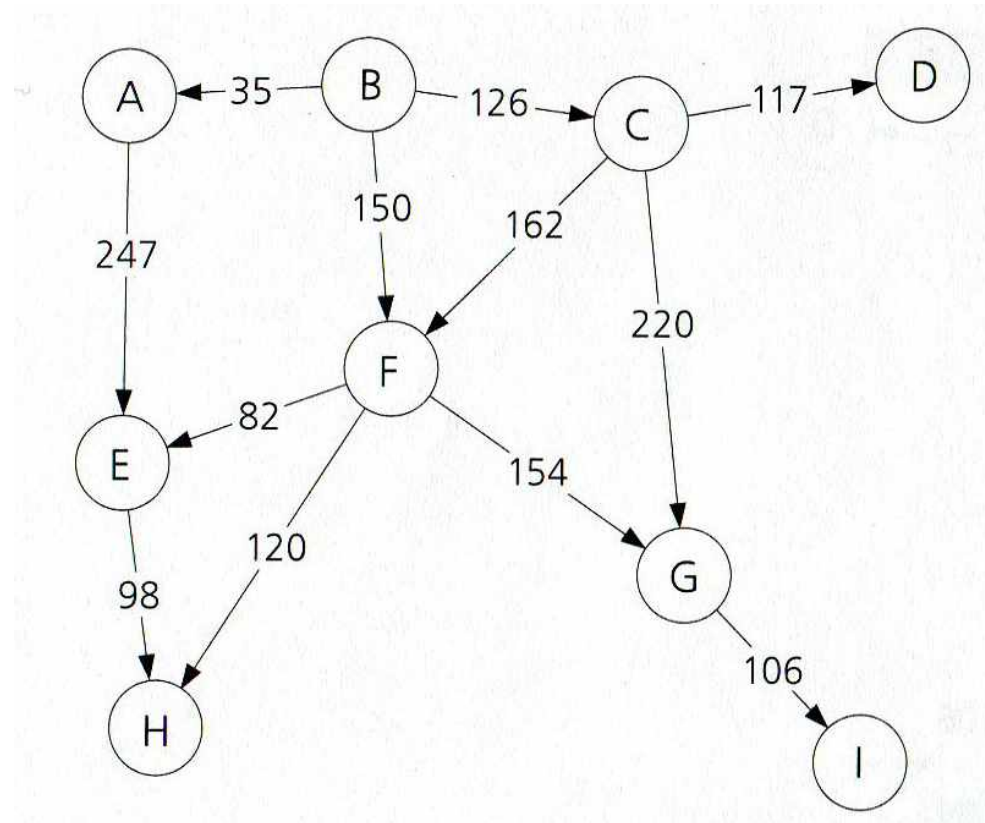
2.6 Breadth-first search

- Basic strategy
  - In visiting a vertex v,
  - Mark all the adjacent vertices as visited
  - Add the vertices to the queue
  - Use a queue

- Compare to Depth-first search
  - Visit connected nodes as far as possible
  - Implemented using a recursive call
  - Use a stack

2.6 Breadth-first search

- Recall: Queue
  - Property of Queue
    - First-In First-Out

  - Important points of Queue
    - Front
    - Rear

  - Operations of Queue
    - Insert ( )
    - Remove ( )

2.6 Breadth-first search

- Algorithm

```
procedure bfs ( G, s )
Input: Graph G = (V, E), s = start vertex
Output: For all vertices u reachable from s, dist[u] is
set to the distance from s to u

For all u ∈ V
    dist[u] = ∞;

dist[s] = 0;
Q.insert ( s );
while ( !Q.is_empty ( ) ) {
    u = Q.remove ( );
    for all edges (u, v) ∈ E
        if ( dist[v] == ∞ )
            Q.insert ( v );
            dist[v] = dist[u] + 1;

}
```

– Example

visit & insert 0

0	*0*
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞

dist[u]

queue

0							
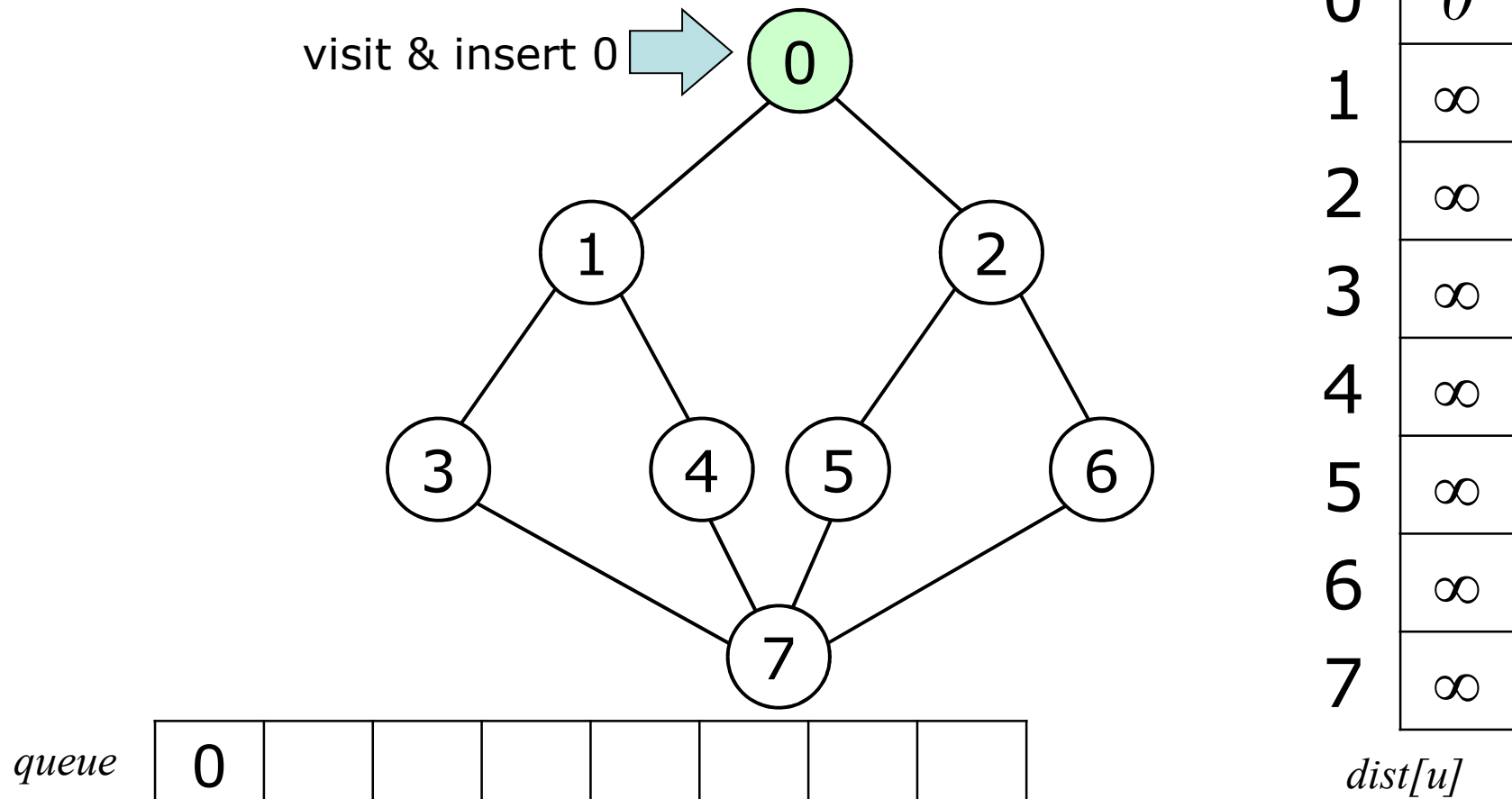
2.6 Breadth-first search

– Breadth-first spanning tree
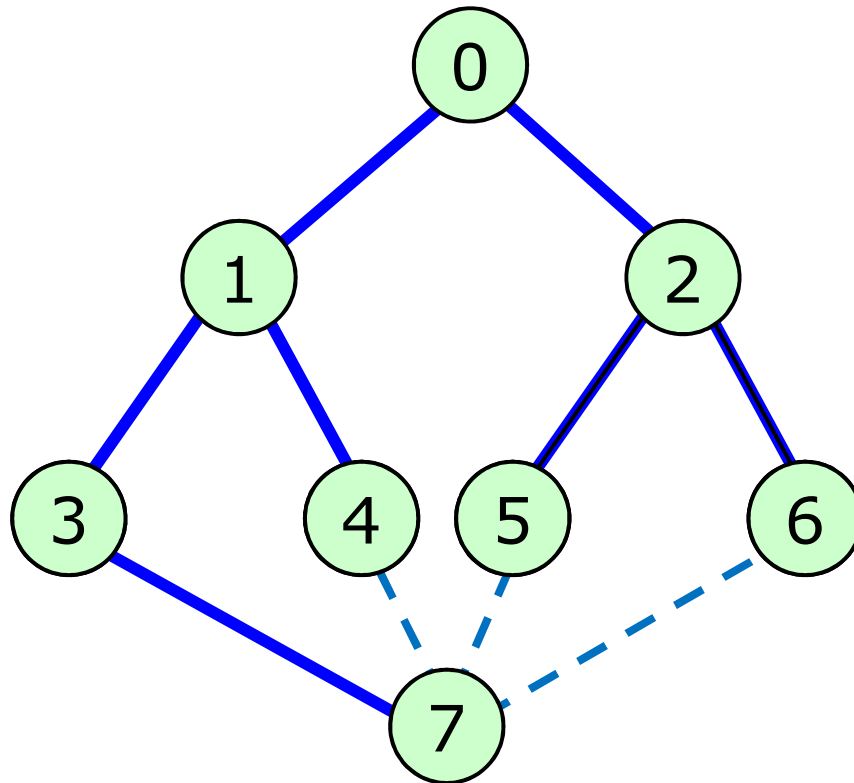
2.7 Single source shortest path

(1) Basic concept (1)

- Find the shortest path from a node to all other nodes in a graph
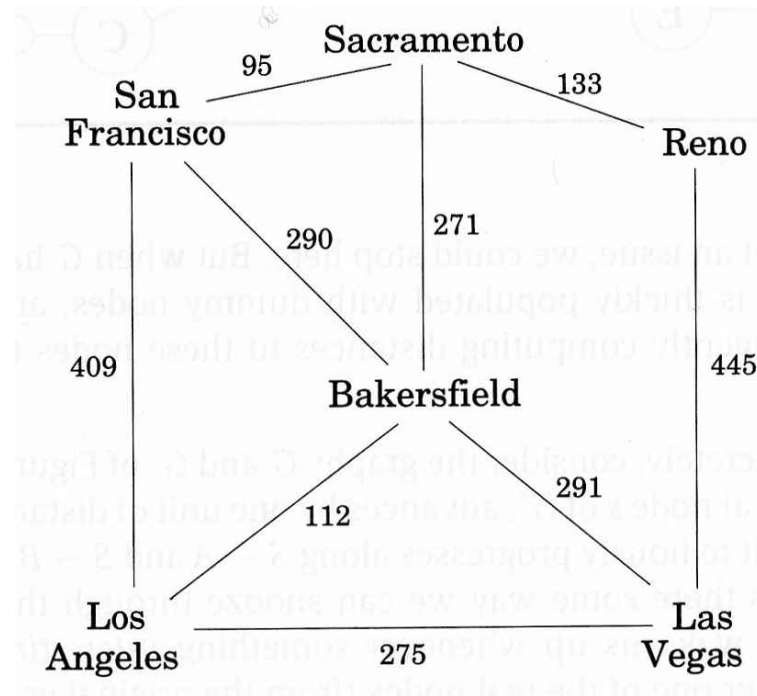
2.7 Single source shortest path

(1) Basic concept (2)

- Single-source shortest path
  - A path from v_0 to u: v_0, v_1, ..., v_k, u
  - Path: a set of edges composed of
    (v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k), (v_k, u)

  - Cost of path: sum of weights of the edges on the path
    $w(v_0, v_1)+w(v_1, v_2)+...+w(v_{k-1}, v_k)+w(v_k, u)$

2.7 Single source shortest path

(2) Simple approach (1)

- Adaptation of breadth-first search
  - Modifying weighted graph to non-weighted graph

> For any edge e = (u, v) of weight l_e, replace e with l_e
> edges of weight 1_e by adding l – 1 vertices between u & v

(2) Simple approach (2)

- Problem: Inefficiency
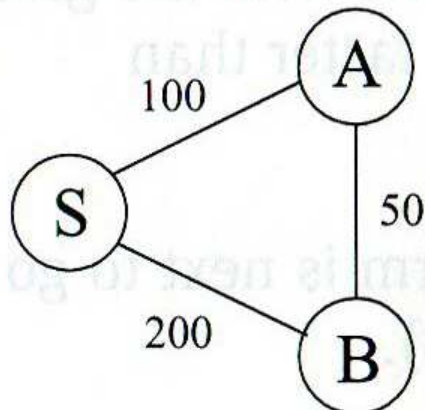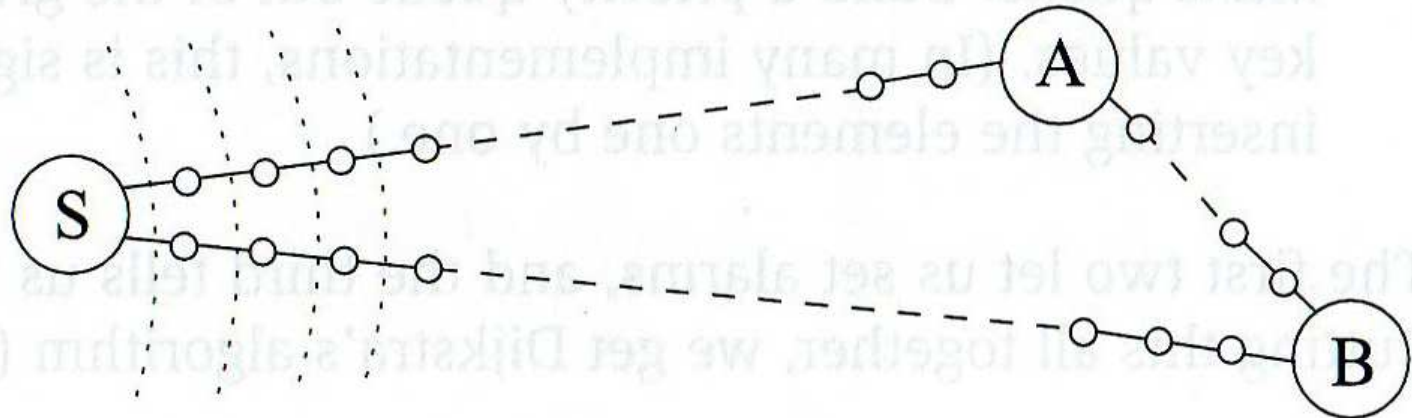
2.7 Single source shortest path

(3) Alarm clocks (1)

- A motivation to Dijkstra's algorithm
- Strategy
  - On leaving S (t = 0), we set two alarms for A (t = 100) and B (t = 200).
  - At arriving A at t = 100, we estimate the distance from A to B and reset the alarm for B (t = 150).

G:

A

100

S

50

200

B
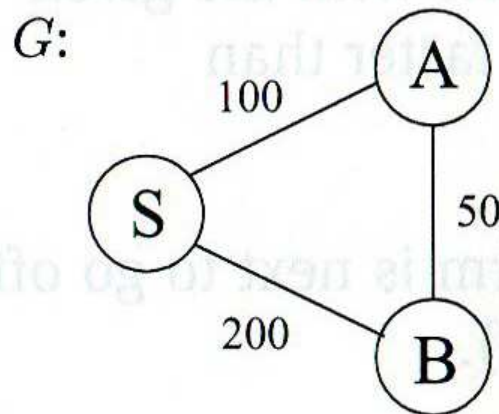
2.7 Single source shortest path

(3) Alarm clocks (2)

- General strategy

```
1. Set an alarm clock for node s at time 0.
2. Repeat until there are no more alarms.
3. If the next alarm goes off at time T for node u,
     3.1 The distance from s to u is T.
     3.2 For each neighbor v of u
          3.2.1 If no alarm set to v, then set as T + l(u, v).
          3.2.2 If v's alarm > T + l(u, v), then
                  set v's alarm as T + l(u, v).
```

2.7 Single source shortest path

(4) Dijkstra's algorithm (1)

- Single source all destination shortest path

- Directed graph

- Non-negative edge

2.7 Single source shortest path

(4) Dijkstra's algorithm (2)

```
procedure Dijkstra ( G, s )

for all u ∈ V
    dist[u] = ∞;
    prev[u] = NULL;


dist[s] = 0;


H = makequeue ( V );
while ( H is not empty )
    u = get_min ( H );
    for all edges (u, v) ∈ E
        if ( dist[v] > dist[u] + l(u, v) )
            dist[v] = dist[u] + l(u, v);
            prev[v] = u;
            modify_H ( H, v );
```

2.7 Single source shortest path

(4) Dijkstra's algorithm (3)

- Example: Initial step

A: 0	D: ∞
B: 4	E: ∞
C: 2	

(4) Dijkstra's algorithm (3)

- Example: Step 1

A: 0	D: 6
B: 3	E: 7
C: 2	
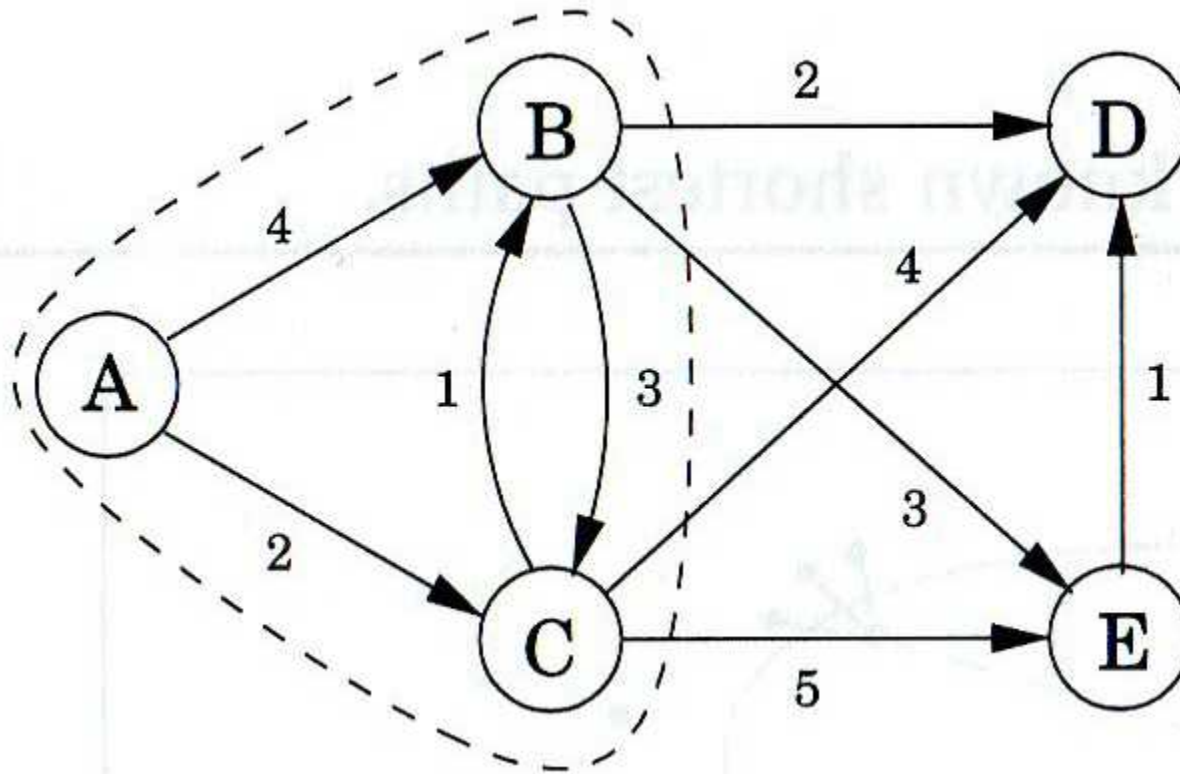
(4) Dijkstra's algorithm (3)

- Example: Step 2

A: 0	D: 5
B: 3	E: 6
C: 2	

(4) Dijkstra's algorithm (3)

- Example: Step 3
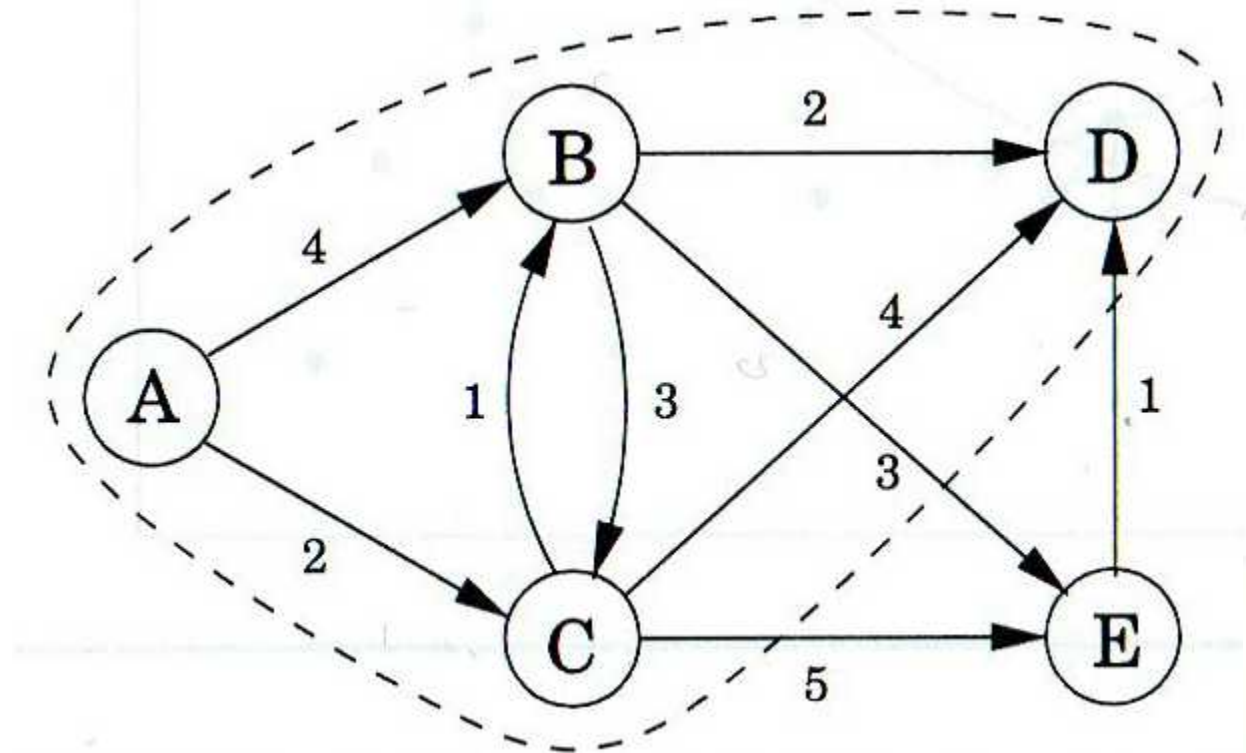
A: 0	D: 5
B: 3	E: 6
C: 2	

(4) Dijkstra's algorithm (3)

- Example: Result
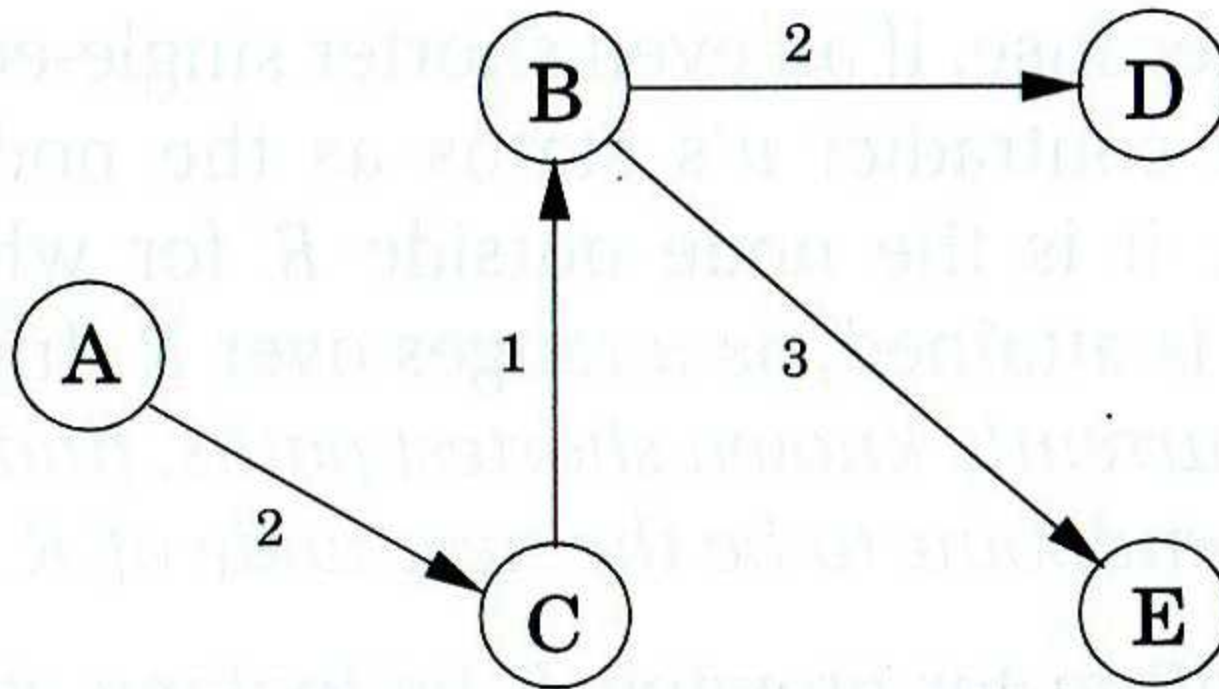
(4) Dijkstra's algorithm (3)

- Example:
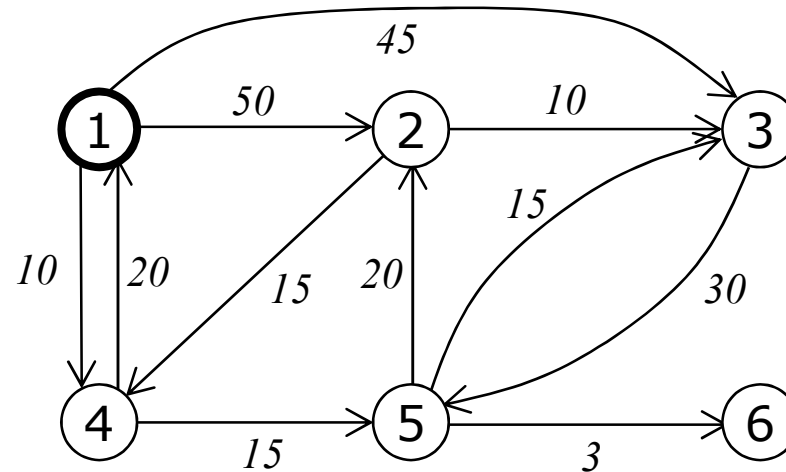
2.7 Single source shortest path

(4) Dijkstra's algorithm (4)

- Performance analysis

```
procedure Dijkstra ( G, s )
for all u ∈ V
    dist[u] = ∞;            O(n)
    prev[u] = NULL;


dist[s] = 0;


H = makequeue ( V );                    O(?) …… A
while ( H is not empty )
    u = get_min ( H );                  O(n * ?) … B
    for all edges (u, v) ∈ E
        if ( dist[v] > dist[u] + l(u, v) )
            dist[v] = dist[u] + l(u, v);   O(n * ?) … C
            prev[v] = u;
            modify_H ( H, v );
```
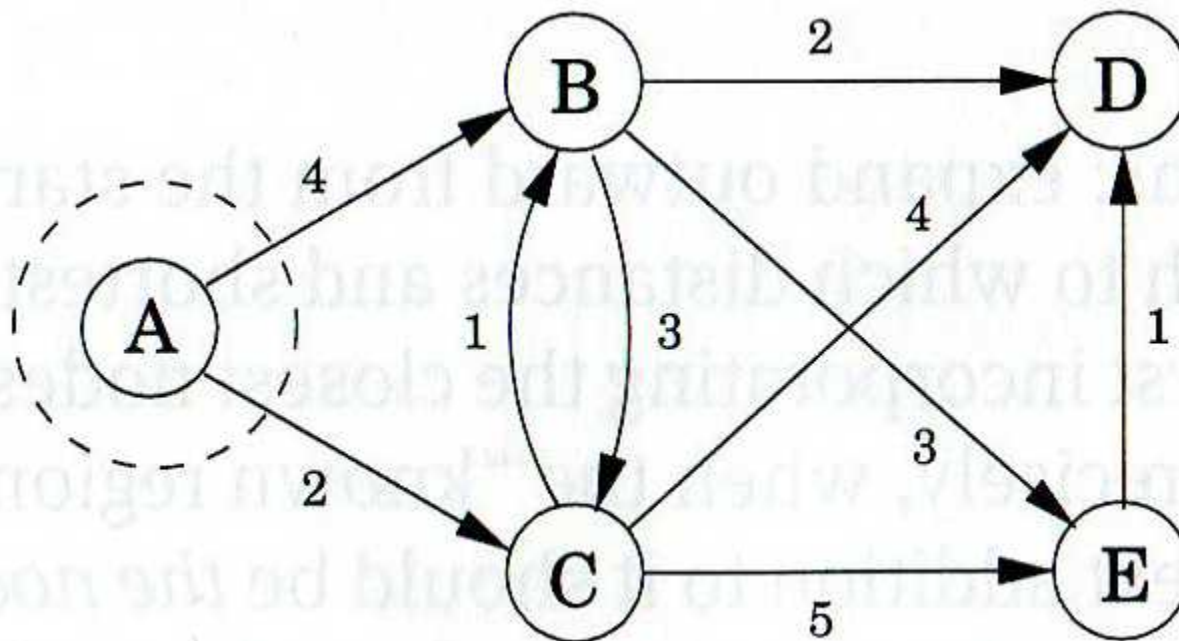
2.7 Single source shortest path

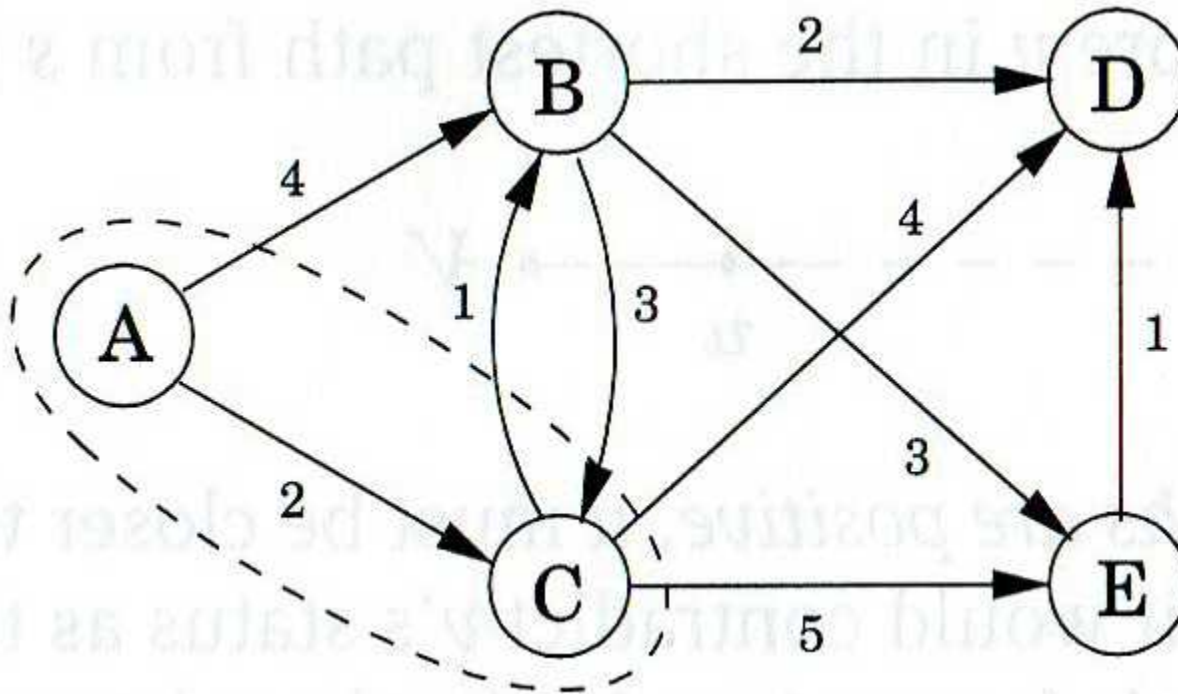## (4) Dijkstra's algorithm (5)

- Performance depends on queue & complexity

queue \ complexity	array	Priority queue
$|E| = O(|V|^2)$	A: O(n)	A: O(n log n)
	B: $O(n^2)$	B: O(n log n)
	C: $O(|E|) = O(n^2)$	C: $O(|E|*\log n) = O(n^2 *\log n)$ $= O(n^2 \log n)$
	Total: **$O(n^2)$**	Total: **$O(n^2 \log n)$**
$|E| = O(1)$	A: O(n)	A: O(n log n)
	B: $O(n^2)$	B: O(n log n)
	C: $O(|E|) = O(n)$	C: $O(|E|*\log n) = O(n *\log n)$ $= O(n \log n)$
	Total: **$O(n^2)$**	Total: **$O(n \log n)$**

2.8 All pairs shortest path

(1) Basic concept (1)

- The problem of finding shortest paths for every two vertices v to u.

- Solving single-source shortest path for all vertices in G

- Floyd's algorithm

2.8 All pairs shortest path

(2) Floyd's algorithm (1)

- Finding the all-pair's shortest path.
  - Input: adjacency matrix of a graph.

  - The weight of a path between two vertices is the sum of the weights of the edges along that path.

  - Negative weight is allowed.

  - Negative cycle is not allowed.

2.8 All pairs shortest path

(2) Floyd's algorithm (2)

- $A^k[i][j]$:
  - The cost of the shortest path from vertex i to j, using only those intermediate vertices with an index $\leq$ k.

- $A^{-1}[i][j]$
  - the weight of an edge connecting vertex i and vertex j

2.8 All pairs shortest path

(2) Floyd's algorithm (3)
- – Basic idea:
  - Starting from A^{-1}, successively generate the matrices to A^1, A^2, …, A^n.

$$A^k[i][j] = min \{ A^{k-1}[i][j],$$
$$A^{k-1}[i][k] + A^{k-1}[k][j] \}$$

$$A^{-1}[i][j] = cost[i][j]$$

2.8 All pairs shortest path

(2) Floyd's algorithm (4)
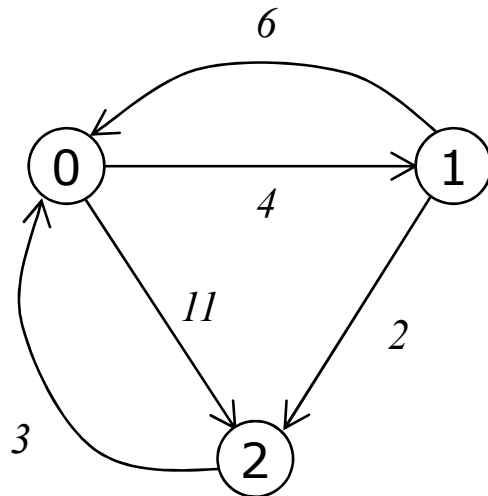
```c
void Floyd ( int cost[][], int dist[][], int n )
{
    int i, j, k;
    for ( i = 0; i < n; i++ )
        for ( j = 0; j < n; j++ )
            dist[i][j] = cost[i][j];

    for ( k = 0; k < n; k++ ) {
        for ( i = 0; i < n; i++ )
            for ( j = 0; j < n; j++ )
                if ( dist[i][k] + dist[k][j] < dist[i][j] )
                    dist[i][j] = dist[i][k] + dist[k][j];
    }
}
```

(2) Floyd's algorithm (5)

- Example:

A^{-1}	0	1	2
0			
1			
2			

A^0	0	1	2
0			
1			
2			

A^1	0	1	2
0			
1			
2			

A^2	0	1	2
0			
1			
2			

2. Graph

2.0 Introduction

2.1 Why graph?

2.2 Depth-first search in undirected graph

2.3 Depth-first search in directed graphs

2.4 Strongly connected components

2.5 Distances

2.6 Breadth-first search

2.7 Single source shortest path

2.8 All pairs shortest path

Contents

0. Prologue

1. Divide & conquer

2. Graph

3. Greedy algorithm

4. Dynamic programming