Data Structures and Algorithms

Lecture 14.1: Graphs (cont.)

Nopadon Juneam
Department of Computer Science
Kasetsart university

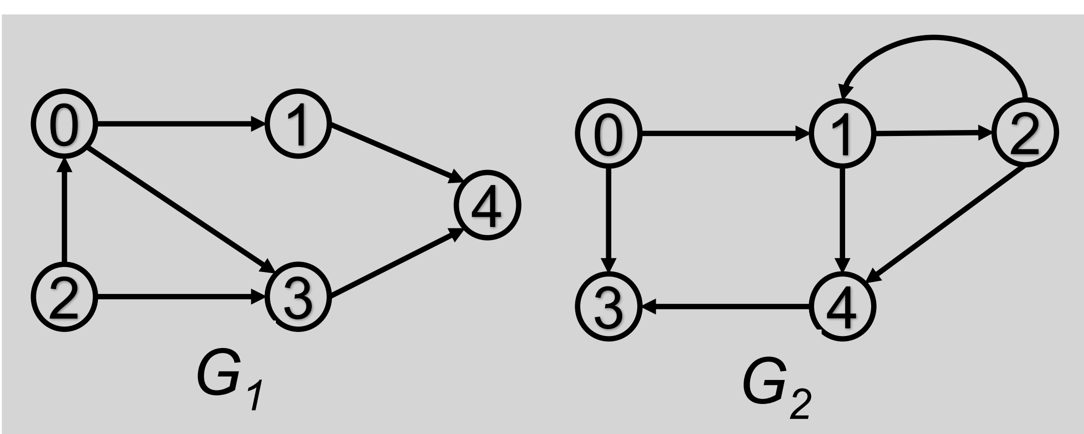
Outlines

- Graph representations: adjacency matrix & adjacency list (recaps)
- Basic operations on graphs (detailed)
- More operations on graphs
- Graph representations: pros & cons

Adjacency Matrix

- The ***adjacency-matrix representation*** is a 2D array of size *n* x *n*, where *n* is the number of vertices in a graph.

- The (*i*, *j*)-th entry of the array is 1 if there is an edge from vertex *i* to vertex *j*; otherwise, the (*i*, *j*)-th entry is 0.



$$A_{G_1} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \ A_{G_2} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Adjacency List (s)

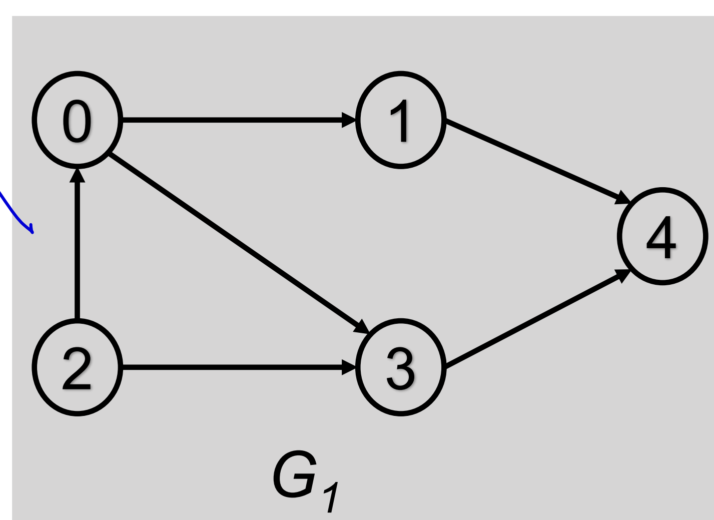
เป็น Array ของ n lists

- The ***adjacency-list representation*** of a graph *G* consists of an array *Adj* of *n* (จำนวน Node) lists, where *n is* the number of vertices in a graph; one list for each vertex in V.

- For each *u* in *V*, the adjacency list *Adj*[*u*] contains all the vertices *v* such that there is an edge (*u*, *v*) in *E.* In other words, *Adj*[*u*] consists of all the vertices adjacent to *u.*

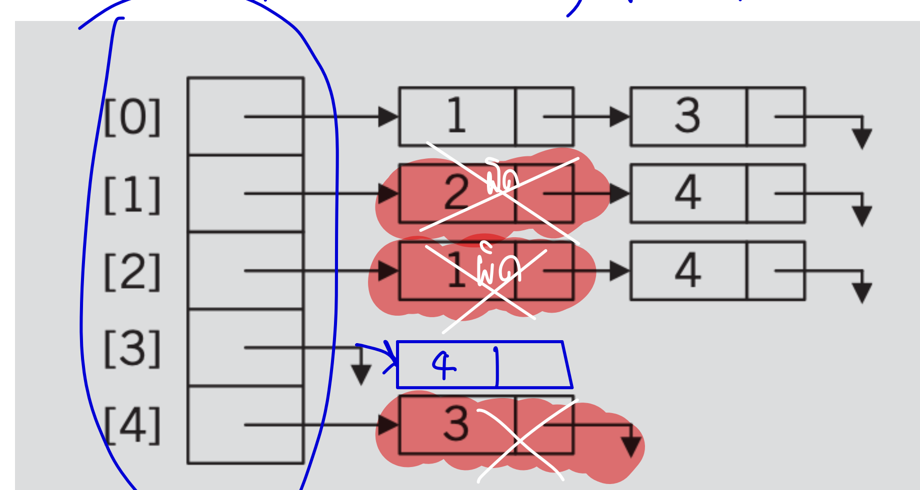
มี 5 Node

มี Array 5 list



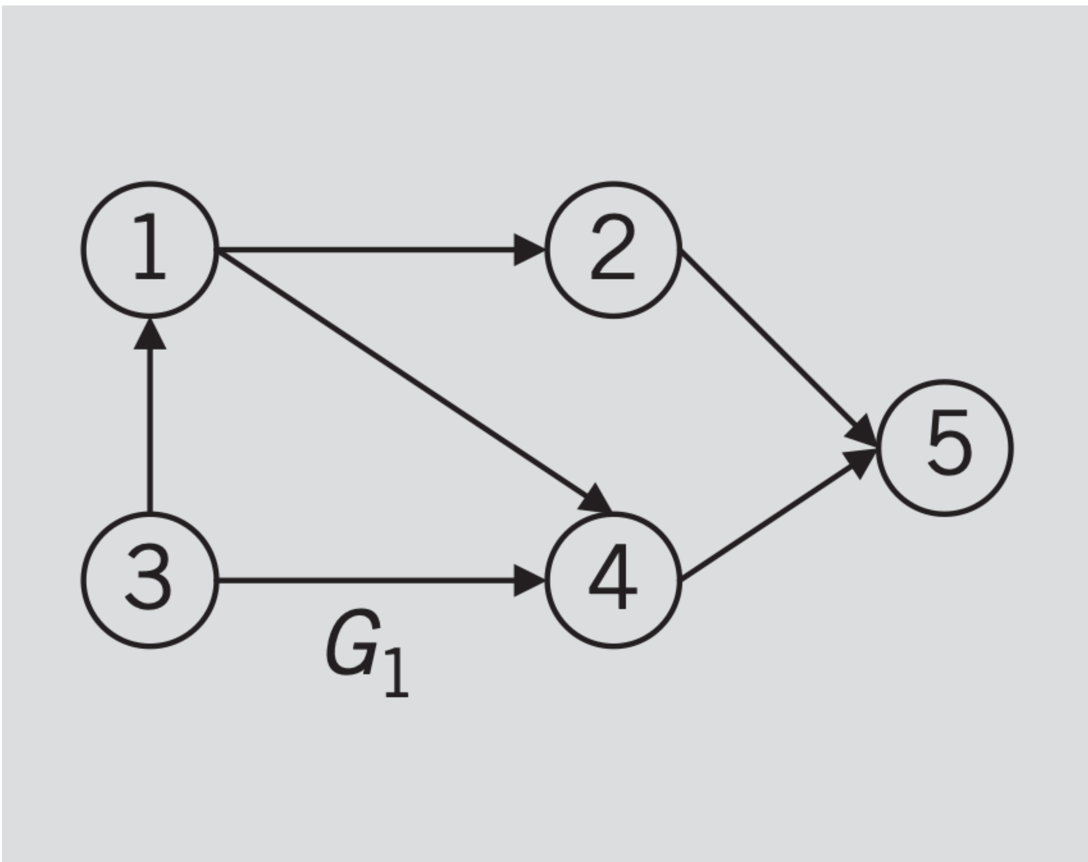
G_1

Array สามารถเป็น Array หรือ list



$$\begin{array}{c|c}
 & \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}
\end{array}$$

Programming Exercise



- Let’s try to create the above graph.

Basic Operations on Graphs

- Basic operations commonly performed on a graph:
  - `createGraph(n)`: create the empty graph with *n* isolated vertices;
  - `addEdge(G,u,v)`: add the edge from vertex *u* to vertex *v* in the graph *G*;
  - `printGraph(G)`: print the graph *G*;
  - `deleteGraph(G)`: delete the graph *G*;

Operation: createGraph

- createGraph(n): create the empty graph with n isolated vertices;

0x1 address

0x2 address

```
// Create the graph using adjacency-matrix representation
int** createGraph(const int n) {
    // Return 2D array of size n*n
    int** adjMatrix = malloc(sizeof(int*)*n);
    for (int i=0; i<n; i++) {
        adjMatrix[i] = malloc(sizeof(int)*n);
        for (int j=0; j<n; j++)
            adjMatrix[i][j] = 0;
    }
    return adjMatrix;
}
```

สร้าง 2Day Array

$O(|V^2|)$

```
// Create the graph using adjacency-list representation
struct Node** createGraph(int n){
    // Return array of n lists (vectors)
    struct Node** adjList = malloc(sizeof(struct Node*)*n);
    for(int i=0; i<n; i++)
        adjList[i] = NULL;
    return adjList;
}
```

$O(|V|)$

```
// Implementation of
singly linked list
struct Node
{
    int adj_vertex;
    struct Node* next;

};
```

หมายเลข vertex

Operation: addEdge

- `addEdge(G,u,v)`: add the edge from vertex *u* to vertex *v* in the graph *G*;

```
// Add the edge to the graph using
// adjacency-matrix representation
void addEdge(int** adjMatrix, int u,
int v) {
    adjMatrix[u][v] = 1;
}
```

} O(1)

คือ constant time

```
// Add the edge to the graph using adjacency-
// list representation
void addEdge(struct Node** adjList, int u, int v)
{
    struct Node* node = adjList[u];
    if(node == NULL) {
        node = malloc(sizeof(struct Node));
        node->adj_vertex = v;
        node->next = NULL;
        adjList[u] = node;
    } else {
        while(node->next != NULL)
            node =  node->next;
        struct Node* new_node =
malloc(sizeof(struct Node));
        new_node->adj_vertex = v;
        new_node->next = NULL;
        node->next = new_node;
    }
}
```

&*(adjList + u)

worst case

$O(\text{out.deg}(u))$

↓

$O(|V|)$

out.deg ≤ |V| − 1

$\sum out.deg(u) = |E|$

worst case

u → $n-1$

Operation: printGraph

มี out degree ได้สูงสุด n-1 edges

$0 \le |E| \le n(n-1)$

$O(n^2)$

- printGraph(G): print the graph *G*;

```
// Print the adjacency-matrix representation of the graph
void printGraph(int** adjMatrix, int n)
{
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++)
            printf("%d ", adjMatrix[i][j]);
        printf("\n");
    }
}
```

$O(|V|^2)$

```
// Print the adjacency list representation of the graph
void printGraph(struct Node** adjList, int n)
{
    for (int u = 0; u < n; u++) {
        printf("[%d] head: ", u);
        struct Node* node = adjList[u];
        while(node) {
            printf("-> %d ", node->adj_vertex);
            node = node->next;
        }
        printf("-> NULL \n");
    }
}
```

จำนวน Node

วนลูปเพื่อไปแต่ละ Node

$O(|V|+|V|^2) = O(|V|^2)$

ดีกว่า เพราะ complexity ต่ำกว่า n^2 ได้

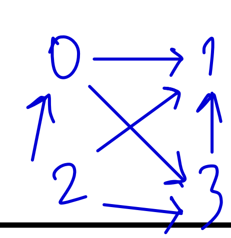
$O(out.deg(u))$

$\sum_{u \in V} O(outdeg(u))$

$O(|V|+|E|)$

จะทำงาน 4+6 ครั้ง

V	E
0	1 3
1	
2	0 1 3
3	1



Operation: deleteGraph

- deleteGraph(G): delete the graph *G*;

```c
// Delete the adjacency-matrix representation of the graph
void deleteGraph(int** adjMatrix, int n){
    for (int i=0; i<n; i++) {
        free(adjMatrix[i]);
    }
    free(adjMatrix);
}
```

$O(|V|)$

```c
// Delete the adjacency list representation of the graph
void deleteGraph(struct Node** adjList, int n) {
    for (int u=0; u<n; u++) {
        struct Node* node = adjList[u];
        while(node != NULL) {
            struct Node* next_node = node->next;
            free(node);
            node = next_node;
        }
    }
    free(adjList);
}
```

เก็บตัว Next Node

เพื่อให้ไปตัวถัดไปได้

$O(|V|+|E|)$

More Operations on Graphs

- removeEdge(G, u, v): remove the existing edge from vertex *u* to vertex *v*.

ให้ nodes คือ 0, 1, ..., n-1

- addVertex(G, u): add the new vertex *u* to the graph *G*;
- removeVertex(G, u): remove the existing vertex *u* from the graph *G*;

มี Edge จาก u → v หรือไม่

- isAdjacent(G, u, v): check whether vertices *u* and *v* are adjacent in *G*;
- inDegree(*G,* u): return the *in-degree* of vertex *u* in G;
- outDegree(*G,* u): return the *out-degree* of vertex *u* in G;

Operation: removeEdge

- removeEdge(G, u, v): remove the existing edge from vertex *u* to vertex *v*;

```
// Remove the existing edge from
// the graph using adjacency-matrix
// representation
void removeEdge(int** adjMatrix,
int u, int v) {
    adjMatrix[u][v] = 0;        O(1)
}
```

```
// Remove the existing edge from the graph using
// adjacency-list representation
void removeEdge(struct Node** adjList, int u,
int v) {
    struct Node* node = adjList[u];
    if(node->adj_vertex == v) {
        adjList[u] = node->next;
        free(node);        ให้ตัวถัดไปเป็น head แทน
    } else {
        struct Node* prev_node = node;
        node = node->next;
เข้าไป list u
        while(node->adj_vertex != v) {
เพื่อไปเอา v ออก
            prev_node = node;
            node = node->next;
        }   node ->adj_vertex == v แล้ว
        prev_node->next = node->next;
        free(node);
    }
}
```

Operation: addVertex

- addVertex(G, u): add the new vertex *u* to the graph *G*;

```
// Add the new vertex to the graph
// using adjacency-matrix
// representation
int** addVertex(int** adjMatrix, int *n,
int u) {
    int** adjMatrix_new =
createGraph(u+1);
    for (int i=0; i<*n; i++) {
        for (int j=0; j<*n; j++)
            if(adjMatrix[i][j] == 1)
                adjMatrix_new[i][j] = 1;
    }
    deleteGraph(adjMatrix, *n);
    *n = u+1;
    return adjMatrix_new;
}
```

Copy ของเดิมที่มีอยู่

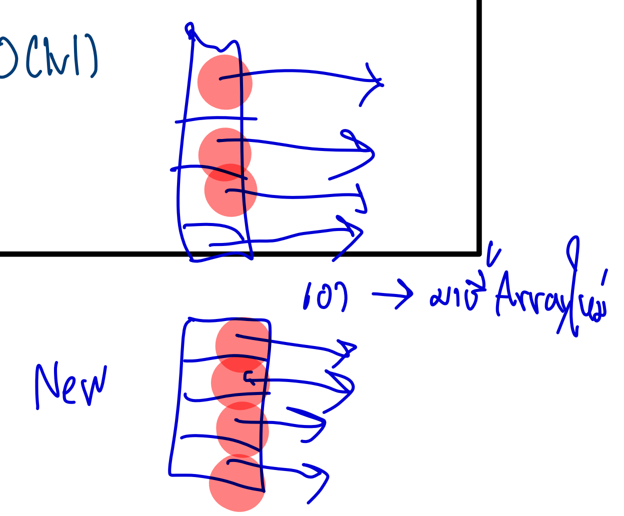
```
// Add the new vertex to the graph
// graph using adjacency-list
// representation
struct Node** addVertex(struct Node** adjList,
int *n, int u) {
    struct Node** new_adjList =
createGraph(u+1);
    for (int i=0; i<*n; i++) {
        new_adjList[i] = adjList[i];
        adjList[i] = NULL;
    }
    deleteGraph(adjList, *n);
    *n = u+1;
    return new_adjList;
}
```

createGraph(u+1); → $O(|V|)$

Copy ของเดิม $O(|V|)$

deleteGraph(adjList, *n); $O(|V|)$

ดังนั้นจึงเป็น $O(|V|)$



Operation: removeVertex

- removeVertex(G, u): remove the existing vertex u from the graph G

```c
// Remove the existing vertex from the graph
// using adjacency-matrix representation
int** removeVertex(int** adjMatrix, int* n, int u)
{
    for (int i=0; i<*n; i++)
        for (int j=u; j<*n-1; j++)
            adjMatrix[i][j] = adjMatrix[i][j+1];

    for (int i=0; i<*n; i++)
        for (int j=u; j<*n-1; j++)
            adjMatrix[j][i] = adjMatrix[j+1][i];

    int** adjMatrix_new = createGraph(*n-1);
    for (int i=0; i<*n-1; i++)
        for (int j=0; j<*n-1; j++)
            adjMatrix_new[i][j] = adjMatrix[i][j];
    deleteGraph(adjMatrix, *n);
    (*n)--;
    return adjMatrix_new;
}
```

$\rightarrow O(|V|^2)$

$\} \; O(|V|^2)$

$\rightarrow O(|V|^2)$

$\Rightarrow O(|V|^2)$

$O\left(\sum_{v \in V} \text{outdeg}(v)\right) = O(|E|)$

```c
// Remove the existing vertex
// from the graph using
// adjacency-list representation
void removeVertex(struct Node** adjList,
int *n, int u) {

    for(int v=0; v<*n; v++) {
        if(isAdjacent(adjList, v, u) == 1)
            removeEdge(adjList, u, v);
    }
    struct Node* node = adjList[u];
    while(node != NULL) {
        struct Node* next_node = node-
>next;
        free(node);
        node = next_node;
    }
    adjList[u] = NULL;
    if(u < *n-1)
        return;
    (*n)--;
}
```

Operation: isAdjacent

- isAdjacent(G, u, v): check whether vertices *u* and *v* are adjacent in *G*;

เข้าไปที่ list ของ u ดูทั้งหมดว่ามี v ไหม

```c
// Check whether the two vertices
// are adjacent using adjacency-
// matrix representation
int isAdjacent(int** adjMatrix, int u,
int v) {        เช็คแถว u คอลัมน์ v == 1
    if(adjMatrix[u][v] == 1)
        return 1;                         O(1)
    else
        return 0;
}
```

```c
// Check whether the two vertices
// are adjacent using adjacency-
// list representation
int isAdjacent(struct Node** adjList, int u,
int v) {
    struct Node* node = adjList[u];
    int ret = 0;
    while(node != NULL) {
        if(node->adj_vertex == v)
            ret = 1;  เจอ Return ได้เลย      O(out·deg(|u|))
        node = node->next;
    }
    return ret;
}
```

ไม่เร็ว
ไอดาวน์น

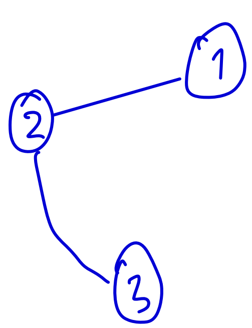
Operation: inDegree

- **`inDegree(`*`G`*`, u)`**: return the *in-degree* of vertex *u* in G;

```
// Report the in-degree of the
// vertex using adjacency-
// matrix representation
int inDegree(int** adjMatrix, const
int n, int u) {
    int in_deg = 0;
    for(int i=0; i<n; i++) {
        if(adjMatrix[i][u])
            in_deg++;
    }
    return in_deg;
}
```

$O(|V|)$

$u = 2$



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

ดูจาก Out Node ที่ว่ามี u หรือไม่

```
// Report the in-degree of the
// vertex using adjacency-
// list representation
int inDegree(struct Node** adjList, int n,
int u) {
    int in_deg = 0;

    for(int i=0; i<n; i++) {
        struct Node* node = adjList[i];
        while(node) {
            if(node->adj_vertex == u)
                in_deg++;
            node = node->next;
        }
    }
    return in_deg;
}
```

$O(|V|+|E|)$

Operation: outDegree

- outDegree(G, u): return the *out-degree* of vertex *u* in G;

```
// Report the out-degree of the
// vertex using adjacency-
// matrix representation
int outDegree(int** adjMatrix, const
int n, int u) {
    int out_deg = 0;
    for(int i=0; i<n; i++) {
        if(adjMatrix[u][i])
            out_deg++;
    }
    return out_deg;
}
```

$O(|V|)$

```
// Report the out-degree of the
// vertex using adjacency-
// list representation
int outDegree(struct Node** adjList, int
n, int u) {
    struct Node* node = adjList[u];
    int out_deg = 0;
    while(node != NULL) {
        out_deg++;
        node = node->next;
    }
    return out_deg;
}
```

$\Theta(\text{out.deg}(u))$

Graph Representations: Complexity of Operations

Operation	Adjacency Matrix	Adjacency List
createGraph (space to store graph)	$O(\|V\|^2)$	$O(\|V\|)$
addEdge	$O(1)$	$O(\|V\|)$
addVertex	$O(\|V\|^2)$	$O(\|V\|)$
removeVertex	$O(\|V\|^2)$	$O(\|E\|)$
removeEdge	$O(1)$	$O(\|V\|)$
isAdjacent	$O(1)$	$O(\|V\|)$
inDegree	$O(\|V\|)$	$O(\|V\|+\|E\|)$
outDegree	$O(\|V\|)$	$O(\|V\|)$
space to store graph Remarks	$O(\|V\|^2)$ Slow to add/remove vertices as matrix must be resized/copied	$O(\|V\|+\|E\|)$ → เก็บเท่ากับจำนวน graph จริงๆ [illegible] Slow to remove edges because it needs to iterate all the adjacent vertices

Graph Representations: Pros & Cons

- *Remarks*:
  - ***Adjacency matrix:*** Slow to add/remove vertices because matrix must be resized/copied
  - **Adjacency list**: Slow to remove edges because it needs to iterate through all the adjacent vertices
- *Conclusions:* Adjacency list is generally preferred if the graph is ~~sparse~~ ห่าง *sparse*, i.e., when $|E| << |V|^2$; กราฟห่าง พิจารณาจาก $|E|$ ถ้าห่างจาก n^2 เยอะ = ห่าง Adjacency matrix is preferred if the graph is *dense,* i.e., when $|E| \approx |V|^2$ = แน่น

เข้าใกล้

O(|V|)

$O(|V| + |E|)$ → เก็บเท่ากับจำนวน graph จริงๆ กับเส้นเชื่อม

emove edges because
o iterate all the

มีผลกับตรงนี้ เพราะถ้าแน่น adj list ก็จะใกล้กับ adj matrix

จะต้องเช็คอีก