Directed graphs

anhtt-fit@mail.hut.edu.vn

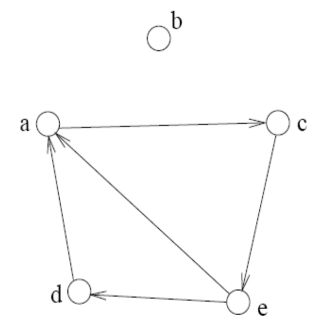
Terminology

- Connected graph
  - A graph is connected if and only if there exists a path between every pair of distinct vertices
- Sub-graph
  - A graph with the vertex and edge set being subsets of the original graph
- Connected Components
  - A connected component of a graph is a maximally connected subgraph of a graph
- Cycle
  - A path in a graph that starts and ends at the same vertex
- Tree
  - A graph G is a tree if and only if it is connected and acyclic
- Directed Graph
  - A graph whose the edges (arcs) are directional
- Directed Acyclic Graph
  - A directed graph with no directed cycles

Directed Graphs

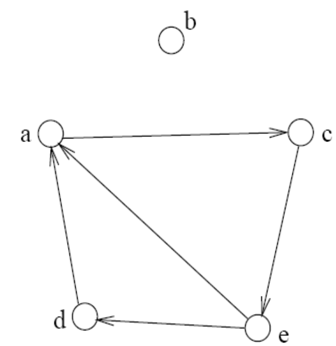
- A directed graph can be represented by an adjacency matrix/list the same way as in undirected graph, except:
  - An arc (u, v) only contributes to 1 entry in the adj. matrix or 1 node in the adj. list

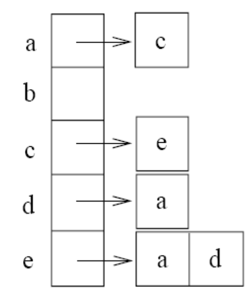
1. Adjacency Matrix



	a	b	c	d	e
a	0	0	1	0	0
b	0	0	0	0	0
c	0	0	0	0	1
d	1	0	0	0	0
e	1	0	0	1	0

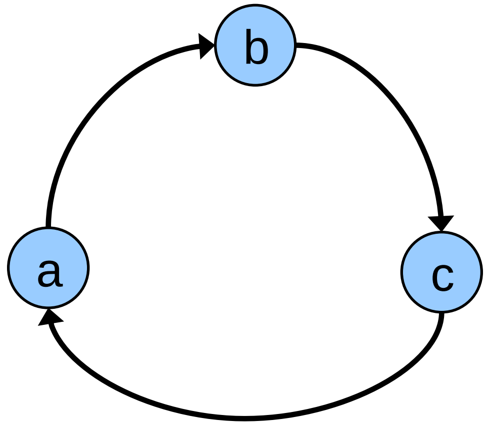
2. Adjacency List

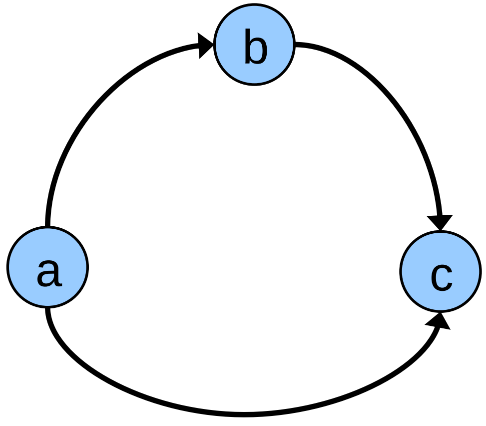




Paths/Cycles

- A directed graph can also contain paths and cycles ("directed paths" and "directed cycles")
  - Graph on top has directed paths and directed cycle
  - Graph on bottom has directed paths but NO directed cycle (acyclic)





Graph traversal

- BFS and DFS can be used to traverse a directed graph, the same way as in undirected graph

DAG

- DAG is directed graph without directed cycles
- To check for cycles in a directed graph
  - run DFS using an arbitrary vertex as the source. If the source can be visited twice, then there is a cycle in the graph.

A complete graph API

- In the current graph API, only the edges are managed. Therefore we can not know how many vertices there are in the graph. Each vertex need also a name for identification.
- Redefine the graph structure in order the vertices data are stored in a tree as the following

```
typedef struct {
    JRB edges;
    JRB vertices;
} Graph;
```

API (cont.)

```
Graph createGraph();
void addVertex(Graph graph, int id, char* name);
char *getVertex(Graph graph, int id);
void addEdge(Graph graph, int v1, int v2);
int hasEdge(Graph graph, int v1, int v2);
int indegree(Graph graph, int v, int* output);
int outdegree(Graph graph, int v, int* output);
int DAG(Graph graph);
void dropGraph(Graph graph);
```

Quiz 1

- Implement the directed graph API based on the given specification
- Test your new API with the following example

```
Graph g = createGraph();
addVertex(g, 0, "V0");
addVertex(g, 1, "V1");
addVertex(g, 2, "V2");
addVertex(g, 3, "V3");
addEdge(g, 0, 1);
addEdge(g, 1, 2);
addEdge(g, 2, 0);
addEdge(g, 1, 3);
if (DAG(g)) printf("The graph is acycle\n");
else printf("Have cycles in the graph\n");
```

Solution

- directed_graph.c

Topological Sort

- One can make use of the direction in the directed graph to represent a dependent relationship
  - COMP104 is a pre-requisite of COMP171
  - Breakfast has to be taken before lunch
- A typical application is to schedule an order preserving the order-of-completion constraints following a topological sort algorithm
  - We let each vertex represents a task to be executed. Tasks are inter-dependent that some tasks cannot start before another task finishes
  - Given a directed acyclic graph, our goal is to output a linear order of the tasks so that the chronological constraints posed by the arcs are respected
  - The linear order may not be unique

Topological Sort Algorithm

1. Build an "indegree table" of the DAG
2. Output a vertex v with zero indegree
3. For vertex v, the arc (v, w) is no longer useful since the task (vertex) w does not need to wait for v to finish anymore
   - So after outputting the vertex v, we can remove v and all its outgoing arcs. The result graph is still a directed acyclic graph. So we can repeat from step 2 until no vertex is left

Demo

- demo-topological.ppt

Pseudocode

```
Algorithm TSort(G)
Input: a directed acyclic graph G
Output: a topological ordering of vertices
Initialize Q to be an empty queue;
For each vertex v
     do if indegree(v) = 0
                then enqueue(Q,v);
While Q is non-empty
     do v := dequeue(Q);
          output v;
          for each arc(v,w)
               do indegree(w) = indegree(w)-1;
                    if indegree(w) = 0
                         then enqueue(w);
```

Quiz 2

- Let a file describe the perquisites between classes as the following

```
CLASS CS140
  PREREQ CS102
CLASS CS160
  PREREQ CS102
CLASS CS302
  PREREQ CS140
CLASS CS311
  PREREQ MATH300
  PREREQ CS302
```

- Use the last graph API to write a program to give a topological order of these classes

Hint

- Create a new function in the for topological sort.
  - void topologicalSort(Graph g, int* output, int* n);
- Reuse the following example to test the function.

```
Graph g = createGraph();
addVertex(g, 0, "CS102"); addVertex(g, 1, "CS140");
addVertex(g, 2, "CS160"); addVertex(g, 3, "CS302");
addVertex(g, 4, "CS311"); addVertex(g, 5, "MATH300");
addEdge(g, 0, 1); addEdge(g, 0, 2);
addEdge(g, 1, 3); addEdge(g, 5, 4); addEdge(g, 3, 4);
if (!DAG(g)) {
   printf("Can not make topological sort\n");
   return 1; }
topologicalSort(g, output, &n);
printf("The topological order:\n");
for (i=0; i<n; i++)
    printf("%s\n", getVertex(g, output[i]));
```

Solution

- topological_sort.c