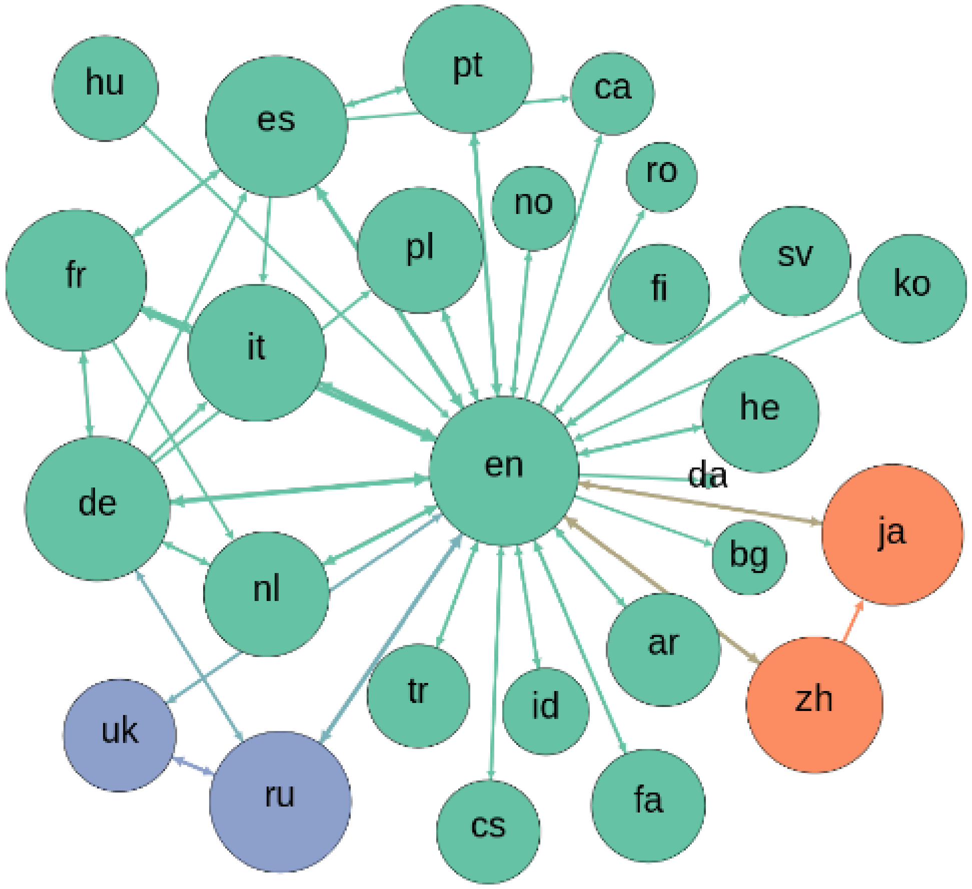
Graphs: Connectivity

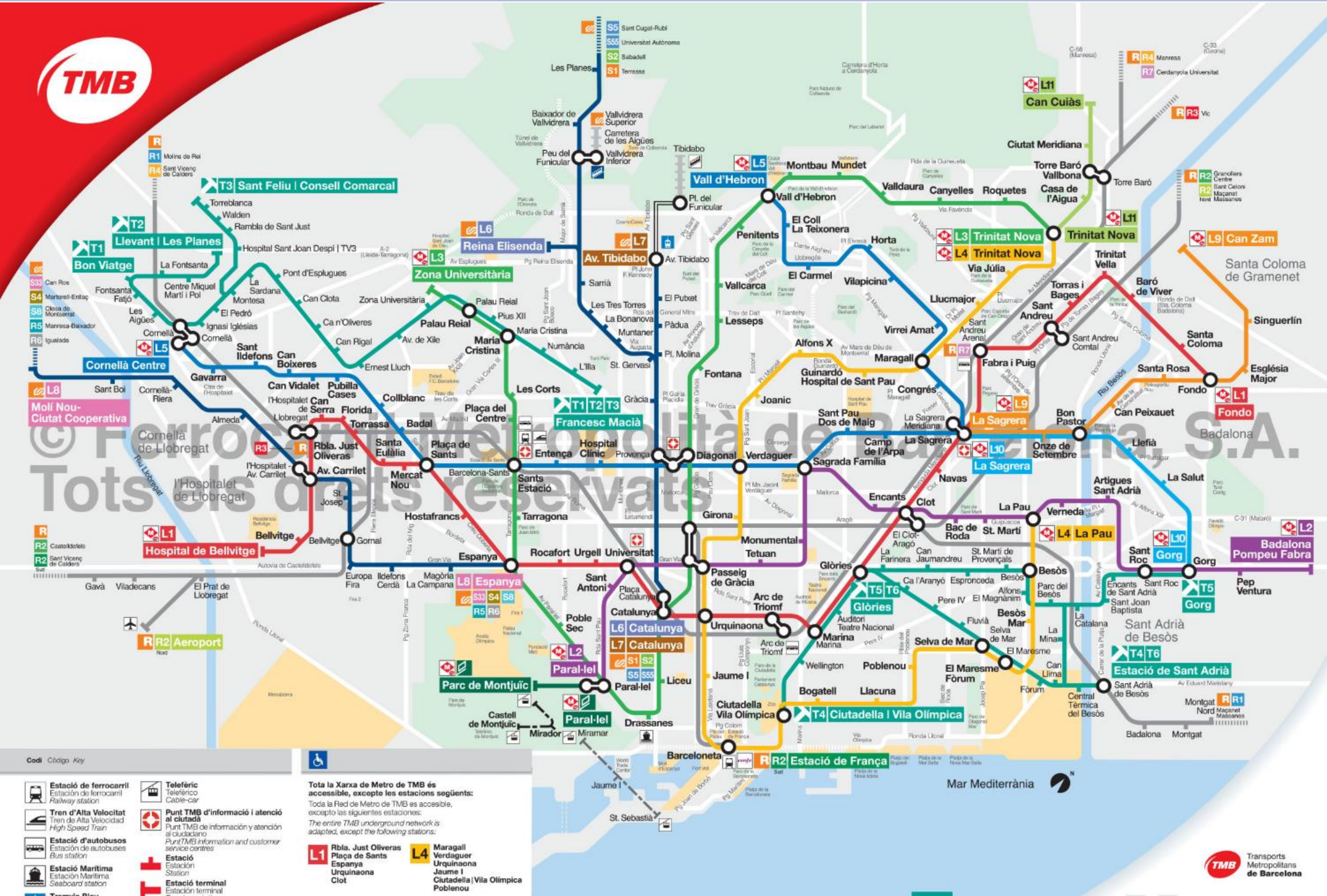
Jordi Cortadella and Jordi Petit
Department of Computer Science

A graph



Source: Wikipedia
The network graph formed by Wikipedia editors (edges) contributing to different Wikipedia language versions (vertices) during one month in summer 2013

Transportation systems



Codi Código Key

Estació de ferrocarril / Estación de ferrocarril / Railway station

Tren d'Alta Velocitat / Tren de Alta Velocidad / High Speed Train

Estació d'autobusos / Estación de autobuses / Bus station

Estació Marítima / Estación Marítima / Seaboard station

Telefèric / Teleférico / Cable-car

Punt TMB d'informació i atenció al ciutadà / Punt TMB de información y atención al ciudadano / Punt TMB information and customer service centres

Estació / Estación / Station

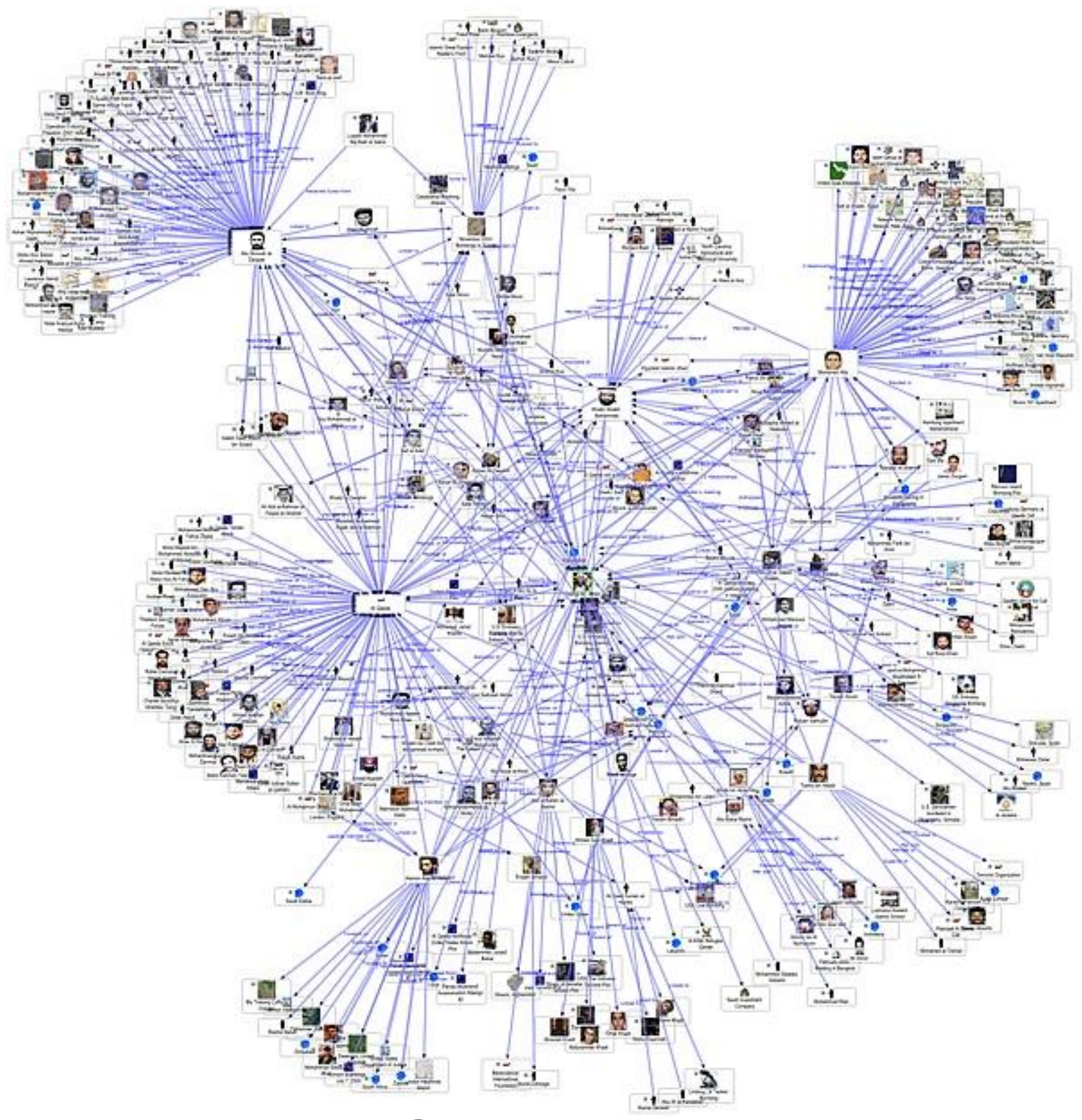
Estació terminal / Estación terminal

Tota la Xarxa de Metro de TMB és accessible, excepte les estacions següents:
Toda la Red de Metro de TMB es accesible, excepto las siguientes estaciones:
The entire TMB underground network is adapted, except the following stations:

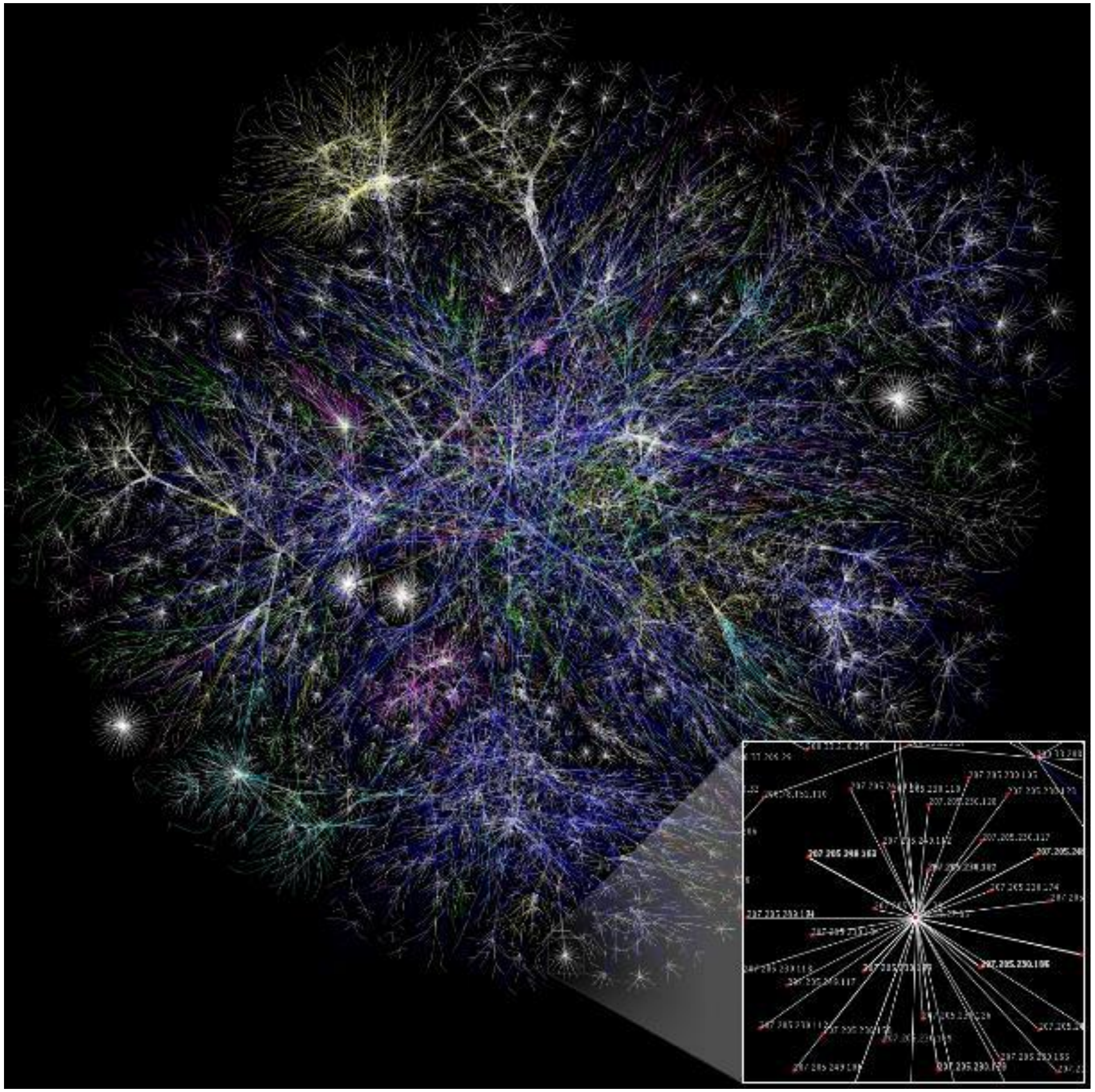
L1 Rbla. Just Oliveras, Plaça de Sants, Espanya, Urquinaona, Clot

L4 Maragall, Verdaguer, Urquinaona, Jaume I, Ciutadella | Vila Olímpica, Poblenou

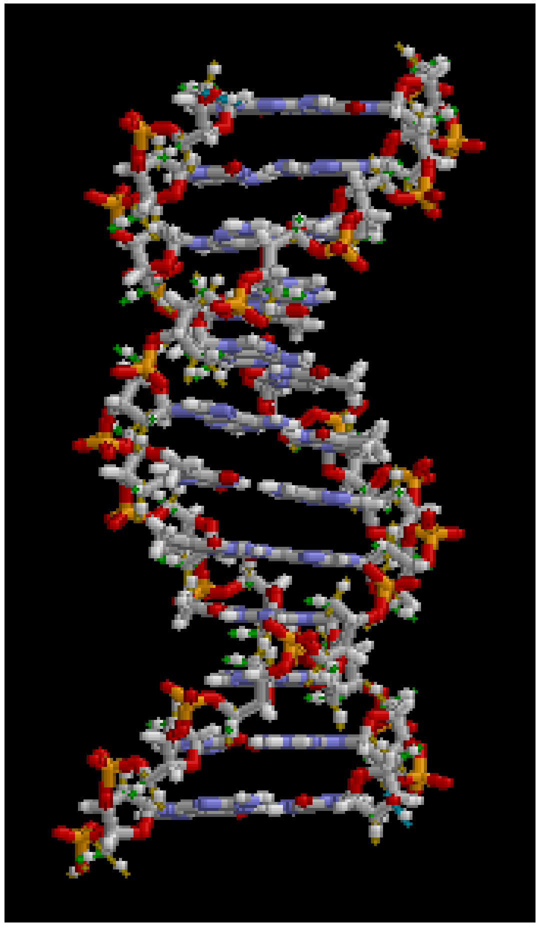
Social networks

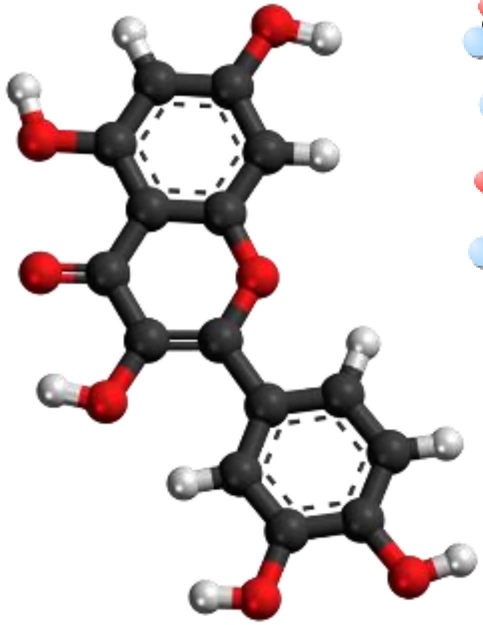


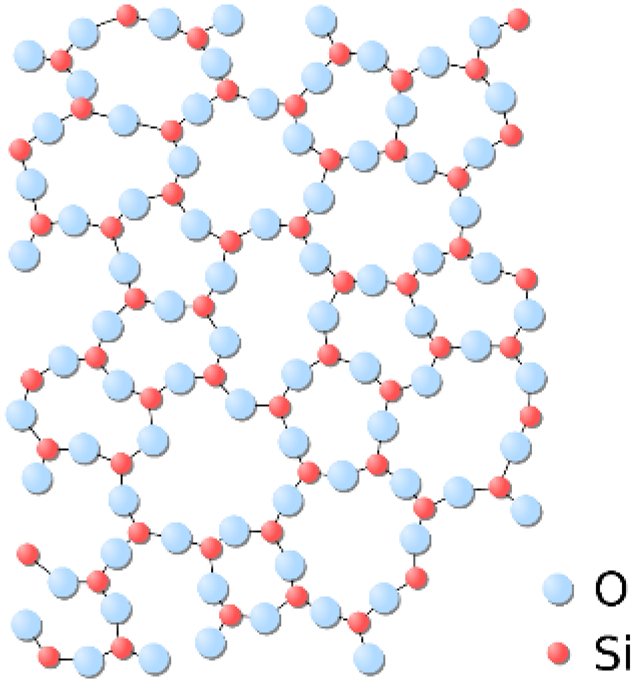
World Wide Web



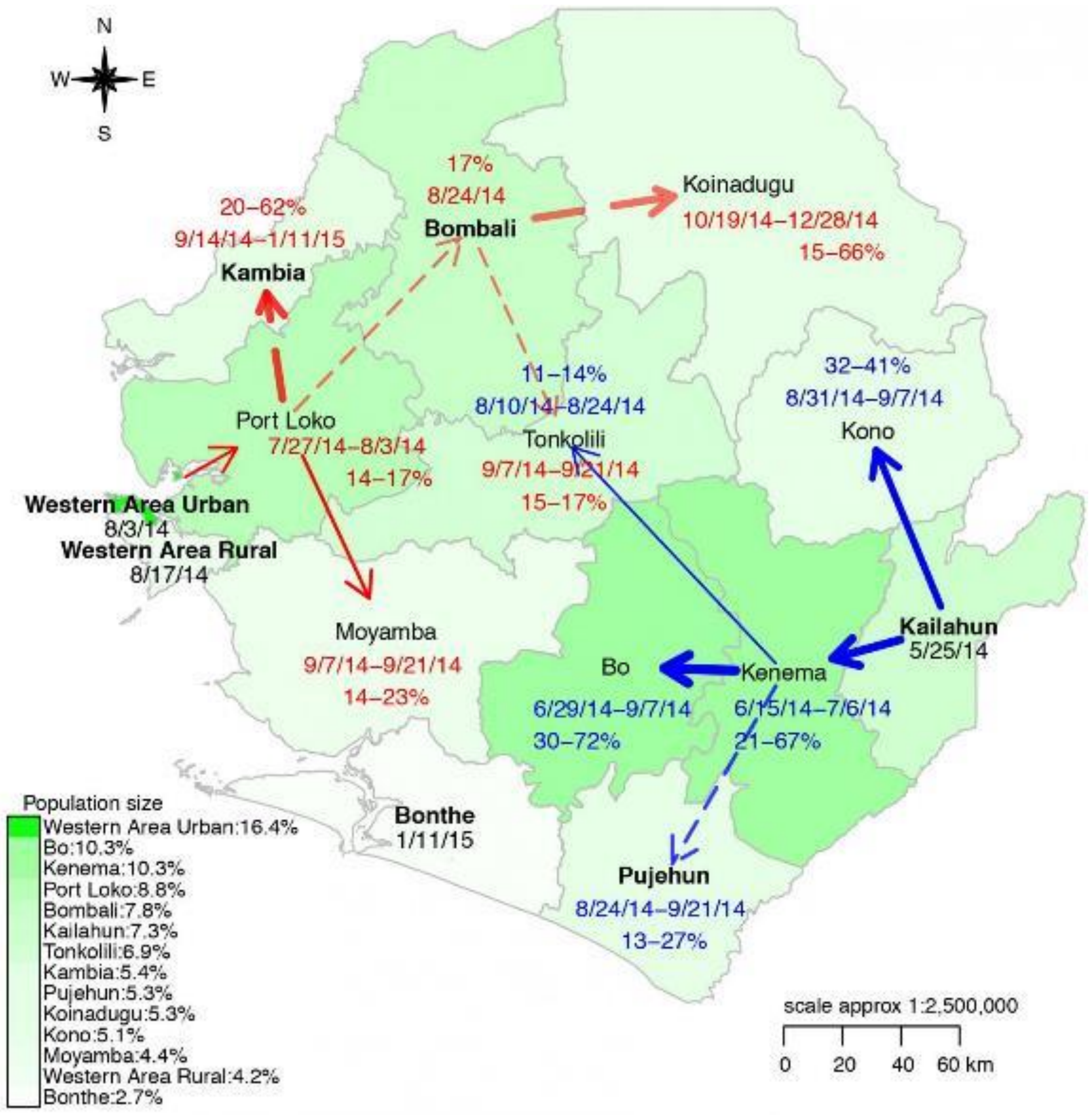
Biology





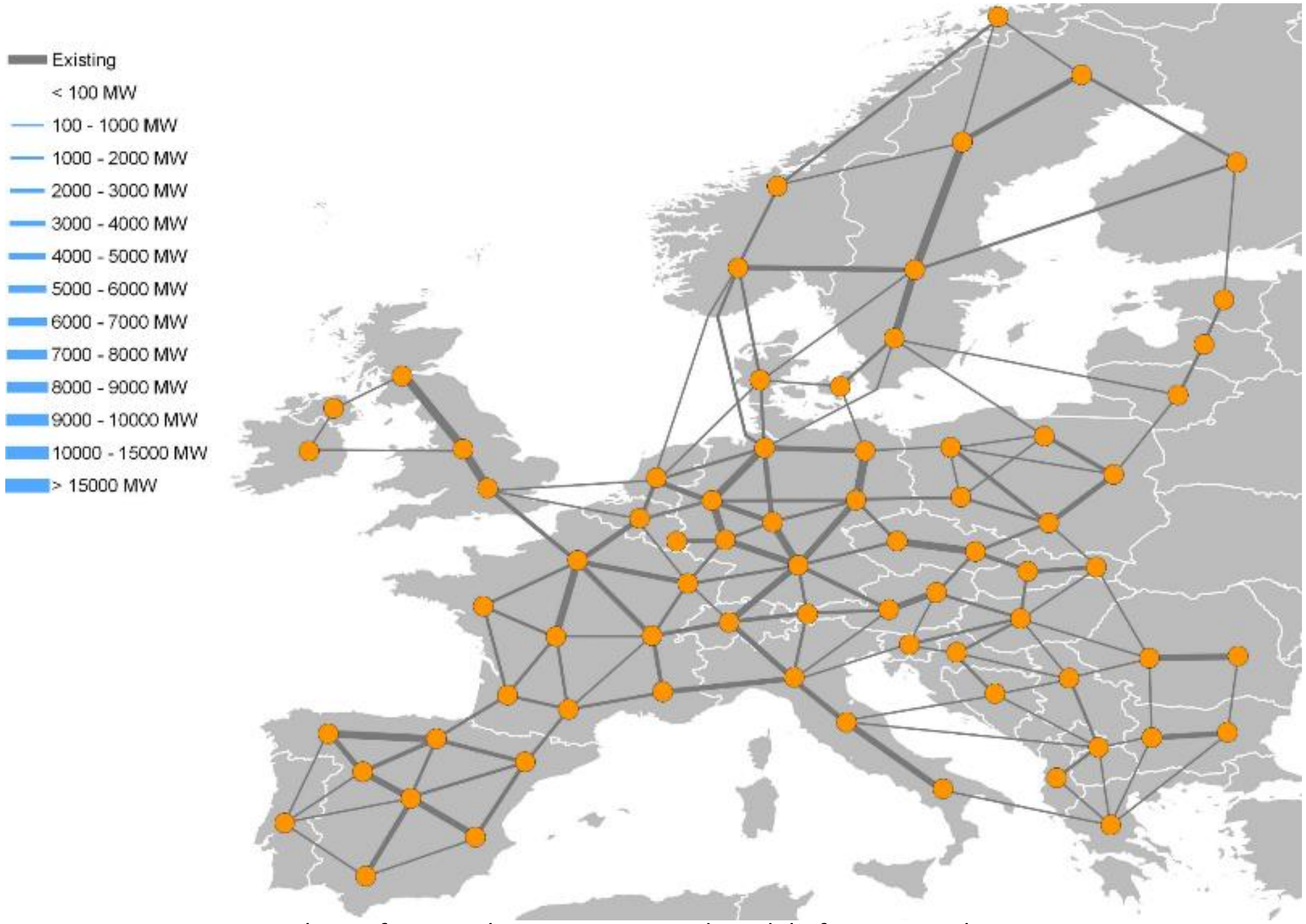


Disease transmission network



https://medicalxpress.com/news/2015-11-reveals-deadly-route-ebola-outbreak.html

Transmission of renewable energy



Topology of regional transmission grid model of continental Europe in 2020
https://blogs.dnvgl.com/energy/integration-of-renewable-energy-in-europe

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?

Credits

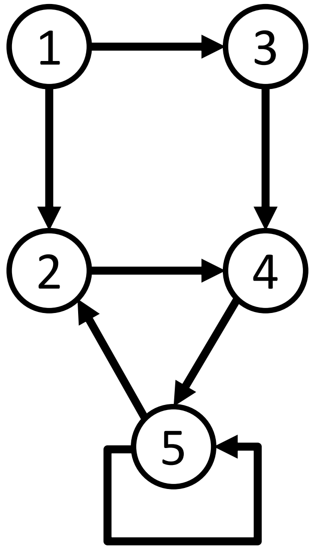
A significant part of the material used in this chapter has been inspired by the book:

Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani, *Algorithms*, McGraw-Hill, 2008. [DPV2008]

(several examples, figures and exercises are taken from the book)

Graph definition

A graph is specified by a set of vertices (or nodes) V and a set of edges E.



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

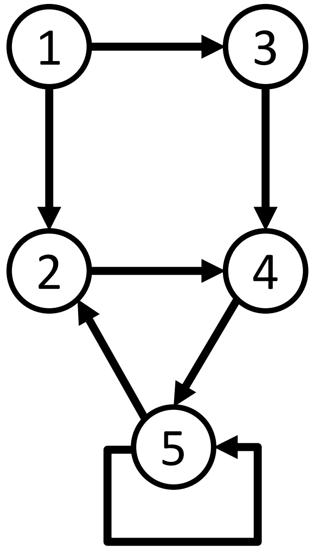
$$E = \{(1,2),(1,3),(2,4),(3,4),\\ (4,5),(5,2),(5,5)\}$$

Graphs can be directed or undirected.
Undirected graphs have a symmetric relation.

Graph representation: adjacency matrix

A graph with $n = |V|$ vertices, $v_1, \cdots, v_n$, can be represented by an $n \times n$ matrix with:

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



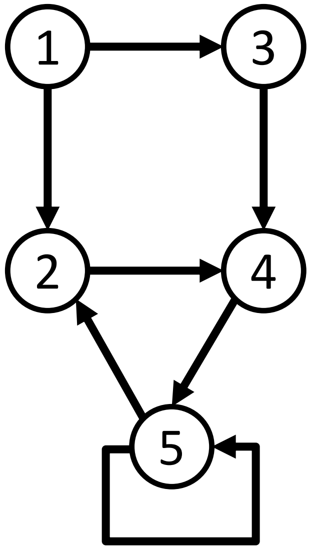
$$a = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

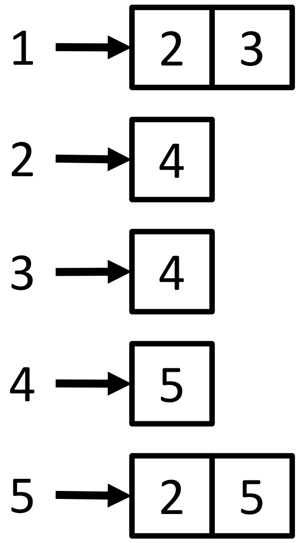
Space: $O(n^2)$

For undirected graphs, the matrix is symmetric.

Graph representation: adjacency list

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.





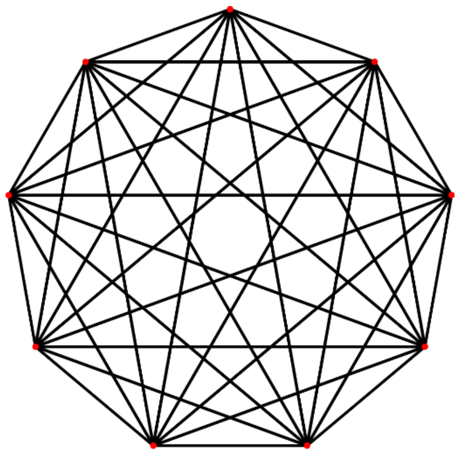
The lists can be implemented in different ways (vectors, linked lists, …)

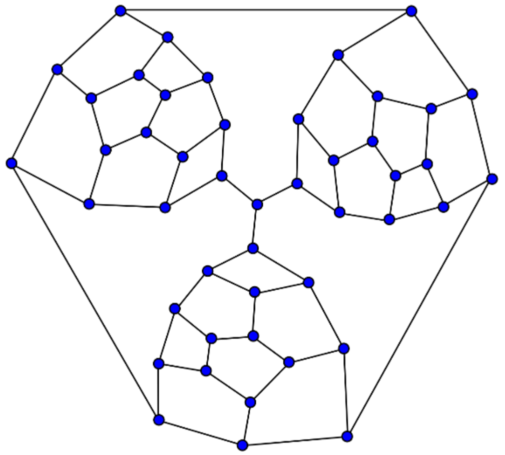
Space: $\mathrm{O}(|E|)$

Undirected graphs: use bi-directional edges

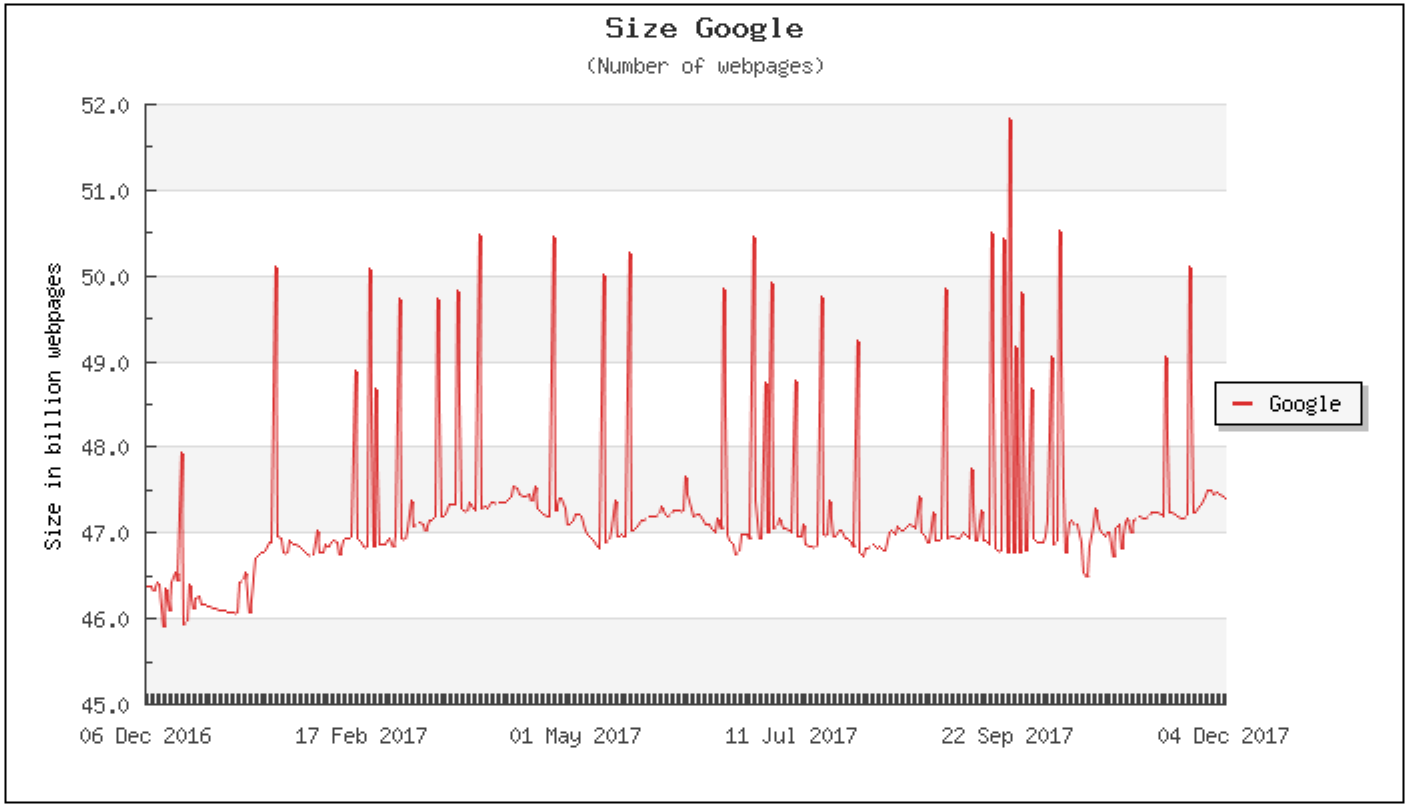
Dense and sparse graphs

- 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

Size of the World Wide Web



www.worldwidewebsize.com

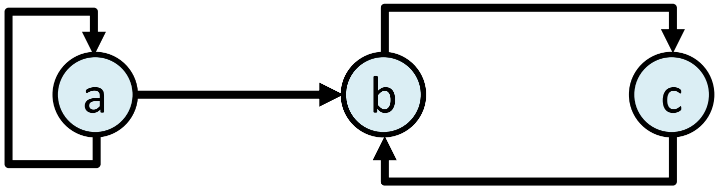
- December 2017: 50 billion web pages (50×10^9).
- Size of adjacency matrix: 25×10^{20} elements.
  (not enough computer memory in the world to store it).
- Good news: The web is very sparse. Each web page has about half a dozen hyperlinks to other web pages.

Adjacency matrix vs. adjacency list

- Space:
  - Adjacency matrix is $\mathrm{O}(|V|^2)$
  - Adjacency list is $\mathrm{O}(|E|)$
- 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 → adjacency matrix
  - For sparse graphs → 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)

Graph usage: example

```
// Declaration of a graph that stores
// a string (name) for each vertex
Graph<string> G;

// Create the vertices
int a = G.addVertex("a");
int b = G.addVertex("b");
int c = G.addVertex("c");

// Create the edges
G.addEdge(a,a);
G.addEdge(a,b);
G.addEdge(b,c);
G.addEdge(c,b);

// Print all edges of the graph
for (int src = 0; src < G.numVertices(); ++src) { // all vertices
  for (auto dst: G.succ(src)) {  // all successors of src
    cout << G.info(src) << " -> " << G.info(dst) << endl;
  }
}
```



	info	succ	pred
0	"a"	{0,1}	{0}
1	"b"	{2}	{0,2}
2	"c"	{1}	{1}

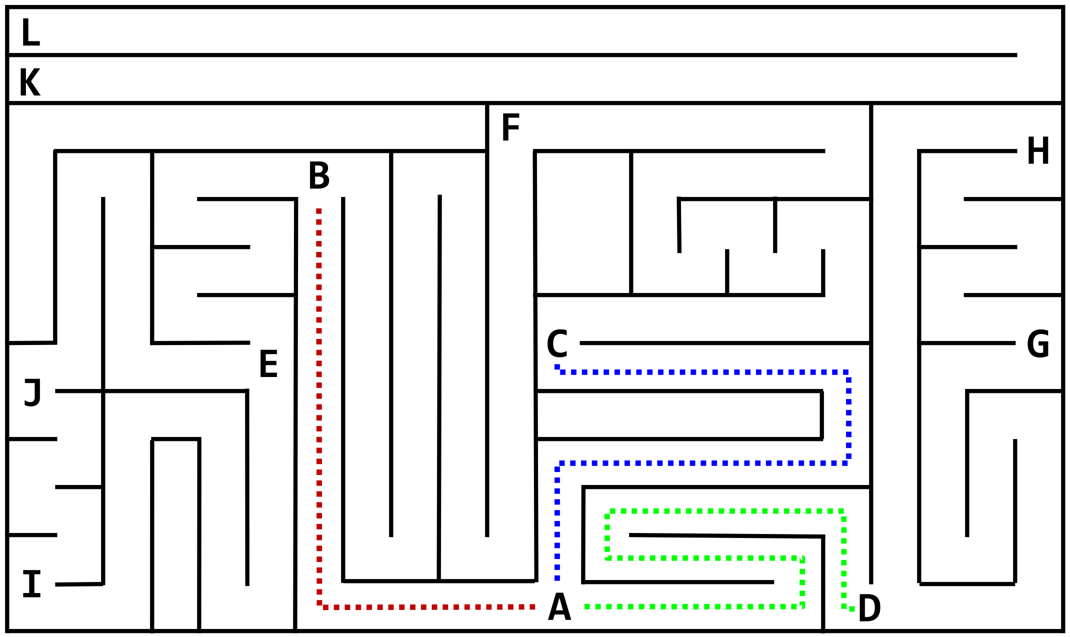
Graph implementation

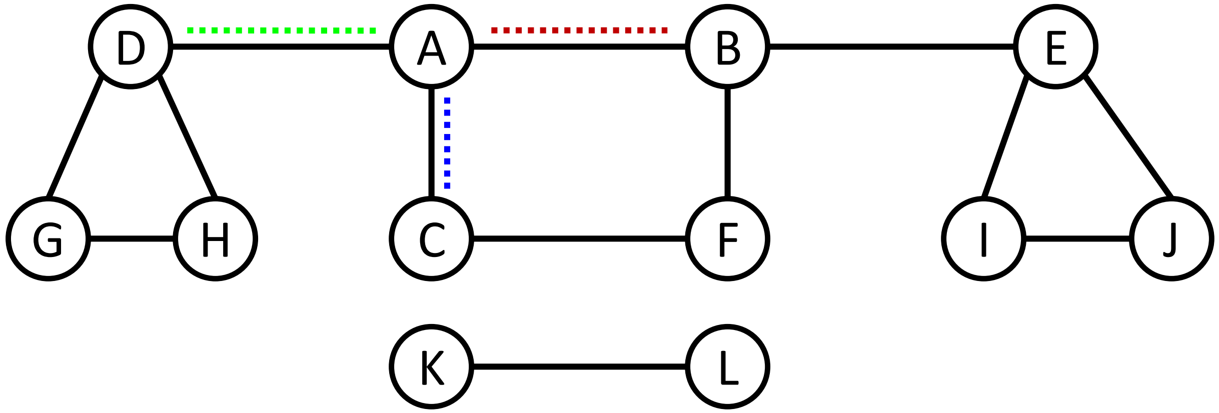
```cpp
template<typename vertexType>
class Graph {
private:
  struct Vertex {
    vertexType info;  // Information of the vertex
    vector<int> succ; // List of successors
    vector<int> pred; // List of predecessors
  };

  vector<Vertex> vertices; // List of vertices

public:
  /** Constructor */
  Graph() {}

  /** Adds a vertex with information associated to the vertex.
      Returns the index of the vertex */
  int addVertex(const vertexType& info) {
    vertices.push_back(Vertex{info});
    return vertices.size() - 1;
  }
```

Graph implementation

```
    /** Adds an edge src → dst */
    void addEdge(int src, int dst) {
      vertices[src].succ.push_back(dst);
      vertices[dst].pred.push_back(src);
    }

    /** Returns the number of vertices of the graph */
    int numVertices() const {
      return vertices.size();
    }

    /** Returns the information associated to vertex v */
    const vertexType& info(int v) const {
      return vertices[v].info;
    }

    /** Returns the list of successors of vertex v */
    const vector<int>& succ(int v) const {
      return vertices[v].succ;
    }

    /** Returns the list of predecessors of vertex v */
    const vector<int>& pred(int v) const {
      return vertices[v].pred;
    }
  };
```

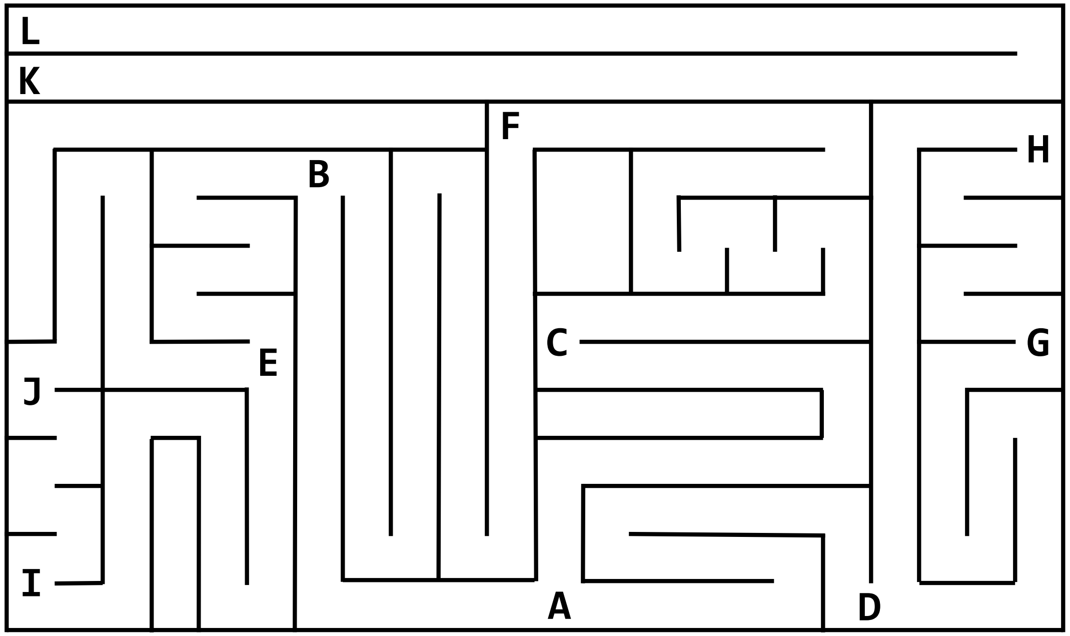
Reachability: exploring a maze





Which vertices of the graph are reachable from a given vertex?

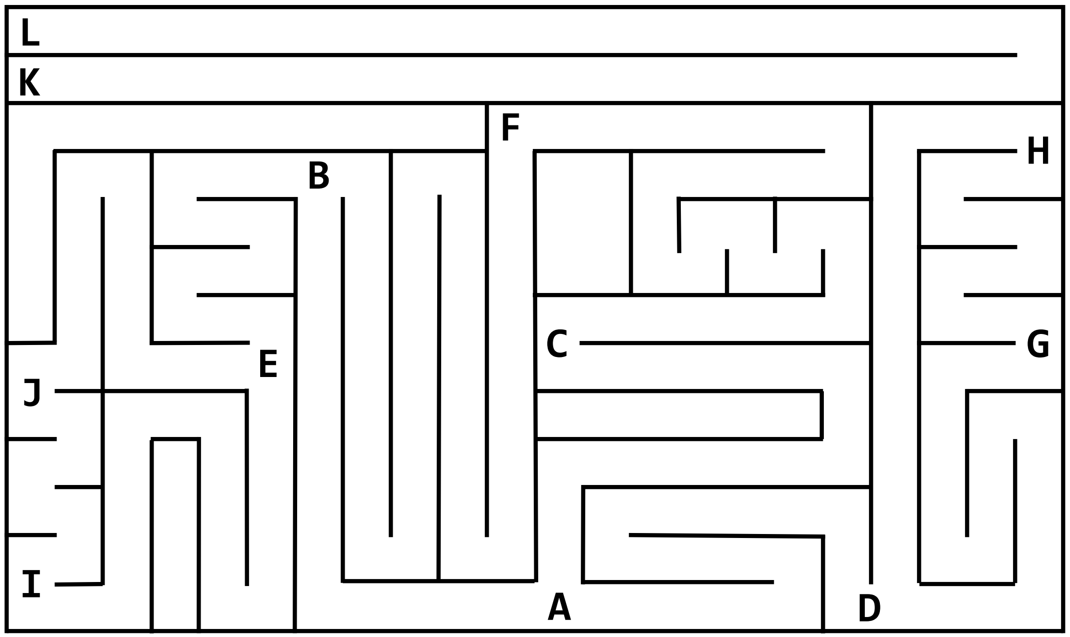
Reachability: exploring a maze



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.

Reachability: exploring a maze



How to simulate the string and the chalk with an algorithm?

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

Note: the stack can be simulated with recursion.

Finding the nodes reachable from another node

```
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 $(v, u) \in E$:
    if not visited($u$): explore($G, u$)
  postvisit($v$)
```

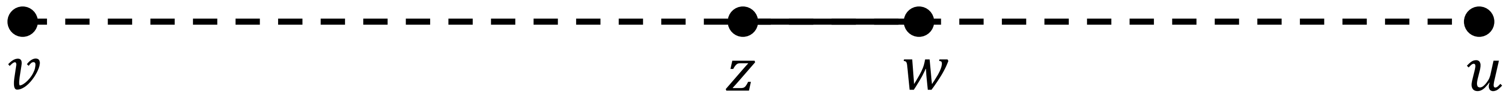
Notes:

- Initially, visited(v) is assumed to be *false* for every $v \in V$.
- pre/postvisit functions are not required now.

Finding the nodes reachable from another node

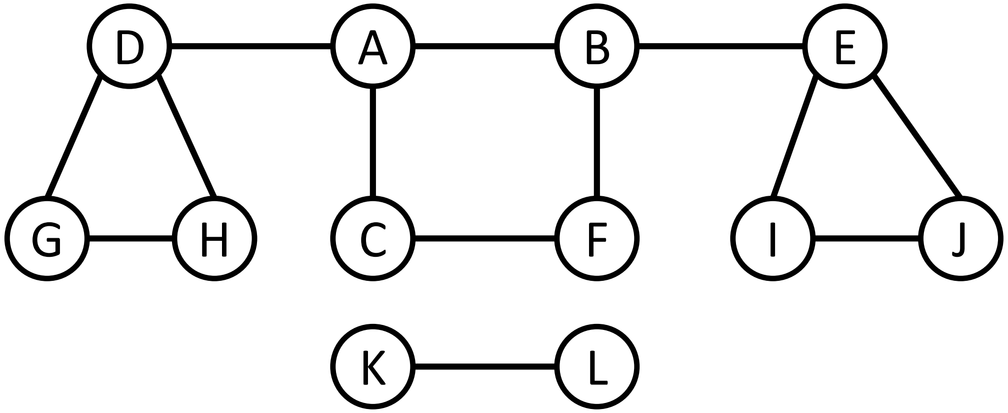
```
function explore($G$, $v$):
  visited($v$) = true
  for each edge $(v,u) \in E$:
    if not visited($u$): explore($G,u$)
```

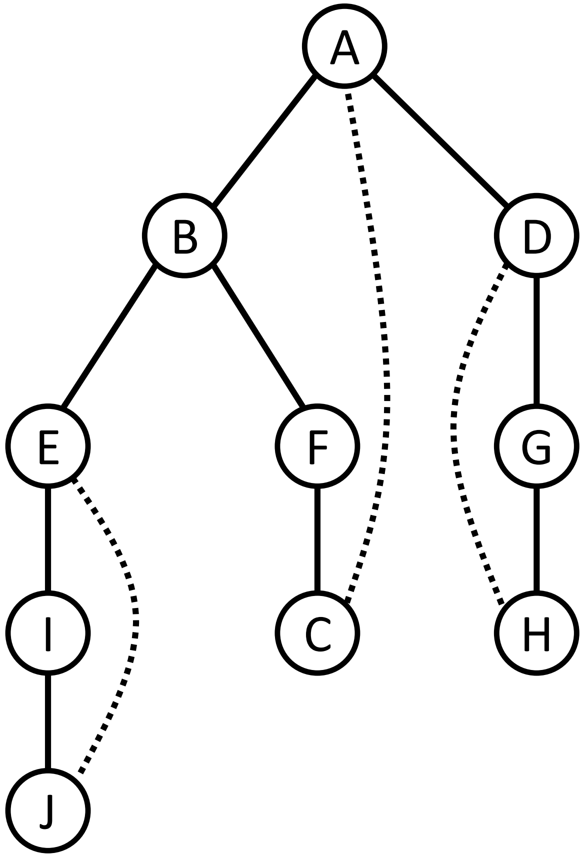
- All visited nodes are reachable because the algorithm only moves to neighbors and cannot jump to an unreachable region.
- Does it miss any reachable vertex? No. Proof by contradiction.
  - Assume that a vertex u is missed.
  - Take any path from v to u and identify the last vertex that was visited 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 $(v,u) \in E$:
    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.

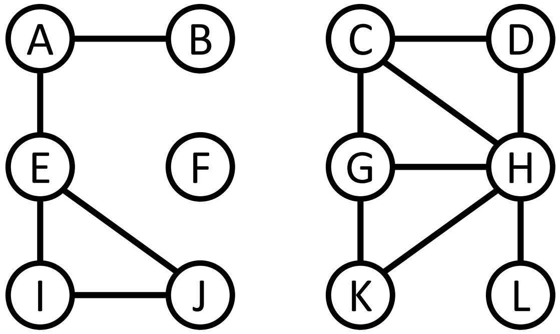
Complexity:

- 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

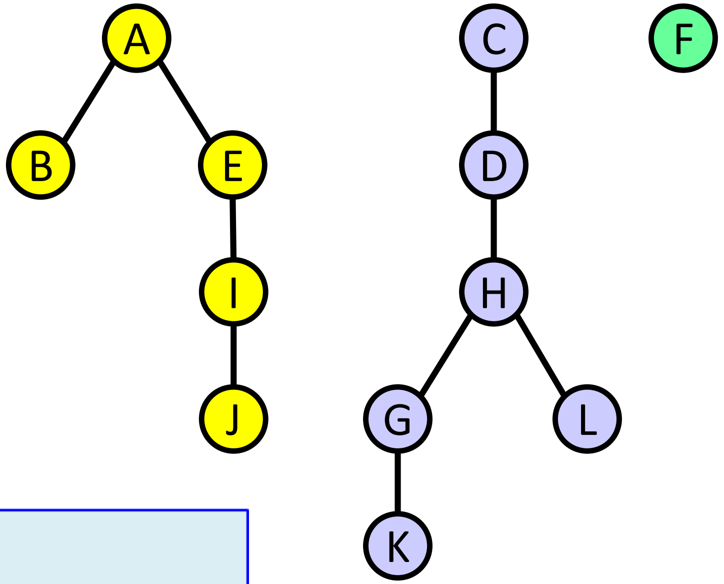
Running time is $O(|V| + |E|)$.
Difficult to improve: reading a graph already takes $O(|V| + |E|)$.

DFS example

Graph



DFS forest

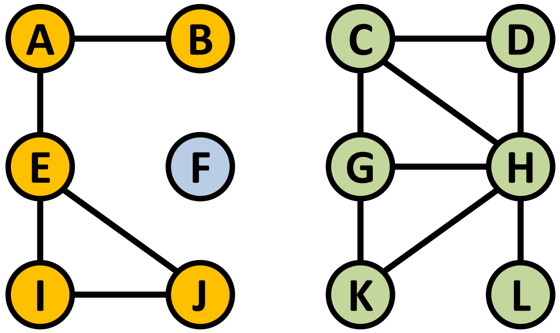


```
function DFS($G$):
  for all $v \in V$:
    visited($v$) = false
  for all $v \in V$:
    if not visited($v$): explore($G$,$v$)
```

- 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 *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

```
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 $(v,u) \in E$:
    if ccnum[$u$] == 0: 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 $v \in V$: ccnum[$v$] = 0; // Clean cc numbers
  cc = 1; // Identifier of the first CC
  for all $v \in V$:
    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 $(v, u) \in E$:
    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.

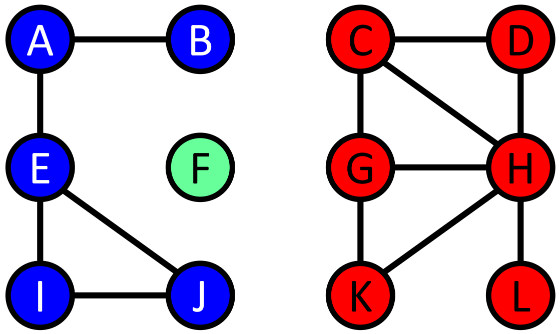
```
function previsit($v$):
  pre[$v$] = clock
  clock = clock + 1

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

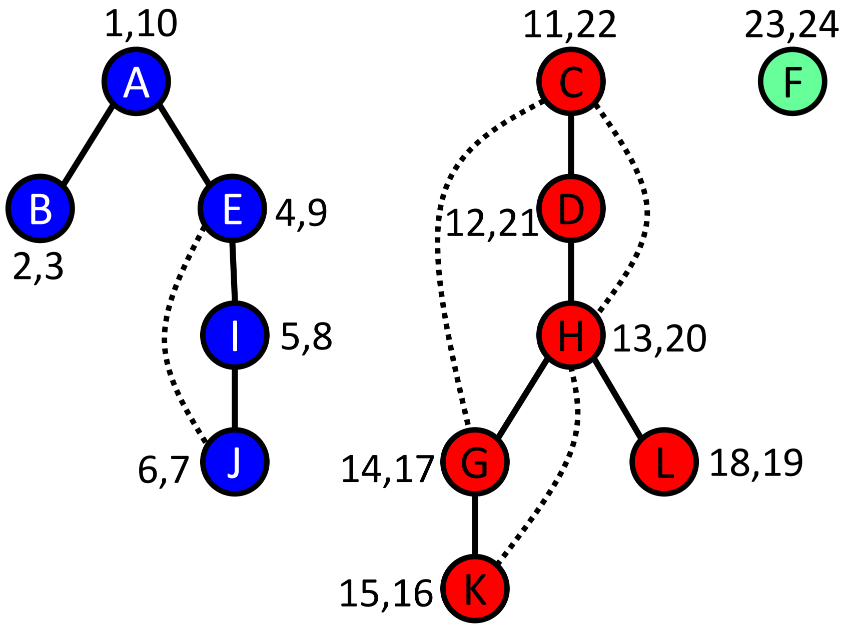
Every node v will have an interval (pre[v], post[v]) that will indicate the time the node was first visited (pre) and the time of departure from the exploration (post).

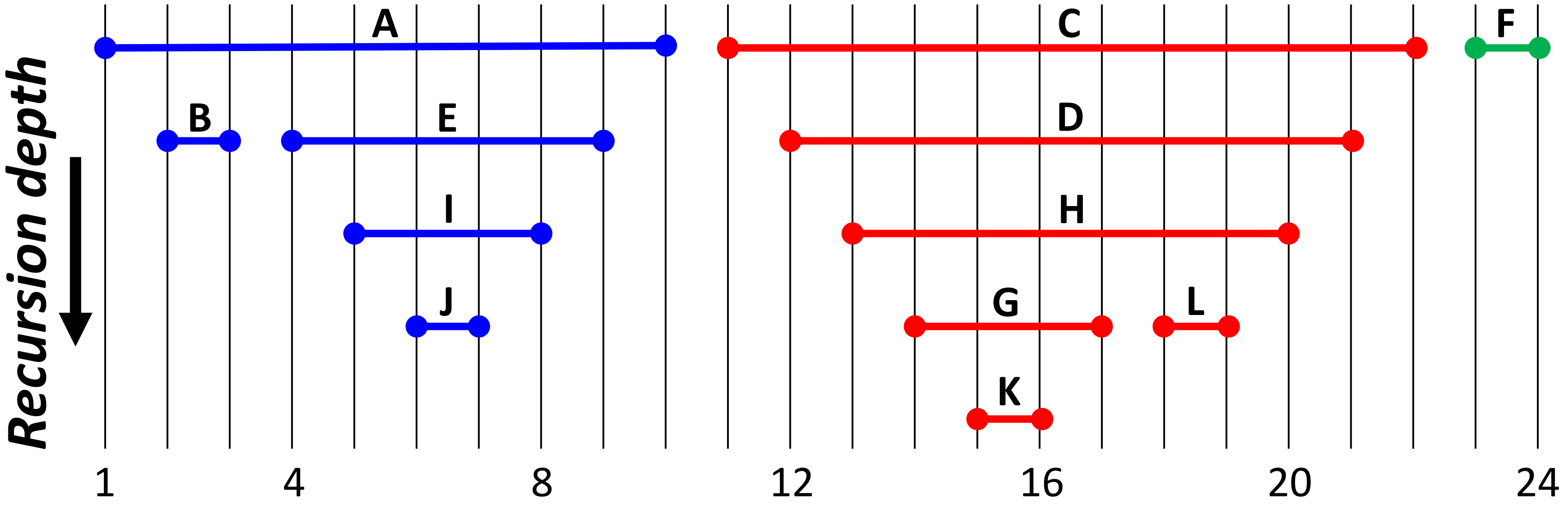
Property: Given two nodes u and v, the intervals (pre[u], post[u]) and (pre[v], post[v]) are either disjoint or one is contained within the other.

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

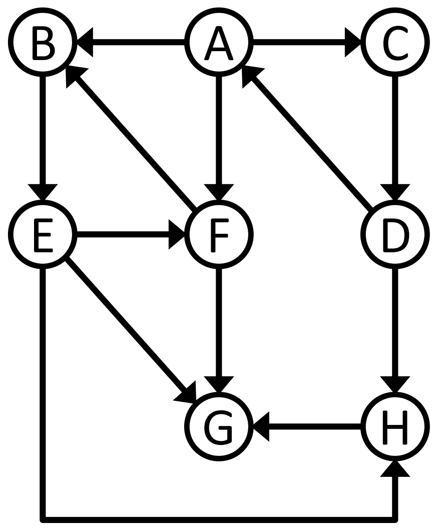
Example of pre/postvisit orderings

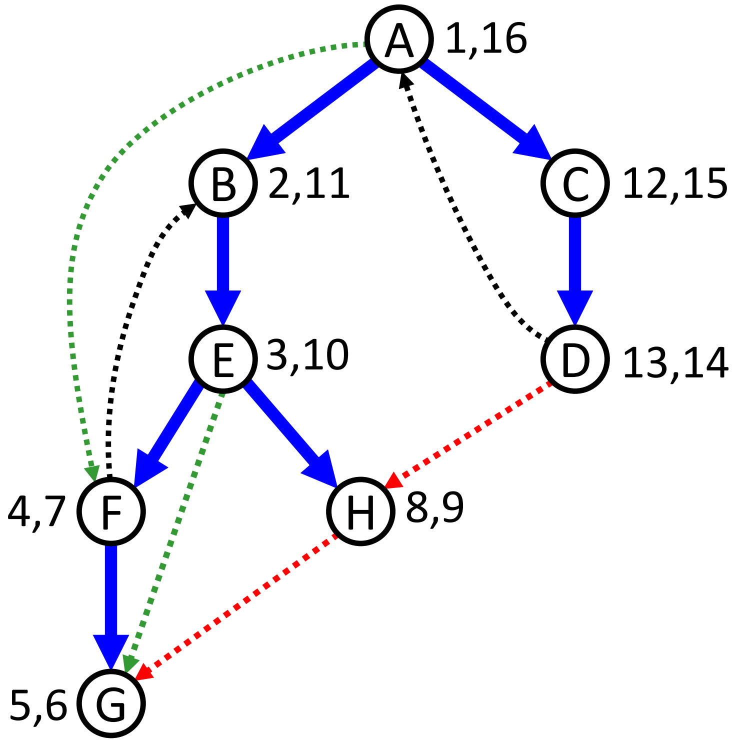


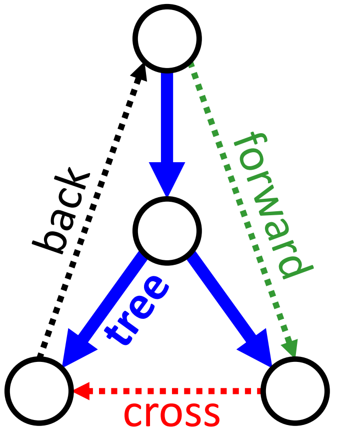




DFS in directed graphs: types of edges



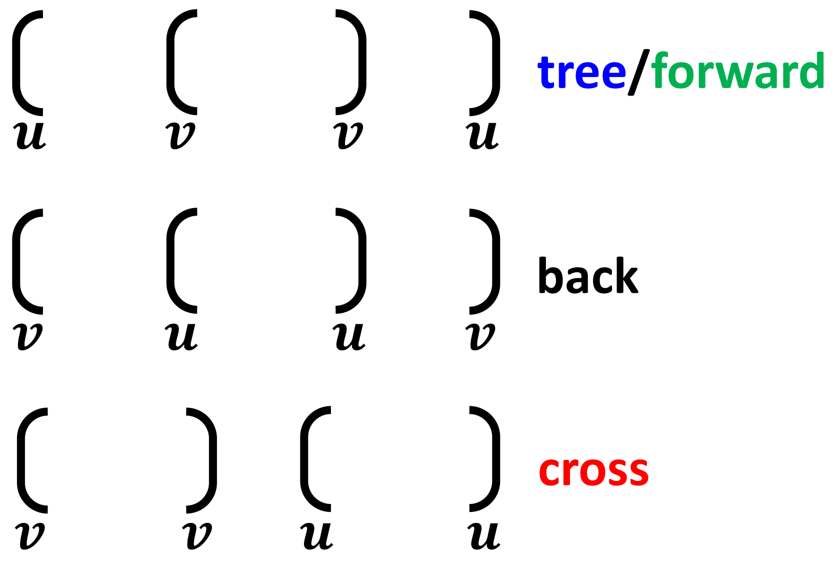


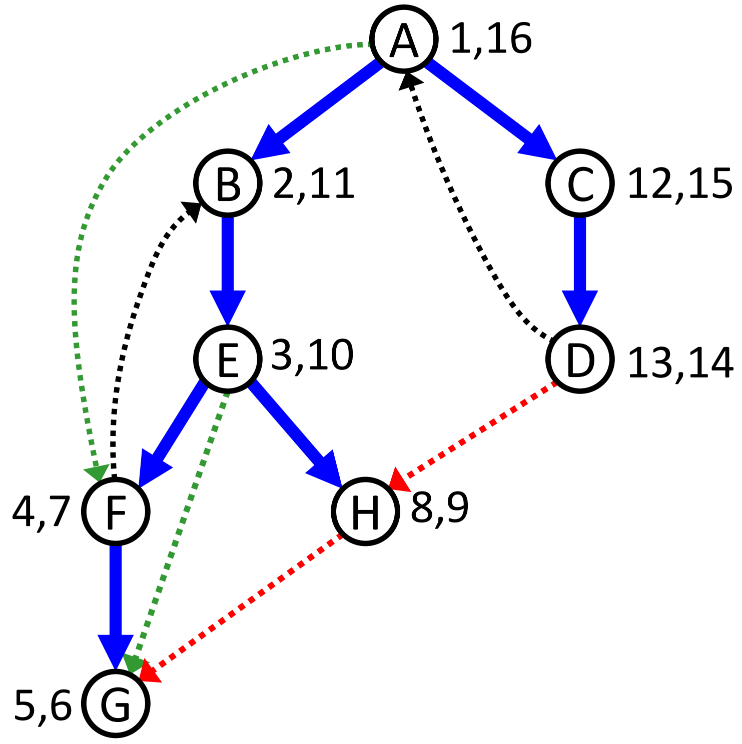


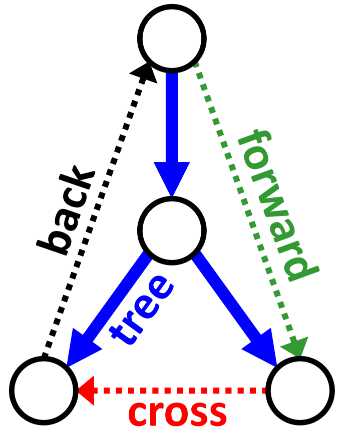
- **Tree edges:** those in the DFS forest.
- **Forward edges:** lead to a nonchild descendant in the DFS tree.
- **Back edges:** lead to an ancestor in the DFS tree.
- **Cross edges:** lead to neither descendant nor ancestor.

DFS in directed graphs: types of edges

pre/post ordering for (u, v)

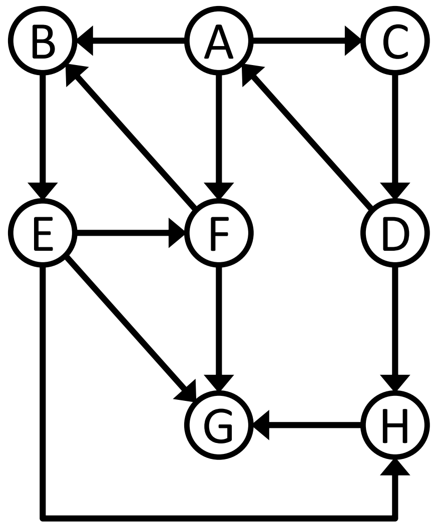






- **Tree edges:** those in the DFS forest.
- **Forward edges:** lead to a nonchild descendant in the DFS tree.
- **Back edges:** lead to an ancestor in the DFS tree.
- **Cross edges:** lead to neither descendant nor ancestor.

Cycles in graphs



A ***cycle*** is a circular path:

$$v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0.$$

Examples:

$$B \to E \to F \to B$$
$$C \to D \to A \to C$$

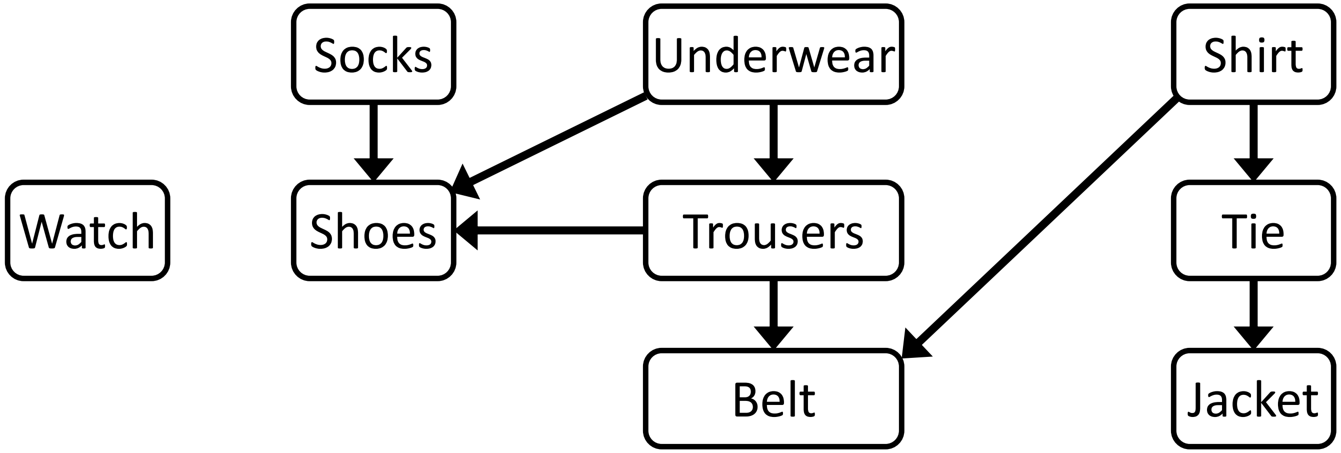
Property: A directed graph has a cycle *iff* its DFS reveals a back edge.

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 $v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0$. 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 $v_{i-1} \to v_i$ leads from a vertex to its ancestor and is thus a back edge.

Getting dressed: DAG representation

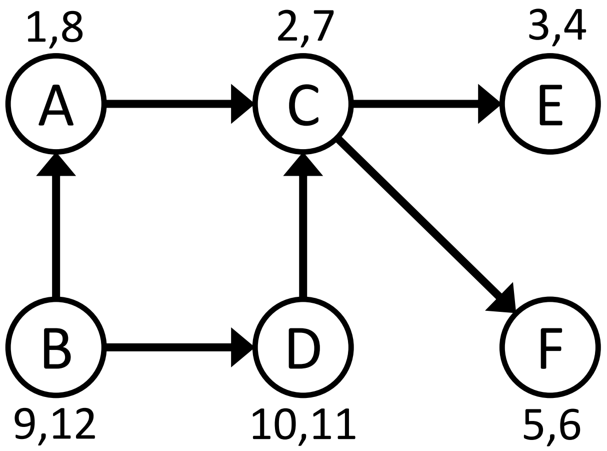


A list of tasks that must be executed in a certain order (cannot be executed if it has cycles).

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

Underwear	Socks	Trousers	Shoes	Watch	Shirt	Belt	Tie	Jacket
Watch	Socks	Shirt	Tie	Jacket	Underwear	Trousers	Belt	Shoes

Directed Acyclic Graphs (DAGs)



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 post$[u]$ < 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($v$)
```

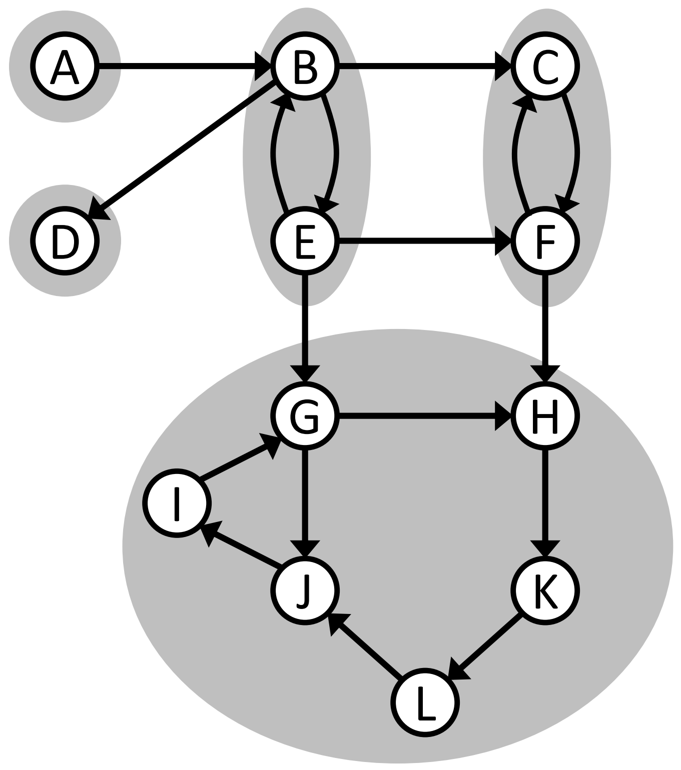
```
Initially: TSort = $\emptyset$

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?

Strongly Connected Components



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*.

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 $E \to \cdots \to A$.

Strongly Connected Components

$\{A\}$

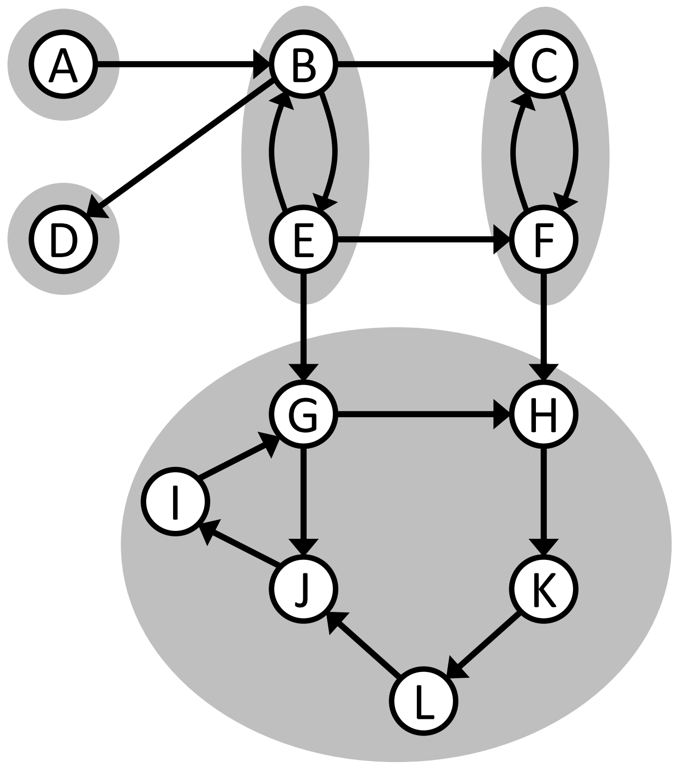
$\{B, E\}$

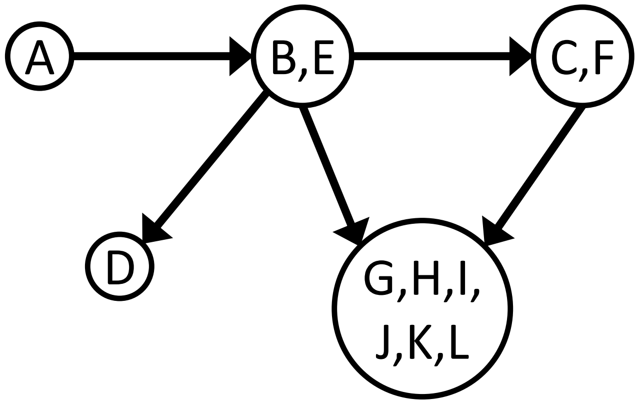
$\{C, F\}$

$\{D\}$

$\{G, H, I, J, K, L\}$

Strongly Connected Components





Property: every directed graph is a DAG of its strongly connected components. (Exercise: prove it!)

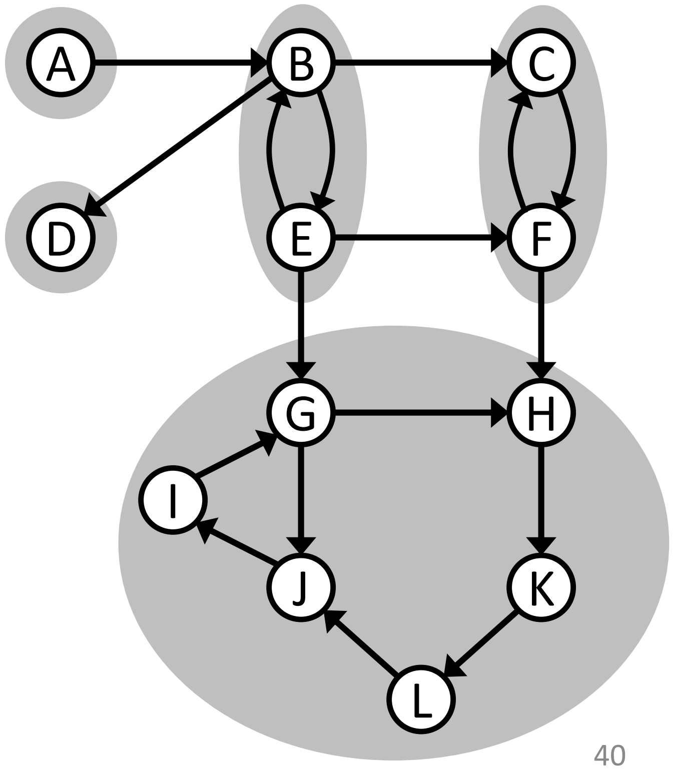
A directed graph can be seen as a 2-level structure. At the top we have a DAG of SCCs. At the bottom we have the details of the SCCs.

Every directed graph can be represented by a ***meta-graph***, where each meta-node represents a strongly connected component.

Properties of DFS and SCCs

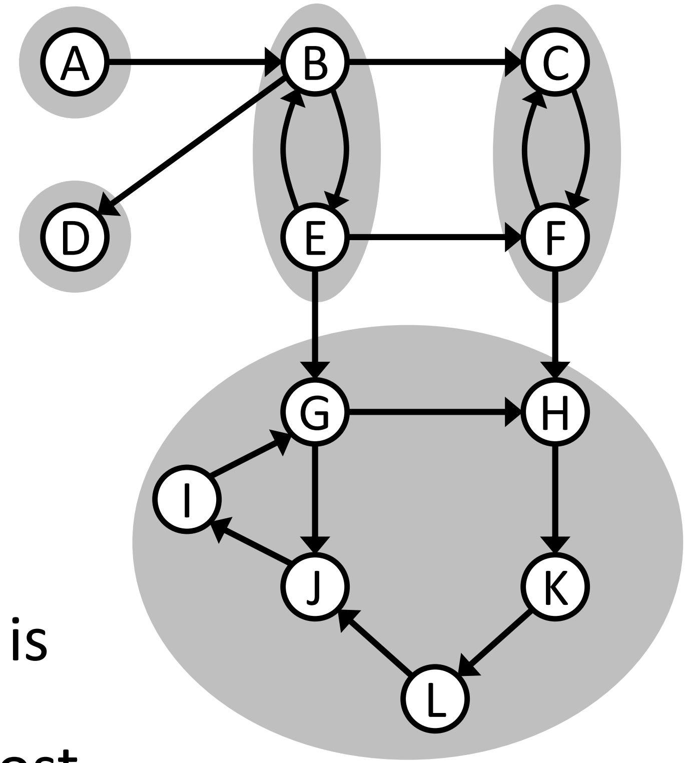
- **Property:** If the *explore* function starts at u, it will terminate when all vertices reachable from u have been visited.
  - If we start from a vertex in a sink SCC, it will retrieve exactly that component.
  - If we start from a non-sink SCC, it will retrieve the vertices of several components.

- **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.

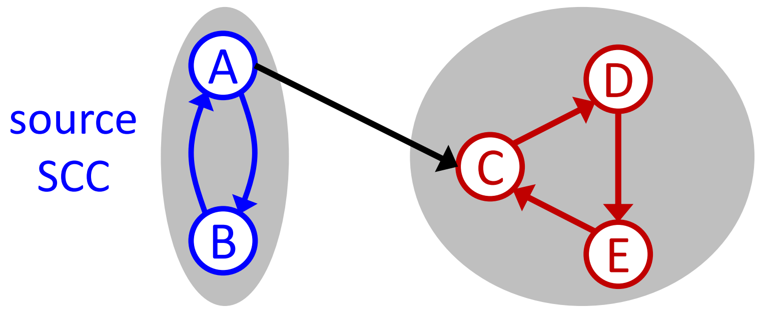


Properties of DFS and SCCs

- **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:** If C and C' are SCCs and there is an edge $C \to C'$, then the highest post number in C is bigger than the highest post number in C'.
- **Property:** The vertex with the highest DFS post number lies in a source SCC.

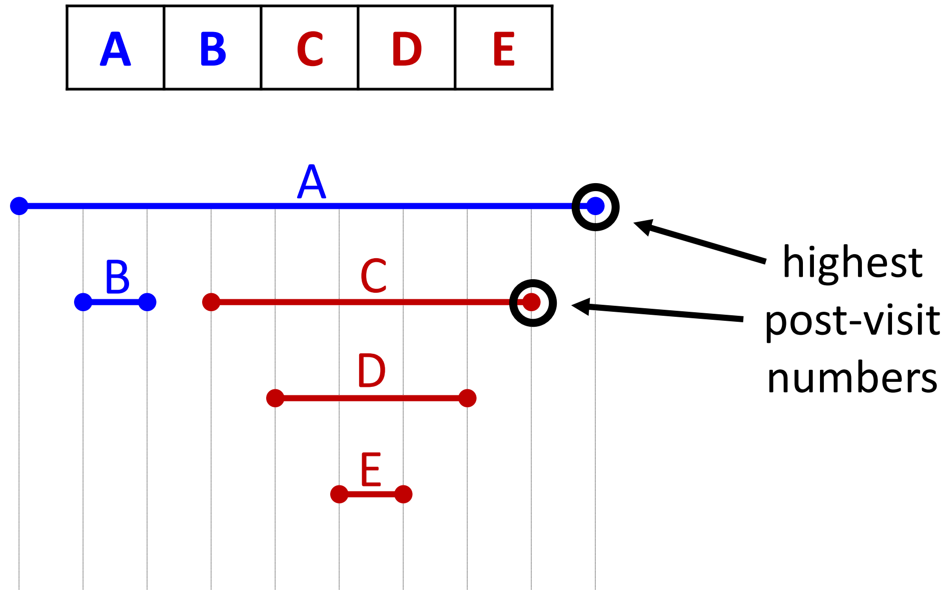


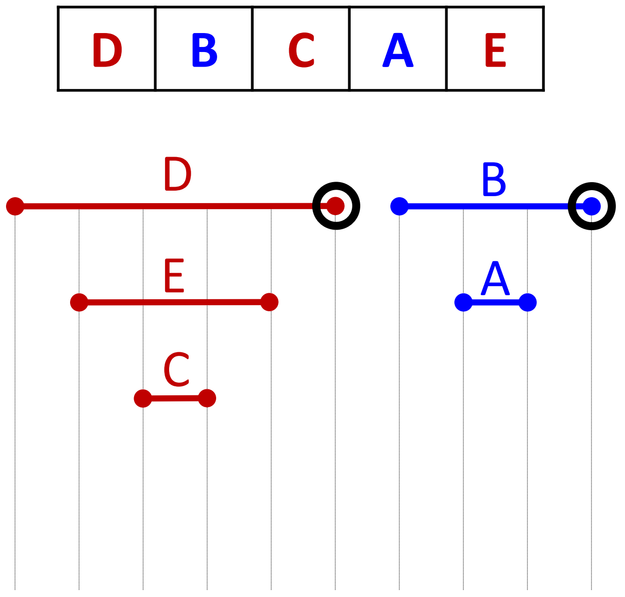
Properties of DFS and SCCs



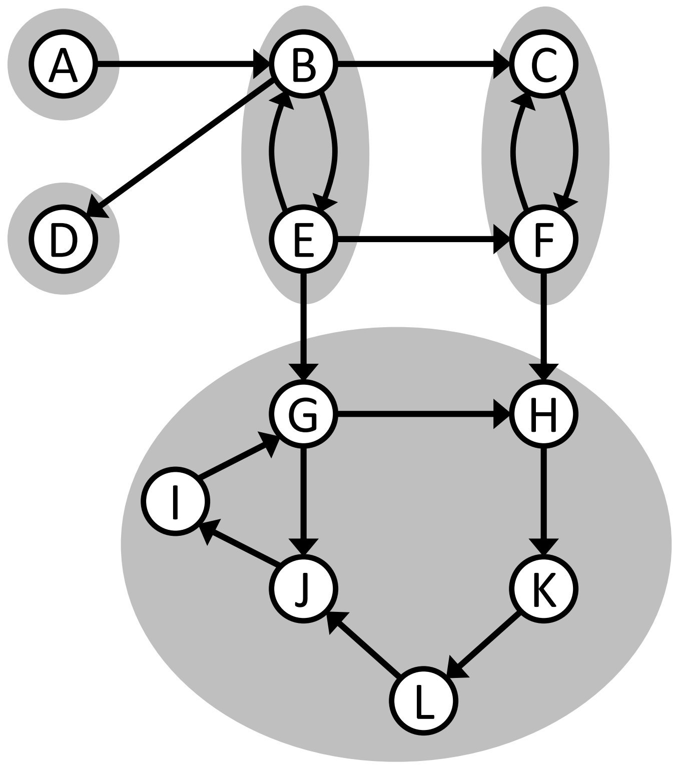
But we would like executing DFS starting from the sink nodes!

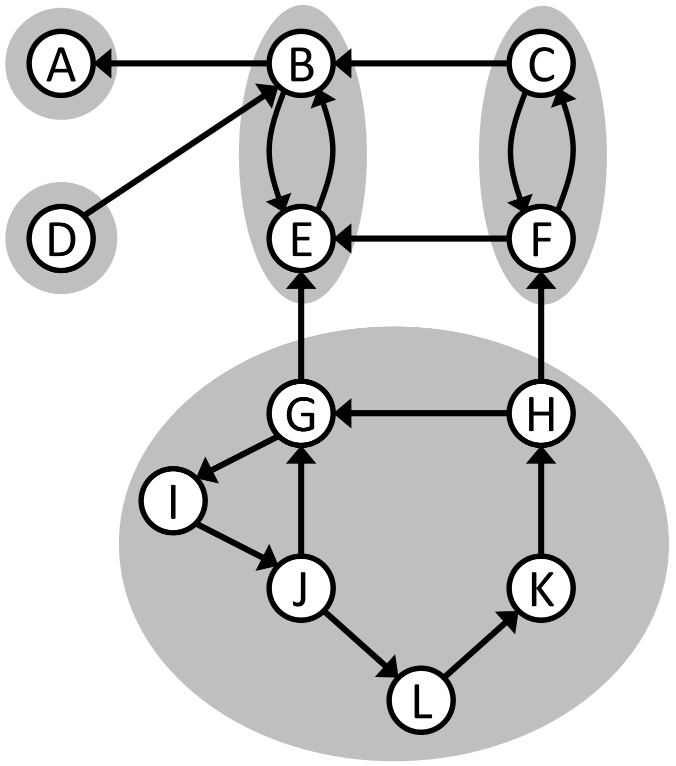
How can we do that?

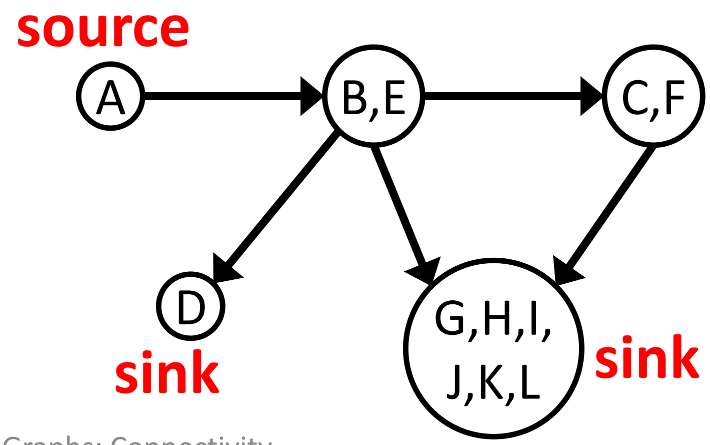


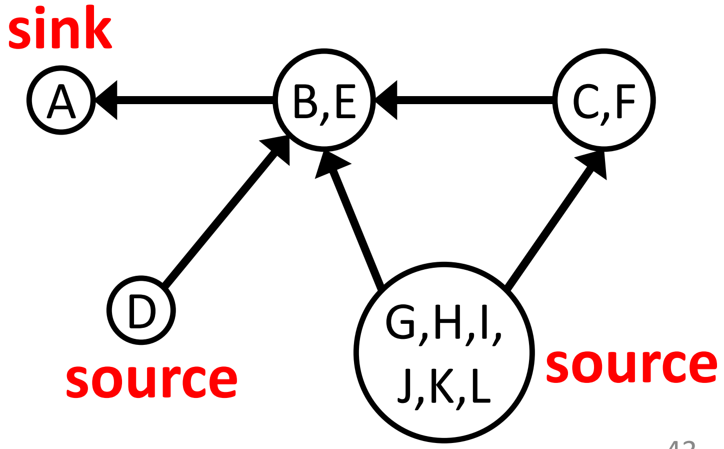


Reverse graph (G^R)









SCC algorithm

```
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

```
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 $v \in$ edges$[u]$:
      edgesR[$v$].insert($u$)
  return $(V,$edgesR$)$
```

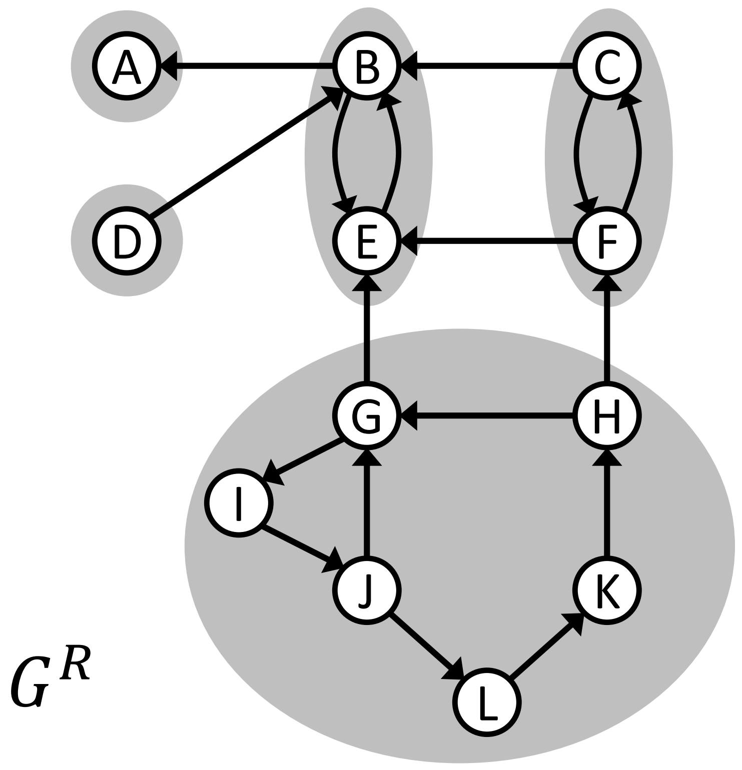
Sorting V in linear time

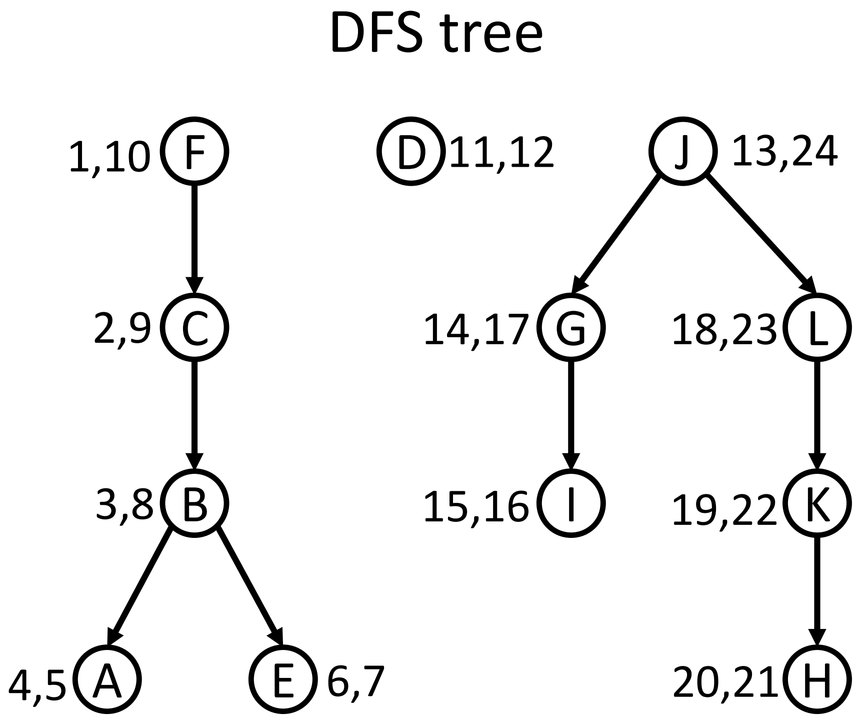
```
function SCC($G$):
// Input: $G(V,E)$ a directed graph
// Output: each vertex $v$ has an SCC number in ccnum[$v$]
   $G^R$= reverse($G$)
   DFS($G^R$) // calculates post numbers
   sort $V$   // decreasing order of post number
   ConnComp($G$)
```

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.

Sorting V in linear time





Assume the initial order:
$F, A, B, C, D, E, J, G, H, I, K, L$

Vertex:	**J**	**L**	**K**	**H**	**G**	**I**	**D**	**F**	**C**	**B**	**E**	**A**
Post:	24	23	22	21	17	16	12	10	9	8	7	5

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.

EXERCISES

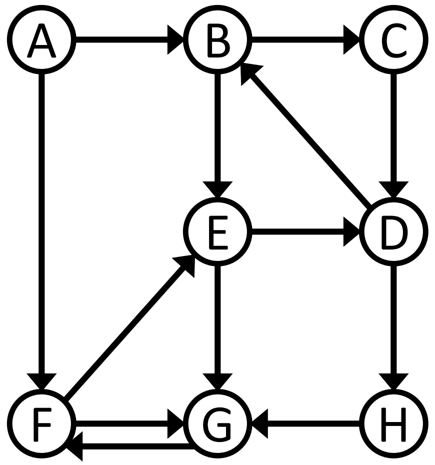
DFS: stack overflow

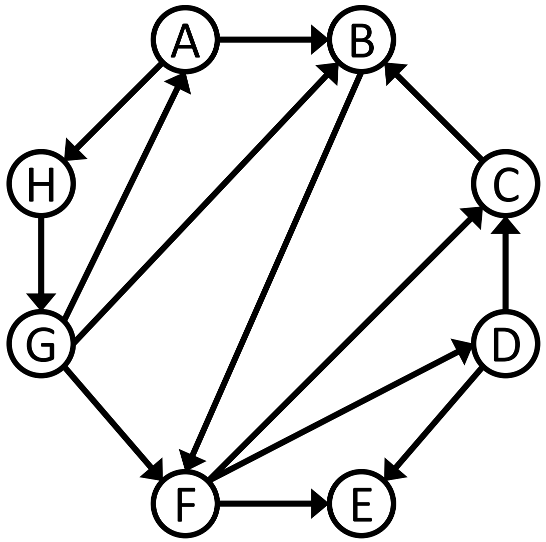
- DFS can be implemented with an elegant recursive algorithm, but it may experiment *stack overflow* problems. Explain why.

- Design an iterative version of DFS.

- Challenge: do not to search in stack**overflow**

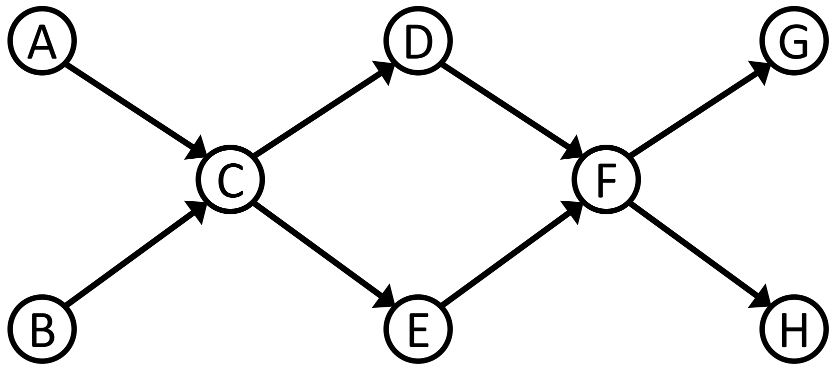
DFS (from [DPV2008])





Perform DFS on the two graphs. Whenever there is a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge, forward edge, back edge or cross edge, and give the `pre` and `post` number of each vertex.

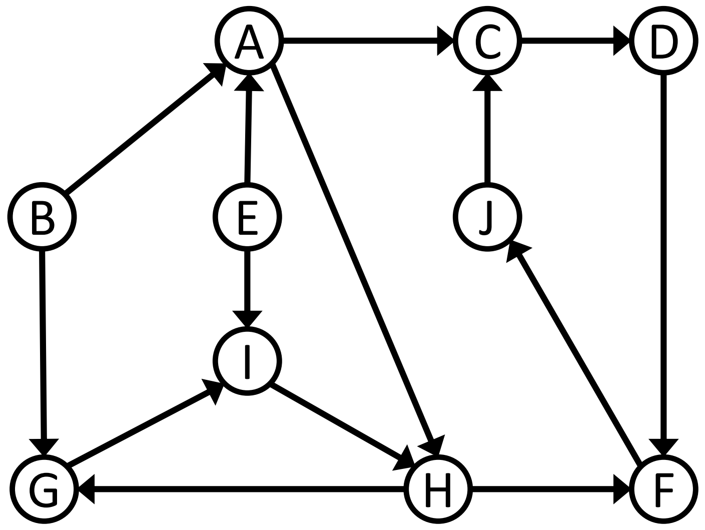
Topological ordering (from [DPV2008])

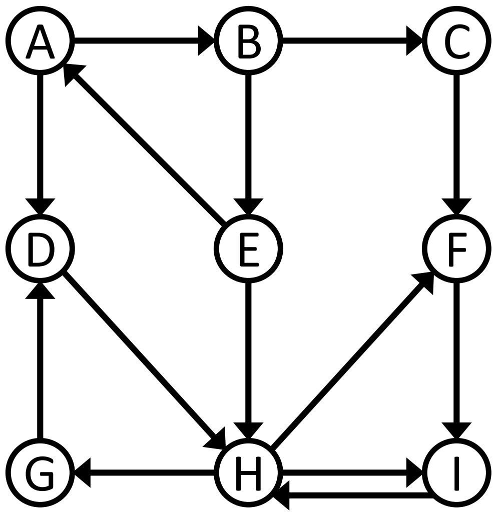


Run the DFS-based topological ordering algorithm on the graph. Whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.

1. Indicate the `pre` and `post` numbers of the nodes.
2. What are the sources and sinks of the graph?
3. What topological order is found by the algorithm?
4. How many topological orderings does this graph have?

SCC (from [DPV2008])





Run the SCC algorithm on the two graphs. When doing DFS of G^R: whenever there is a choice of vertices to explore, always pick the one that is alphabetically first. For each graph, answer the following questions:

1. In what order are the SCCs found?
2. Which are source SCCs and which are sink SCCs?
3. Draw the meta-graph (each meta-node is an SCC of G).
4. What is the minimum number of edges you must add to the graph to make it strongly connected?

Streets in Computopia (from [DPV2008])

The police department in the city of Computopia has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. A computer program is needed to determine whether the mayor is right. However the city elections are coming up soon, and there is just enough time to run a *linear-time* algorithm.

a) Formulate this problem graph-theoretically, and explain why it can indeed be solved in linear time.

b) Suppose it now turns out that the mayor's original claim is false. She next claims something weaker: if you start driving from town hall, navigating one-way streets, then no matter where you reach, there is always a way to drive legally back to the town hall. Formulate this weaker property as a graph-theoretic problem, and carefully show how it too can be checked in linear time.

Pouring water (from [DPV2008])

We have three containers whose sizes are 10 pints, 7 pints and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. We are allowed one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that leaves exactly 2 pints in the 4-pint container.

a) Model this as a graph problem: give a precise definition of the graph involved and state the specific question about this graph that needs to be answered.

b) What algorithm should be applied to solve the problem?

c) Give a sequence of pourings, if it exists, or prove that it does not exist any sequence.

Hint: A vertex of the graph can be represented by a triple of integers.