[CSE3081(2반)] 알고리즘 설계와 분석

2020학년도 2학기

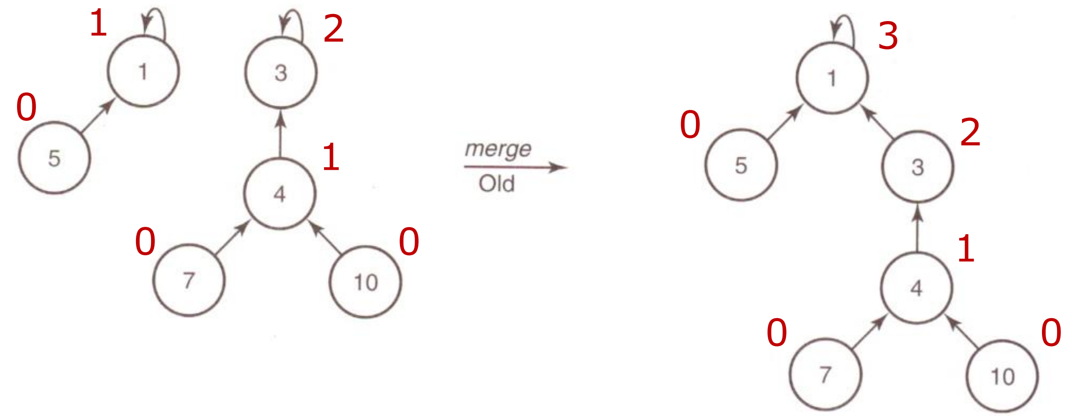
강의자료

(2020.11.19 목요일)

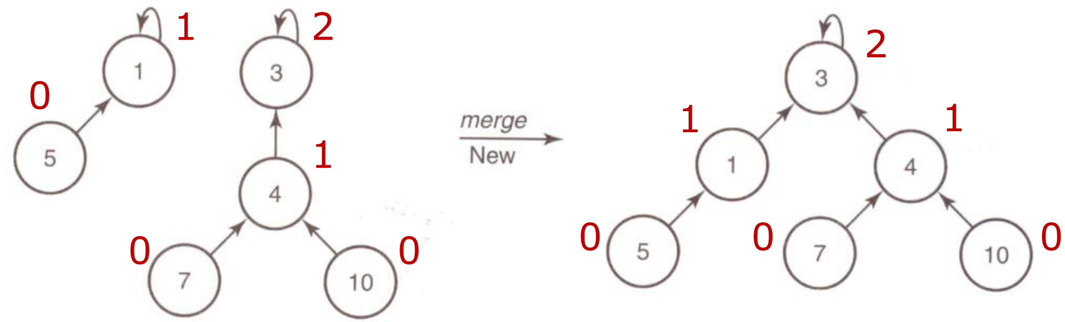
서강대학교 공과대학 컴퓨터공학과
임 인 성 교수

[주제 5]
Greedy Methods

- **Two ways of implementing the Union operation**



Time complexity: $O(n)$



Time complexity: $O(\log n)$

Union by rank

- Always attach the smaller tree to the root of the larger tree.
- The **rank** increases by one only if two trees of the same rank are merged.
  - ✓ The rank of a one-element tree is zero.
- The Union and Find operations can be done in O(log n) in the worst case.
  - ✓ The number of elements in a tree of rank r is at least 2^r. (Proof by induction)
  - ✓ The maximum possible rank of a tree with n elements is O(log n).

Disjoint set의 path compression 연산에 대해서도 알아볼 것.

– S = Find(x)

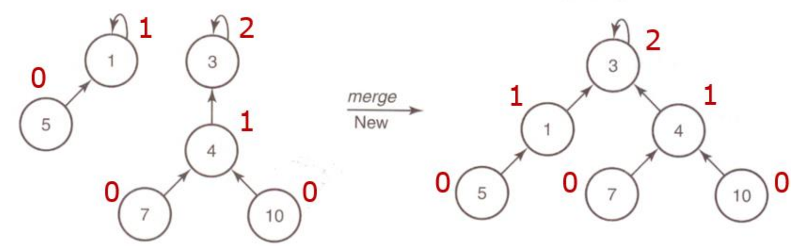
Time complexity: O(depth of x in the tree)

```
Find(x) {
  while (x != parent(x))
    x := parent(x)
  return x
}
```

```
Find(x) {
  if (x == parent(x))
    return x
  else
    return Find(parent(x))
}
```

– Union(x, y)

```
Union(x, y) {
  x0 := Find(x)
  y0 := Find(y)
  if (x0 == y0)
    return
  if (rank(x0) > rank(y0))
    parent(y0) := x0
  else
    parent(x0) := y0
    if (rank(x0) == rank(y0))
      rank(y0) := rank(y0)+1
}
```



Time complexity:

2 Find op's + O (1)

Scheduling with Deadlines

- **Problem**
  - Let $J = \{1, 2, \ldots, n\}$ be a set of jobs to be served.
  - Each job takes one unit of time to finish.
  - Each job has a **deadline** and a **profit**.
    - If the job starts before or at its deadline, the profit is obtained.
  - ➢ Schedule the jobs so as to **maximize** the total profit (not all jobs have to be scheduled).

1 2 3

- **Example:**

Job	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40

Schedule	Total Profit
[1, 3]	30 + 25 = 55
[2, 1]	35 + 30 = 65
[2, 3]	35 + 25 = 60
[3, 1]	25 + 30 = 55
[4, 1]	☞ 40 + 30 = 70
[4, 3]	40 + 25 = 65

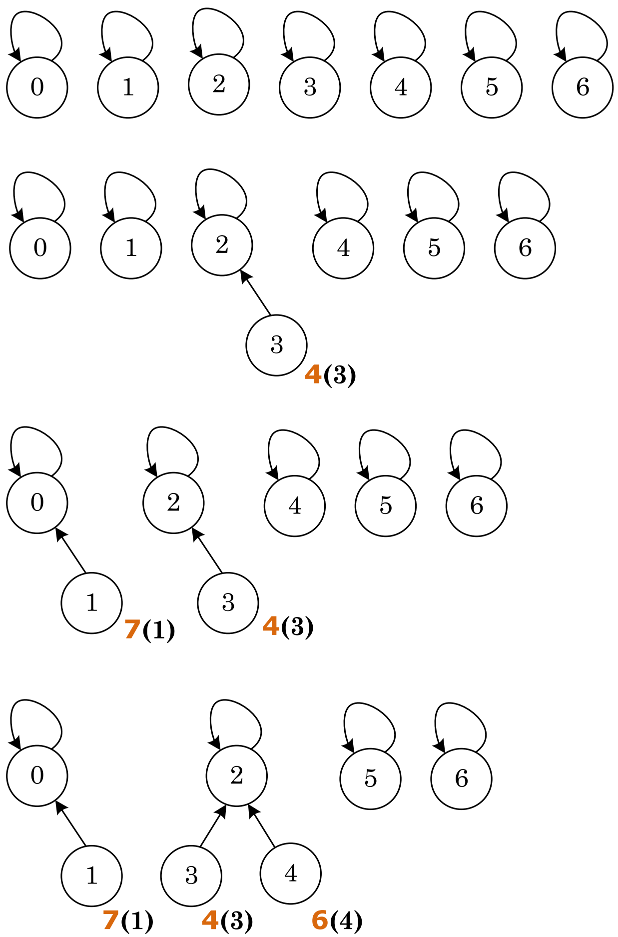
Another Algorithm Based on Disjoint Sets

- **Method**
  - ✓ d_{max} : the maximum of the deadlines for n jobs.
  - ✓ Add a job as late as possible to the schedule being built, but no later than its deadline.

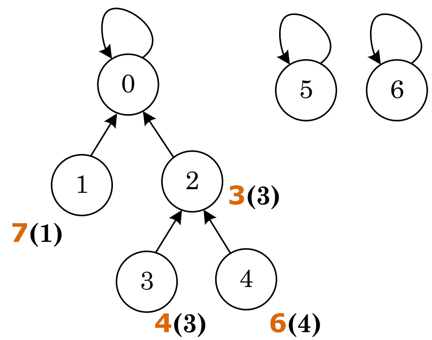
  - Sort the jobs in non-increasing order by profit.
  - Initialize $d_{max}+1$ disjoint sets, containing the integers $0, 1, 2, \ldots, d_{max}$.
  - For each job in the sorted order,
    - Find the set S containing the minimum of its deadline and n.
      - If $\text{small}(S) = 0$, reject the job.
      - Otherwise, schedule it at time $\text{small}(S)$, and merge S with the set containing $\text{small}(S)\text{-}1$.

- Time complexity
  - $O(n \log m)$ for the disjoint set manipulation, where m is the minimum of n and d_{max}.
  - $O(n \log n)$ for sorting the profits.

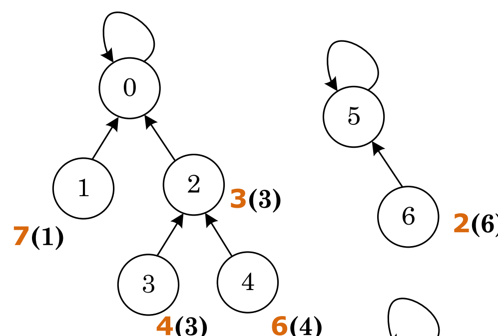
- **Example**



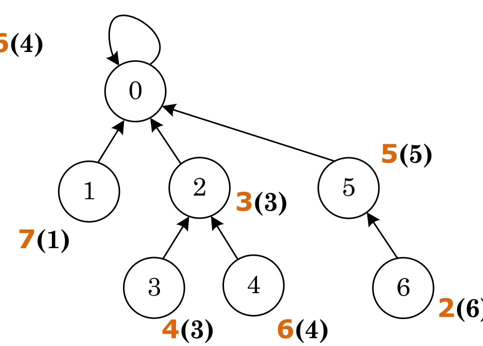
1(1) : rejected



8(2) : rejected



9(4) : rejected

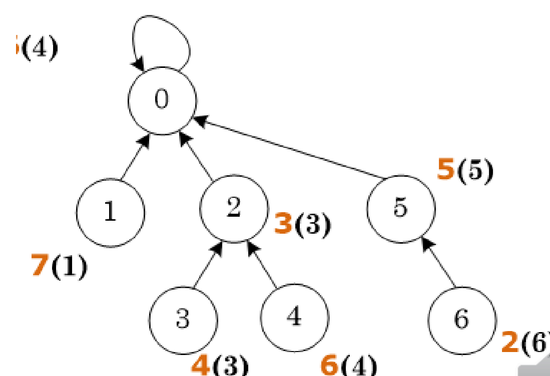


Job	Deadline	Profit
1	1	100
2	6	80
3	3	90
4	3	120
5	5	40
6	4	105
7	1	115
8	2	85
9	4	50

Another Algorithm Based on Disjoint Sets

- **Method**
  - ✓ d_{max} : the maximum of the deadlines for n jobs.
  - ✓ Add a job as late as possible to the schedule being built, but no later than its deadline.

  - Sort the jobs in non-increasing order by profit.
  - Initialize d_{max} +1 disjoint sets, containing the integers $0, 1, 2, \ldots, d_{max}$.
  - For each job in the sorted order,
    - Find the set S containing the minimum of its deadline and n.
      - If $\text{small}(S) = 0$, reject the job.
      - Otherwise, schedule it at time $\text{small}(S)$, and merge S with the set containing $\text{small}(S)$-1.



- Time complexity
  - $O(n \log m)$ for the disjoint set manipulation, where m is the minimum of n and d_{max}.
  - $O(n \log n)$ for sorting the profits.

[주제 6]

Graph Algorithms

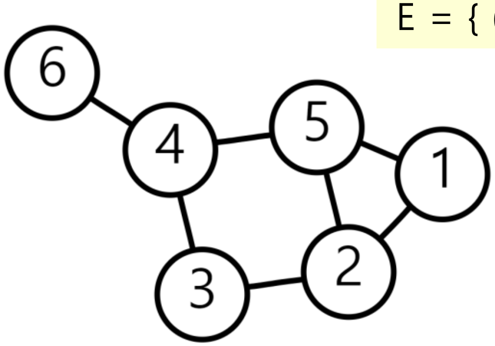
Basic Things to Know about Graph as a CSE Undergraduate

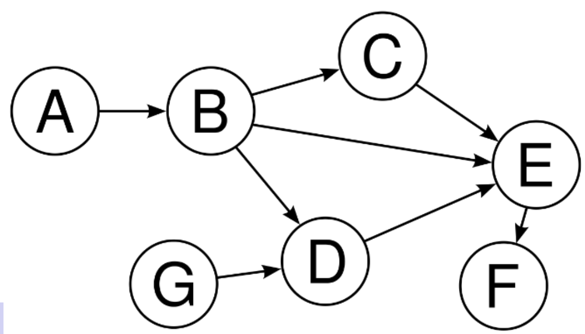
- Definitions and representations
- Graph traversal algorithms
  - Depth-first search
  - Breadth-first search
- Connectivity
  - Simple connectivity
  - Strong connectivity
  - Biconnectivity
  - Transitive closure
- Biconnected component algorithms
- **Shortest path algorithm**
  - All-pairs shortest path algorithm
  - Single-source shortest path algorithm
- **Minimum spanning tree algorithm**
  - Prim's minimum spanning tree algorithm
  - Kruskal's minimum spanning tree algorithm

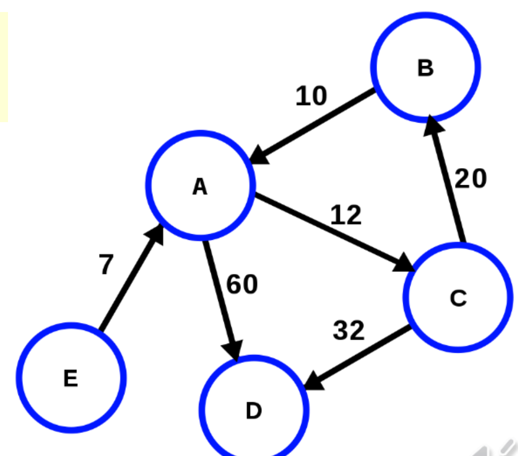
Definitions

- An (*undirected*, *simple*) *graph* G is defined to be a pair of (V, E) , where V is a non-empty finite set of elements called *vertices*, and E is a finite set of unordered pairs of distinct elements of V called edges.
  - $G = (V, E) = (V(G), E(G))$
  - Graphs that allow loops and multiple edges are often called a *general graphs*.
- A (*simple*) *digraph* D is defined to be a pair (V, A), where V is a non-empty finite set of elements called *vertices*, and A is a finite set of ordered pairs of distinct elements of V called *(directed) edges* or *(directed) arcs*.
- A *weighted graph* is a graph in which a number, called the *weight*, is assigned to each edge.

V = { 1, 2, 3, 4, 5, 6 }
E = { (1, 2), (1, 5), (2, 3), (2, 5), (3, 4), (4, 5), (4, 6) }

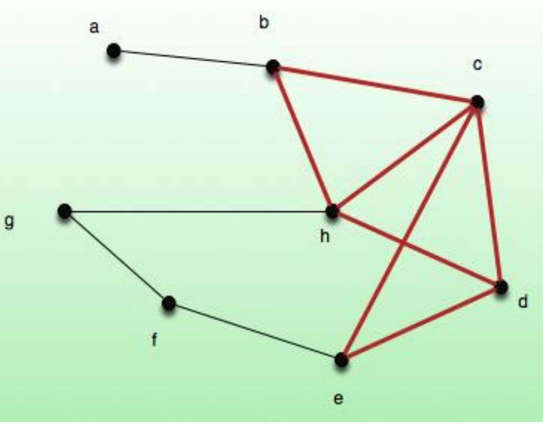






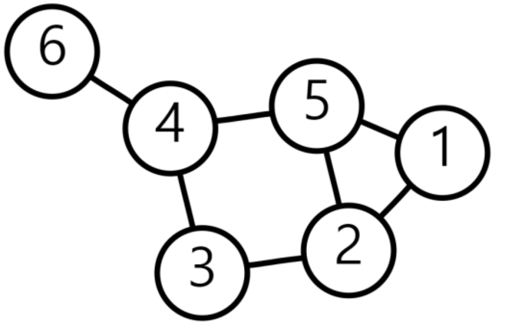
From Wikipedia

- A *subgraph* of a graph G is simply a graph, all of whose vertices belong to $V(G)$ and all of whose edges belong to $E(G)$.



- Adjacency and incidence
  – Two vertices v and w of a graph G are said to be *adjacent* if there is an edge joining them.
  – Two distinct edges of G are *adjacent* if they have at least one vertex in common.
  – The vertices v and w are then said to be *incident* to such an edge.
  – The *degree* of a vertex v of G is the number of edges incident to v.

From Wikipedia



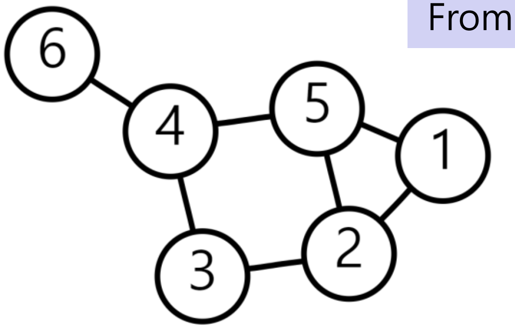
- **Walk, trail, circuit, path, and cycle**
  - A *walk* (or *edge-sequence*) is an alternating sequence of vertices and edges, starting and ending at a vertex, in which each edge is adjacent in the sequence to its two endpoints.
  - The *length* of a walk is the number of edges in it.
  - A *trail* is a walk in which all the edges are distinct from one another.
  - A walk is *closed* if it starts and ends at the same vertex.
  - A *circuit* is a trail that is closed.
  - A *path* is a walk in which all the vertices are distinct from one another.
  - A *cycle* is a path containing at least one edge with an exception that the first and last vertices coincide.

4 -> 5 -> 2 -> 3-> 4 -> 6

6 -> 4 -> 5 -> 2 -> 1

4 -> 5 -> 1 -> 2 -> 3 -> 4

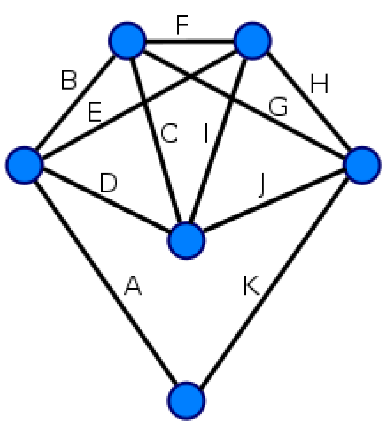
From Wikipedia



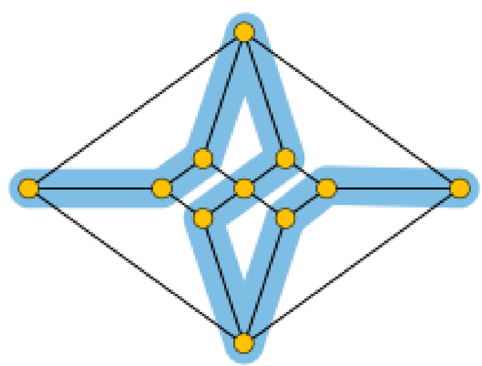
➢ *The definitions differ by various textbooks!!!*

	Walk	Trail	Circuit	Path	Cycle
Openness	Open/Closed	Open	Closed	Open	Closed
Vertex repetition	Allowed	Allowed	Allowed	Disallowed	Disallowed
Edge repetition	Allowed	Disallowed	Disallowed	Disallowed	Disallowed

- An *Eulerian trail* is a trail that visits every edge exactly once.
- An *Eulerian circuit* is an Eulerian trail that starts and ends on the same vertex.
- A *Hamiltonian path* is a path that visits each vertex exactly once.
- A *Hamiltonian cycle* is a Hamiltonian path that is a cycle.



From Wikipedia

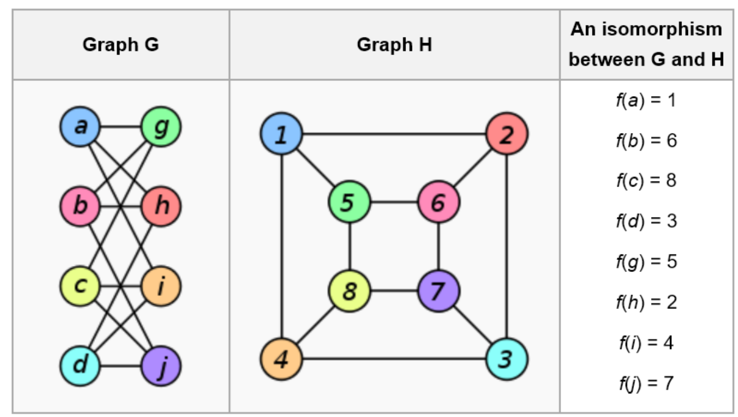


- An Eulerian circuit exists in a connected graph G if the degree of every vertex is even, and can be found in O(|E|) time.

- Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem, which is NP-complete.

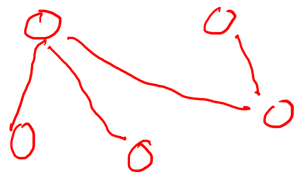
- Examples of graphs
  - A *null graph* is a graph whose edge-set is empty.
  - A *regular graph* is a graph in which each vertex has the same degree.
  - A *complete graph* is a graph in which each pair of vertices is joined by an edge.
  - A *bipartite graph* is a graph in which its vertex set can be partitioned into two sets V_1 and V_2, in such a way that every edge of the graph joins a vertex of V_1 to a vertex of V_2.
  - A *connected graph* is an undirected graph, in which, given any pair of vertices v and w, there is a path from v to w.
    - An arbitrary graph can split up into disjoint connected subgraphs called *connected components*.
  - A *tree* is a connected graph with no cycles.
  - A *forest* is a graph with no cycles.

Courtesy of Wikipedia

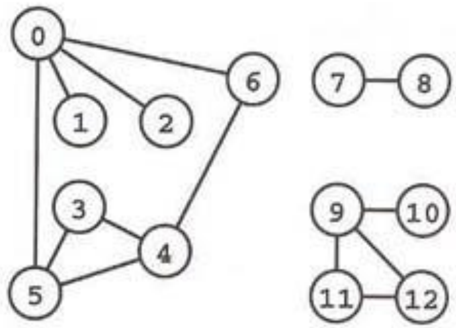


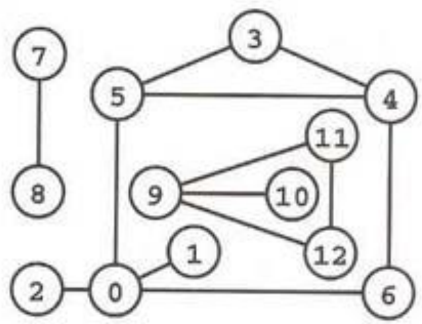
- Graph isomorphism
  - Two graphs G_1 and G_2 are *isomorphic* if there is a one-to-one correspondence between the vertices of G_1 and those of G_2 with the property that the number of edges joining any two vertices of G_1 is equal to the number of edges joining the corresponding vertices of G_2.

Graph Representation 1: Adjacency List



$0 \le m \le \frac{n(n-1)}{2}$





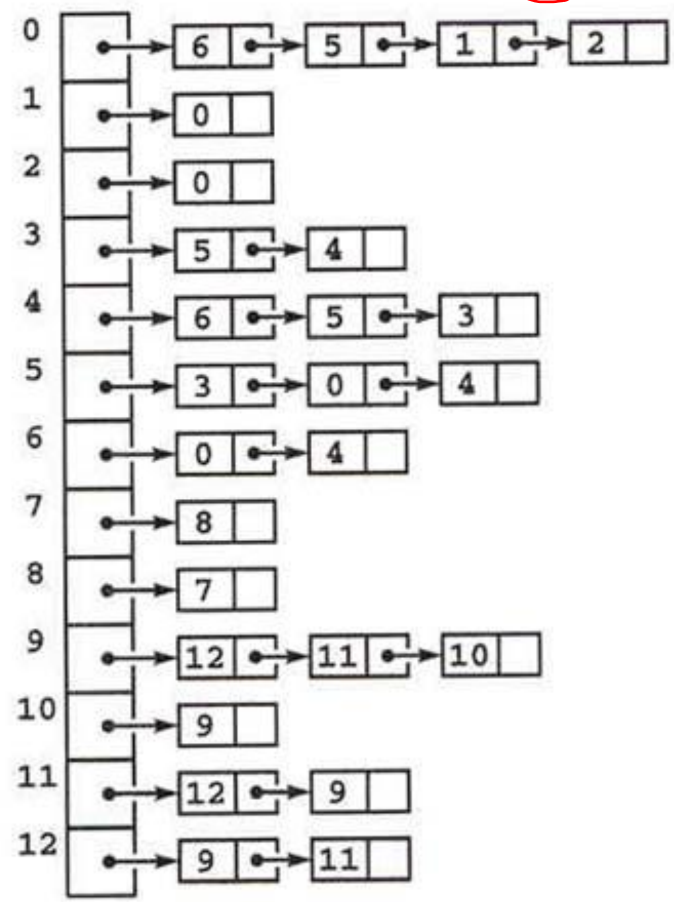
0-5	5-4	7-8
4-3	0-2	9-11
0-1	11-12	5-3
9-12	9-10	
6-4	0-6	

0: 1 2 5 6
1: 0
2: 0
3: 4 5
4: 3 5 6
5: 0 3 4
6: 0 4
7: 8
8: 7
9: 10 11 12
10: 9
11: 9 12
12: 9 11

In mathematics

$n = |V|$

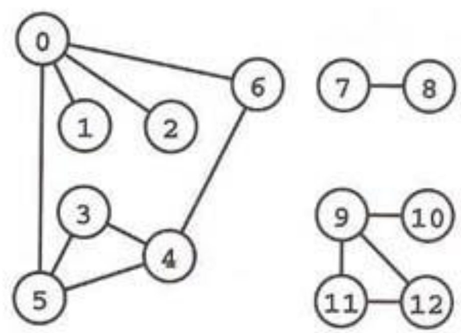
$m = |E|$

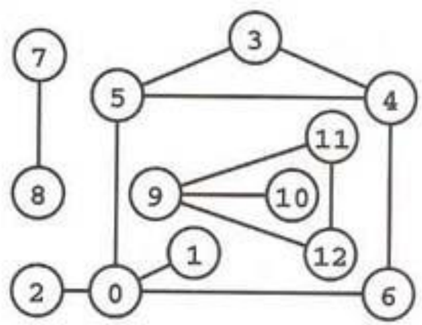


In C

$O(n+m)$

Graph Representation 2: Adjacency Matrix





0-5	5-4	7-8
4-3	0-2	9-11
0-1	11-12	5-3
9-12	9-10	
6-4	0-6	

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

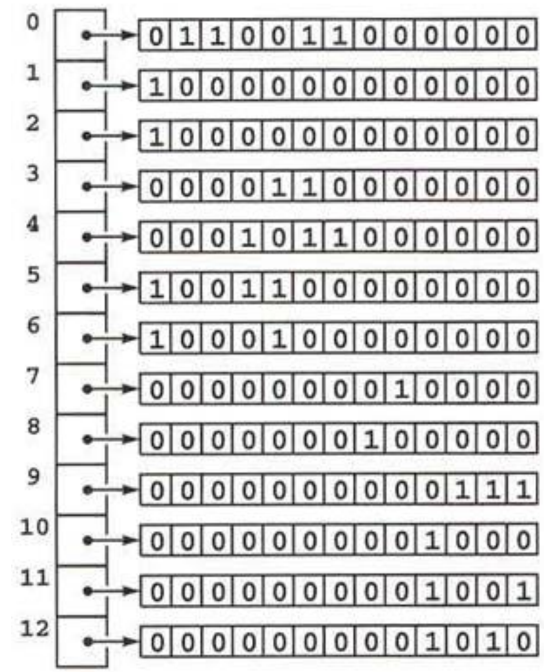
In mathematics

```
for int A[3][5];

  A[i][j]
= *(A[i] + j)
= (*(A + i))[j]
= *((*(A + i)) + j)
= *(&A[0][0] + 5*i + j)
```

$O(n^2)$

$n = |V|$



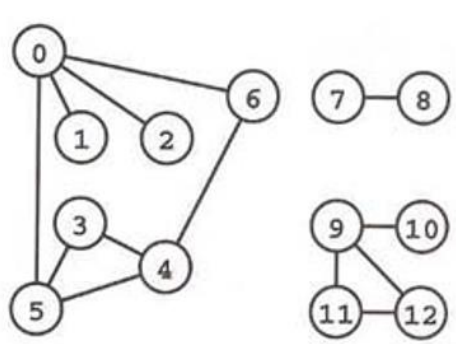
In C

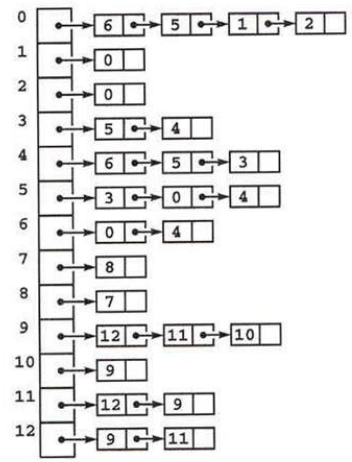
Various Costs for a Graph G = (V, E)

Sparse ← → Dense

$$0 \le |E| \le \frac{|V|(|V|-1)}{2}$$

	Adjacency list	**Adjacency matrix**
Space	O(|V| + |E|)	O(|V|²)
Initialize empty	O(|V|)	O(|V|²)
Copy after initialization	O(|E|)	O(|V|²)
Destroy	O(|E|)	O(|V|) or O(1)
Insert vertex u	O(1)	O(|V|) or O(|V|²)
Insert edge (u, v)	O(1)	O(1)

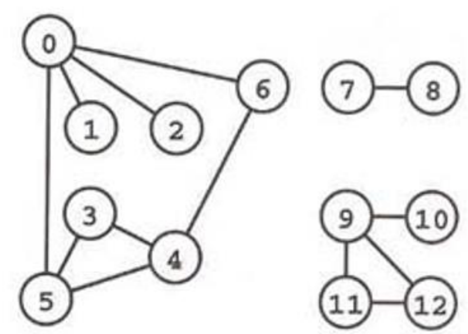
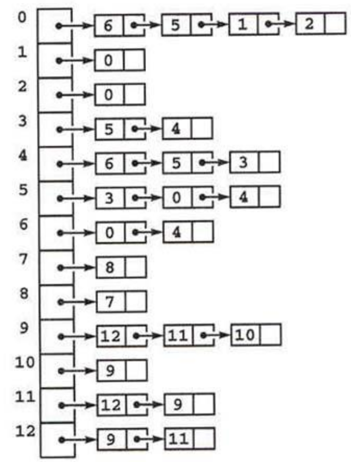




	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

Sparse ← → Dense

$$0 \le |E| \le \frac{|V|(|V|-1)}{2}$$

	Adjacency list	Adjacency matrix
		(see matrix below)
Remove vertex u	O(|E|)	O(|V|²)
Find/remove edge (u, v)	O(|V|)	O(1)
Are u and v adjacent?	O(|V|)	O(1)
Is *v* isolated?	O(1)	O(|V|)
Find a path from *u* to *v*	O(|V| + |E|)	O(|V|²)

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0