



Department of Computer Science and Engineering
The Chinese University of Hong Kong

CSCI2100B CSCI2100S DATA STRUCTURES

.....
Spring 2011

Elementary Graph Algorithms



Applications of Graphs

Many computational applications naturally involve both a set of **items** and a set of **connections** between the items.

In these situations, we **abstract objects** by graphs.

Graph is widely used in many different fields:

- Transactions
- Matching
- Networks
- Program Structure
- Maps
- Circuits
- Schedules

Graph: Basic Definition

- Graph is all about **connectivity**
 - How far are the two places?
 - How much money I owe you?
- We represent the objects as **vertices**(vertex).
- If two objects are connected / have some relationship, there is an **edge** between the vertices.




Definition: A graph is a set of vertices plus a set of edges that connect pairs of distinct vertices (with at most one edge connecting any pairs).

Graph: Vertices and Edges

- A graph $G = (V, E)$ consists of a set of **vertices** V and a set of **edges**, E .
- We use names 0 to $V - 1$ for the vertices.
 - In applications, you may **build a symbol table** to map vertex number with actual object names.
- Each **edge is a pair (v, u)** , where $v, u \in V$.
 - Sometimes we may write an edge as $v-u$.
- Vertex u is **adjacent** to v if and only if $(v, u) \in E$.
- An edge may have **weights** or **costs**.
- If there is no ambiguity, we denote the number of vertices and edges by V and E respectively.

Simple Graphs & Directed Graphs

- A graph is simple if there is no parallel and self-loop edges.
 - In this note, we assume all graphs are simple.
- A simple graph with V vertices has at most $V(V-1) / 2$ edges. 
- If the pair is ordered, the graph is directed, also known as digraph with directed edges.
 - Otherwise, the graph is undirected.

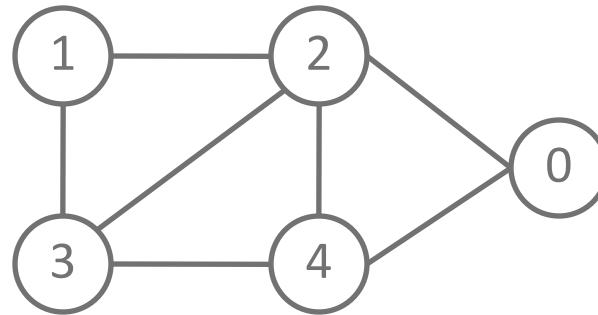
Some more terminologies

Graph	Implementation
Vertex	Node
Edge	Link

Graph: Example

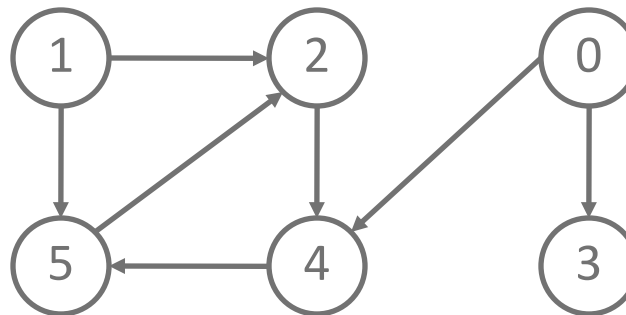
simple, undirected

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



simple, directed

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



Note: There are many ways to draw the same graph on paper.

Graph: Paths

- A path is a **sequence** of vertices v_1, v_2, \dots, v_N such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < N = |V|$.
 - The path is **simple** if the vertices and edges are distinct.
- A cycle is a simple path that the first and the final vertices are the same.
 - In a directed graph, a **cycle** is a path of length at least 1 such that $v_1 = v_N$.
- A directed acyclic graph is named as **DAG**.
- The **length** of a path/cycle is the number of edges on the path.

Adjacency Matrix Representation

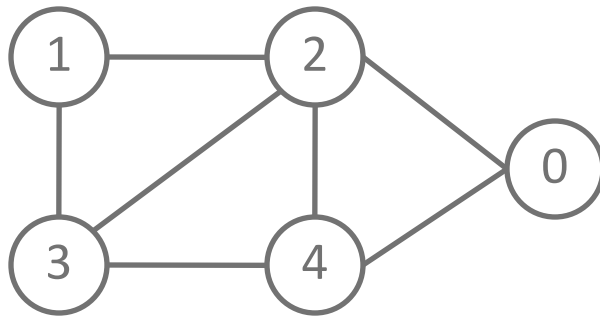
- Suitable for representing **dense** graphs: 
 E is close to V^2 .

- Then the adjacency-matrix representation of a graph G consists of a $V \times V$ matrix $A = (a_{ij})$ such that

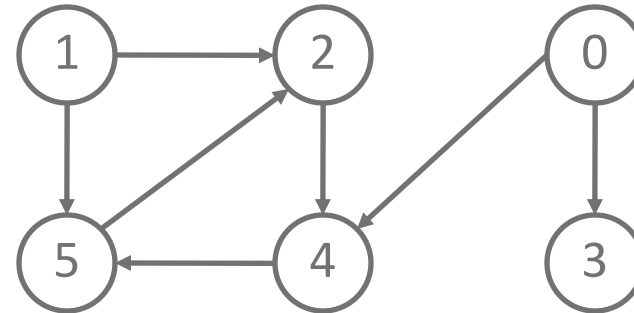
$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- Memory requirement: **$O(V^2)$** , independent of E .
- For undirected graph, the size of the matrix can be reduced to half (theoretically).

Adjacency Matrix: Examples



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



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

Adjacency Matrix Implementation

```
struct graph_t {  
    int V;  
    int E;  
    int **adj;  
};
```

```
typedef struct {  
    int v;  
    int w;  
} edge_t;
```

```
static int **matrix_init(int r, int c, int val){  
    int i, j;  
    int **t = malloc(r * sizeof(int *));  
    for (i = 0; i < r; i++)  
        t[i] = malloc(c * sizeof(int));  
    for (i = 0; i < r; i++)  
        for (j = 0; j < c; j++)  
            t[i][j] = val;  
    return t;  
}  
graph graph_init(int V){  
    graph G = malloc(sizeof *G);  
    G->V = V;  
    G->E = 0;  
    G->adj = matrix_init(V, V, 0);  
    return G;  
}
```

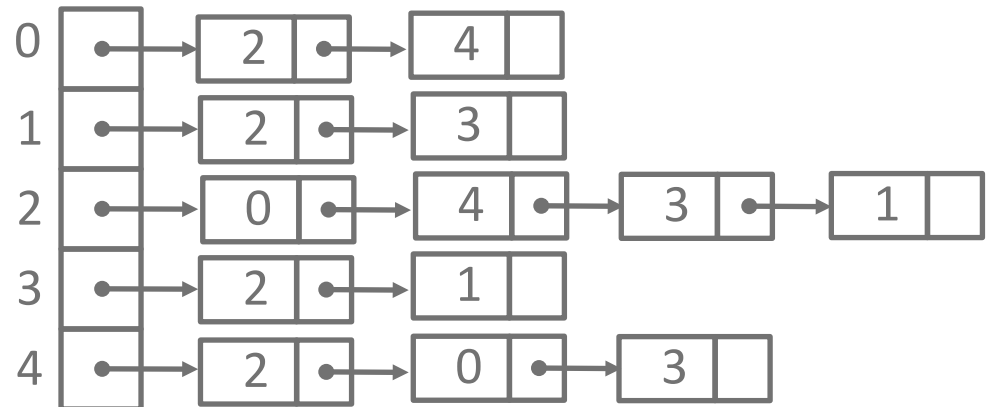
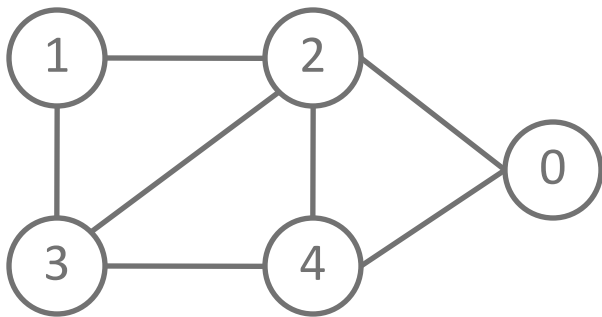
Adjacency Matrix Implm. (2)

```
void graph_insert_e(graph G, edge_t e){
    int v = e.v, w = e.w;
    if (G->adj[v][w] == 0)
        G->E++;
    G->adj[v][w] = 1;
    G->adj[w][v] = 1;
}

void graph_remove_e(graph G, edge_t e){
    int v = e.v, w = e.w;
    if (G->adj[v][w] == 1)
        G->E--;
    G->adj[v][w] = 0;
    G->adj[w][v] = 0;
}
```

Adjacency List Representation

- Suitable for representing **sparse** graphs:
 $E \ll V^2$.
- Consists of an array ***adj[]*** of V linked lists, one for each vertex.
- The **adjacency list *adj[u]*** contains all the vertices v such that there is an edge $(u, v) \in E$.
- The vertices are stored in an **arbitrary** order.



Properties of Adjacency List

- If G is directed, the sum of the lengths of all the adjacency list is E
 - An edge (u, v) is represented by having v in $adj[u]$.
- If G is an undirected graph, the sum of the lengths is $2E$, since (u, v) appears in both $adj[u]$ and $adj[v]$.
- Desirable Memory requirement: $O(V + E)$.
- It can be adopt to weighted graphs with a weight function $w: E \rightarrow R$
- **✗ Disadvantage**: it is not easy to tell a given edge (u, v) is present in the graph.
You need to search list $adj[u]$ in $O(V)$ time.

Adjacency List Implementation

```
struct graph_t {  
    int V;  
    int E;  
    node **adj;  
};
```

```
typedef struct node_s node;  
struct node_s {  
    int v;  
    node *next;  
};
```

```
graph graph_init(int V){  
    int i;  
    graph G = malloc(sizeof *G);  
    G->V = V;  
    G->E = 0;  
    G->adj = malloc(V * sizeof(node *));  
    for (i = 0; i < V; i++)  
        G->adj[i] = NULL;  
    return G;  
}
```

Adjacency List Implementation (2)

graph_free: remember to free the graph after use.

```
void graph_free(graph G){
    int i;
    node *t, *u;
    for (i = 0; i < G->V; i++)
        for (t = G->adj[i]; t != NULL; t = u){
            u = t->next;
            free(t);
        }
    free(G->adj);
}
```

graph_insert_edge: extract the adj. list and add to that list.

```
void graph_insert_e(graph G, edge_t e){
    int v = e.v, w = e.w;
    G->adj[v] = insert(w, G->adj[v]);
    G->adj[w] = insert(v, G->adj[w]);
    G->E++;
}
```

Adjacency List Implementation (3)

graph_show: print out the adj. lists for inspection/debugging.

```
void graph_show(graph G){
    int i;
    node *t;
    printf("%d vertices, %d edges\n", G->V, G->E);
    for (i = 0; i < G->V; i++){
        printf("%2d:", i);
        for (t = G->adj[i]; t != NULL; t = t->next)
            printf(" %2d", t->id);
        printf("\n");
    }
}
```

Exercise:

Can you write a procedure to count the number of edges in a graph, based on the adjacency list representation?

Exploring a Maze

- How to explore a maze without **getting lost**?
 - Unroll a ball of **string** → always find way out
- How to ensure we explore **every part** of the maze (treasure hunting)?
 - **Do not retrace steps** unless we have to
 - We need to mark places that we have visited

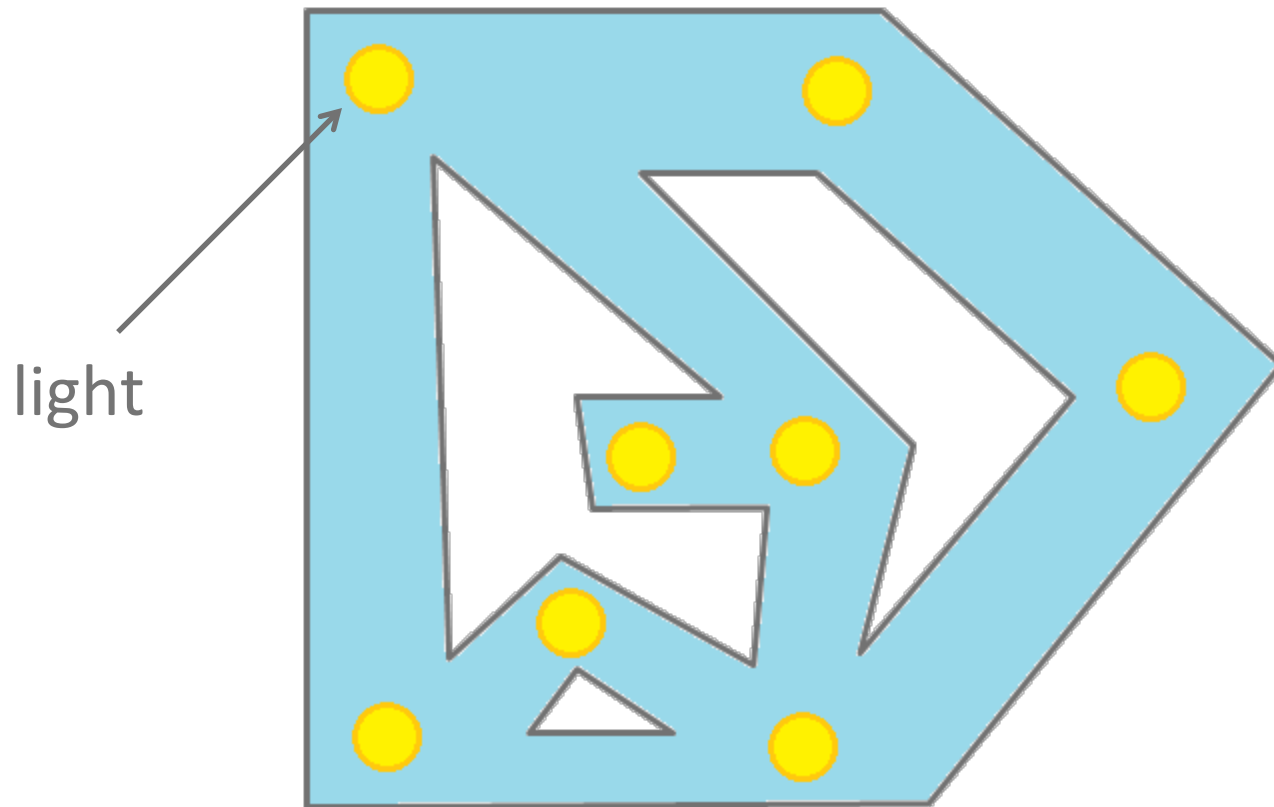
Maze Properties

Lights (initially off) in every intersections

Doors (initially close) at both ends of every passage

Windows on doors that the light is strong enough we can whether see another end is lit or not.

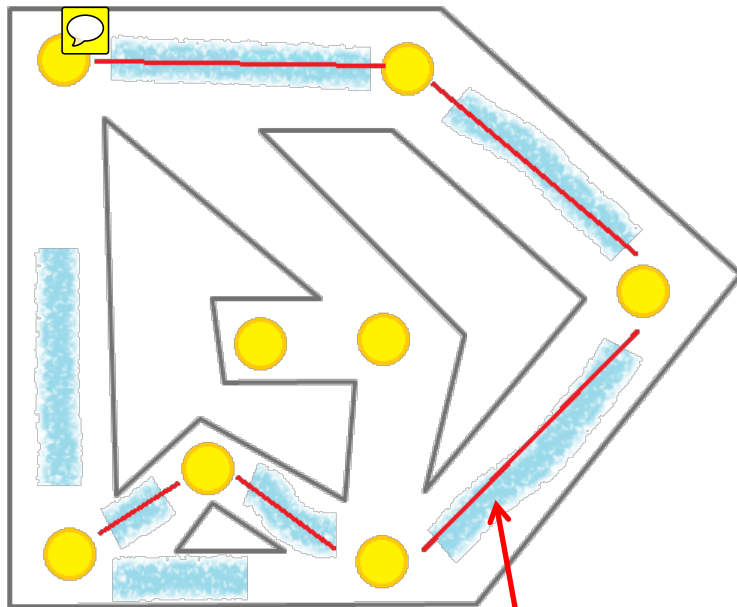
Maze Example



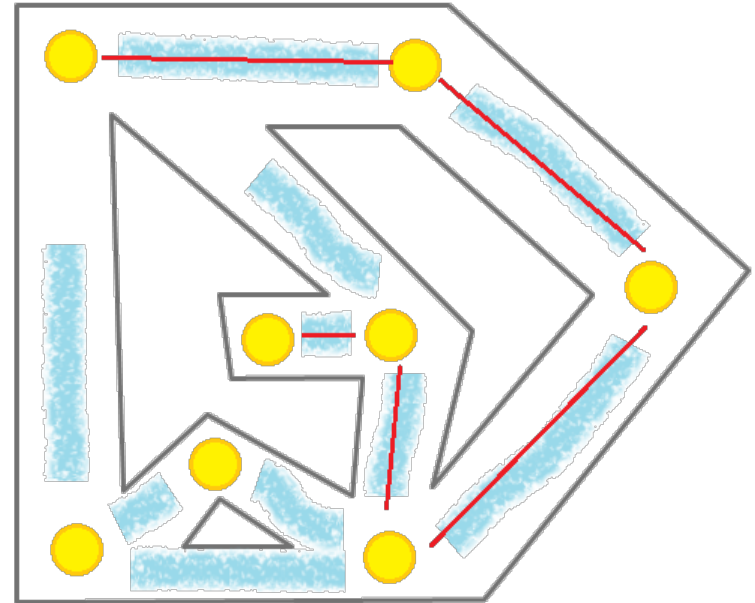
Trèmaux Exploration

- 1) If there is no closed doors at the current intersection, go to step (3).
Otherwise, open any closed door and leave it open.
- 2) If you see another the other end is already lighted, try another door.
Otherwise, follow the passage and unroll the string as you go, turn on the light and go to step (1).
- 3) If all the doors are open, check whether you are back to the start point. If so, stop.
If not, use the string to go back. Rolling the string as you go, and look for another closed door (repeat step (1)).

Trèmaux Exploration: Illustration



string



Depth-First Search (DFS)

- The strategy in depth-first search (DFS) is to search **deeper** in the graph whenever possible.
- We explore out of the most recent discovered vertex v that still has **unexplored** edges leaving it.
- When all v 's edges have been explored, the search **backtracks** to explore edges leaving the vertex from which v is discovered.
- If any discovered vertices remain, then one of them is selected as a new source and the search is **repeated**.
- The entire process is repeated until all vertices are discovered.

DFS: Code

```
int cnt, *pre; /* global variables for recursion */
void graph_dfs(graph G){
    int i;
    pre = malloc(sizeof(int) * G->V);
    cnt = 0;
    for (i = 0; i < G->V; i++)
        pre[i] = -1;
    for (i = 0; i < G->V; i++)
        if (pre[i] == -1)
            dfs_visit(G, new_edge(i, i));
    free(pre);
}
```

Initialization: $O(V)$

For each unvisited vertex in the graph, we start the **recursive** call `dfs_visit()`.

```
void dfs_visit(graph G, edge_t e){
    node *t;
    int w = e.w;
    pre[w] = cnt++;

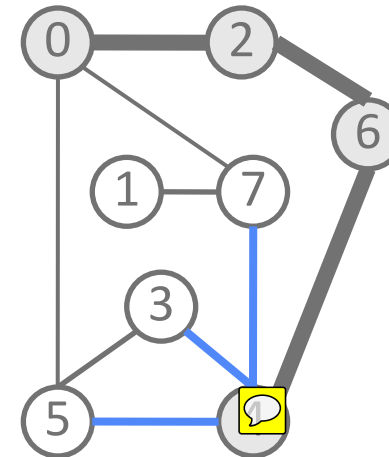
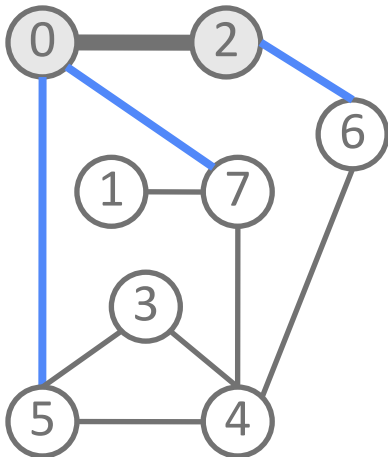
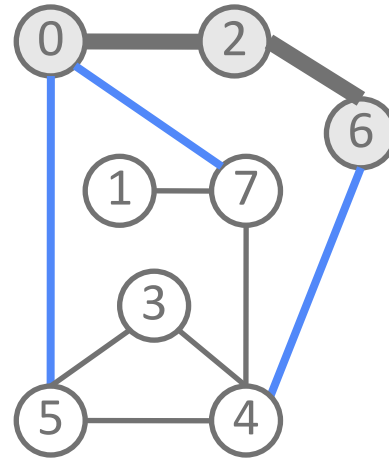
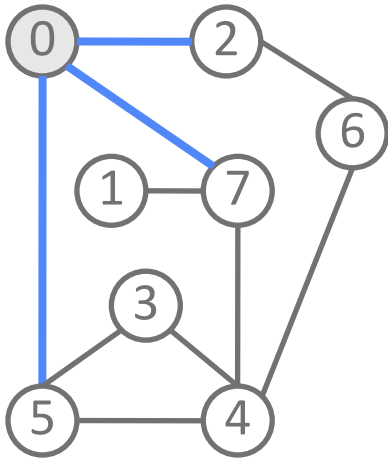
    for (t = G->adj[w]; t != NULL; t = t->next)
        if (pre[t->v] == -1)
            dfs_r(G, new_edge(w, t->v));
}
```

Mark visited: record the order of visit

Continue to traverse unvisited vertices



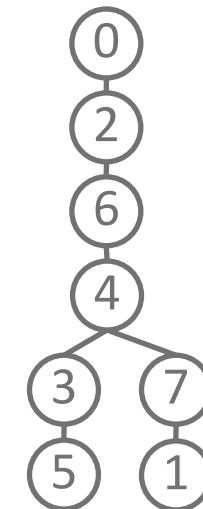
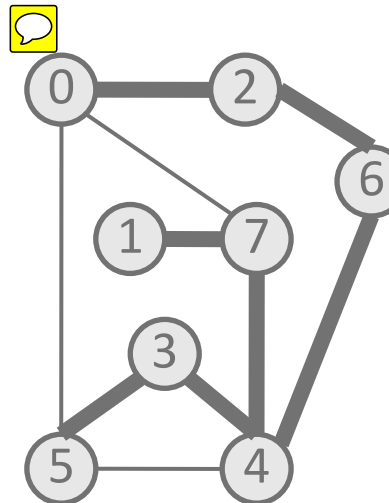
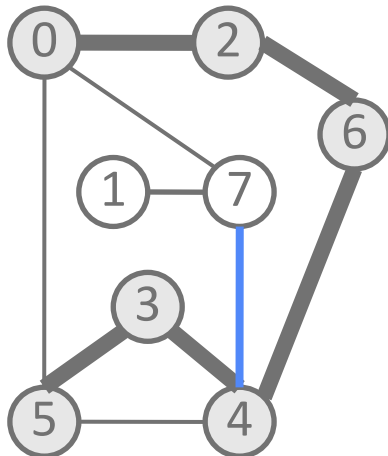
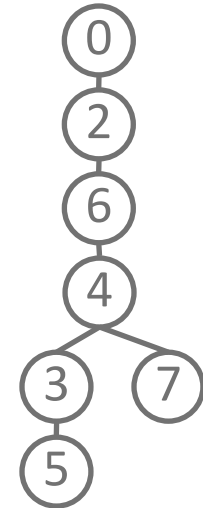
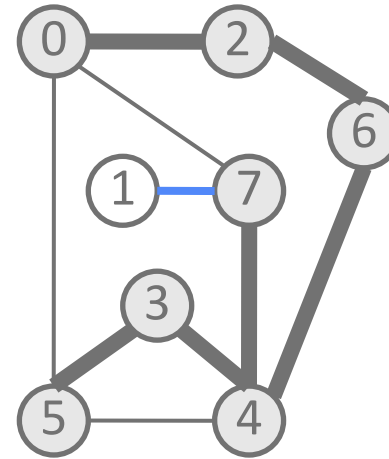
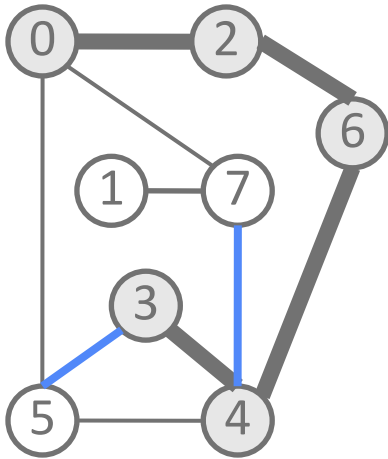
DFS: Example



DFS Tree



DFS: Example (2)



DFS: Analysis

- Initializations and first loop execute in $O(V)$ time.
- dfs_visit() executes **once for each** vertex.
- Inside dfs_visit(), the loop executes $|adj[v]|$ times. Since

$$\sum_{v \in V} |adj[v]| = O(E)$$

- The total cost of line 23-28 is $O(E)$.
- The **running time of DFS is $O(V + E)$** .
- Would this take longer if adjacency matrix is used instead?



Breadth-First Search (BFS)

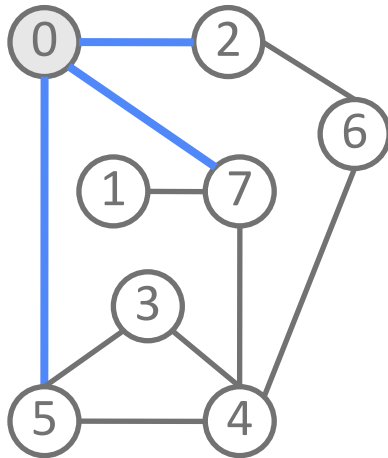
- Breadth-first search (BFS) is one of the simplest algorithms for **searching** a graph.
- Given a graph $G = (V, E)$, we pick a distinguished **source vertex** s , BFS systematically explores the edges of G to **discover every vertex** that is reachable from s .
- It computes the **shortest distance** (smallest number of edges) from s to each reachable vertex.
- Works on both directed or undirected graphs.
- **Breadth**: discovers all vertices at distance k from s before discovering any vertex **at distance $k + 1$.**

BFS: Procedure

- To keep track of progress, breadth-first search **marks** each vertex as visited or unvisited.
- A vertex is discovered the first time, it will be marked visited.
- A queue of edges is used to **make sure** the search proceeds in a breadth-first manner.
 - Take edges from the queue until it is **empty**.
 - **Visit** that vertex; put all edges that go from the vertex to unvisited vertices onto the queue.

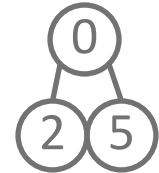
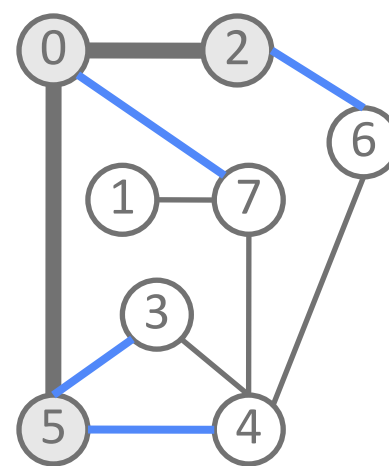


BFS: Example

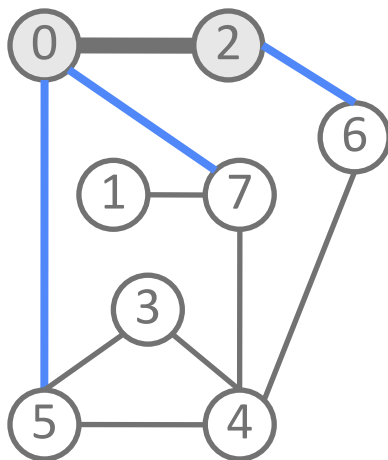


0

0-2 0-5 0-7

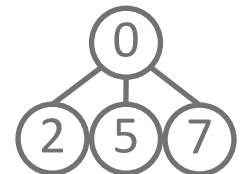
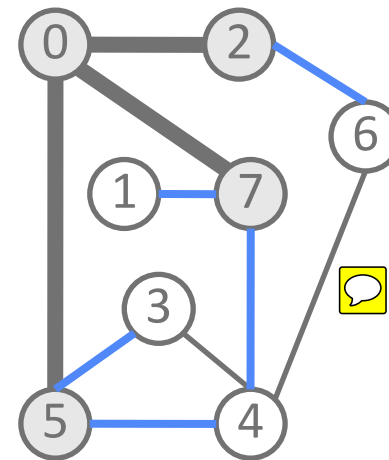


0-7 2-6 5-3 5-4



0
2

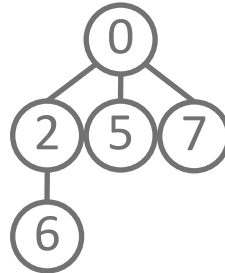
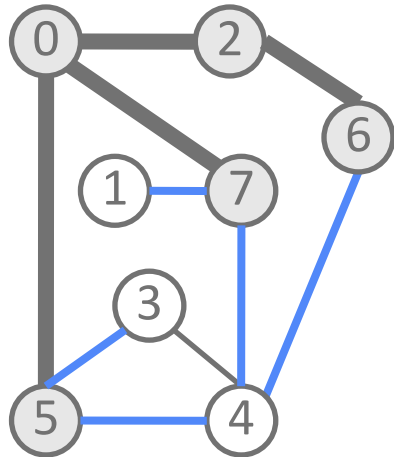
0-5 0-7 2-6



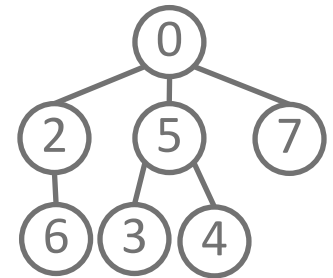
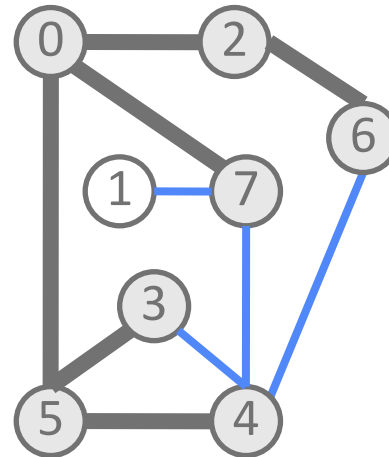
2-6 5-3 5-4 7-1 7-4



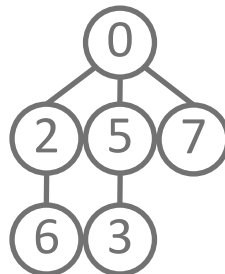
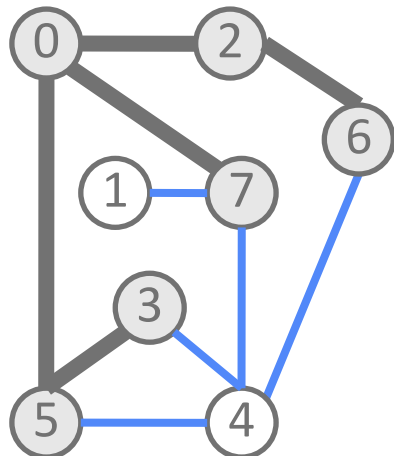
BFS: Example 2



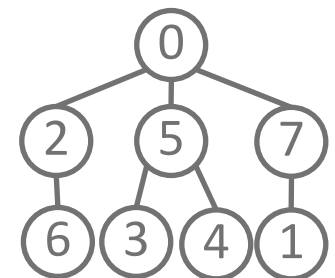
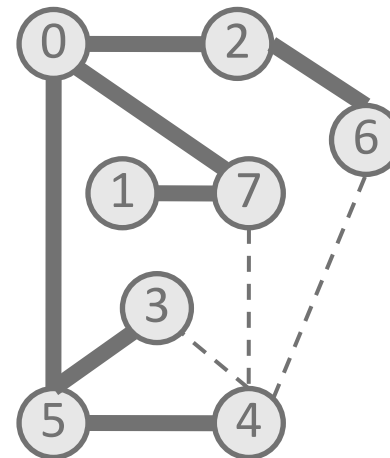
5-3 5-4 7-1 7-4 6-4



7-1 7-4 6-4 3-4



5-4 7-1 7-4 6-4 3-4



7-4 6-4 3-4

BFS: Code (1)

- We need a **queue** to keep track on the vertices that are on the frontier (initially contains s only).
- The queue implementation is skipped here.

```
void graph_bfs(graph G){
    int i;

    pre = malloc(sizeof(int) * G->V);
    st = malloc(sizeof(int) * G->V);
    cnt = 0;
    for (i = 0; i < G->V; i++){
        pre[i] = -1;
        st[i] = -1;
    }
    queue_init(G->V);

    bfs_visit(G, new_edge(0, 0));

    queue_free();
    free(pre);
    free(st);
}
```

BFS: Code (2)

- Take a vertex w from the queue, go through $adj[w]$ and find all vertices v connected to w .
- If v is unvisited, record it is visited (new frontier) and enqueue the edge (w, v) .
- $st[]$ is used to store the predecessors of the vertices during the search.

```
void bfs_visit(graph G, edge_t e){
    int w; node *t;
    queue_enq(e);
    pre[e.w] = cnt++;
    while (!queue_is_empty()){
        e = queue_deq();
        w = e.w;
        st[w] = e.v;
        for (t = G->adj[w]; t != NULL; t = t->next)
            if (pre[t->v] == -1){
                queue_enq(new_edge(w, t->v));
                pre[t->v] = cnt++;
            }
    }
}
```

BFS: Analysis

- Initialization: $O(V)$
- Every node is enqueued and dequeued **exactly once**.
- Since **enqueue** and **dequeue** run in $O(1)$ time, the total time used to queue operations is $O(V)$
- The adj. list of each node is traversed at most once, thus the total time in scanning the adj. lists is the sum of lengths of all lists = $O(E)$.
- Total running time is $O(V + E)$.

Question

How to modify the code so that it records the shortest-path distances $\delta(s, v)$, i.e. the min. number of edges in any path from vertex s to v ?

Summary

- **Graph**: a general model for problem solving in computer science
 - Adjacency **matrix**: suitable for dense graph, $O(1)$ access to edges
 - Adjacency **list**: suitable for sparse graph, $O(V)$ access to edges
- Graph **search/traversal**
 - **Depth-first** search (DFS): maze exploration. generates depth-first trees
 - **Breadth-first** search (BFS): computes shortest-unit-path distances