Introduction to Algorithms and Data Structures

Lecture 14: Graphs, BFS, DFS

Mary Cryan

School of Informatics
University of Edinburgh

Directed and Undirected Graphs

- A *graph* is a mathematical structure consisting of a set of *vertices* and a set of *edges* connecting the vertices.

  Formally: $G = (V, E)$, where V is a set and $E \subseteq V \times V$.

- $G = (V, E)$ *undirected* if for all $v, w \in V$:

  $$(v, w) \in E \iff (w, v) \in E.$$

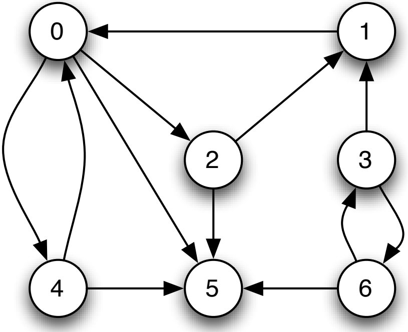
  Otherwise *directed*.

  Directed $\sim$ *arrows* (one-way)
  Undirected $\sim$ *lines* (two-way)

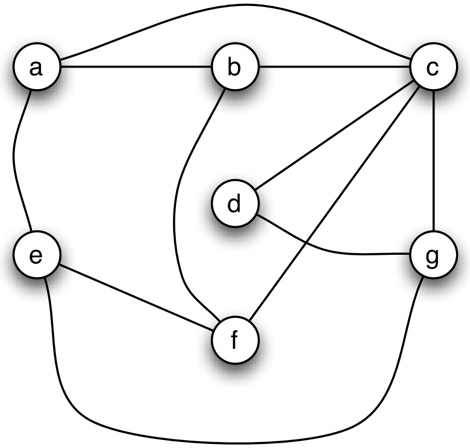
A directed graph

$G = (V, E)$ with vertex set $V = \{0, 1, 2, 3, 4, 5, 6\}$ and edge set

$$E = \{(0,2), (0,4), (0,5), (1,0), (2,1), (2,5), (3,1), (3,6), (4,0), (4,5), (6,3), (6,5)\}.$$



An undirected graph



Examples of graphs in "real life"

Road Maps.
Edges represent streets and vertices represent crossings.

Examples (cont'd)

Computer Networks.
Vertices represent computers and edges represent network connections (cables) between them.

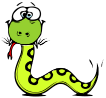
The World Wide Web.
Vertices represent webpages, and edges represent hyperlinks.

Adjacency matrices

Let $G = (V, E)$ be a graph with n vertices, with vertices numbered $0, \dots, n-1$.

The *adjacency matrix* of G is the $n \times n$ matrix $A = (a_{ij})_{0 \le i,j \le n-1}$ with

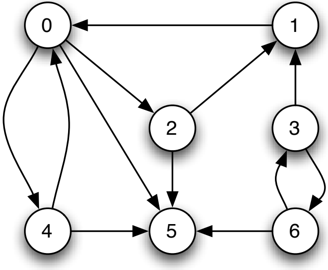
$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from vertex } i \text{ to vertex } j \\ 0 & \text{otherwise.} \end{cases}$$



Python arrays are a bit strange to work with, being set up as “lists of lists”. Alternatives are:

- Import `NumPy`, and use their true 2D arrays
- Define a mapping $(i, j) \to i * n + j$ (where n is the number of vertices) and work with a $n * n$ or $n * (n + 1)/2$ sized 1D array in python.

Adjacency matrix (Example)

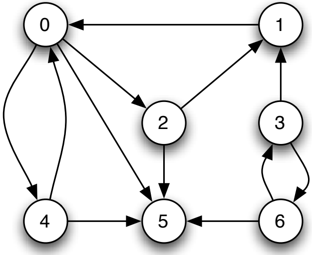


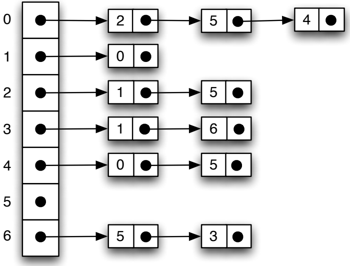
$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Adjacency lists

Array with one entry for each vertex v, which is a list of all vertices adjacent to v.

Example





Lists or matrices?

Given: graph $G = (V, E)$, with $n = |V|$, $m = |E|$.
For $v \in V$, we write $in(v)$ for in-degree, $out(v)$ for out-degree.

Questions to think about:?

1. Which data structure has faster (asymptotic) worst-case running-time, for *checking if w is adjacent to v*, for a given pair of vertices?
2. Which data structure has faster (asymptotic) worst-case running-time, for *visiting all vertices w adjacent to v*, for a given vertex v?

Adjacency Matrices vs Adjacency Lists

	adjacency matrix	adjacency list
Space	$\Theta(n^2)$	$\Theta(n+m)$
Time to check if w adjacent to v	$\Theta(1)$	$\Theta(out(v))$
Time to visit all w adjacent to v.	$\Theta(n)$	$\Theta(out(v))$
Time to visit all edges	$\Theta(n^2)$	$\Theta(n+m)$

Sparse and dense graphs

$G = (V, E)$ graph with n vertices and m edges

Observation: $m \leq n^2$

- G *dense* if m close to n^2
- G *sparse* if m much smaller than n^2

What about *planar* graphs?

"Kuratowski's criterion" for Planar graphs has the Corollary that

$$|E| \ \leq \ 3|V| - 6.$$

Graph traversals

A *traversal* is a strategy for visiting all vertices of a graph.

$$\boxed{\textit{BFS} = \textit{breadth-first search}}$$

$$\boxed{\textit{DFS} = \textit{depth-first search}}$$

General strategy:

1. Let v be an arbitrary vertex
2. Visit all vertices reachable from v
3. If there are vertices that have not been visited, let v be such a vertex and go back to 2.

BFS

Visit all vertices reachable from v in the following order:

- v
- all neighbours of v
- all neighbours of neighbours of v that have not been visited yet
- all neighbours of neighbours of neighbours of v that have not been visited yet
- etc.

BFS (using a Queue)

Algorithm bfs(G)

1. Initialise Boolean array $visited$, setting all entries to FALSE.
2. Initialise *Queue* Q
3. **for all** $v \in V$ **do**
4.   **if** $visited[v] = \text{FALSE}$ **then**
5.   bfsFromVertex(G, v)

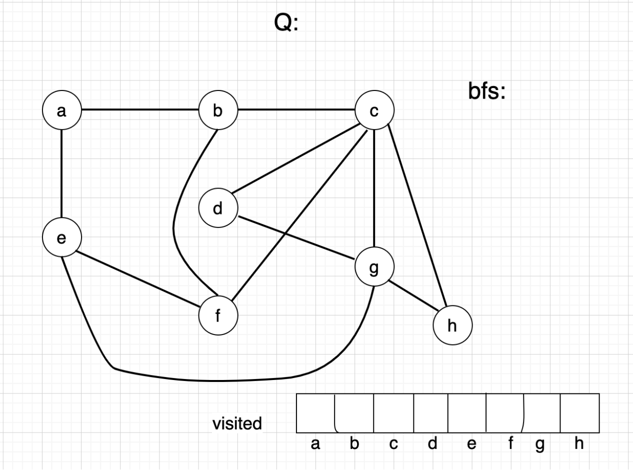
BFS (using a Queue)

Algorithm bfsFromVertex(G, v)

```
visited[v] = TRUE
Q.enqueue(v)
while not Q.isEmpty() do
        u ← Q.dequeue()
        for all w adjacent to u do
                if visited[w] = FALSE then
                        visited[w] = TRUE
                        Q.enqueue(w)
```

Example for Breadth-First Search

The video recording shows a demonstration of bfs on the graph below:



DFS

Visit all vertices reachable from v in the following order:

- v
- some neighbor w of v that has not been visited yet
- some neighbor x of w that has not been visited yet
- etc., until the current vertex has no neighbour that has not been visited yet
- Backtrack to the first vertex that has a yet unvisited neighbour v'.
- Continue with v', a neighbour, a neighbour of the neighbour, etc., backtrack, etc.

DFS (using a stack)

Algorithm dfs(G)

Initialise Boolean array $visited$, setting all to FALSE
Initialise *Stack* S
for all $v \in V$ **do**
    if $visited[v] = \text{FALSE}$ **then**
        dfsFromVertex(G, v)

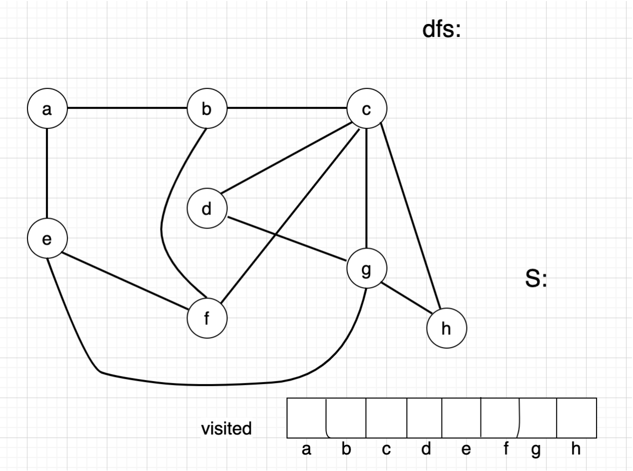
DFS (using a stack)

Algorithm dfsFromVertex(G, v)

```
S.push(v)
while not S.isEmpty() do
    u ← S.pop()
    if visited[u] = FALSE then
        visited[u] = TRUE
        for all w adjacent to u do
            S.push(w)
```

Example for Depth-First Search

The video recording shows a demonstration of dfs on the graph below:



Reading

Today's lecture:

- Graph representations in Section 22.1
- Breadth-first search Section 22.2, Depth-first search Section 22.3

Next week:

- Showing why Depth-first search (and Breadth-first search) run in $\Theta(n+m)$ time.
- Computing connected components and topological sort Section 22.4