GRAPH I

Jason
Graph

• Consists of a set of vertices and a set of edges
• Each edge connects a pair of vertices

Mathematically, we often write $G = (V, E)$
  • $V$: set of vertices, so $|V| = \text{number of vertices}$
  • $E$: set of edges, so $|E| = \text{number of edges}$
Usage

• To present the relations between different objects/elements in a mathematical way
• Objects -> Vertices (or nodes)
• Relations -> Edges
• Examples:
  • Social Networks
  • Maps
  • Grids
  • States
Usage - Example

- Social Networks
  - Alice and Bob are friends
  - Bob and Carol are friends
  - Bob and Dan are friends
  - Dan and Eve are friends
  - Alice and Eve are friends
Usage - Example

• Cities connected by some bridges
Usage - Example

- Maze
Graph

Directed graph:

Undirected graph:

• You may use two directed edges to represent an undirected edge (For most of the time)
Graph

Weighted graph: 

Unweighted graph:

• You may treat unweighted edges to be weighted edges of equal weights
Content

- Graph Adjacency Representation
  - Adjacency Matrix
  - Edge List
  - Adjacency List
- Grid Graph
- Depth-First Search
  - Flood Fill
- Breadth-First Search
  - Unweighted shortest path
  - Multisource BFS
- States
Adjacency matrix

- Use a 2D array to store the edges
- $A[i][j] = 0$ if there are no edges from vertex $i$ to vertex $j$
- $A[i][j] = 1$ if there are edges from vertex $i$ to vertex $j$


diagram.png

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Adjacency matrix

• Simple to implement
• Memory Complexity : $O(|V|^2)$
• Time for finding all adjacent nodes of one node: $O(|V|)$
• Not efficient if the graph is sparse, i.e. number of edges ($|E|$) are much less than square of number of vertices ($|V|^2$)

A 1 2 3 4 5 6
1 | 0 0 1 0 1 0
2 | 0 0 0 1 1 0
3 | 0 0 0 0 0 0
4 | 0 0 0 0 0 0
5 | 0 0 0 1 0 0
6 | 0 1 1 0 1 0
Edge List

- A list of edges
- Storing the edges are from where to where

Memory Complexity : $O(|E|)$
Edge List

- You may sort the edges in ascending order of x so that all adjacency nodes of a given node can be easily found.

- Time Complexity: $O(|E| \log |E|)$ or $O(|E|)$

<table>
<thead>
<tr>
<th>id</th>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>
Adjacency Lists

• Use $|V|$ lists to store the edges
• The i-th list stores the vertices vertex i is adjacent to
Adjacency Lists

- Can use a 2D array to imitate the $|V|$ lists

- Memory Complexity: $O(|V|^2)$
Adjacency Lists

- $|E|$ is usually much less than $|V|^2$ when $|V|$ is large
- Use $|V|$ linked lists to store the adjacent nodes instead
- Memory complexity: $O(|E|)$
Adjacency Lists - Implementation

```
void initialize()
{
    for (int i=1; i<=n; i++) first[i] = -1;
}
```
Adjacency Lists - Implementation

- 1->3

<table>
<thead>
<tr>
<th>id</th>
<th>y</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>-1</td>
</tr>
</tbody>
</table>

```c
void new_edge(int from, int to, int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```
Adjacency Lists - Implementation

1->3

<table>
<thead>
<tr>
<th>id</th>
<th>y</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

void new_edge(int from, int to, int weight){
  v[m] = to;
  w[m] = weight;
  next[m] = first[from];
  first[from] = m;
  m++;
}
Adjacency Lists - Implementation

- 6->3

<table>
<thead>
<tr>
<th>node</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>-1</td>
</tr>
</tbody>
</table>

```java
void new_edge(int from, int to, int weight) {
    v[m] = to;
    w[m] = weight;
    next[m] = first[from];
    first[from] = m;
    m++;
}
```
Adjacency Lists - Implementation

- Adjacency List Representation:
  
<table>
<thead>
<tr>
<th>node</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

- Code Example:

```c
void new_edge(int from, int to, int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```
Adjacency Lists - Implementation

- 1->5

```
void new_edge(int from, int to, int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```
Adjacency Lists - Implementation

- **1->5**

<table>
<thead>
<tr>
<th>node</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

```java
void new_edge(int from, int to, int weight) {
    v[m] = to;
    w[m] = weight;
    next[m] = first[from];
    first[from] = m;
    m++;
}
```
void new_edge(int from, int to, int weight) {
    v[m] = to;
    w[m] = weight;
    next[m] = first[from];
    first[from] = m;
    m++;
}
Adjacency Lists - Implementation

- 2->5

```
void new_edge(int from, int to, int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```
Adjacency Lists - Implementation

- 2->4

<table>
<thead>
<tr>
<th>node</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

```c
void new_edge(int from, int to, int weight) {
    v[m] = to;
    w[m] = weight;
    next[m] = first[from];
    first[from] = m;
    m++;
}
```
Adjacency Lists - Implementation

- 2->4

<table>
<thead>
<tr>
<th>node</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

```c
void new_edge(int from, int to, int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```
Adjacency Lists - Implementation

- 6->2

<table>
<thead>
<tr>
<th>node</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

```c
void new_edge(int from, int to, int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```
Adjacency Lists - Implementation

- **6->2**

<table>
<thead>
<tr>
<th>node</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>y</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

```cpp
void new_edge(int from, int to, int weight) {
    v[m] = to;
    w[m] = weight;
    next[m] = first[from];
    first[from] = m;
    m++;
}
```
Adjacency Lists - Implementation

- 5->4

<table>
<thead>
<tr>
<th>node</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>y</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>-1</td>
</tr>
</tbody>
</table>

```c
void new_edge(int from, int to, int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```
Adjacency Lists - Implementation

- 6->5

<table>
<thead>
<tr>
<th>node</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

```
void new_edge(int from, int to, int weight){
    v[m]=to;
    w[m]=weight;
    next[m]=first[from];
    first[from]=m;
    m++;
}
```
Adjacency Lists - Implementation

- Query 6’s neighbors

<table>
<thead>
<tr>
<th>node</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>y</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

```c
void query(int x){
    //Querying all adjacent nodes of a given node x
    for (int i=first[x];i!= -1;i=next[i])
        printf("%d\n",v[i]);
}
```
Adjacency Lists - Implementation

```c
int v[10000], w[10000], next[10000];
int first[1000];
int n, m = 0;

void initialize(){
    for (int i = 1; i <= n; i++) first[i] = -1;
}

void new_edge(int from, int to, int weight){
    v[m] = to;
    w[m] = weight;
    next[m] = first[from];
    first[from] = m;
    m++;
}

void query(int x){
    // Querying all adjacent nodes of a given node x
    for (int i = first[x]; i != -1; i = next[i])
        printf("%d
", v[i]);
}
```
Adjacency Lists - Implementation

• We can use vector in C++ to implement the lists

```cpp
#include<vector>
using namespace std;

int n, m;
vector<int> v[10000], w[10000];
void new_edge(int from, int to, int weight){
    v[from].push_back(to);
    w[from].push_back(weight);
}

void query(int x){
    //Querying all adjacent nodes of a given node (x)
    for (int i=0; i<v[x].size(); i++)
        printf("%d\n", v[x][i]);
}
```
Grid Graph

• A graph that the vertices form a regular tiling
• E.g. rectangles, cubes or triangles
Grid Graph - Vertex

- No need to give new indices to the vertices
- Use their coordinates instead
Grid Graph - Vertex

- Some vertices may be invalid (e.g. a wall in a maze)
- Skip the process when visiting an invalid vertex

```c
int valid(int x, int y){
    if (x <= 0 || x > n) return 0;
    if (y <= 0 || y > m) return 0;
    if (isWall[x][y]) return 0;
    ...
    return 1;
}
```
Grid Graph - Edge

- The number of edges for each vertex is small
- No need to use the previous representations to store the edges
- The coordinates of the adjacent vertices can be calculated
Grid Graph – Example 1

• A rectangular maze
• You are only allowed to move one cell horizontally or vertically in one move
• Find the shortest path from one to another
Grid Graph – Example 1

• The possible adjacent vertices of vertex \((x, y)\) are \((x - 1, y)\), \((x + 1, y)\), \((x, y - 1)\) and \((x, y + 1)\)
Grid Graph – Example 1

• Hardcode the edges

```c
int dx[4]={-1,1,0,0};
int dy[4]={0,0,-1,1};
...
// find nodes adjacent to (x, y)
for (int i = 0; i < 4; i++){
    newX = x + dx[i];
    newY = y + dy[i];
    if (!valid(newX, newY)) continue;
}
...
```

• Use Breath-first Search to find the shortest path
Grid Graph – Example 2

- You are allowed to move one cell horizontally, vertically or diagonally in one move
Grid Graph – Example 2

- Hardcode

```c
int dx[8] = {-1, -1, -1, 0, 0, 1, 1, 1};
int dy[8] = {-1, 0, 1, -1, -1, -1, 0, 1};

// find nodes adjacent to (x, y)
for (int i = 0; i < 8; i++){
    newX = x + dx[i];
    newY = y + dy[i];
    if (!valid(newX, newY)) continue;
    ...
}
...
Grid Graph – Example 2

• Or use 2 for loops

```java
// find nodes adjacent to (x, y)
for (int i = -1; i <= 1; i++)
    for (int j = -1; j <= 1; j++){
        newX = x + i;
        newY = y + j;
        if (newX == x && newY == y) continue;
        if (!valid(newX, newY)) continue;
        ...
    }
```
Graph Traversal

• The process of visiting (checking and/or updating) each vertex in a graph.

• Classified by the order in which the vertices are visited
  • Depth-first Search
  • Breadth-first Search
Depth-first Search

- Starts at a node
- Explores as far as possible along each branch before backtracking
- Recursion is often used
Depth-first Search - example
Depth-first Search - example

- Perform DFS on node 1

DFS(1)

Function Calls
Depth-first Search - example

- Node 2 is not visited, perform DFS on node 2

DFS(1)
Function Calls
Depth-first Search - example

• Node 4 is not visited, perform DFS on node 4
Depth-first Search - example

- Node 2 is visited, no need to perform DFS on node 2

Function Calls
- DFS(1)
- DFS(2)
- DFS(4)
- DFS(5)
Depth-first Search - example

• DFS(4) is finished, go back to DFS(2)
Depth-first Search - example

- Node 6 is not visited, perform DFS on node 6
Depth-first Search - example

- Node 5 is not visited, perform DFS on node 5

Function Calls

DFS(1)
DFS(2)
DFS(6)
Depth-first Search - example

- Node 3 is not visited, perform DFS on node 3

Function Calls

DFS(5)
DFS(6)
DFS(2)
DFS(1)
Depth-first Search - example

- No neighbors of node 3 is not visited, go back to DFS(5)
Depth-first Search - example

- ... and so on

Function Calls:
- DFS(1)
- DFS(2)
- DFS(5)
- DFS(6)
Depth-first Search - example

• ... and so on

Function Calls

1

DFS(1)

2

DFS(2)

3

DFS(3)

4

DFS(4)

5

DFS(5)

6

DFS(6)

7

DFS(7)

8

DFS(8)
Depth-first Search - example

- ... and so on

Function Calls:
- DFS(1)
- DFS(2)
- DFS(5)
- DFS(6)
Depth-first Search - example

• ... and so on

Function Calls

DFS(1)
DFS(2)
DFS(6)
Depth-first Search - example

• ... and so on

Function Calls

DFS(1)
DFS(2)
Depth-first Search - example

• ... and so on

DFS(1)
Function Calls
Depth-first Search - example

• Done

Function Calls
Depth-first Search - Implementation

procedure DFS(vertex v){
    label v as discovered
    for all w adjacent to v do
        if w is not labeled as discovered then
            DFS(w)
    }

Depth-first Search

• Time Complexity:
  • $O(|V|^2)$ when using adjacency matrix
  • $O(|V|+|E|)$ when using adjacency list / edge list
Depth-first Search - Example

- Flood fill
- Determine the number of nodes (or area) connected to a given node

E.g. Find the area connected to the cell (5, 5) with the same color in the following grid
Flood fill

• Perform DFS on (5, 5)
Flood fill

- Perform DFS on (4, 5)
Flood fill

- Perform DFS on (3, 5)
Flood fill

- ... and so on
Flood fill

• ... and so on
Flood fill

- ... and so on
Flood fill

• ... and so on

1,1
2,1
3,1
4,1
5,1
1,2
2,2
3,2
4,2
5,2
1,3
2,3
3,3
4,3
5,3
1,4
2,4
3,4
4,4
5,4
1,5
2,5
3,5
4,5
5,5
Flood fill

- ... and so on
Flood fill

- Count the # of visited nodes during DFS
Flood fill - Implementation

Procedure floodFill(x, y){
    label node(x, y) as discovered
    counter = counter + 1
    For each node(newX, newY) in neighbors of node(x, y) do
        If node(newX, newY) is valid and not discovered then
            floodFill(newX, newY)
Breadth-first Search

• Starts at a node
• Explores the neighbor nodes first before moving to the next level neighbors
• The vertices closest to the source are evaluated first, and the most distant vertices are evaluated last
• Use a queue to implement
Breadth-first Search - example

• Push 1 into the queue and mark node 1 as visited first
Breadth-first Search - example

- Perform BFS on node 1

```
1
```

```
2 3
```

```
4 5
```

```
6 7
```

```
8
```
Breadth-first Search - example

- Node 2 and 3 are adjacent to node 1 and they are not visited
Breadth-first Search - example

• Mark node 2 and 3 as visited and push them into the queue
Breadth-first Search - example

- BFS on node 1 is finished, pop 1 from the queue
Breadth-first Search - example

- Perform BFS on node 2
Breadth-first Search - example

- Node 4 and 6 are adjacent to node 2 and they are not visited
Breadth-first Search - example

• Mark node 4 and 6 as visited and push them into the queue
Breadth-first Search - example

• BFS on node 2 is finished, pop 2 from the queue
Breadth-first Search - example

• ... and so on

1 2 3 4 6
Breadth-first Search - example

• ... and so on
Breadth-first Search - example

• ... and so on

| 1 | 2 | 3 | 4 | 6 | 5 |   |   |   |   |   |

1 -- 2 -- 3
   |   |   |
4 -- 5 -- 6
   |   |   |
   |   |
7 -- 8
Breadth-first Search - example

• ... and so on
Breadth-first Search - example

... and so on
Breadth-first Search - example

• ... and so on
Breadth-first Search - example

• ... and so on
Breadth-first Search - example

• ... and so on
Breadth-first Search - example

• ... and so on
Breadth-first Search - example

• ... and so on

1 2 3 4 6 5 7
Breadth-first Search - example

• Since the queue is empty now, the search can be stopped
Breadth-first Search - code

Procedure bfs(vertex v) {
    mark v is visited
    push v into the queue
    while the queue is not empty do
        t = front of the queue
        for all vertex w adjacent to t do
            if w is not visited
                mark w as visited
                push w into the queue
        pop t from the queue
}
Breadth-first Search

• Time Complexity:
  • $O(|V|^2)$ when using adjacency matrix
  • $O(|V|+|E|)$ when using adjacency list / edge list
Breadth-first Search - application

• Find the shortest path of all nodes from one node, with path length measured by number of edges
Breadth-first Search - example

• Find the shortest paths from node 1
Breadth-first Search - example

<table>
<thead>
<tr>
<th>Node</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>
Breadth-first Search - example

<table>
<thead>
<tr>
<th>Node</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

Diagram:
- Node 1 is the starting point.
- Node 3 is connected to Node 1, with a distance of 1.
- Node 2 is connected to Node 3, with a distance of 1.
- Node 4 is connected to Node 2, with a distance of ∞.
- Node 5 is connected to Node 3, with a distance of ∞.
- Node 6 is connected to Node 5, with a distance of ∞.
- Node 7 is connected to Node 5, with a distance of ∞.
- Node 8 is connected to Node 6, with a distance of ∞.
Breadth-first Search - example

<table>
<thead>
<tr>
<th>Node</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

Diagram:
- Node 1
- Node 2
- Node 3
- Node 4
- Node 5
- Node 6
- Node 7
- Node 8

Connections:
- Node 1 to Node 2
- Node 1 to Node 3
- Node 2 to Node 3
- Node 2 to Node 4
- Node 3 to Node 4
- Node 4 to Node 6
- Node 5 to Node 6
- Node 7 to Node 8
- Node 8 to Node 6
**Breadth-first Search - example**

<table>
<thead>
<tr>
<th>Node</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>5</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

![Graph](image)
Breadth-first Search - example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Distance</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

Graph:
- Node 1 is connected to nodes 2 and 3.
- Node 2 is connected to nodes 1 and 3.
- Node 3 is connected to nodes 1 and 4.
- Node 5 is connected to nodes 4 and 6.
- Node 6 is connected to node 5.
- Node 7 is connected to node 6.
- Node 8 is connected to node 7.
Breadth-first Search - example

<table>
<thead>
<tr>
<th>Node</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

![Graph with nodes and distances](image)
Breadth-first Search - example

<table>
<thead>
<tr>
<th>Node</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>∞</td>
</tr>
</tbody>
</table>

Graph representation:

1 -- 2 -- 3
     
2 -- 4

5 -- 6

7

8
Breadth-first Search - example

<table>
<thead>
<tr>
<th>Node</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>∞</td>
</tr>
</tbody>
</table>
Breadth-first Search - example

<table>
<thead>
<tr>
<th>Node</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>∞</td>
</tr>
</tbody>
</table>
Practice problem

• 01035 Patrol Area
• 01067 Maze
• M1311 Dokodemo Door
Multisource BFS

- Sometimes you may want to start searching from more than one source

- Method 1:
  - Perform BFS once for each source

- Time Complexity: $O(N*(|V|+|E|))$, where $N$ is the number of sources
- Too slow if there are many sources
Multisource BFS

- Method 2:
- Push all sources into the queue first
- Perform BFS once
- Time Complexity: $O(|V| + |E|)$
Multisource BFS

- Find the black area connected to (4, 2), (4, 4) and (2, 3)
- Push (4, 2), (4, 4) and (2, 3) into the queue and marked them as visited
Multisource BFS
Multisource BFS
### Multisource BFS

<table>
<thead>
<tr>
<th>4, 2</th>
<th>4, 4</th>
<th>2, 3</th>
<th>4, 1</th>
<th>3, 4</th>
<th>4, 5</th>
<th>1, 3</th>
<th>3, 3</th>
</tr>
</thead>
</table>

![Diagram of multisource BFS](image.png)
# Multisource BFS

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4, 2</td>
<td>4, 4</td>
<td>2, 3</td>
<td>4, 1</td>
<td>3, 4</td>
<td>4, 5</td>
<td>1, 3</td>
<td>3, 3</td>
</tr>
<tr>
<td>3, 1</td>
<td>5, 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram of a multisource BFS.
Multisource BFS
# Multisource BFS

| 4, 2 | 4, 4 | 2, 3 | 4, 1 | 3, 4 | 4, 5 | 1, 3 | 3, 3 | 3, 1 | 5, 1 | 3, 5 | 5, 5 |  |

![Multisource BFS Diagram](image_url)
Multisource BFS
**Multisource BFS**

| 4, 2 | 4, 4 | 2, 3 | 4, 1 | 3, 4 | 4, 5 | 1, 3 | 3, 3 | 3, 1 | 5, 1 | 3, 5 | 5, 5 | 1, 2 | 1, 1 |

![Multisource BFS Diagram](image-url)
States

- A vertex can represent a state
- The edges are the possible transitions and the weights are the cost of the transitions
States – Example 1

• Water Jug Problem

• There are two jugs with a capacity of N and M liters
• Both are empty initially
• Each time you can only do one of the three operations:
  • Empty a jug
  • Fill a jug completely
  • Pour water from one jug to another until either one jug becomes empty or the other becomes full
• Find the minimum number of operations to get a specific volume in one jug
States – Example 1

• The state \((x, y)\) represents that there are \(x\) and \(y\) liter water in the first and second jug respectively

• The state \((x, y)\) can transit to six states:
  • \((x, 0)\) and \((0, y)\) by emptying a jug
  • \((N, y)\) and \((x, M)\) by filling a jug completely
  • \((x + y - M, M)\) or \((0, x + y)\) by pouring water from the first jug to second
  • \((N, x + y - N)\) or \((x + y, 0)\) by pouring water from the second jug to first

• A graph can be built by using states as vertices and transitions as edges
• For example, when $N = 3$, $M = 5$ and the required volume is 4:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
States – Example 1

Diagram showing a grid of states with arrows indicating transitions between states.

States:
- (0, 0)
- (0, 1)
- (0, 2)
- (0, 3)
- (0, 4)
- (0, 5)
- (1, 0)
- (1, 1)
- (1, 2)
- (1, 3)
- (1, 4)
- (1, 5)
- (2, 0)
- (2, 1)
- (2, 2)
- (2, 3)
- (2, 4)
- (2, 5)
- (3, 0)
- (3, 1)
- (3, 2)
- (3, 3)
- (3, 4)
- (3, 5)
States – Example 1

• The answer will be the shortest path from (0, 0) to these nodes
States – Example 1

• The answer will be the shortest path from (0, 0) to these nodes
States – Example 2

• A rectangular maze
• You are only allowed to move one cell horizontally or vertically in one move
• Now you have some bombs to destroy the wall, and each bomb can destroy a wall of exactly one cell
• Find the shortest path from one to another
States – Example 2

• The state \((x, y, b)\) represents the you move to the cell \((x, y)\) and still have \(b\) bombs

• \((x, y, b)\) can transit to \((x, y + 1, b)\) if the cell \((x, y + 1)\) is not a wall, and \((x, y, b)\) can transit to \((x, y + 1, b - 1)\) if the cell \((x, y)\) is a wall and \(b\) is greater than zero

• Similar for other directions
Practice problem

• T022 Bomber Man
• M0911 Theseus and the Minotaur
• 30422 Knights in FEN