Graph

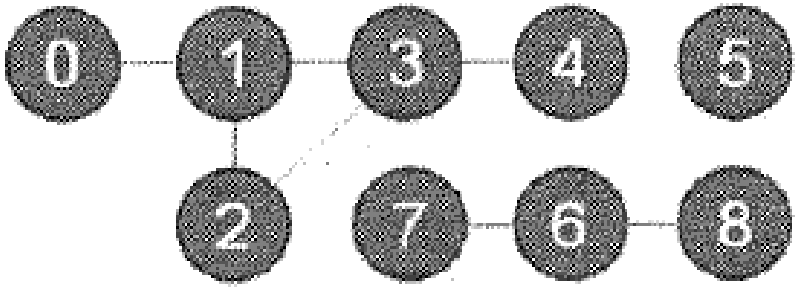
เนื้อหาส่วนที่นำมาจาก ดร.ภิญโญ จะปรากฏข้อความจากต้นฉบับไม่มีการปรับเปลี่ยน

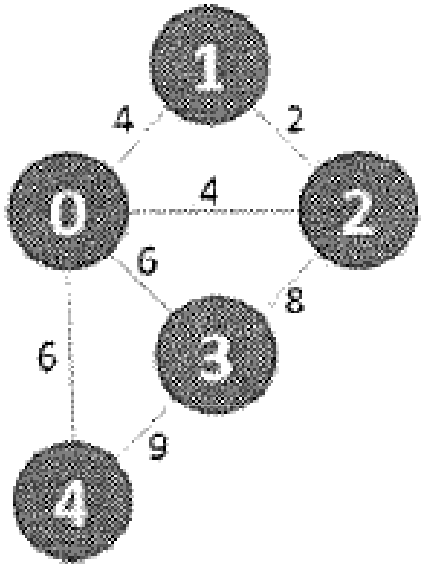
รู้จักกับกราฟ

- กราฟเป็นโครงสร้างข้อมูลแบบหนึ่ง
- โดยตัวของมันเองไม่ใช่อัลกอริทึม ซึ่งจุดนี้จะต่างกับ ...
  - Greedy Algorithm
  - Divide and Conquer
  - Dynamic Programming
- โครงสร้างข้อมูลมักจะประกอบด้วยของสองส่วนหลักคือ
  - โหนด / จุดยอด (Node / Vertex)
  - เส้นเชื่อม / ขอบ (Edge)
    - เส้นเชื่อมอาจจะมีทิศทาง (directional edge)
    - เส้นเชื่อมอาจจะมีน้ำหนัก (weight) หรือค่าใช้จ่าย (cost) กำหนดไว้ด้วย

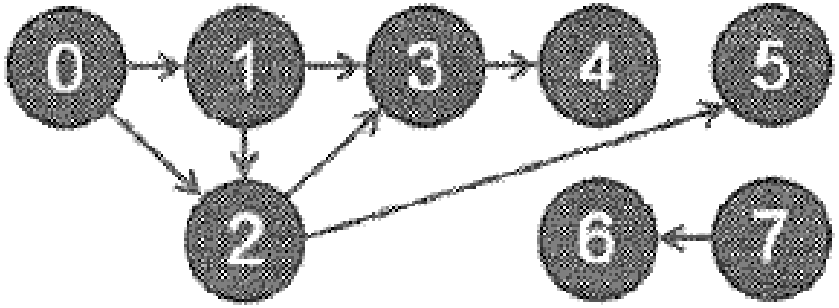
มโนภาพของโครงสร้างข้อมูลแบบกราฟ

- ตัวอย่างกราฟที่เส้นเชื่อมไม่มีทิศทาง





- ตัวอย่างกราฟที่เส้นเชื่อมมีทิศทางกำหนด



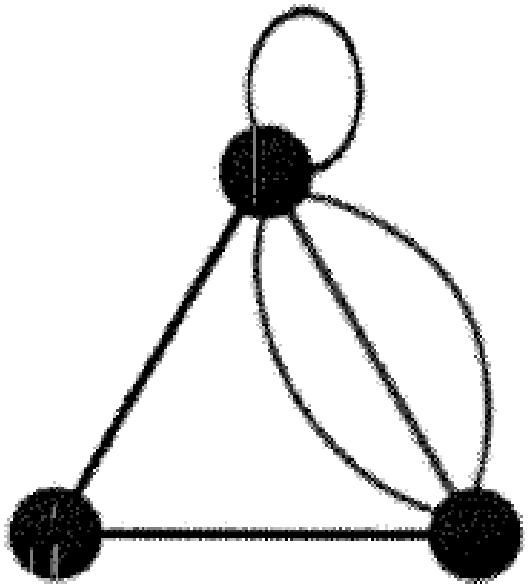
ภาพจาก Competitive Programming 2nd Edition

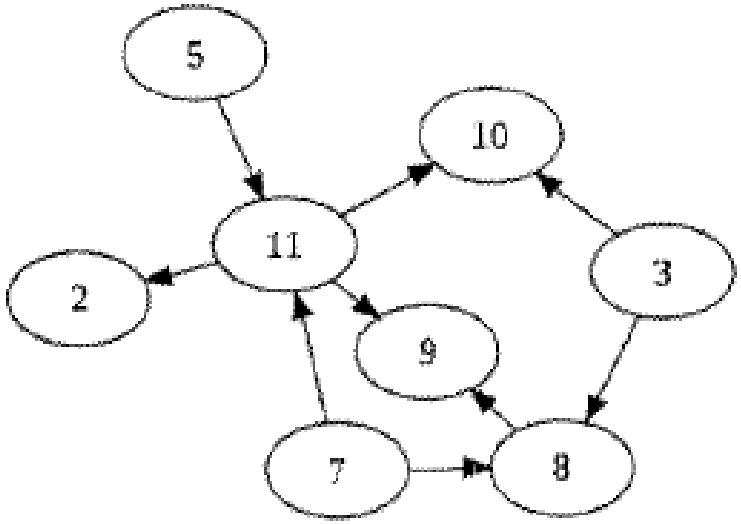
ประเภทของกราฟ

- เราสามารถแบ่งกราฟได้เป็นหลายประเภท
  - ขึ้นอยู่กับมุมมองที่เราเลือกใช้สำหรับการแบ่งประเภท
  - แต่ในระดับพื้นฐาน เรามักแบ่งตามลักษณะของเส้นเชื่อม
    - ➢ ถ้าเส้นเชื่อมมีทิศทางกำหนดเรามักเรียกกราฟว่ากราฟมีทิศทาง (directional graph) ไม่เช่นนั้นจะเรียกว่ากราฟไม่มีทิศทาง (undirectional graph)
- แต่วิธีแบ่งประเภทกราฟก็มีอีกหลากหลาย เช่น
  - ถ้ากราฟมีทิศทาง เราก็อาจแบ่งว่ามีวังวน (cycle) ในกราฟหรือไม่ ถ้าไม่มีเราเรียกว่า Directional Acyclic Graph (DAG)
  - โหนดคู่ใด ๆ ในกราฟมีเส้นเชื่อมได้มากกว่า 1 เส้นหรือไม่ ถ้ามีเราเรียกว่า Multigraph

ภาพตัวอย่างกราฟแบบต่าง ๆ

- กราฟบางแบบก็มีเส้นเชื่อมที่วกเข้าหาตัวเอง (self loop) [เส้นเชื่อมสีน้ำเงิน]
- ชุดเส้นเชื่อมที่ทำให้เป็น multigraph คือเส้นเชื่อมสีแดง





Directed Acyclic Graph (DAG)

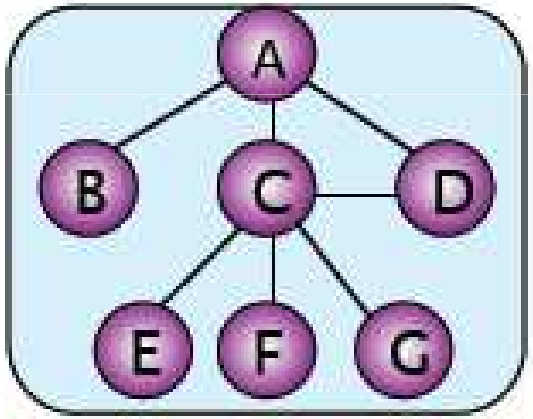
ภาพจาก wikipedia

การอธิบายโครงสร้างกราฟ

- เราสามารถแบ่งกราฟได้เป็นหลายประเภท
  - ขึ้นอยู่กับมุมมองที่เราเลือกใช้สำหรับการแบ่งประเภท
  - แต่ในระดับพื้นฐาน เรามักแบ่งตามลักษณะของเส้นเชื่อม
    - ถ้าเส้นเชื่อมมีทิศทางกำหนดเรามักเรียกกราฟว่ากราฟมีทิศทาง (directional graph) ไม่เช่นนั้นจะเรียกว่ากราฟไม่มีทิศทาง (undirectional graph)
- แต่วิธีแบ่งประเภทกราฟก็มีอีกหลากหลาย เช่น
  - ถ้ากราฟมีทิศทาง เราก็อาจแบ่งว่ามีวังวน (cycle) ในกราฟหรือไม่ ถ้าไม่มีเราเรียกว่า Directional Acyclic Graph (DAG)
  - โหนดคู่ใด ๆ ในกราฟมีเส้นเชื่อมได้มากกว่า 1 เส้นหรือไม่ ถ้ามีเราเรียกว่า Multigraph
- เรามักแทนเซ็ตของโหนดด้วย V และเซ็ตของเส้นเชื่อมด้วย E

Directed & Undirected Graph

Undirected Graph แสดงสายการบินของ Air Asia



Undirected Graph แสดงโหนดและเส้นเชื่อมของกราฟรูปหนึ่ง

เรื่องที่ต้องคิดในทางปฏิบัติ

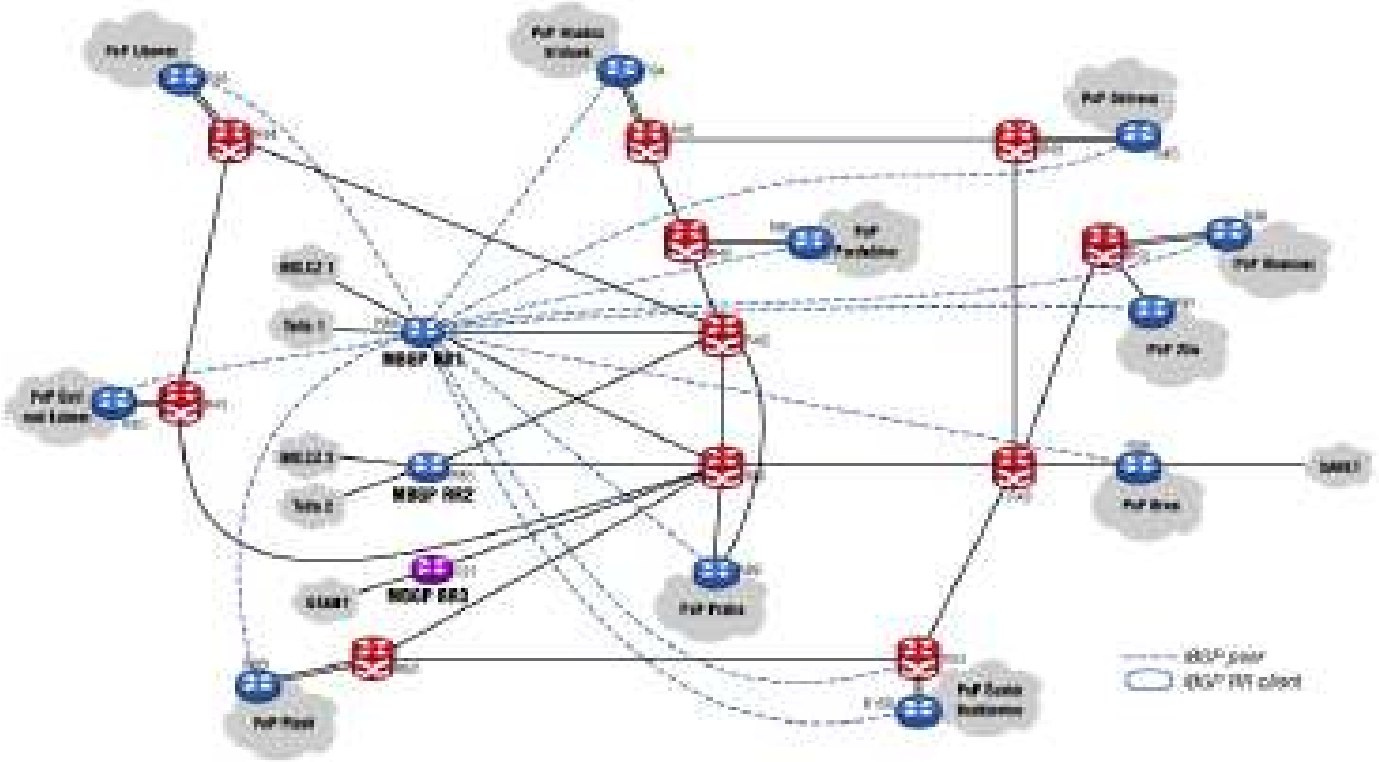
- เราเห็นการอธิบายกราฟด้วย Edge listing กับ Adjacency list มาแล้ว
  - แต่คำถามก็คือว่าเราจะเก็บข้อมูลพวกนี้ในหน่วยความจำอย่างไร (How to keep graph description in memory?)
  - แล้วเราจะทำอะไรกับข้อมูลพวกนี้บ้าง (What operations are we going to do with a graph?)
- ธรรมชาติของโครงสร้างข้อมูลที่ดีก็คือว่า มันต้องสามารถทำในสิ่งที่เราต้องการทำบ่อย ๆ ได้อย่างมีประสิทธิภาพ
  - เรามักถามว่าโหนดหมายเลข x มีเส้นเชื่อมต่อกับโหนดหมายเลข y หรือไม่
  - เรามักถามว่ามีเส้นทางจาก โหนดหมายเลข x ไปโหนดหมายเลข y หรือไม่

ประโยชน์ของกราฟ
(Routing การหาเส้นทาง)

- สายการบิน (การเชื่อมต่อของสายการบิน ตารางบิน)

ประโยชน์ของกราฟ
(Routing การหาเส้นทาง)

- Network (การเชื่อมต่อของอุปกรณ์ Router)
- เพื่อใช้ในการรับส่งข้อมูลในเครือข่าย



ประโยชน์ของกราฟ
(Algorithm Design)

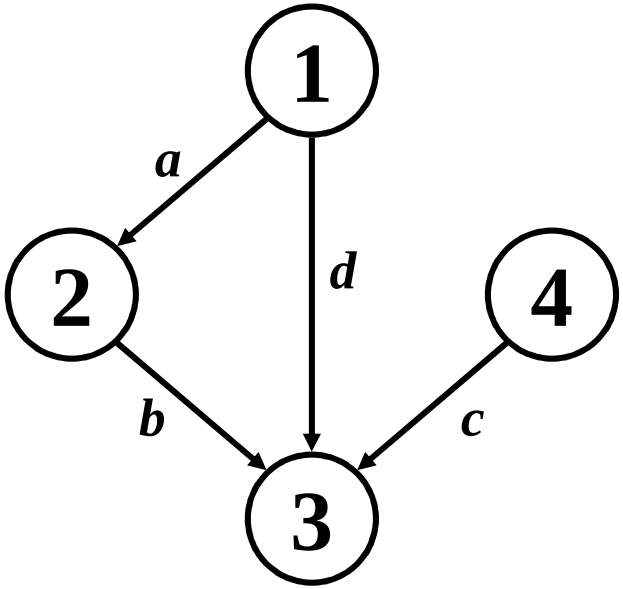
- Map Coloring คือวิธีการระบายสีในแผนที่โดยใช้สีน้อยที่สุด
- พื้นที่ติดกันห้ามใช้สีเดียวกัน

Representing Graphs

- A graph G = (V, E)
  - V = set of vertices
  - E = set of edges = subset of $V \times V$
  - Thus $|E| = O(|V|^2)$
- Assume V = {1, 2, …, *n*}
- Adjacency matrix *n* x *n* matrix A
  - A[*i*, *j*] = 1 if edge $(i, j) \in E$ (or weight of edge)
    = 0 if edge $(i, j) \notin E$

Graphs: Adjacency Matrix

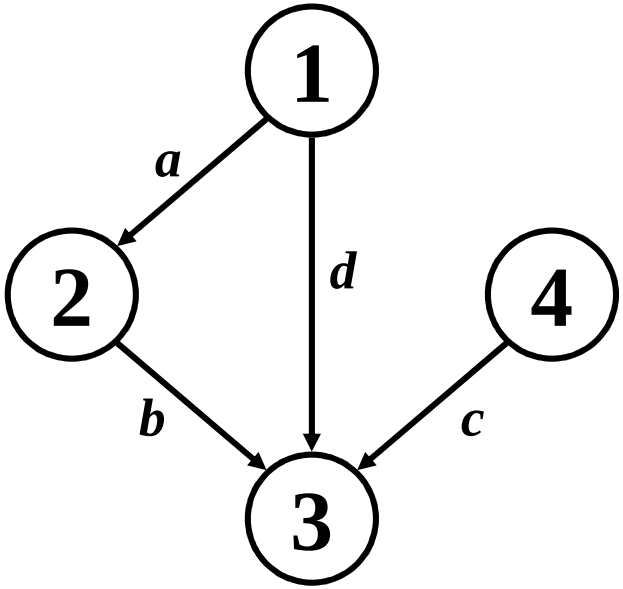
- Example:



A	1	2	3	4
1				
2				
3			??	
4				

Graphs: Adjacency Matrix

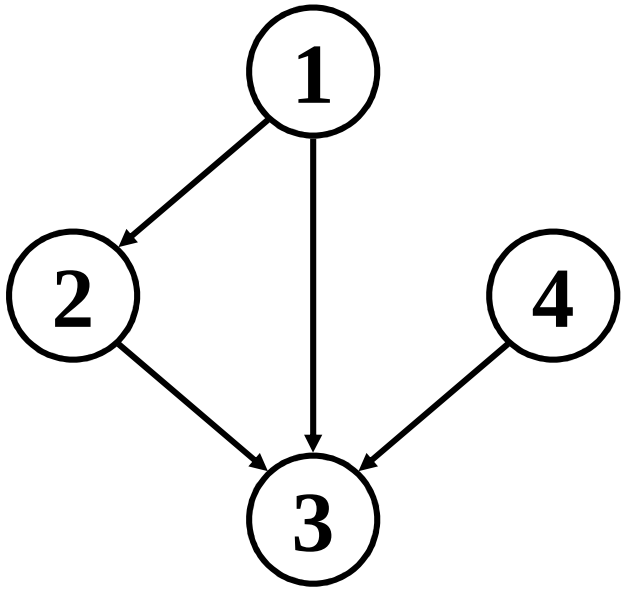
- Example:



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

Graphs: Adjacency List

- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to *v*
- Example:
  - Adj[1] = {2,3}
  - Adj[2] = {3}
  - Adj[3] = {}
  - Adj[4] = {3}
- Variation: can also keep a list of edges coming *into* vertex



ตัวอย่างการเก็บ Adjacency list

0	1
1	0 2 3
2	1 3
3	1 2 4
4	3
5	
6	7 8
7	6
8	6

```
public class GraphStructDemo {
   int[][] arNode;

   private void prepareSpace() {
     arNode = new int[9][];

     arNode[0] = new int[1];
     arNode[1] = new int[3];
     arNode[2] = new int[2];
     arNode[3] = new int[3];
     arNode[4] = new int[1];

     arNode[5] = new int[0];
     arNode[6] = new int[2];
     arNode[7] = new int[1];
     arNode[8] = new int[1];
   }
   .........
}
```

ประกาศอาเรย์สำหรับเก็บข้อมูลการเชื่อมต่อไว้

เริ่มสร้างอาเรย์ แต่อย่าเพิ่งรีบบอกขนาดของมิติที่สอง เก็บไว้ทำทีหลังได้ (ทำแบบนี้ได้จริง ๆ นะ)

ระบุขนาดของลิสต์การเชื่อมต่อของแต่ละโหนดทีละอันตามปริมาณที่ต้องใช้จริง

ไฟล์ GraphStructDemojava

ป้อนข้อมูลเข้า Adjacency list

```
private void insertData() {
    arNode[0][0] = 1;
    arNode[1][0] = 0;  arNode[1][1] = 2;  arNode[1][2] = 3;
    arNode[2][0] = 1;  arNode[2][1] = 3;
    arNode[3][0] = 1;  arNode[3][1] = 2;  arNode[3][2] = 4;
    arNode[4][0] = 3;

    //arNode[5][];  // No edge to insert
    arNode[6][0] = 7;  arNode[6][1] = 8;
    arNode[7][0] = 6;
    arNode[8][0] = 6;
}
```

ตรวจว่าโหนดเชื่อมกันหรือไม่

การตรวจดูว่าโหนด x เชื่อมกับโหนด y หรือไม่

```
public boolean isLinked(int x, int y) {
  int numNodes = arNode[x].length;

  for(int i = 0; i < numNodes; ++i) {
    if(arNode[x][i] == y) {
      return true;  // y is found
    }
  }

  return false;      // y not found
}
```

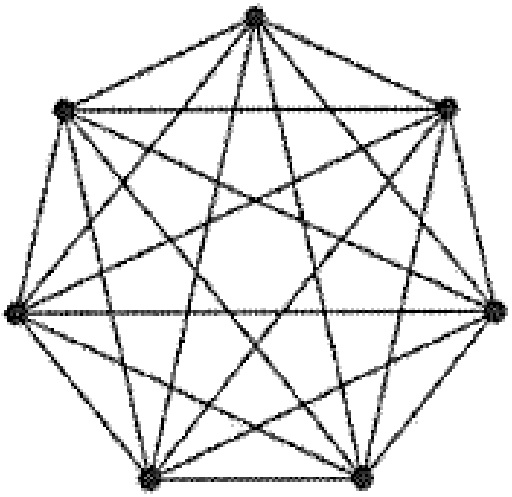
เนื่องจากอาเรย์ของแต่ละโหนดมีความยาวไม่เท่ากัน เราจึงต้องหาความยาวลิสต์ของโหนดมาเก็บไว้ก่อน

จากโครงสร้างของลูป เราเห็นได้ว่าการจะตรวจการเชื่อมต่อของโหนด ในกรณีที่แย่ที่สุดเกิดขึ้นเมื่อไม่พบการเชื่อมต่อ (return false) เพราะเราจะต้องหาตั้งแต่เริ่มจนจบ

มีวิธีที่ทำให้ฟังก์ชันนี้ทำงานเร็วขึ้นหรือไม่ ?

หน่วยความจำที่ต้องใช้ในการอธิบายโครงสร้างกราฟ

- การใช้ Edge listing หรือ Adjacency list เหมาะกับกราฟที่มีเส้นเชื่อมน้อยเมื่อเทียบกับจำนวนโหนด (เหมาะกับ Sparse Graph)
- ลองพิจารณากราฟที่มีเส้นเชื่อมสมบูรณ์ (Complete Graph)



ภาพจาก wikipedia

กราฟมี 7 โหนด 21 เส้นเชื่อม

Edge listing ใช้พื้นที่เก็บเลขจำนวนเต็ม 2 ตัวต่อเส้นเชื่อม

→ ใช้ integer 42 ตัว (168 bytes)

→ ถ้าเส้นเชื่อมมาก ปริมาณหน่วยความจำที่ต้องใช้จะเพิ่มขึ้นมากตาม

→ กราฟจำนวนมากมีปริมาณเส้นเชื่อมเป็น $O(|V|^2)$

[V คือเซ็ตของโหนด $|V|$ ก็คือจำนวนโหนดในเซ็ต]

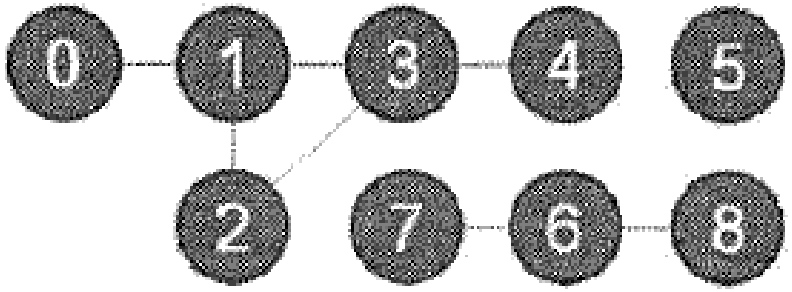
ดังนั้นปริมาณเส้นเชื่อมและหน่วยความจำที่ต้องใช้จึงอยู่ในระดับ $O(|V|^2)$ ไปด้วย

หน่วยความจำที่ต้องใช้ในการอธิบายโครงสร้างกราฟ (2)

- แล้วถ้าเป็น Adjacency matrix ล่ะ
  - เนื่องจากเป็นอาเรย์สองมิติ เราเห็นได้ชัดว่าใช้หน่วยความจำ $O(|V|^2)$
  - แต่ว่าเราไม่จำเป็นต้องเก็บเลขจำนวนเต็มเหมือนกับการใช้ลิสต์
  - ถ้าเราใช้ boolean เราจะเสียพื้นที่ต่อช่อง 1 ไบต์ (ถ้าเป็นจำนวนเต็มแบบลิสต์จะใช้ 4 ไบต์)
  - ถ้าเราใช้ **java.util.BitSet** เราจะใช้พื้นที่ต่อช่องแค่ 1 บิต
- ด้วยเทคนิคด้านการจัดการหน่วยความจำ ถ้ากราฟมีเส้นเชื่อมมาก ๆ การใช้ Adjacency matrix เป็นทางออกที่ดีที่สุดอย่างไม่ต้องสงสัย
  - ตอบคำถามเรื่องการเชื่อมต่อได้เร็วกว่า เข้าใจง่ายกว่า
  - ถ้าเส้นเชื่อมมากพอ มักจะใช้หน่วยความจำน้อยกว่าด้วย

การไปถึงกันได้ระหว่างโหนด

- ก่อนหน้านี้เราพูดถึงการเชื่อมกันของโหนดแบบเชื่อมโดยตรง
- แต่กราฟมักไม่ได้ถูกสร้างมาเพื่อถามเกี่ยวกับการเชื่อมต่อโดยตรง ที่เรามักถามกันบ่อยจริง ๆ คือถามว่า เราสามารถเดินทางจากโหนด x ไปโหนด y ได้หรือไม่
  - เช่นถามว่า เราสามารถเดินทางจากโหนด 0 ไปโหนด 4 ได้หรือไม่
  - หรือ เราสามารถเดินทางจากโหนด 2 ไปโหนด 5 ได้หรือไม่



- เราเรียกการไปถึงกันได้ของโหนดสองโหนดว่า reachable

แล้วจะรู้ได้ไงว่ามันไปถึงกันได้หรือเปล่า

- ใช้วิธีการค้นหา ซึ่งวิธีพื้นฐานก็คือการใช้ Breadth-First Search (BFS) หรือ Depth-First Search (DFS) จากโหนด x แล้วคอยตรวจดูว่าในระหว่างการค้นหามันเคยเจอโหนด y หรือเปล่า
- ถ้าต้องเขียนโค้ดเอง BFS มีแนวโน้มจะเขียนง่ายกว่า เข้าใจง่ายกว่า
- DFS มักถูกอธิบายในทางหลักการออกมาเป็นแบบ recursive
  - ทำให้มือใหม่งงจนทำอะไรไม่ถูก
  - แต่ที่จริงไม่ต้องทำเป็นรีเคอร์ซีฟก็ได้ (แต่ก็ทำให้งงและยุ่งยากกว่าเดิม)
- เราจะเรียนวิธีเขียนโปรแกรมกับทั้งสองแบบเพราะ
  - BFS เป็นพื้นฐานของ Dijkstra's algorithm (หนึ่งในอัลกอริทึมที่ฮิตที่สุด)
  - DFS เป็นพื้นฐานของ Topological sorting

Graph Searching

- Given: a graph G = (V, E), directed or undirected
- Goal: methodically explore every vertex and every edge
- Ultimately: build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - Note: might also build a *forest* if graph is not connected

Breadth-First Search

- “Explore” a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find (“discover”) its children, then their children, etc.

Breadth-First Search

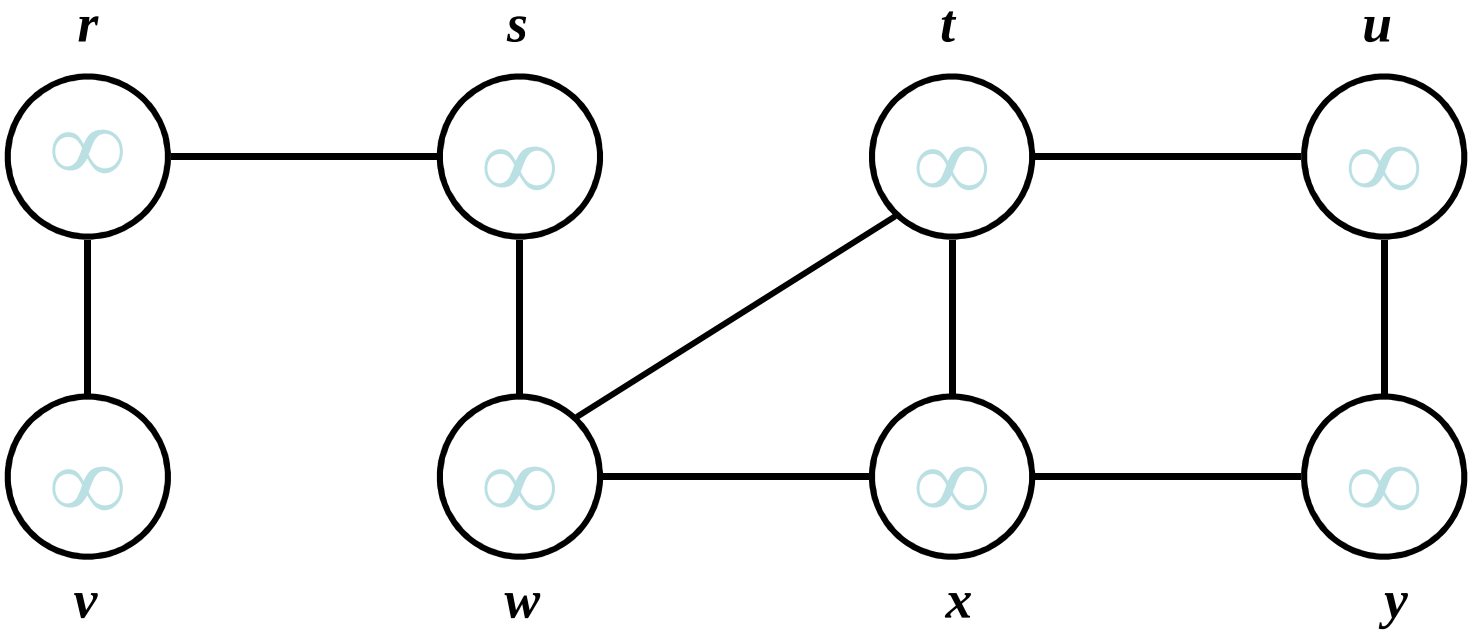
- Again will associate vertex “colors” to guide the algorithm
  - White vertices have not been discovered
    - All vertices start out white
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

Breadth-First Search

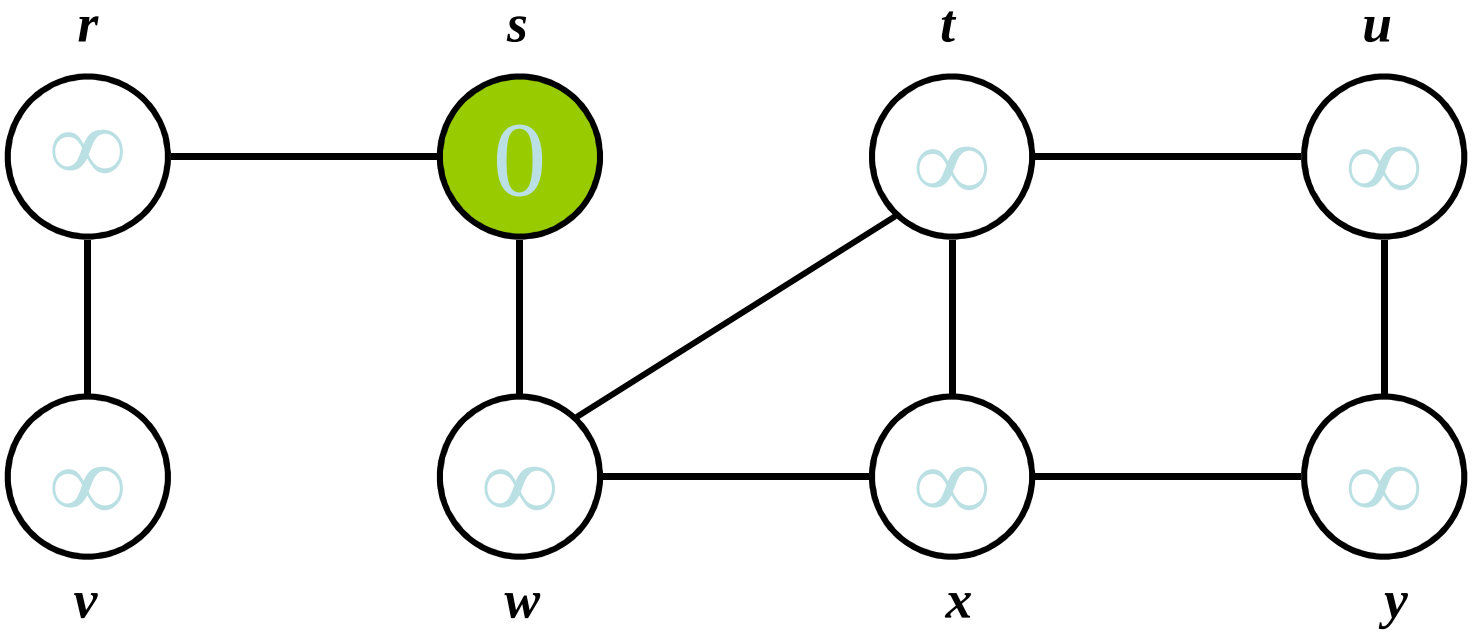
```
BFS(G,s) {
   initialize vertices;
   Q = {s};
   while (Q not empty) {
      u = Dequeue(Q);
      for each v adjacent to u do {
         if (color[v] == WHITE) {
            color[v] = GRAY;
            d[v] = d[u]+1; // compute d[]
            p[v] = u;  // build BFS tree
            Enqueue(Q,v);
      }
   }
   color[u] = BLACK;
}
```

BFS runs in O(V+E)

Breadth-First Search: Example

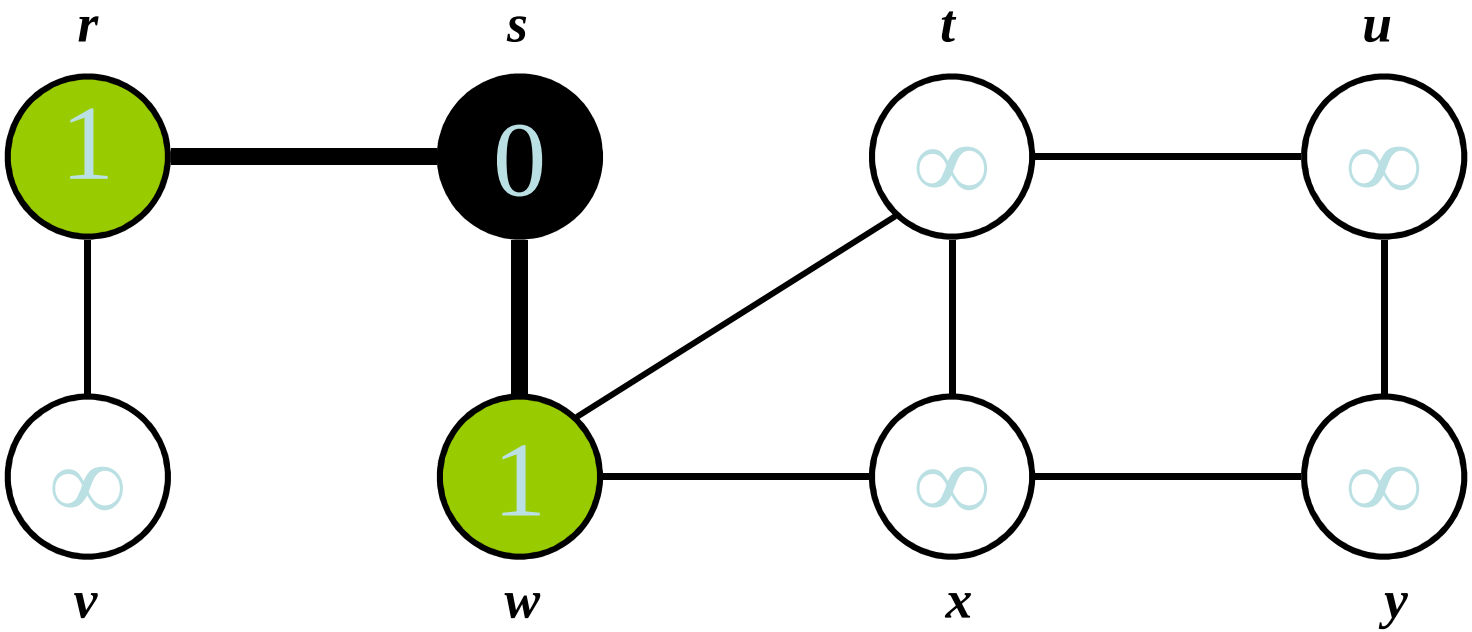


Breadth-First Search: Example



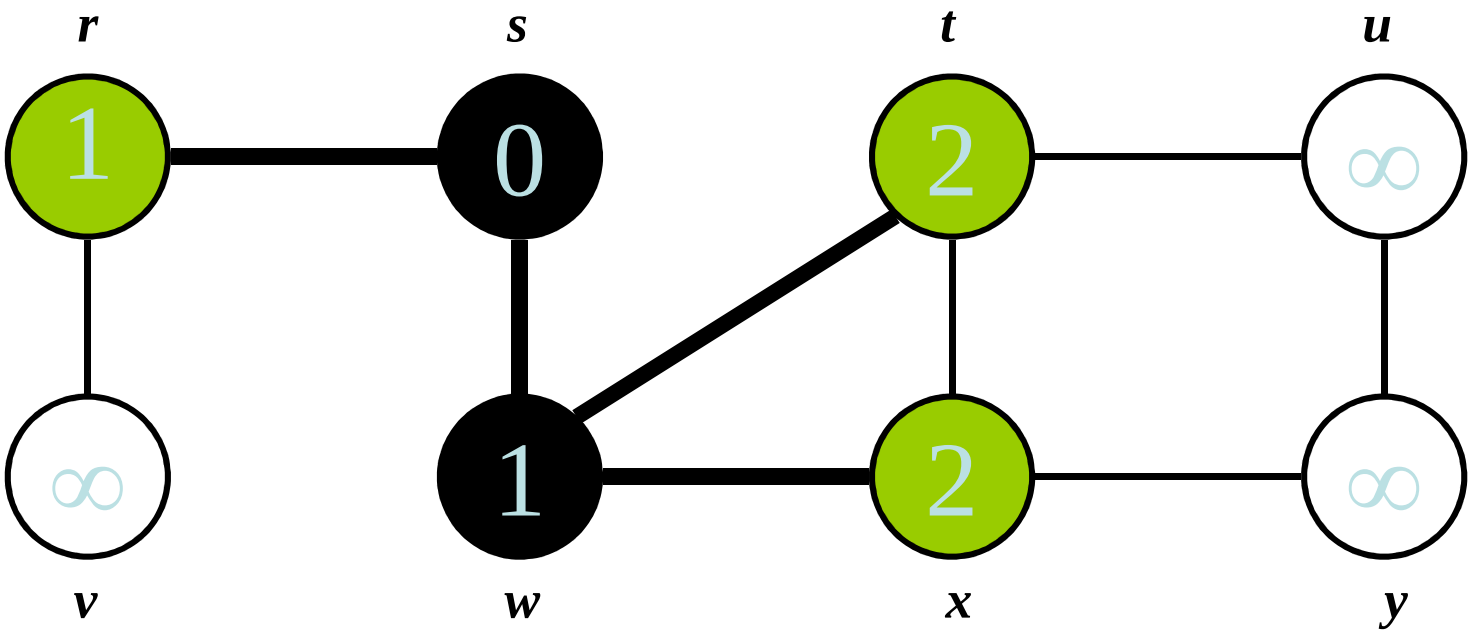
Q: | *s* |

Breadth-First Search: Example



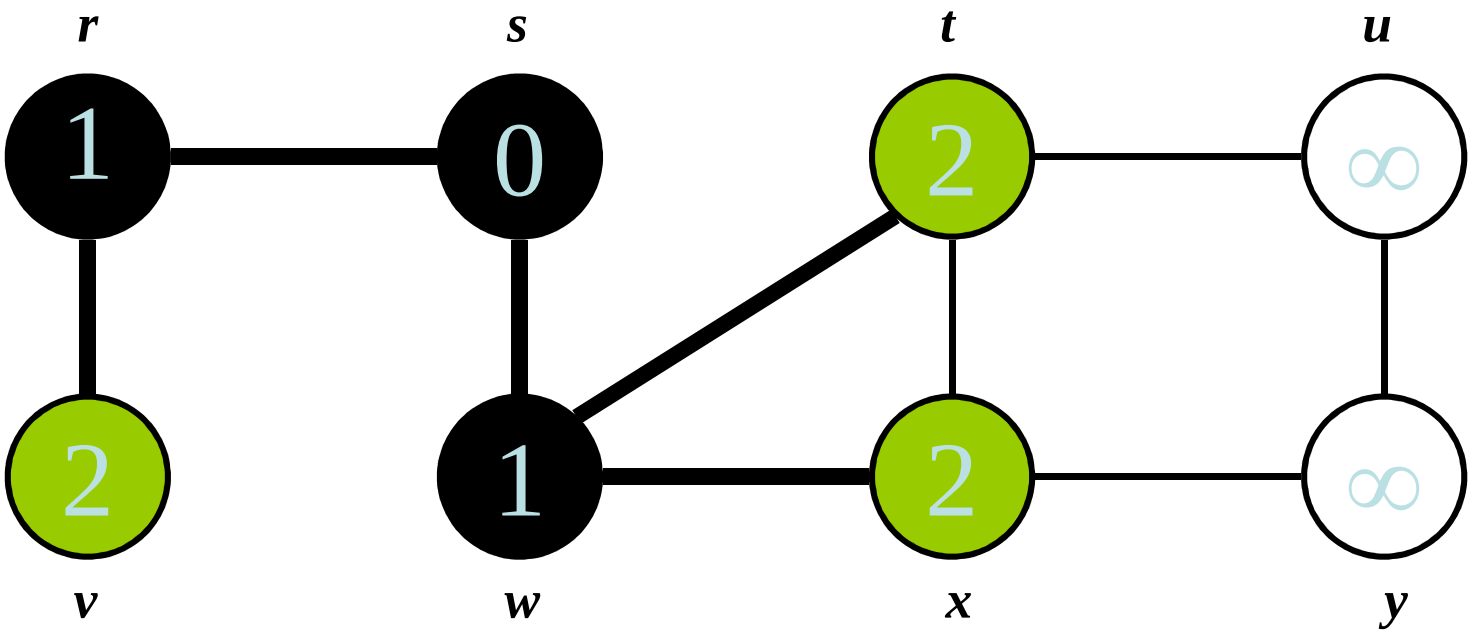
Q: | *w* | *r* |

Breadth-First Search: Example



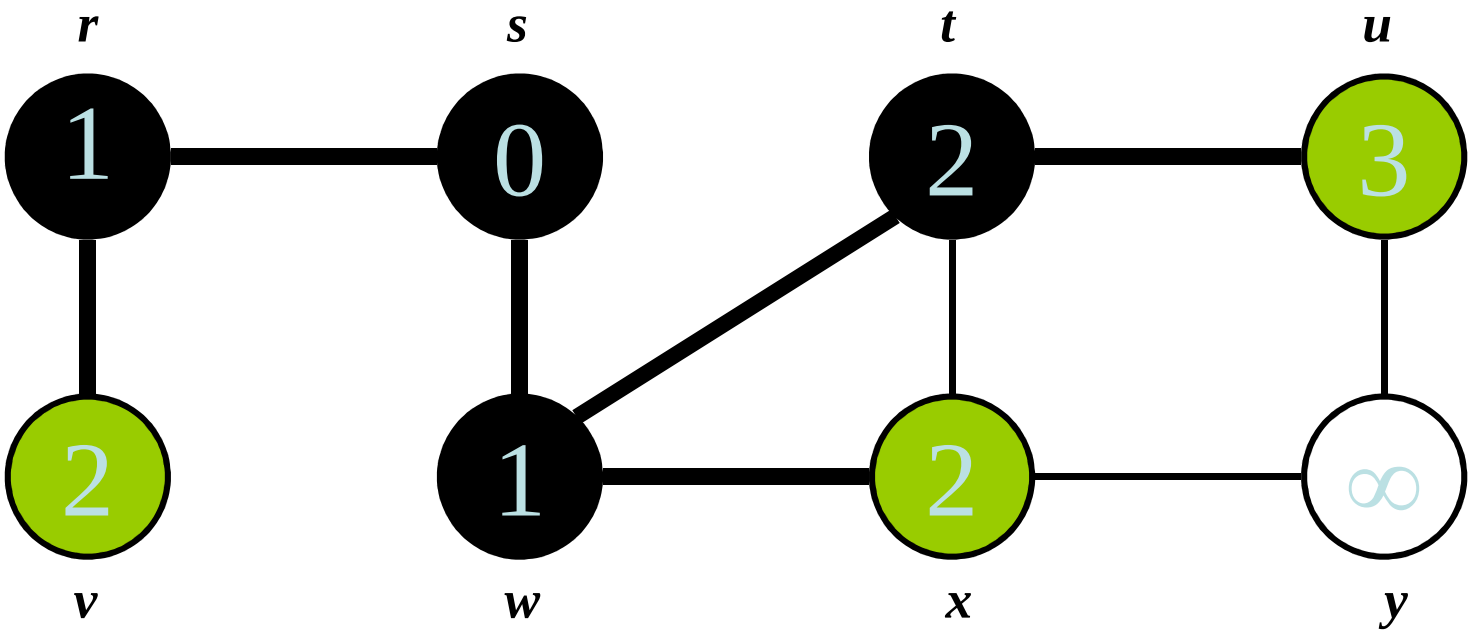
Q: | *r* | *t* | *x* |

Breadth-First Search: Example



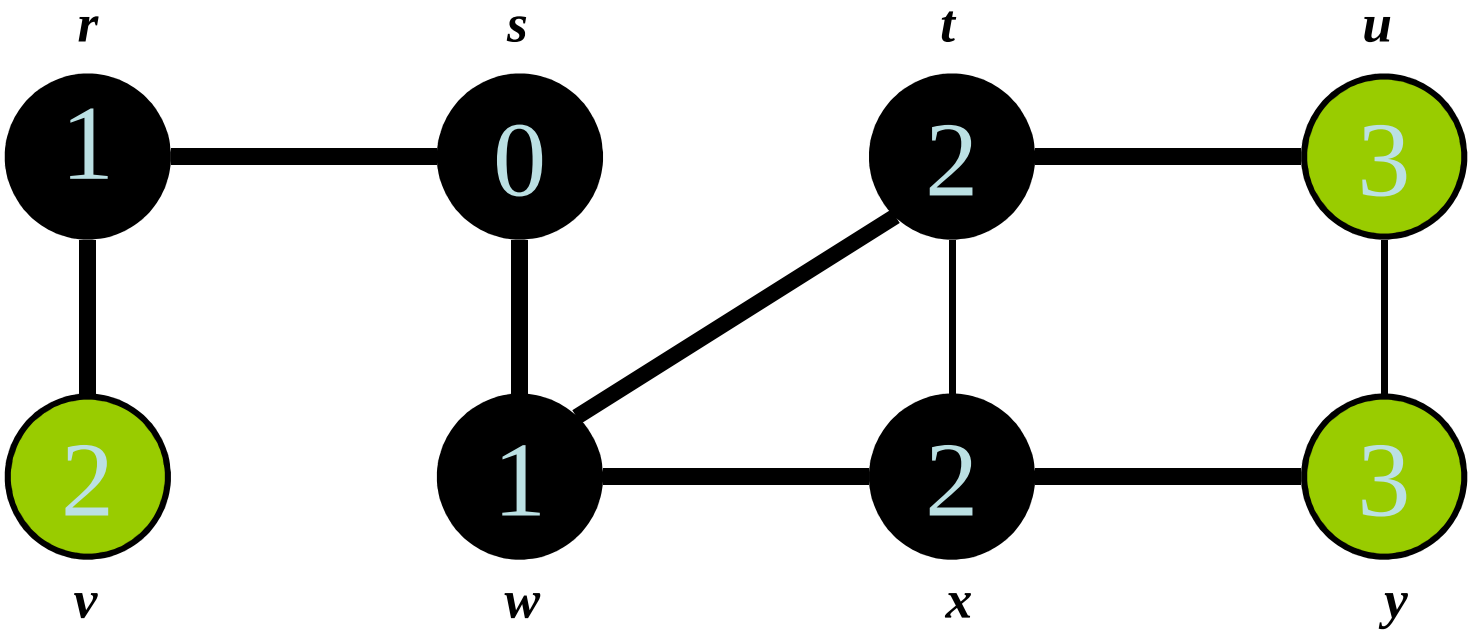
Q: | *t* | *x* | *v* |

Breadth-First Search: Example



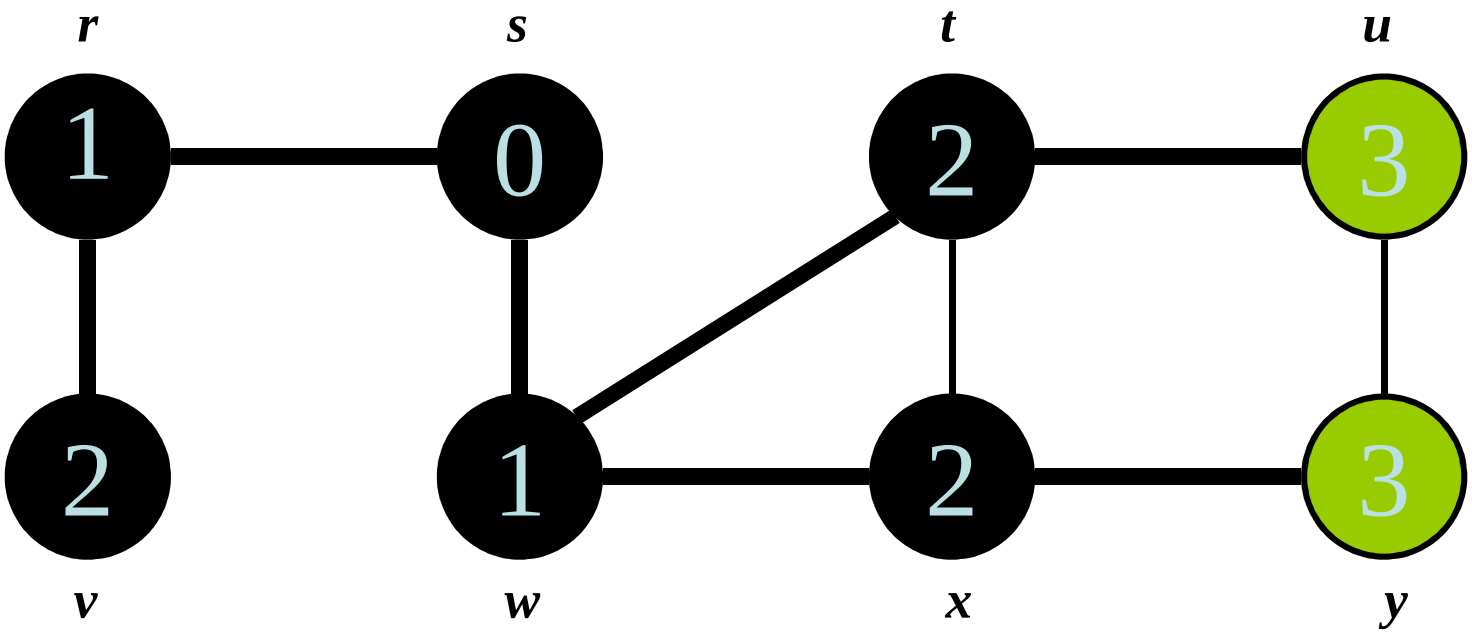
Q: | *x* | *v* | *u* |

Breadth-First Search: Example



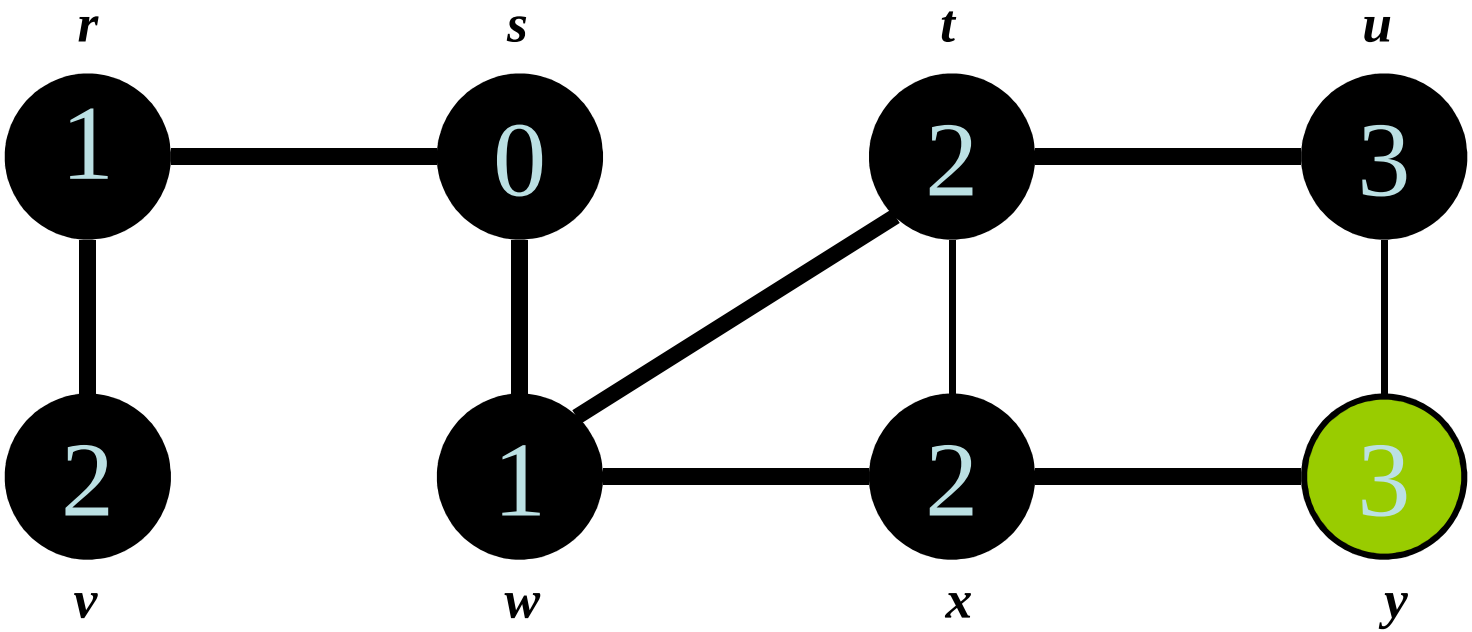
Q: | *v* | *u* | *y* |

Breadth-First Search: Example



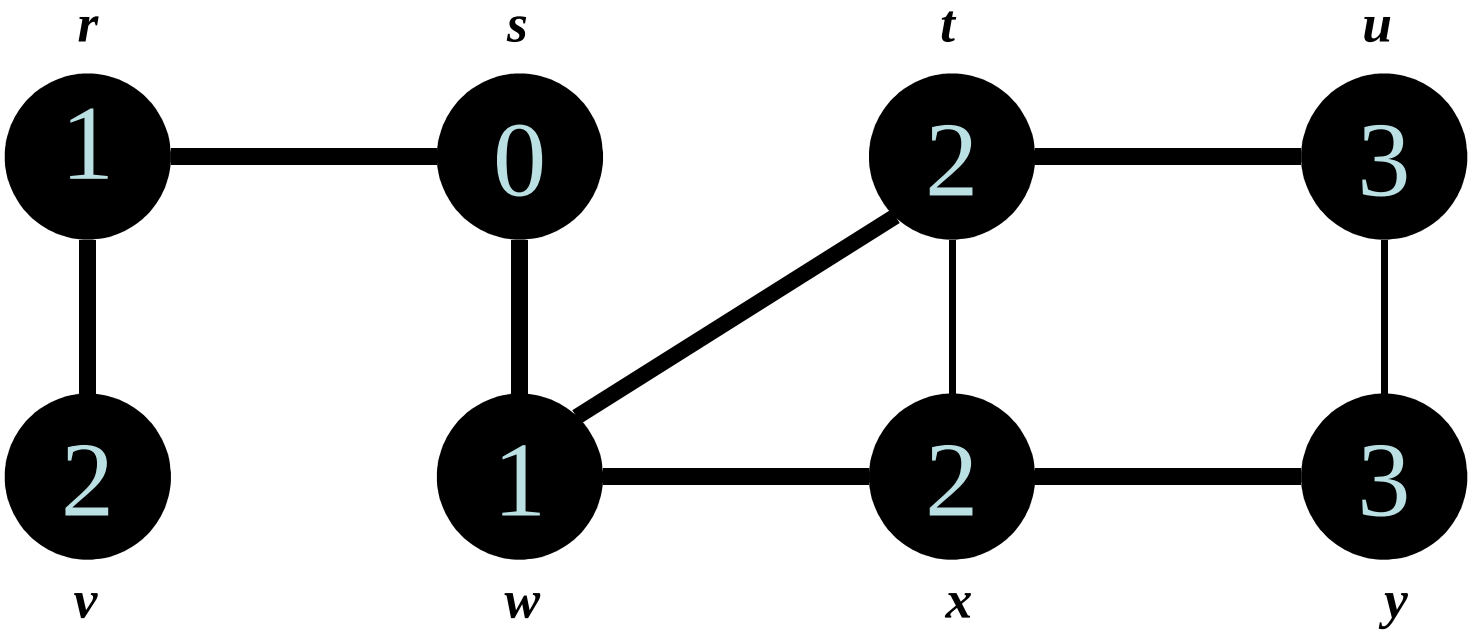
Q: | *u* | *y* |

Breadth-First Search: Example



Q: y

Breadth-First Search: Example



Q: ***Ø***

Breadth-First Search: Properties

- BFS calculates the *shortest-path distance* to the source node
  - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v, or ∞ if v not reachable from s
- BFS builds *breadth-first tree*, in which paths to root represent shortest paths in G
  - Thus can use BFS to calculate shortest path from one vertex to another in O(V+E) time

Some problems

- http://uva.onlinejudge.org/external/3/336.html
  336 - A Node Too Far

- http://uva.onlinejudge.org/external/4/439.html
- **439 - Knight Moves**

- http://uva.onlinejudge.org/external/100/10067.html
- **10067 - Playing with Wheels**