Graphs:
Introduction
CIT5940

Applications

1. Modeling connectivity in computer and communications networks
2. Representing an abstract map as a set of locations with distances between locations. Used to compute shortest routes between locations
3. Modeling flow capacities in transportation networks to find which links create the bottlenecks
4. Finding a path from a starting condition to a goal condition This is a common way to model problems in artificial intelligence applications and computerized game players
5. Modeling computer algorithms, to show transitions from one program state to another
6. Finding an acceptable order for finishing subtasks in a complex activity, such as constructing large buildings

Definitions

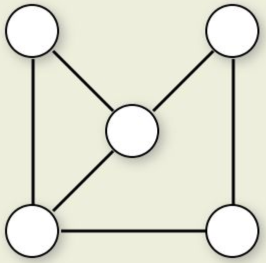
- A graph consists of:
  - A set of nodes
  - A set of edges where an edge connects two nodes
- Flexible data structure

Definitions

- A *graph* **G**=(**V**,**E**) consists of:
  - A set of *vertices* **V**
  - A set of *edges* **E**, such that each edge in **E** is a connection between a pair of vertices in **V**
- The number of vertices is written |**V**|
- The number of edges is written |**E**|
  - Where $0 \le |\mathbf{E}| \le |\mathbf{V}|^2 - |\mathbf{V}|$

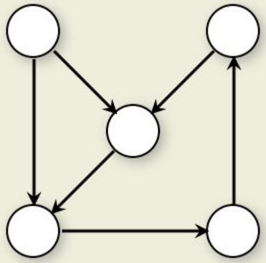
Definitions

- Undirected graph: A *graph* whose *edges* do not have a direction


(a) undirected graph

Definitions

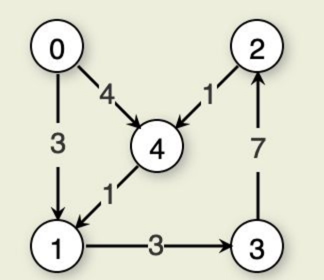
- Directed graph: A *graph* whose *edges* –each- are directed from one of its defining *vertices* to the other.



(b) directed graph

Definitions

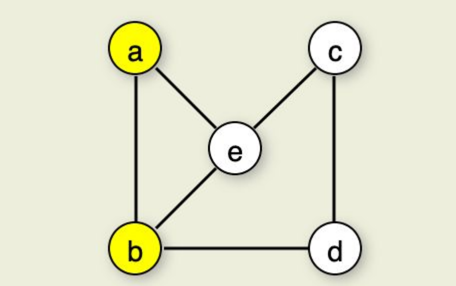
- Labeled graph: A *graph* with labels associated with the *nodes*



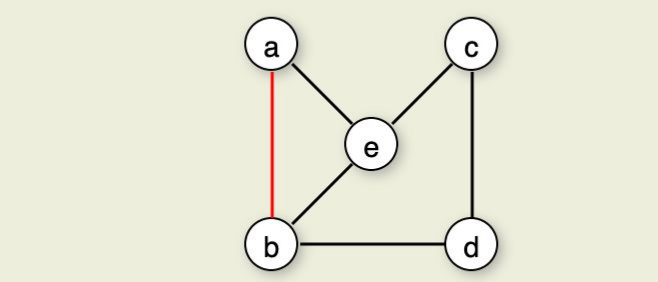
(c) labeled graph

Definitions

- An edge connecting vertices *a* and *b* is said to be *incident* with vertices *a* and *b*. And a and b are said to be *adjacent* *(neighbors)*.



(a) Vertices a and b are *neighbors*



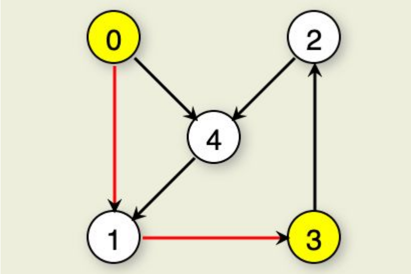
(b) The red edge is *incident* with vertices a and b

Definitions

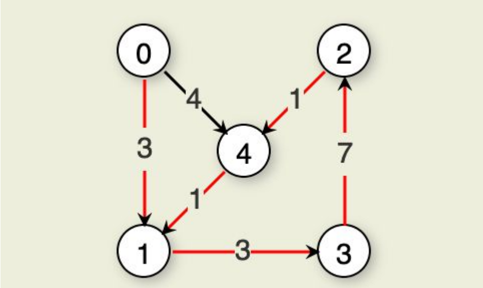
- The degree of a *vertex* is its number of *neighbors*
- In a *directed graph*
  - The *in degree* is the number of edges directed into the vertex
  - The *out degree* is the number of edges directed out of the vertex

Definitions

- A sequence of vertices $v_1, v_2, \ldots, v_n$ forms a *path* of length $n-1$ if there exist edges from v_i to v_{i+1} for $1 \le i < n$.
- A path is a *simple path* if all vertices on the path are distinct



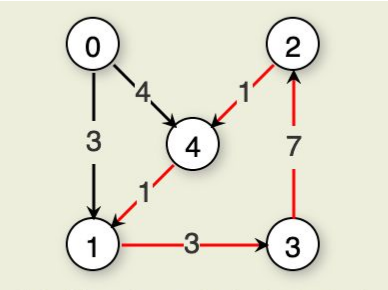
(a) A *simple* path from 0 to 3.



(b) Path 0, 1, 3, 2, 4, 1 is not simple

Definitions

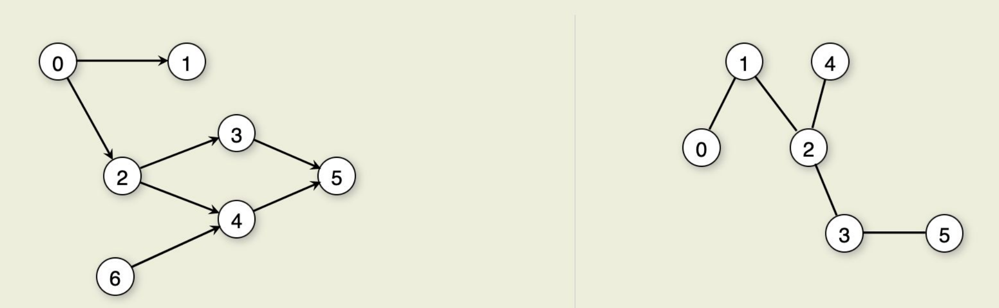
- A *cycle* is a path of length three or more that connects some vertex v_1 to itself.
- A cycle is a *simple cycle* if the path is simple, except for the first and last vertices being the same.



(c) *Simple cycle* 1, 3, 2, 4, 1

Definitions

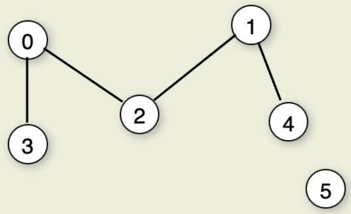
- A graph without cycles is called an *acyclic graph*
- A directed graph without cycles is called a *directed acyclic graph* or *DAG*



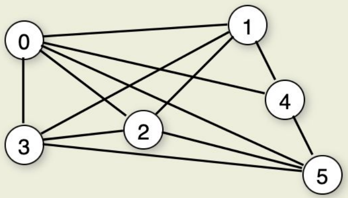
(a) Directed Acyclic Graph

(b) Acyclic Graph

Definitions

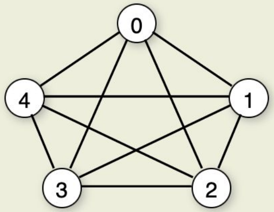


(a) A graph with relatively few edges is called a *sparse graph*.

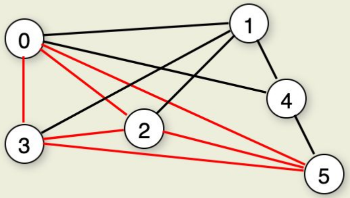


(b) A graph with many edges is called a *dense graph*.

Definitions



(c) A *complete graph* has edges connecting every pair of nodes.

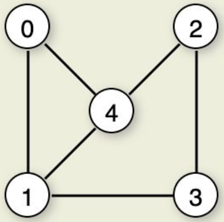


(d) A *clique* is a subset of V where all vertices in the subset have edges to all other vertices in the subset.

Representations: Adjacency Matrix

- The *adjacency matrix* for a graph is a |**V**|×|**V**| array
- The vertices are labeled from v_0 through $v_{|V|-1}$
- Row *i* of the adjacency matrix contains entries for Vertex v_i
- Column *j* in row *i* is marked if there is an edge from v_i to v_j and is not marked otherwise
- The space requirements for the adjacency matrix are $O(|V^2|)$

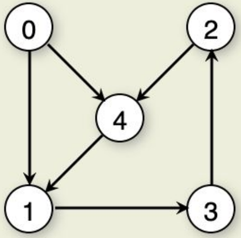
Adjacency Matrix: undirected graph



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

Adjacency Matrix

Adjacency Matrix : directed graph



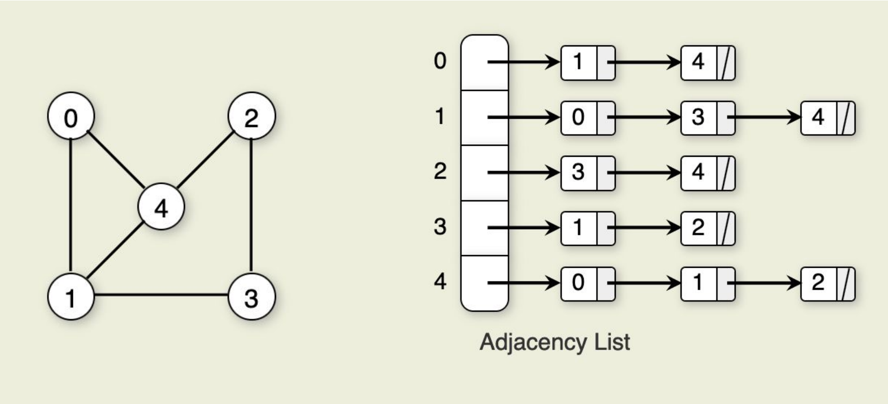
	0	1	2	3	4
0		1			1
1				1	
2					1
3			1		
4		1			

Adjacency Matrix

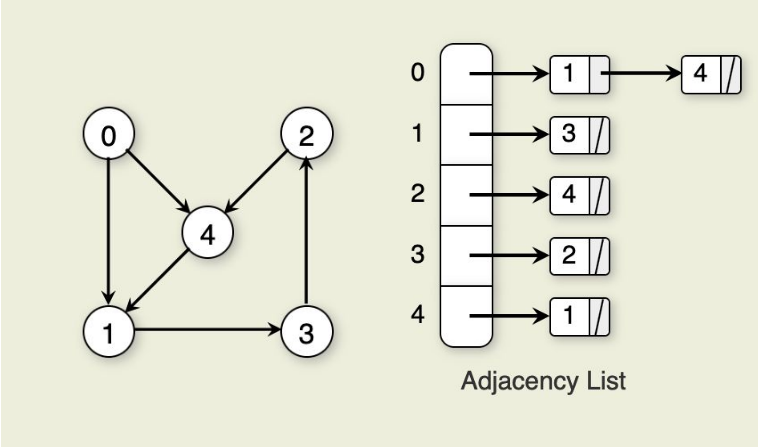
Representations: Adjacency List

- The adjacency list is an array of linked lists
- The array is |**V**| items long, with position *i* storing a pointer to the linked list of edges for Vertex v_i
- The linked list at position i represents the edges by the vertices that are adjacent to Vertex *vi*

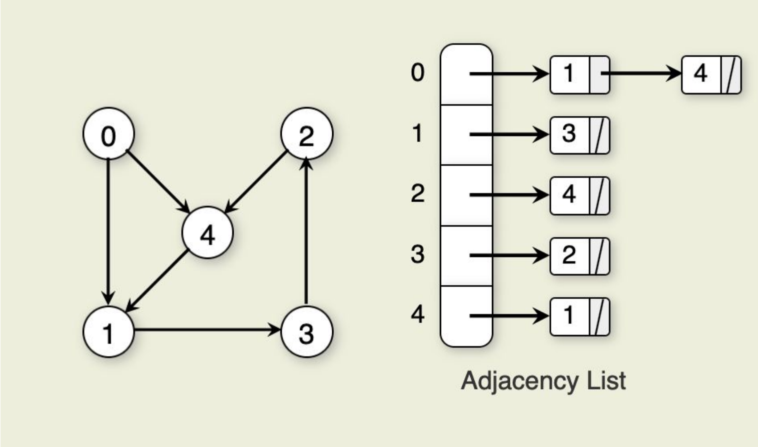
Adjacency List: undirected graph



Adjacency List : directed graph



Adjacency List : directed graph



In practice, we’ll usually use a HashMap<Vertex, List<Vertex>> as opposed to a List<Vertex<List>>

Graph Traversals

Depth-First Search (DFS)

- Whenever a vertex *v* is visited during the search, DFS will recursively visit all of *v* 's unvisited neighbors
- DFS can be implemented using a stack:
  - The neighbor(s) are pushed onto the stack
- The next vertex to be visited is determined by popping the stack and following that edge
- The total cost is $O(|\mathbf{V}|+|\mathbf{E}|)$

Breadth-First Search (BFS)

- Whenever a vertex *v* is visited during the search, BFS will visit all of *v* 's neighbors before visiting vertices further away
- BFS can be implemented using a queue
    - The neighbor(s) are enqueued
- The next vertex to be visited is determined by dequeuing the queue and following that edge
- The total cost is $O(|\mathbf{V}|+|\mathbf{E}|)$

Visitor Pattern & Searches

- Common uses for searching:
  - finding a node matching some criterion
  - modifying all nodes accessible from a given node
  - topological sort
- In each case, the basic pattern of the search stays the same
  - Good use case for a **visitor pattern**
- Essentials of the Visitor Pattern**:**
  - Interface with one method: **visit(Vertex v)**

Strategy Pattern & Searches

- Common searches:
  - BFS
  - DFS
  - Dijkstra's/Bellman-Ford for Shortest Path
  - Greedy searches
- In each case, the basic behavior of the search is the same, varying only the order that we dequeue vertices from the frontier queue.
- Essentials of the Strategy Pattern**:**
  - Interface with one method: **execute()**

Functional Interfaces & Java

- Interfaces with only one method are considered **functional interfaces** in Java
- They can behave just like any normal interface, with implementing classes…
- …or, they can be instantiated **anonymously** using **method references** & **lambda expressions**

Method References

- System.out.println() is a method (println()) belonging to the System.out class.
- To reference the method as a **first-class object,** we can write:
  - System.out::println
  - This is an object belonging to the abstract type **Consumer<Object>**
- Method references allow us to pass along methods as inputs to other methods!

Lambda Expressions

- A way to write short functions without specifying a name/signature

```
(e1, e2) -> e1 == e2 || e1.value == e2.value
```

formal parameter name(s)

arrow token, signifying the start of the function body

a single expression that will be evaluated; this value will be the return value of the function.

Lambda Expressions

- A way to write short functions without specifying a name/signature

```
p -> p.getGender() == Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <=
25
```

formal parameter name(s)

arrow token, signifying the start of the function body

a single expression that will be evaluated; this value will be the return value of the function.

Challenge #1

Can you use **method references** to make the PrintVisitor class redundant?

Challenge #1

Can you use **method references** to make the PrintVisitor class redundant?

g.search(vertex, **System.out::println**)

Challenge #2

Can you use **lambdas** to define a Visitor that changes all vertex labels to uppercase?

Challenge #2

Can you use **lambdas** to define a Visitor that changes all vertex labels to uppercase?

```
g.search(joe, v -> v.label = v.label.toUpperCase());
```

Challenge #3

Can you use **lambdas** to define a Search Strategy that creates a PriorityQueue for the search that chooses the next node from the frontier based on the label of the node?