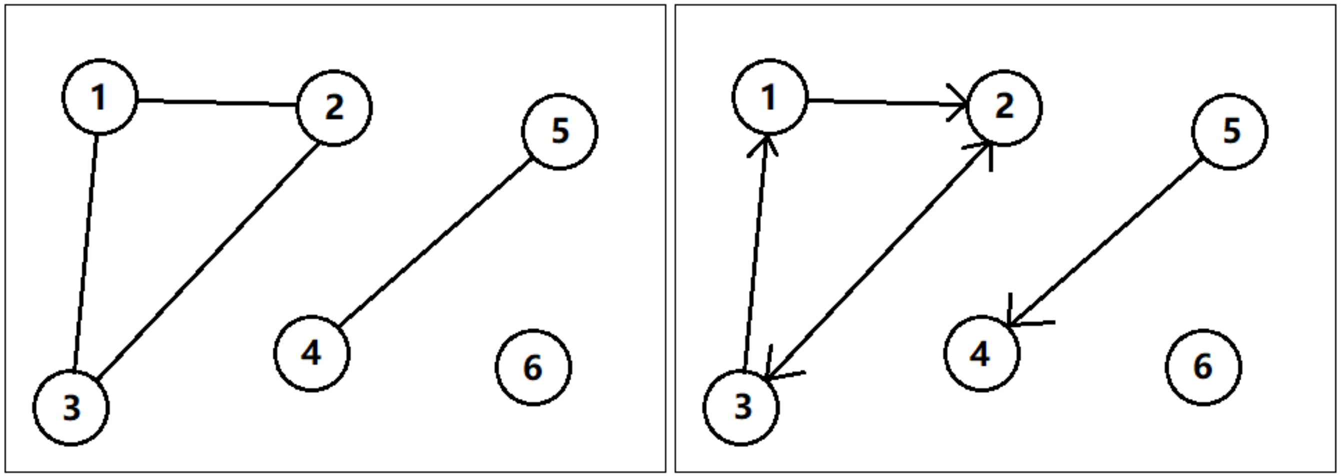
Data Structures and Algorithm Analysis

Lab 13, Graph

Contents

- Implementing graphs.
- Depth-first search and breadth-first search.

Undirected graph and directed graph



The picture on the left is an undirected graph. The picture on the right is a directed graph.

Note that a tree is an undirected graph such that all node are connected and have no cycles.

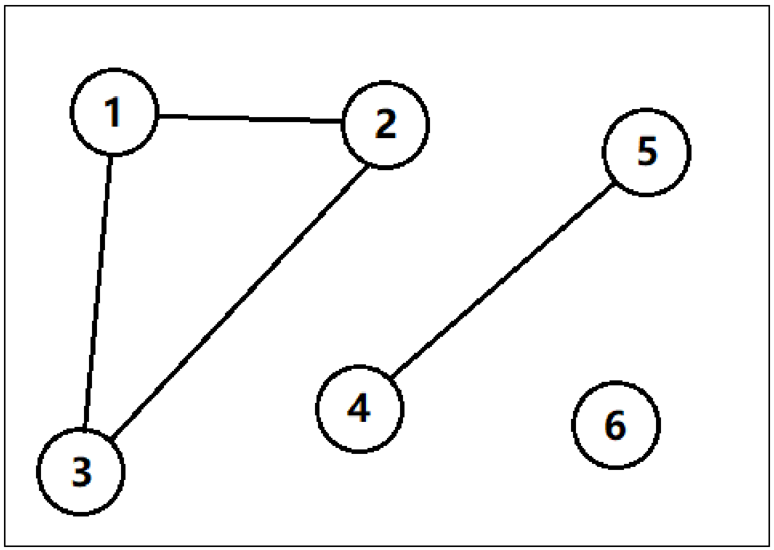
Represent a graph in JAVA

There are many ways to represent a graph in JAVA. Let's first define what a graph should contains.

```java
public interface Graph {
  // return number of vertices.
  int size();

  // add an edge between v1 and v2, where 0 <= v1, v2 <=
    size()-1
  void addEdge( int v1, int v2 );

  // return all vertices that vertex v is connected to.
  Iterable<Integer> adjacency( int v );

  // return whether there is an edge from v1 to v2.
  boolean hasEdge( int v1, int v2 );
}
```

Represent a graph in JAVA

Let's first implement a graph with adjacency matrix.



$$adj = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

If there is a edge from v1 to v2, set adj[v1][v2] to 1.

Graph with adjacency matrix

When implementing graph with adjacency matrix, we usually need to store number of vertices and an adjacency matrix.

```
public class GraphAdjacency implements Graph {

  private int num;
  private boolean[][] adj;

  public GraphAdjacency( int verticesNumber ) {
    num = verticesNumber;
    adj = new boolean[num][num];
  }
```

Graph with adjacency matrix

```
...

  public int size() {
      return num;
  }

  public void addEdge(int v1, int v2) {
      adj[v1][v2] = true;
  }

  public Iterable<Integer> adjacency(int v) {
      LinkedList<Integer> list = new LinkedList<>();
      for( int i = 0; i < num; i ++ )
          if( adj[v][i] )
              list.add(i);
      return list;
  }

  public boolean hasEdge(int v1, int v2) {
      return adj[v1][v2];
  }
```

Graph with adjacency matrix

Advantage:

- Very easy to implement.
- Very fast to add a new edge.
- Very fast to know whether two vertices are connected.

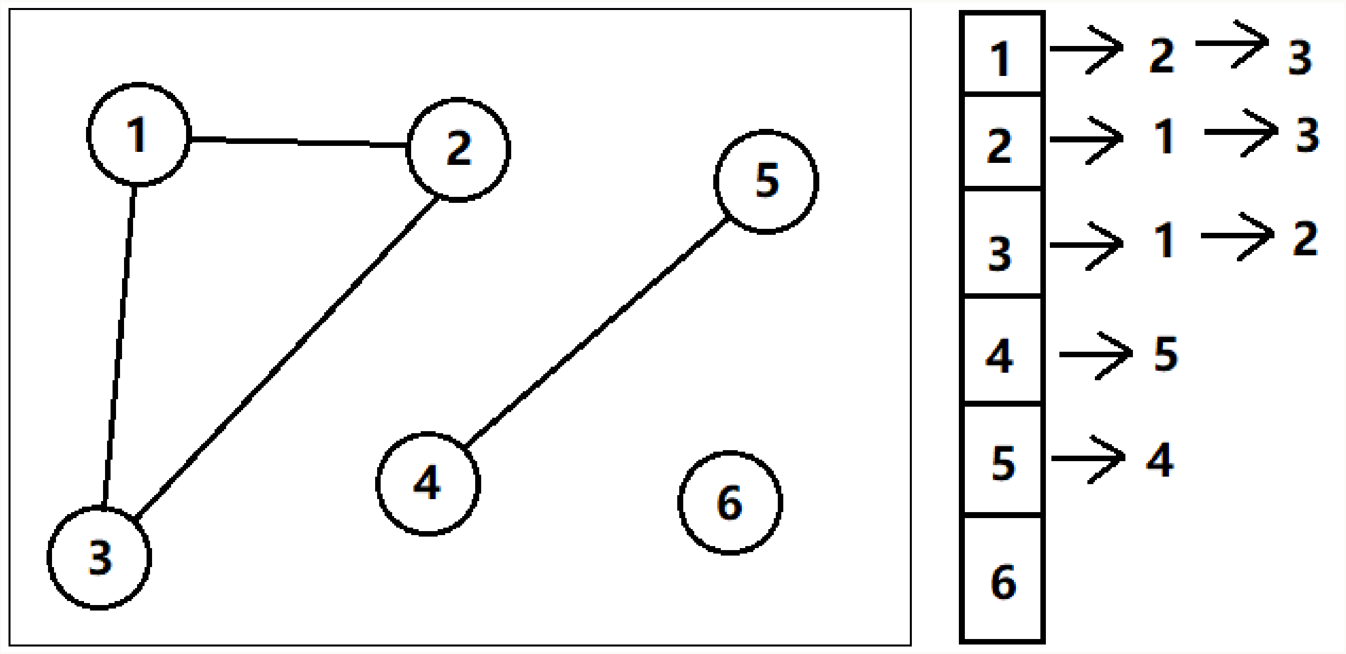
Disadvantage:

- N^2 storage requirement.
- Linear time to know all vertices connected to a vertex.

In conclusion, it is very suitable to be used in a dense graph.

Graph with adjacency lists

Let's implement a graph with adjacency list.



If there is a edge from v1 to v2, the list adj[v1] contains v2.

Graph with adjacency lists

When using linked list to implement this graph, be aware of the generics.

```
public class GraphAdjList implements Graph {

  private int num;
  private LinkedList<Integer>[] adjList;

  @SuppressWarnings (value="unchecked")
  public GraphAdjList( int verticesNumber ) {
      num = verticesNumber;
      adjList = new LinkedList[num];
      for( int i = 0; i < adjList.length; i ++ )
        adjList[i] = new LinkedList<>();
  }

  public int size() {
      return num;
  }
```

Graph with adjacency lists

```java
public void addEdge(int v1, int v2) {
  if( adjList[v1].contains(v2) )
      return;
  adjList[v1].add(v2);
}

public Iterable<Integer> adjacency(int v) {
  return (LinkedList<Integer>)adjList[v].clone();
}

public boolean hasEdge(int v1, int v2) {
  return adjList[v1].contains(v2);
}
```

Graph with adjacency lists

Advantage:

- Using (Vertices+Edges) space.
- Very fast to know all vertices connected to one vertex.

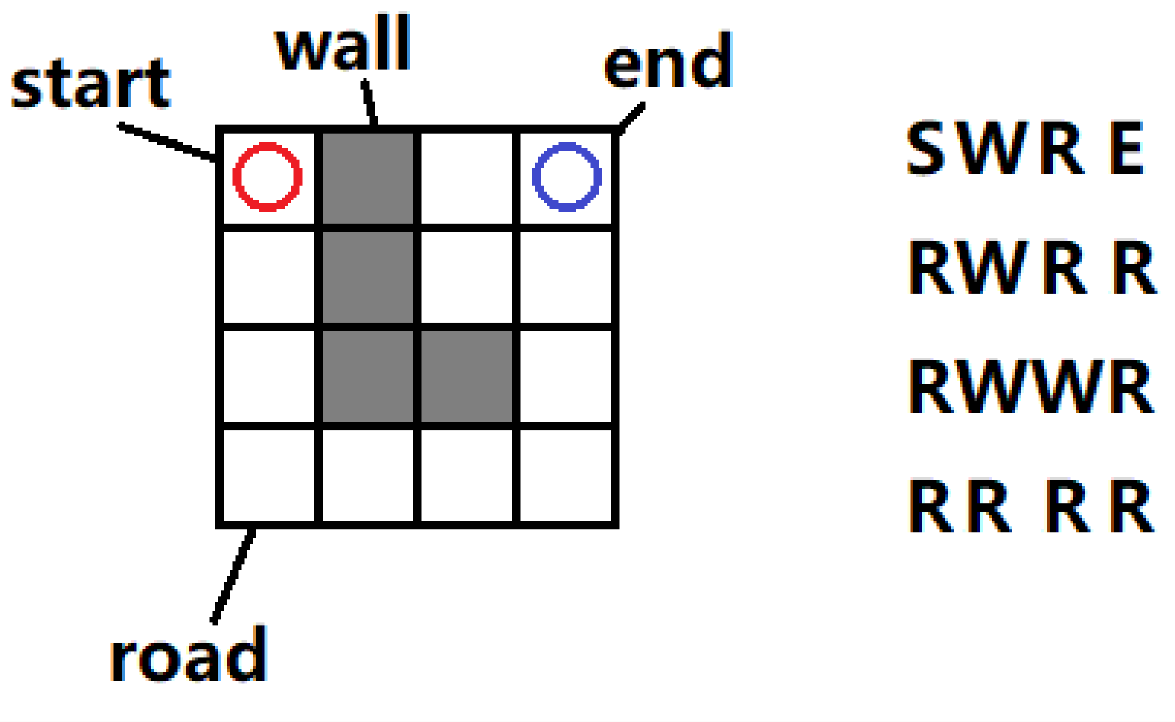
Disadvantage:

- Need to iterate the list to know if two vertices are connected.

In conclusion, it is very suitable to be used in a sparse graph.

Solve maze problem

Let's solve a searching problem. We use characters to represent a maze.



Solve maze problem

Given a maze, can you tell me whether you can reach the "end" from the "start"?

- Build a graph.
- Depth-first search.
- Breadth-first search.

```
public boolean solveMaze( char[][] maze ) {
  // ????
}
```

Solve maze problem

The first thing we need to do is to convert char[][] array into a graph. In this problem it is trivial. We just view each grid (or character) as a vertex. And there are edges between adjacent grids.

```
private static Graph buildGraph( char[][] maze ) {
  int height = maze.length;
  int width = maze[0].length;
  Graph graph = new GraphAdjList(height*width);

  for( int h = 0; h < height; h ++ )
      for( int w = 0; w < width; w ++ ) {
          // add edges between maze[h][w] and its neighbors
            if necessary.
      }
  return graph;
}
```

Depth-first search

Now we have the graph, we can use depth-first search to search from the "start" vertex to every other vertices.

```
boolean dfs( Graph graph, int current, int end, boolean[]
  visited ) {
  for( Integer adj : graph.adjacency(current) ) {
      if( visited[adj] )
          continue;
      visited[adj] = true;
      if( current == end )
          return true;
      if( dfs(graph, adj, end, visited) )
          return true;
  }
  return false;
}
boolean dfs( Graph graph, int start, int end ) {
  boolean[] visited = new boolean[graph.size()];
  return dfs( graph, start, end, visited);
}
```

Breadth-first search

We can also use the breadth-first search.

```
boolean bfs( Graph graph, int start, int end ) {
  boolean[] visited = new boolean[graph.size()];
  Queue<Integer> queue = new LinkedList<>();
  visited[start] = true;
  queue.add(start);
  while( !queue.isEmpty() ) {
      int current = queue.poll();
      if( current == end )
          return true;
      for( int adj : graph.adjacency(current) ) {
          if( visited[adj] )
              continue;
          visited[adj] = true;
          queue.add(adj);
      }
  }
  return false;
}
```

Solve maze problem

We may not need to explicitly define a "graph" structure every time. We may just use the "graph" concept to write algorithms.

```
private static boolean bfs( char[][] maze ) {
  int height = maze.length;
  int width = maze[0].length;
  boolean[][] visited = new boolean[height][width];
  Queue<Point> queue = new LinkedList<>();
  // add start point in queue and set visited
  int[] dh = new int[] { -1, 1,  0, 0 };
  int[] dw = new int[] {  0, 0, -1, 1 };
  while( !queue.isEmpty() ) {
      Point current = queue.poll();
      int h = current.x;
      int w = current.y;
      if( maze[h][w] == 'E' )
          return true;
      for( int i = 0; i < 4; i ++ ) {
          int h2 = h+dh[i];
          int w2 = w+dw[i];
```

```
            // if (h2,w2) is good add it into the queue
        }
    }
    return false;
```