Graphs: Connectivity

Jordi Cortadella and Jordi Petit
Department of Computer Science
The network graph formed by Wikipedia editors (edges) contributing to different Wikipedia language versions (vertices) during one month in summer 2013
Transportation systems
Social networks
World Wide Web
Disease transmission network

Transmission of renewable energy

Topology of regional transmission grid model of continental Europe in 2020
What would we like to solve on graphs?

• Finding paths: which is the shortest route from home to my workplace?

• Flow problems: what is the maximum amount of people that can be transported in Barcelona at rush hours?

• Constraints: how can we schedule the use of the operating room in a hospital to minimize the length of the waiting list?

• Clustering: can we identify groups of friends by analyzing their activity in twitter?
A significant part of the material used in this chapter has been inspired by the book:


(several examples and figures are taken from the book)
A graph is specified by a set of vertices (or nodes) $V$ and a set of edges $E$.

$V = \{1, 2, 3, 4, 5\}$

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

Graphs can be directed or undirected. Undirected graphs have a symmetric relation.
A graph with $n = |V|$ vertices, $v_1, \ldots, v_n$, can be represented by an $n \times n$ matrix with:

$$a_{i,j} = \begin{cases} 
1 & \text{if there is an edge from } v_i \text{ to } v_j \\
0 & \text{otherwise}
\end{cases}$$

For undirected graphs, the matrix is symmetric.

Space: $O(n^2)$
A graph can be represented by $|V|$ lists, one per vertex. The list for vertex $u$ holds the vertices connected to the outgoing edges from $u$.

The lists can be implemented in different ways (vectors, linked lists, ...)

Space: $O(|E|)$

**Undirected graphs**: use bi-directional edges
// Loop visiting all edges of the graph
for (int i = 0; i < v.size() - 1; ++i) // for all vertices
    for (int j = v[i]; j < v[i+1]; ++j) // for all outgoing edges
        visitEdge(i, e[j]);

- Convenient representation for static graphs (not modified).
- An implementation with only one vector is also possible.
- For dynamic graphs, linked lists are preferable.
Dense and sparse graphs

- A graph with $|V|$ vertices could potentially have up to $|V|^2$ edges (all possible edges are possible).

- We say that a graph is **dense** when $|E|$ is close to $|V|^2$. We say that a graph is **sparse** when $|E|$ is close to $|V|$.

- How big can a graph be?

Dense graph

Sparse graph
December 2017: 50 billion web pages ($50 \times 10^9$).

Size of adjacency matrix: $25 \times 10^{20}$ elements. (not enough computer memory in the world to store it).

Good news: The web is very sparse. Each web page has about half a dozen hyperlinks to other web pages.
Adjacency matrix vs. adjacency list

• Space:
  – Adjacency matrix is $O(|V|^2)$
  – Adjacency list is $O(|E|)$

• Checking the presence of a particular edge $(u, v)$:
  – Adjacency matrix: constant time
  – Adjacency list: traverse $u$’s adjacency list

• Which one to use?
  – For dense graphs $\rightarrow$ adjacency matrix
  – For sparse graphs $\rightarrow$ adjacency list

• For many algorithms, traversing the adjacency list is not a problem, since they require to iterate through all neighbors of each vertex. For sparse graphs, the adjacency lists are usually short (can be traversed in constant time)
Reachability: exploring a maze

Which vertices of the graph are reachable from a given vertex?
To explore a labyrinth we need a ball of string and a piece of chalk:

- The chalk prevents looping, by marking the visited junctions.
- The string allows you to go back to the starting place and visit routes that were not previously explored.
How to simulate the string and the chalk with an algorithm?

- Chalk: a boolean variable for each vertex (visited).
- String: a stack
  - push vertex to unwind at each junction
  - pop to rewind and return to the previous junction

**Note:** the stack can be simulated with recursion.
function explore(G, v):
// Input: G = (V,E) is a graph
// Output: visited(u) is true for all the
//         nodes reachable from v

visited(v) = true
previsit(v)
for each edge (v,u) ∈ E:
    if not visited(u): explore(G,u)
postvisit(v)

Note: pre/postvisit functions are not required now.
function explore(G, v):
    visited(v) = true
    for each edge (v, u) ∈ E:
        if not visited(u): explore(G, u)

- All visited nodes are reachable because the algorithm only moves to neighbors and cannot jump to an unreachable region.

- Does it miss any reachable vertex? No. Proof by contradiction.
  - Assume that a vertex u is missed.
  - Take any path from v to u and identify the last vertex that was visited on that path (z). Let w be the following node on the same path. Contradiction: w should have also been visited.
Finding the nodes reachable from another node

```
function explore(G, v):
    visited(v) = true
    for each edge (v,u) ∈ E:
        if not visited(u): explore(G,u)
```

Dotted edges are ignored (*back edges*): they lead to previously visited vertices. The solid edges (*tree edges*) form a tree.
Depth-first search

```
function DFS(G):
    for all \( v \in V \):
        visited(v) = false
    for all \( v \in V \):
        if not visited(v): explore(G, v)
```

DFS traverses the entire graph.

**Complexity:**
- Each vertex is visited only once (thanks to the chalk marks)
- For each vertex:
  - A fixed amount of work (pre/postvisit)
  - All adjacent edges are scanned

**Running time** is \( O(|V| + |E|) \).
Difficult to improve: reading a graph already takes \( O(|V| + |E|) \).
The outer loop of DFS calls *explore* three times (for A, C and F).

Three trees are generated. They constitute a *forest*.
Connectivity

• An undirected graph is connected if there is a path between any pair of vertices.

• A disconnected graph has disjoint \textit{connected components}.

• Example: this graph has 3 connected components:

\{A, B, E, I, J\} \quad \{C, D, G, H, K, L\} \quad \{F\}.
Connected Components

```plaintext
function explore(G, v, cc):
// Input: G = (V, E) is a graph, cc is a CC number
// Output: ccnum[u] = cc for each vertex u in the same CC as v
    ccnum[v] = cc
    for each edge (v, u) ∈ E:
        if ccnum[u] ≠ cc: explore(G, u, cc)

function ConnComp(G):
// Input: G = (V, E) is a graph
// Output: Every vertex v has a CC number in ccnum[v]
    for all v ∈ V: ccnum[v] = 0; // Clean cc numbers
    cc = 1; // Identifier of the first CC
    for all v ∈ V:
        if ccnum[v] = 0: // A new CC starts
            explore(G, v, cc); cc = cc + 1;
```

- Performs a DFS traversal assigning a CC number to each vertex.
- The outer loop of `ConnComp` determines the number of CC’s.
- The variable `ccnum[v]` also plays the role of `visited[v]`.

Revisiting the explore function

```
function explore(G, v):
    visited(v) = true
    previsit(v)
    for each edge (v,u) ∈ E:
        if not visited(u):
            explore(G,u)
    postvisit(v)
```

Let us consider a global variable `clock` that can determine the occurrence times of previsit and postvisit.

```
function previsit(v):
    pre[v] = clock
    clock = clock + 1

function postvisit(v):
    post[v] = clock
    clock = clock + 1
```

Every node `v` will have an interval `(pre[v], post[v])` that will indicate the time the node was first visited (pre) and the time of departure from the exploration (post).

**Property:** Given two nodes `u` and `v`, the intervals `(pre[u], post[u])` and `(pre[v], post[v])` are either disjoint or one is contained within the other.

The pre/post interval of `u` is the lifetime of `explore(u)` in the stack (LIFO).
Example of pre/postvisit orderings

Recursion depth

Graphs © Dept. CS, UPC
DFS in directed graphs: types of edges

- **Tree edges**: those in the DFS forest.
- **Forward edges**: lead to a nonchild descendant in the DFS tree.
- **Back edges**: lead to an ancestor in the DFS tree.
- **Cross edges**: lead to neither descendant nor ancestor.
DFS in directed graphs: types of edges

pre/post ordering for \((u, v)\)

\[
\begin{array}{ccc}
  u & v & u \\
  v & u & v \\
  v & v & u \\
\end{array}
\]

- **Tree edges:** those in the DFS forest.
- **Forward edges:** lead to a nonchild descendant in the DFS tree.
- **Back edges:** lead to an ancestor in the DFS tree.
- **Cross edges:** lead to neither descendant nor ancestor.
A cycle is a circular path: \[ v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_0. \]

Examples:
\[ A \rightarrow F \rightarrow B \rightarrow A \]
\[ C \rightarrow D \rightarrow A \rightarrow C \]

**Property:** A directed graph has a cycle iff its DFS reveals a back edge.

**Proof:**
\[ \Leftarrow \] If \((u, v)\) is a back edge, there is a cycle with \((u, v)\) and the path from \(v\) to \(u\) in the search tree.
\[ \Rightarrow \] Let us consider a cycle \( v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_0 \). Let us assume that \( v_i \) is the first discovered vertex (lowest pre number). All the other \( v_j \) on the cycle are reachable from \( v_i \) and will be its descendants in the DFS tree. The edge \( v_{i-1} \rightarrow v_i \) leads from a vertex to its ancestor and is thus a back edge.
Getting dressed: DAG representation

A list of tasks that must be executed in a certain order (cannot be executed if it has cycles).

Legal task *linearizations* (or *topological sorts*):

Underwear Socks Trousers Shoes Watch Shirt Belt Tie Jacket

Watch Socks Shirt Tie Jacket Underwear Trousers Belt Shoes
A DAG is a directed graph without cycles.

DAGs are often used to represent causalities or temporal dependencies, e.g., task A must be completed before task C.

- Cyclic graphs cannot be linearized.

- All DAGs can be linearized. How?
  - Decreasing order of the post numbers.
  - The only edges \((u, v)\) with \(\text{post}[u] < \text{post}[v]\) are back edges (do not exist in DAGs).

- **Property:** In a DAG, every edge leads to a vertex with a lower post number.

- **Property:** Every DAG has at least one source and at least one sink.
  (source: highest post number, sink: lowest post number).
Topological sort

**function** explore($G, v$):

- visited($v$) = true
- previsit($v$)
- for each edge $(v, u) \in E$:
  - if not visited($u$):
    - explore($G, u$)
- postvisit($v$)

**Initially:** TSort = $\emptyset$

**function** postvisit($v$):

- TSort.push_front($v$)

// After DFS, TSort contains
// a topological sort

Another algorithm:

- Find a source vertex, write it, and delete it (mark) from the graph.
- Repeat until the graph is empty.

It can be executed in linear time. How?
This graph is connected (undirected view), but there is no path between any pair of nodes.

For example, there is no path $K \to \cdots \to C$ or $E \to \cdots \to A$.

The graph is not *strongly connected*.

Two nodes $u$ and $v$ of a directed graph are connected if there is a path from $u$ to $v$ and a path from $v$ to $u$.

The *connected* relation is an equivalence relation and partitions $V$ into disjoint sets of *strongly connected components*.

**Strongly Connected Components**

\[
\begin{align*}
\{A\} \\
\{B, E\} \\
\{C, F\} \\
\{D\} \\
\{G, H, I, J, K, L\}
\end{align*}
\]
Every directed graph can be represented by a **meta-graph**, where each meta-node represents a strongly connected component.

**Property:** every directed graph is a DAG of its strongly connected components.

A directed graph can be seen as a 2-level structure. At the top we have a DAG of SCCs. At the bottom we have the details of the SCCs.
Properties of DFS and SCCs

- **Property:** If the *explore* function starts at $u$, it will terminate when all vertices reachable from $u$ have been visited.
  - If we start from a vertex in a sink SCC, it will retrieve exactly that component.
  - If we start from a non-sink SCC, it will retrieve the vertices of several components.

- **Examples:**
  - If we start at $K$ it will retrieve the component $\{G, H, I, J, K, L\}$.
  - If we start at $B$ it will retrieve all vertices except $A$. 
Properties of DFS and SCCs

- **Intuition for the algorithm:**
  - Find a vertex located in a sink SCC
  - Extract the SCC

- **To be solved:**
  - How to find a vertex in a sink SCC?
  - What to do after extracting the SCC?

- **Property:** The vertex with the highest DFS post number lies in a source SCC.

- **Property:** If $C$ and $C'$ are SCCs and there is an edge $C \rightarrow C'$, then the highest post number in $C$ is bigger than the highest post number in $C'$. 

Graphs © Dept. CS, UPC
Reverse graph \((G^R)\)
SCC algorithm

```java
function SCC(G):
// Input: G(V,E) a directed graph
// Output: each vertex v has an SCC number in ccnum[v]
G^R = reverse(G)
DFS(G^R) // calculates post numbers
sort V // decreasing order of post number
ConnComp(G)
```

Runtime complexity:
- DFS and ConnComp run in linear time O(|V| + |E|).
- Can we reverse G in linear time?
- Can we sort V by post number in linear time?
Reversing $G$ in linear time

```plaintext
function SCC(G):
    // Input: $G(V,E)$ a directed graph
    // Output: each vertex $v$ has an SCC number in ccnum[$v$]
    $G^R$ = reverse($G$)
    DFS($G^R$) // calculates post numbers
    sort $V$ // decreasing order of post number
    ConnComp($G$)

function reverse($G$)
    // Input: $G(V,E)$ graph represented by an adjacency list
    //        edges[$v$] for each vertex $v$.
    // Output: $G(V,E^R)$ the reversed graph of $G$, with the
    //        adjacency list edgesR[$v$].

    for each $u \in V$:
        for each $v \in \text{edges}[u]$:
            edgesR[$v$].insert($u$)
    return ($V$, edgesR)
```

function SCC(G):
    // Input: G(V,E) a directed graph
    // Output: each vertex v has an SCC number in ccnum[v]
    \( G^R = \text{reverse}(G) \)
    DFS(\( G^R \)) // calculates post numbers
    sort V // decreasing order of post number
    ConnComp(G)

Use the explore function for topological sort:
- Each time a vertex is post-visited, it is inserted at the top of the list.
- The list is ordered by decreasing order of post number.
- It is executed in linear time
Assume the initial order:
F, A, B, C, D, E, J, G, H, I, K, L

**Vertex:**

<table>
<thead>
<tr>
<th>Vertex</th>
<th>J</th>
<th>L</th>
<th>K</th>
<th>H</th>
<th>G</th>
<th>I</th>
<th>D</th>
<th>F</th>
<th>C</th>
<th>B</th>
<th>E</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>17</td>
<td>16</td>
<td>12</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>
Crawling the Web

• Crawling the Web is done using depth-first search strategies.

• The graph is unknown and no recursion is used. A stack is used instead containing the nodes that have already been visited.

• The stack is not exactly a LIFO. Only the most “interesting” nodes are kept (e.g., page rank).

• Crawling is done in parallel (many computers at the same time) but using a central stack.

• How do we know that a page has already been visited? Hashing.
Summary

• Big data is often organized in big graphs (objects and relations between objects)

• Big graphs are usually sparse. Adjacency lists is the most common data structure to represent graphs.

• Connectivity can be analyzed in linear time using depth-first search.