Graphs

Lab-STICC

Laurent.lemarchand@univ-brest.fr
http://www.labsticc.univ-brest.fr/pages_perso/lemarch/Cours

Relationshipss

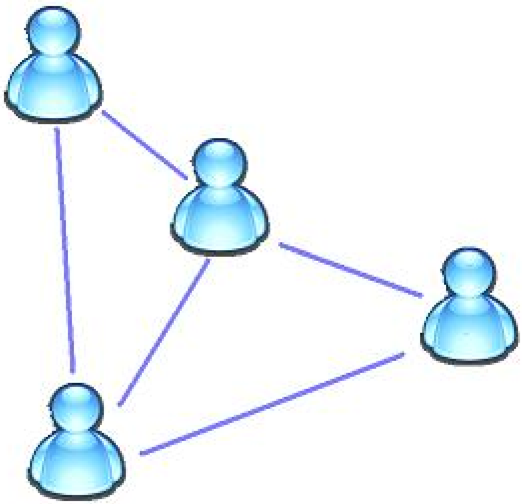
- Relation between 2 entities
  - *Smaller than*
  - *Done before*
  - *Done by*
  - ....
- Drawing :

Symetrical
Relations
or not

Graph !

Symetric relations

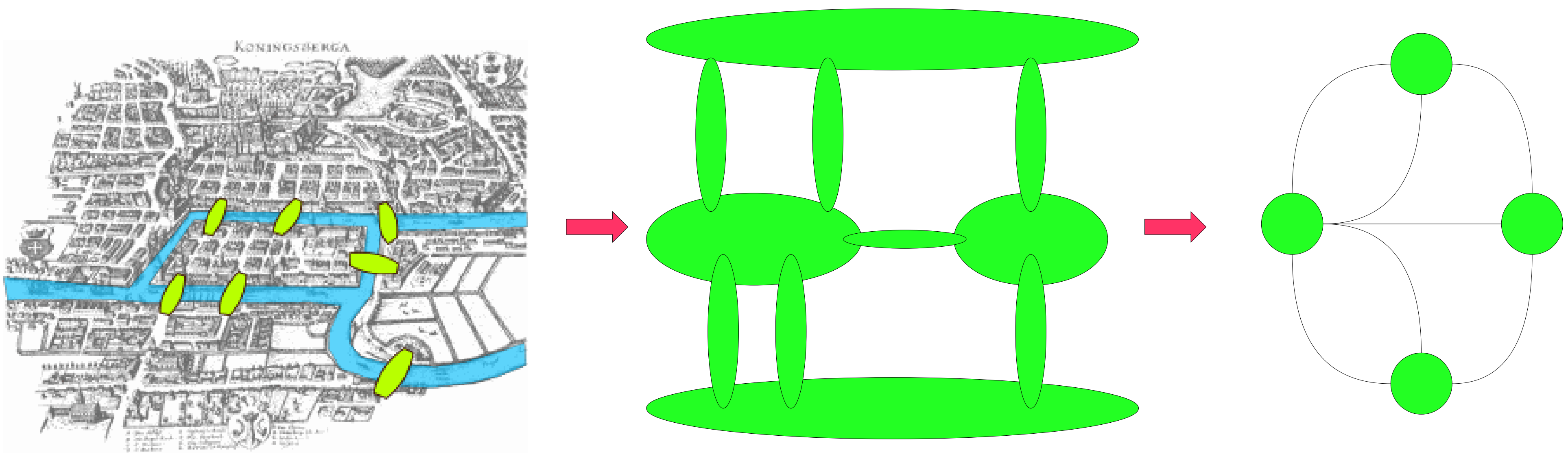
- Relation between 2 entities
  - Know each other
- More global :
  social network



Average degree
=
Dunbar's number
=
125

History (1)
Konigsberg' bridges (Euler, 1736)

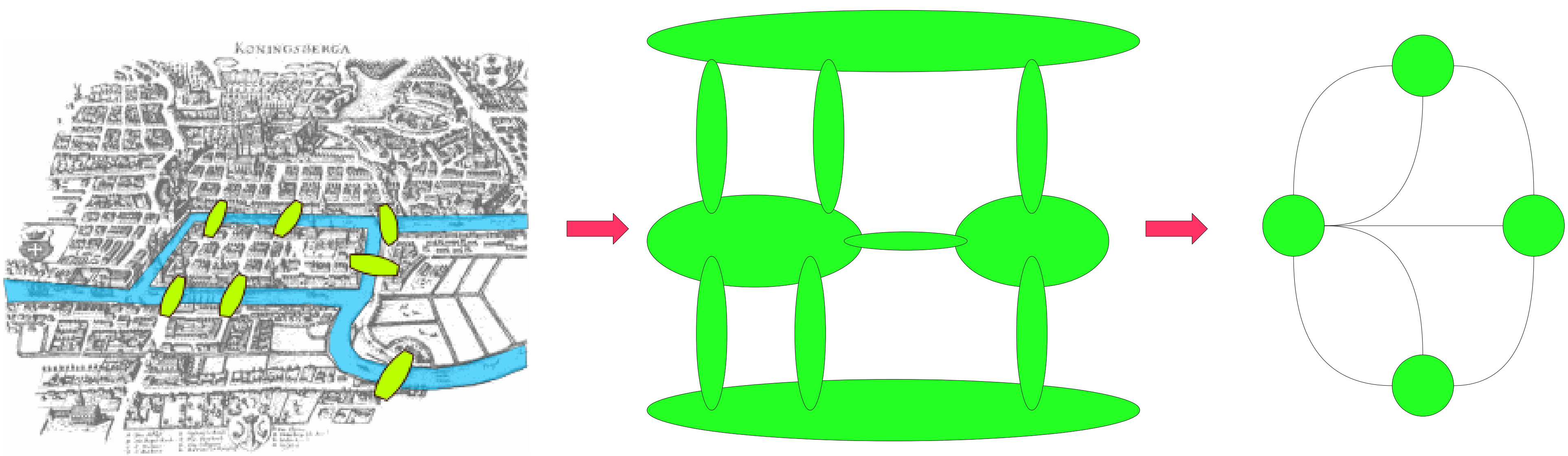
- 2 islands within the town, connected by a bridge, and also connected to the ground.
- Traverse all bridges once, with a round trip



http://www.wikipedia.org

History (2)
Konigsberg' bridges (Euler, 1736)

- How to find an *Eulerian Cycle* : chain including 1 once each edge of a graph
- Impossible if odd degree node exists



http://www.wikipedia.org

Eulerian and hamiltonian traversals

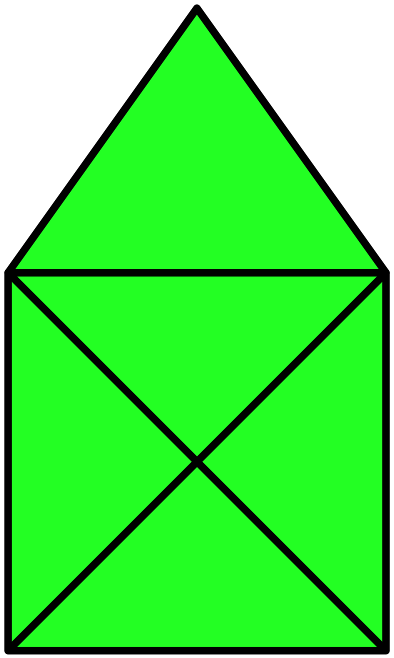
- Cycle finding
  - Eulerian : each edge is included once exactly
  - Hamiltonian : each node is included once exactly
- Associated problems
  - *Shortest length* Eulerian cycle
  - Chinese Postman : Shortest length cycle including each edge (at least one time)
  - Travelling Salesman Problem (TSP) : *Shortest length Hamiltonian* cycle

Eulerian traversal

- Euler Theorem

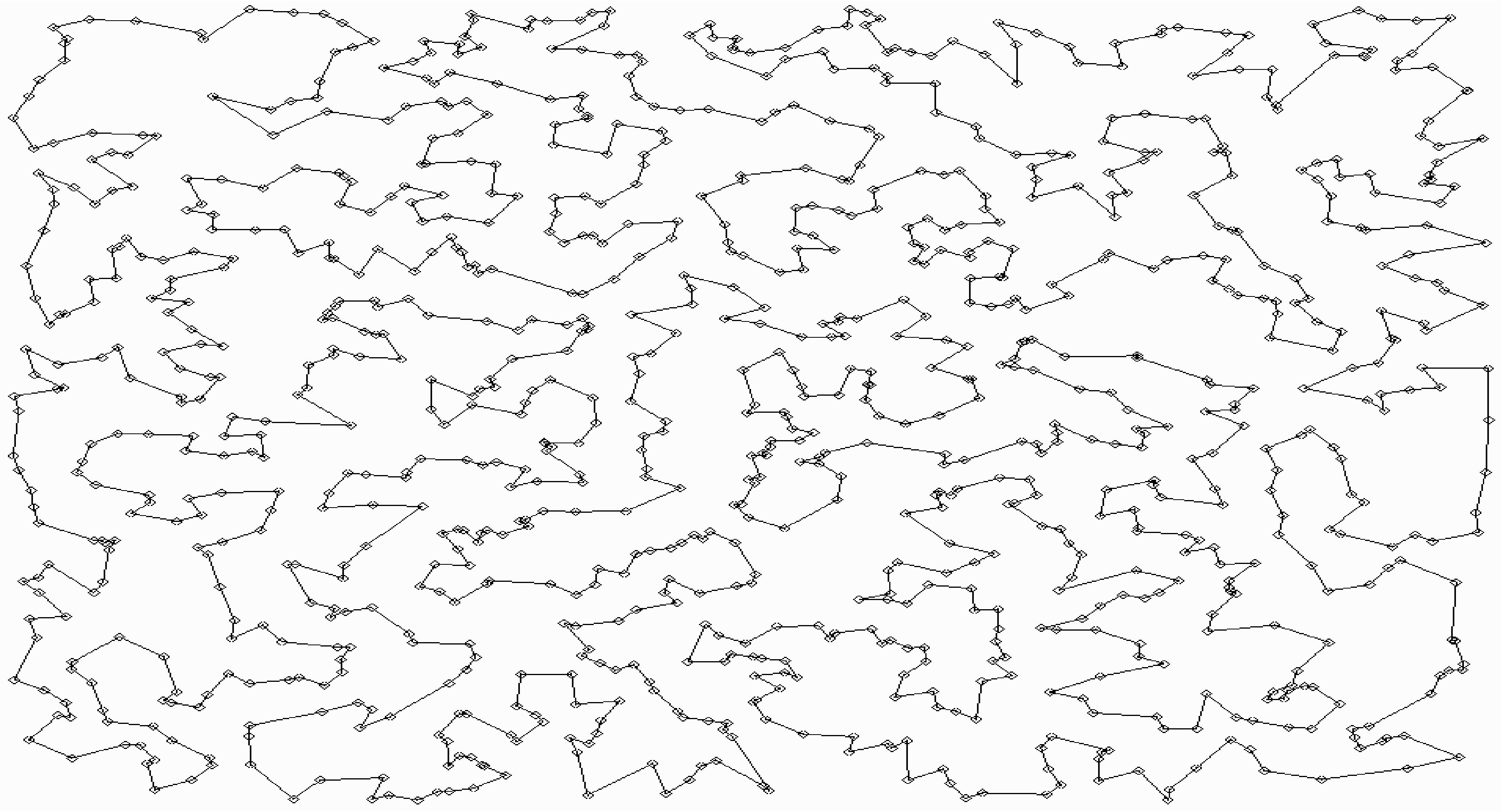
A multigraph includes an eulerian cycle *iff* it is connected and it includes 0 or 2 odd-degree vertices

- Example
  - Draw the hull without lifting your pencil
  - Chinese postman : you can draw a line twice



Hamiltonian traversal

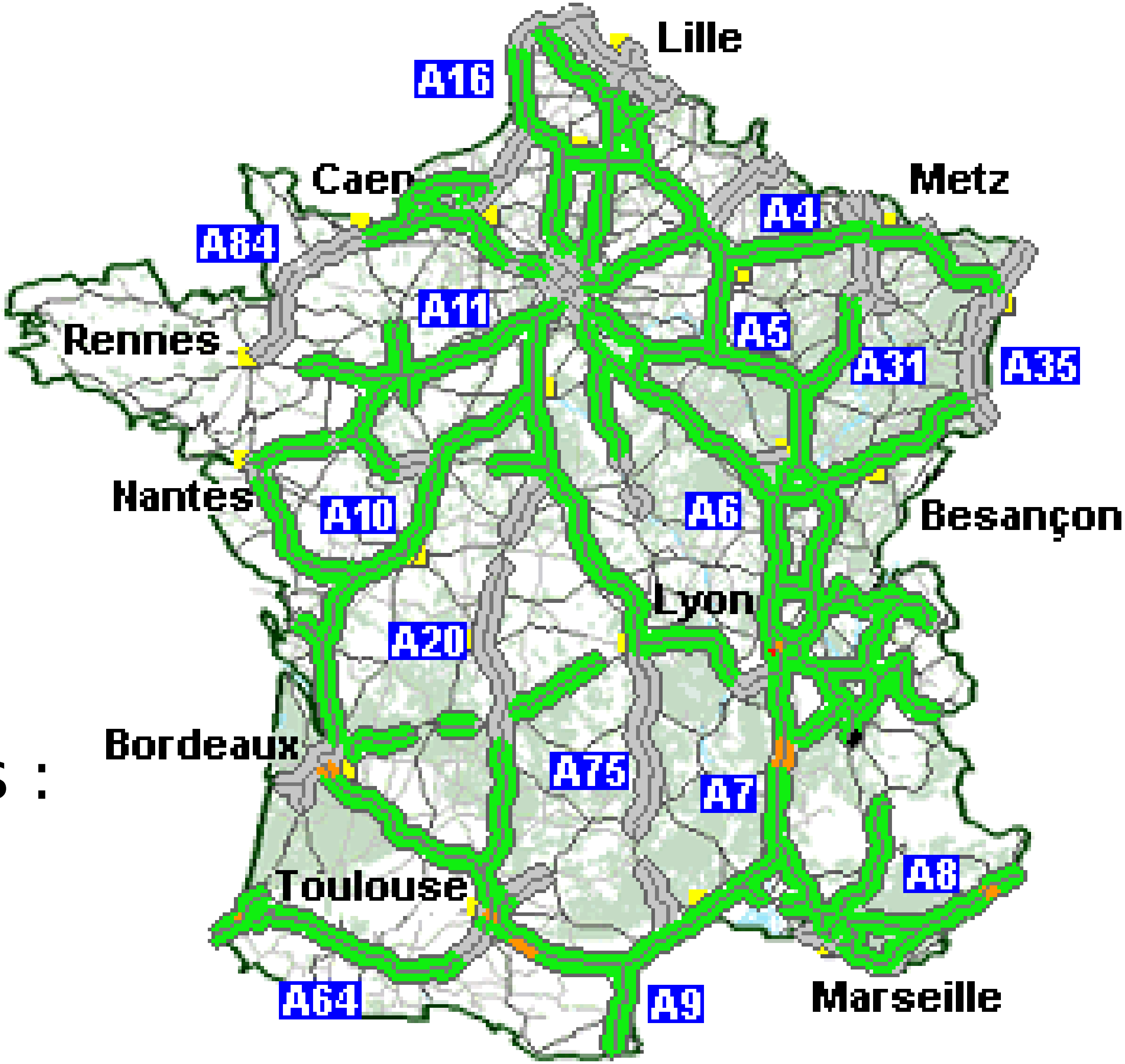
- Looking for a shortest length hamiltonian cycle : *Travelling Salesman Problem (TSP)*



http://www.cs.berkeley.edu/~bonachea

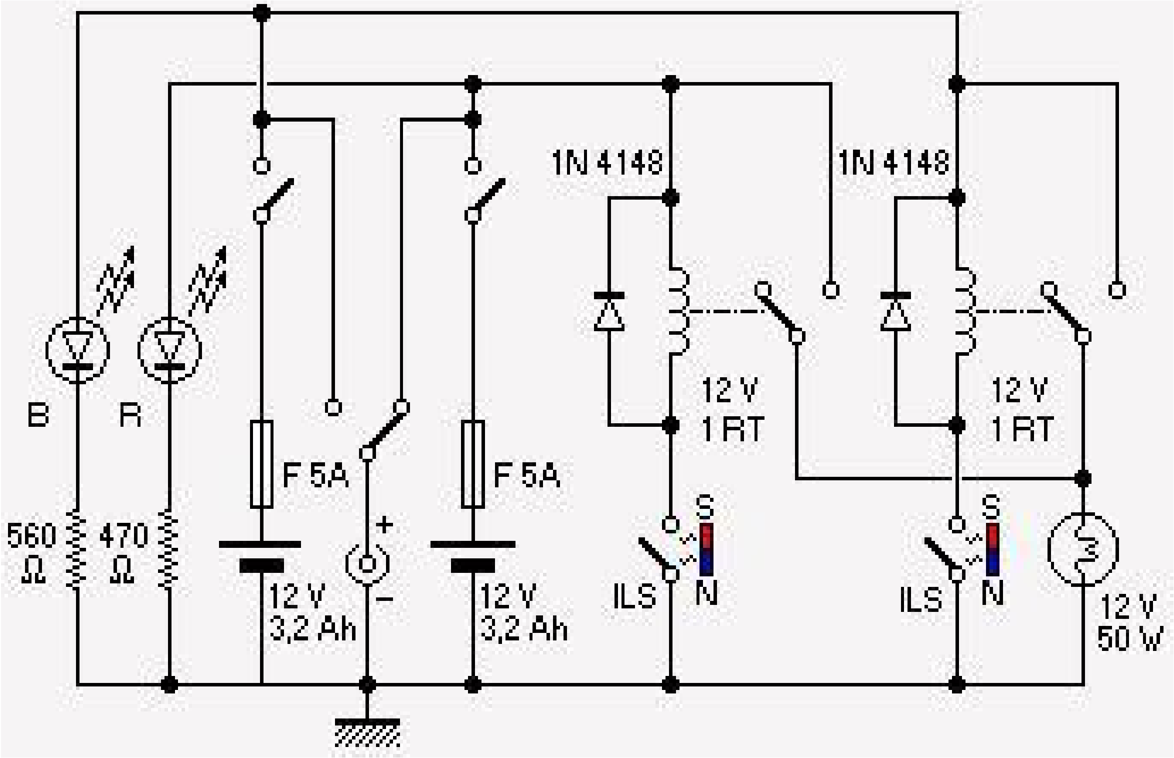
A panel of applications
Maps

- Routes
  - *Which way ?*
  - *Which cost ?*
- Routing
  - Geographic
  - Internet
- Hamiltonian cycles : TSP
- Eulerian cycles : Chinese Postman

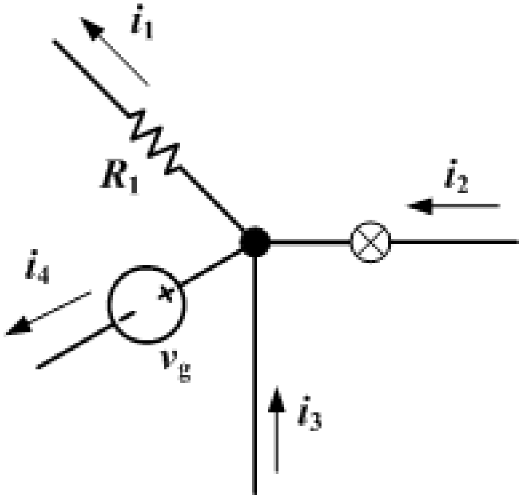


A panel of applications
Electrical Circuits

- Junction nodes in electrical circuits

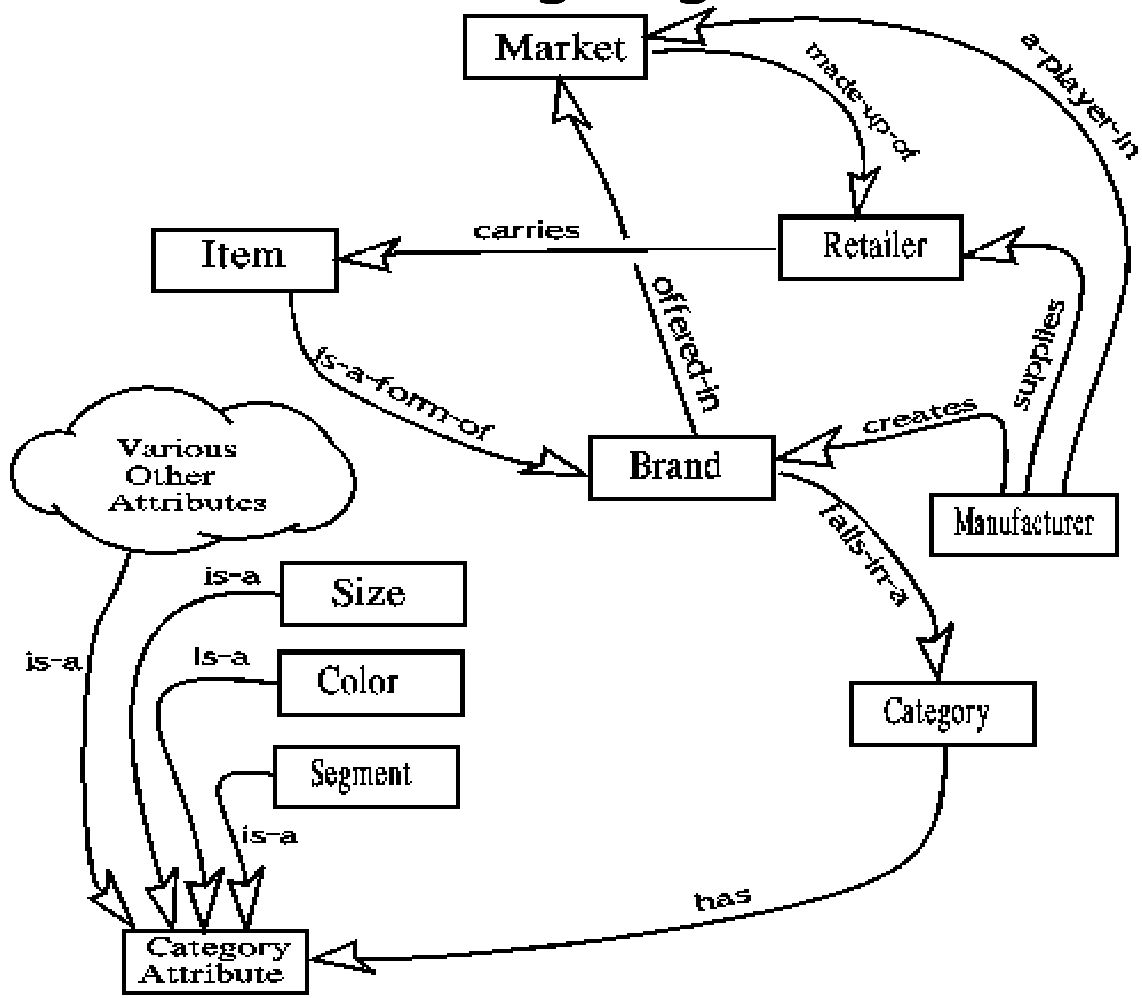


- Kirchoff's point rule
  - *Conservation of energy*
  - *Conservation of electric charge*



A panel of applications
Semantic network

- Semantic relations between concepts
- Applications in Natural Language Processing

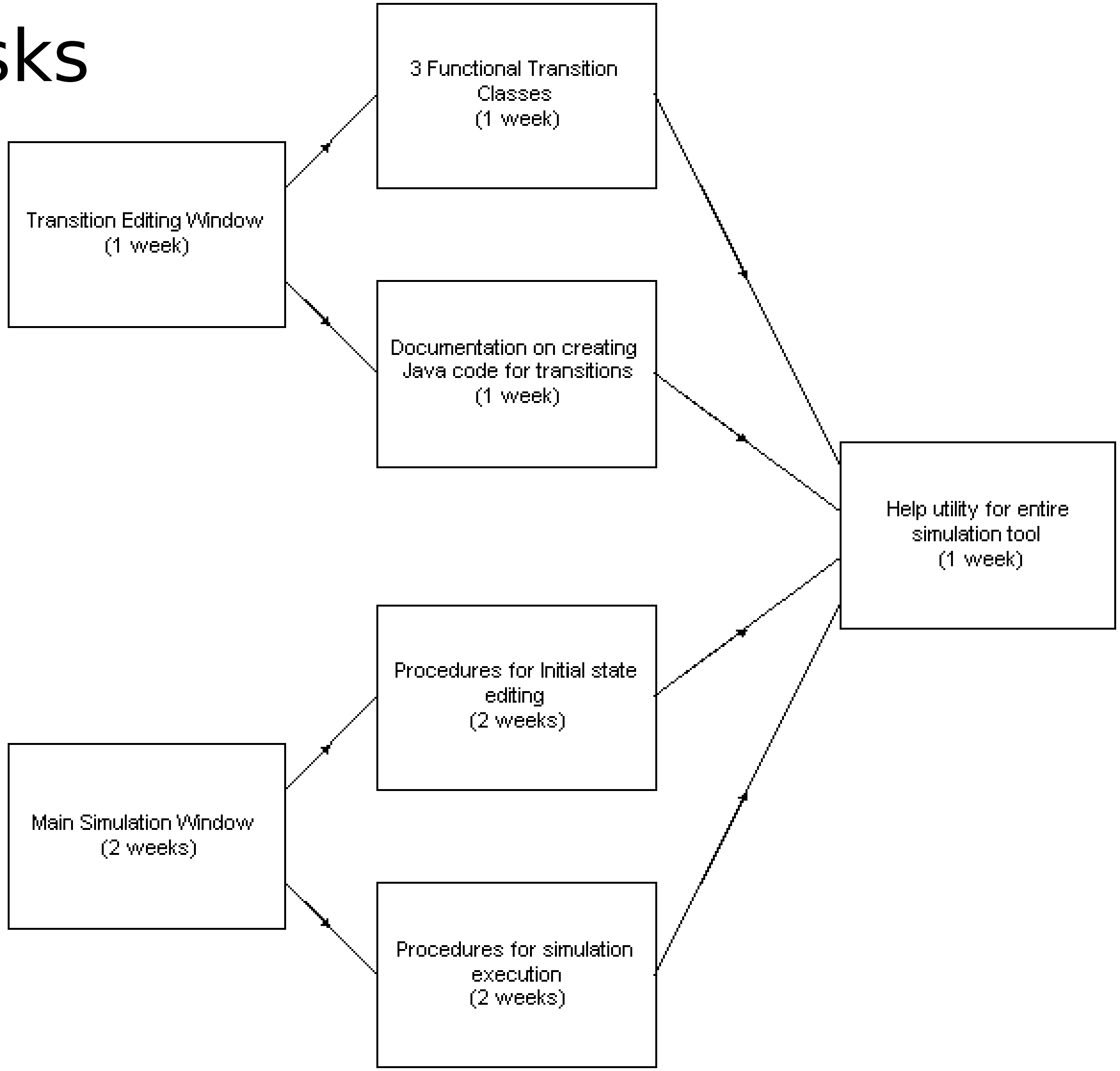


A panel of applications
Project management and scheduling

- Perform activities/tasks
  - *Dependancies*
  - *Times*

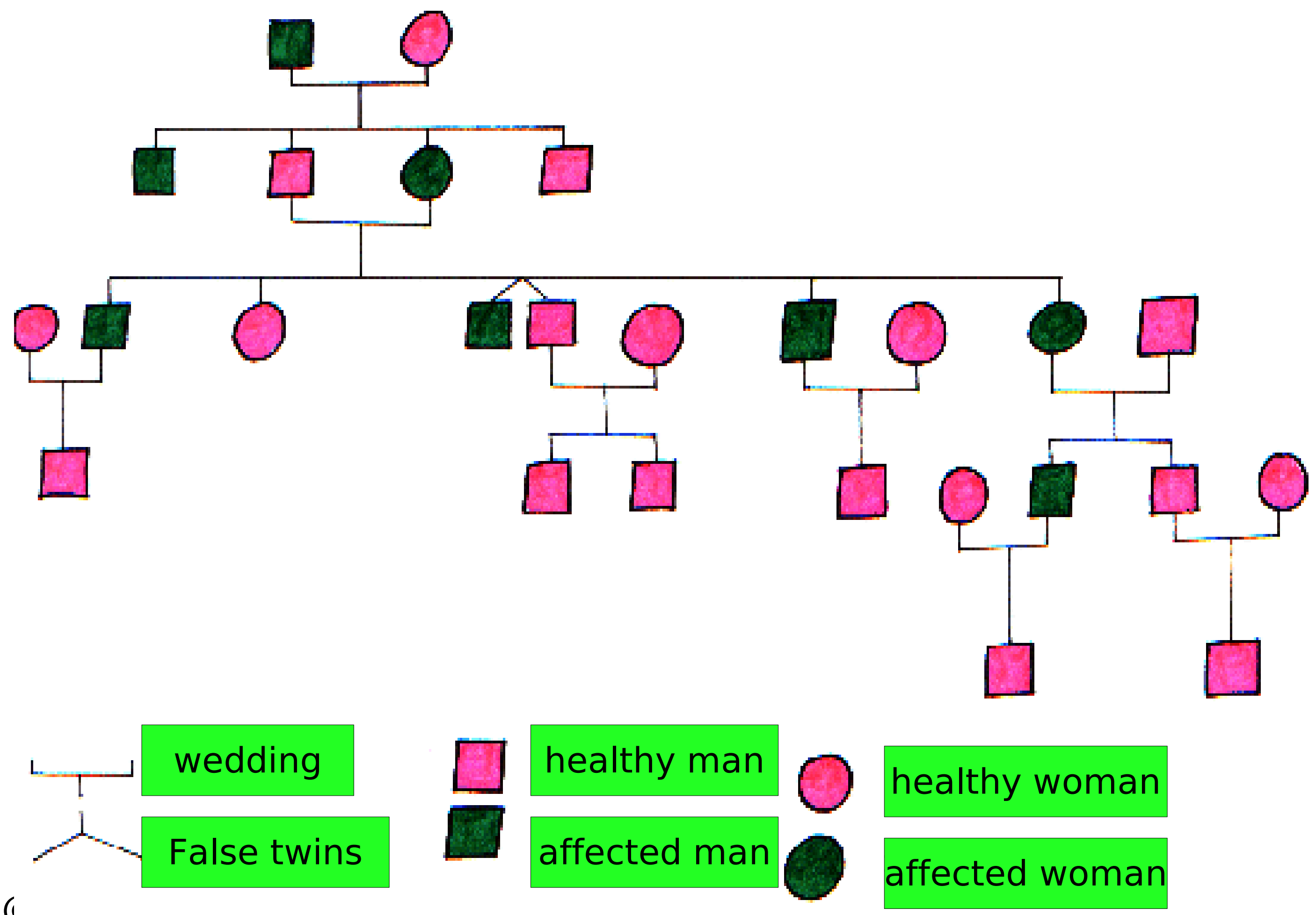
- Longest path time
- Activity network ?

- PERT
- CPM



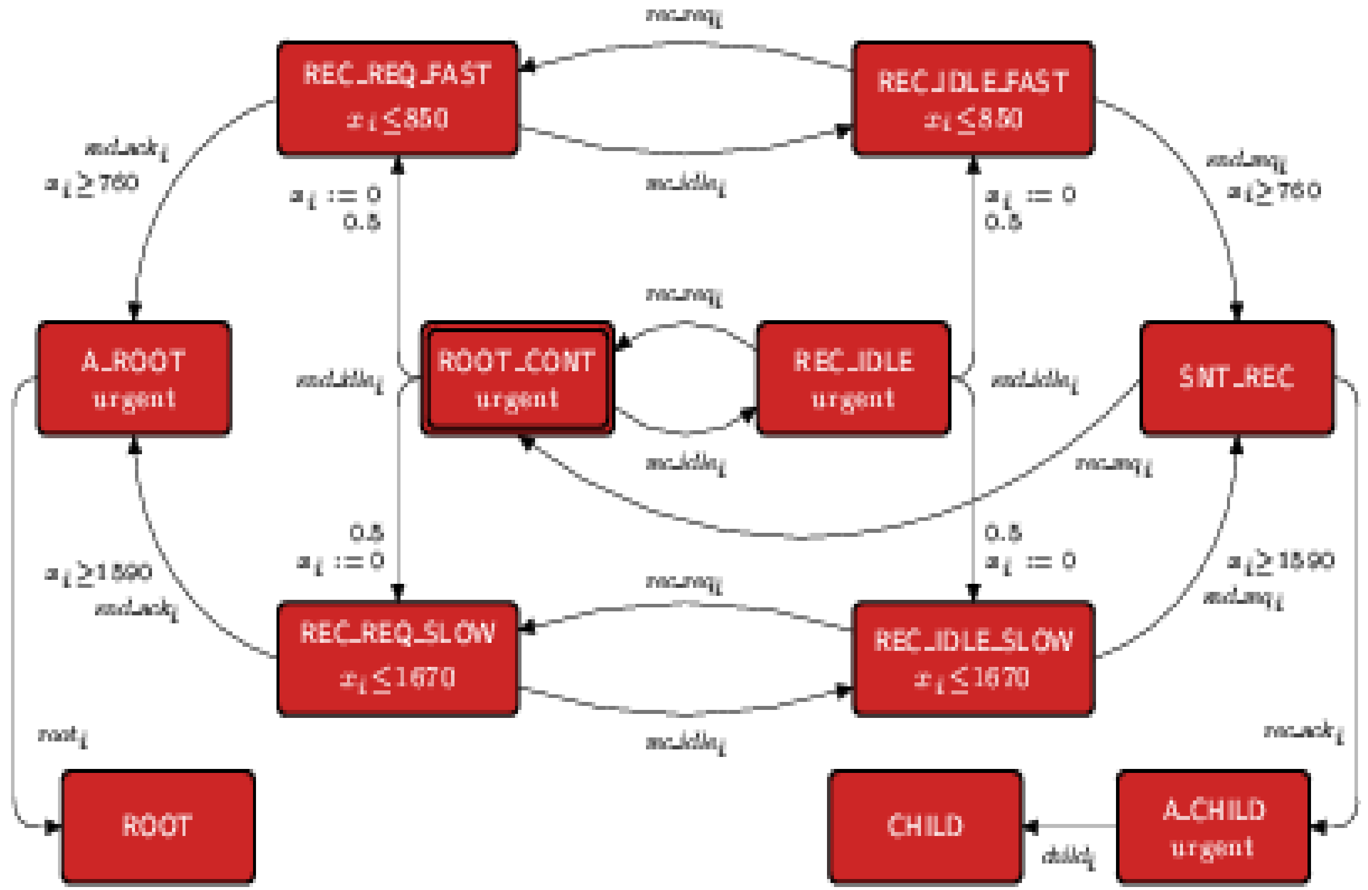
A panel of applications applications Genealogy

- Particular Graphs : trees
- Genetic diseases



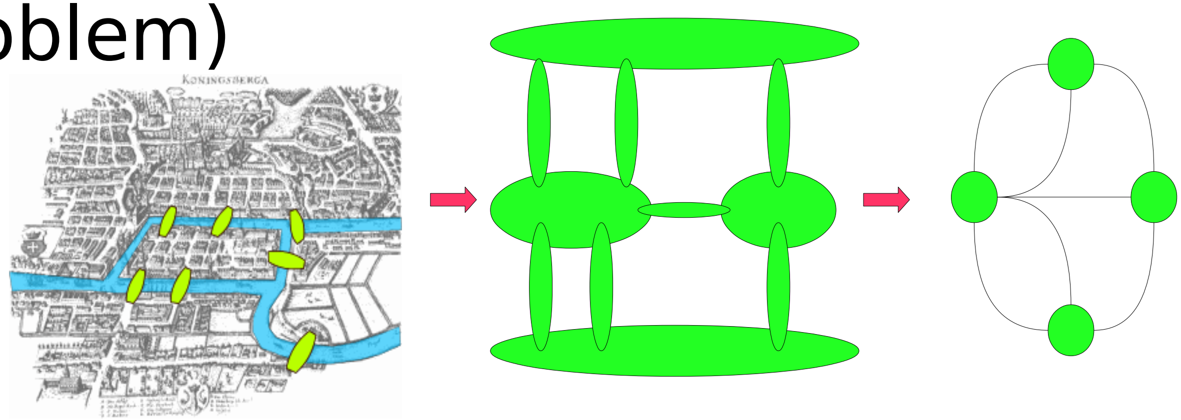
A panel of applications
Automata

- Describe a protocol
  - *TCP*
  - *Telephone*
  - *Traffic lights*
  - *...*
- Concepts
  - States
  - Transitions

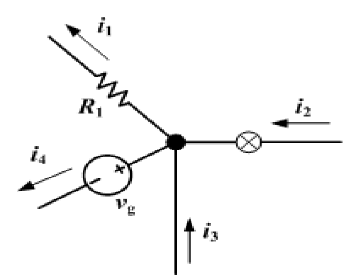


Summary

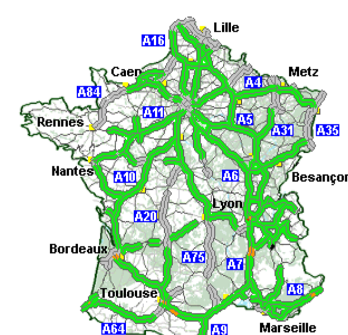
- Graph conception
  - Modelize problems
  - Graphic representation (help you solving the problem)



- Exploiting the model
  - Check properties



  - Solution optimization



Examples of models

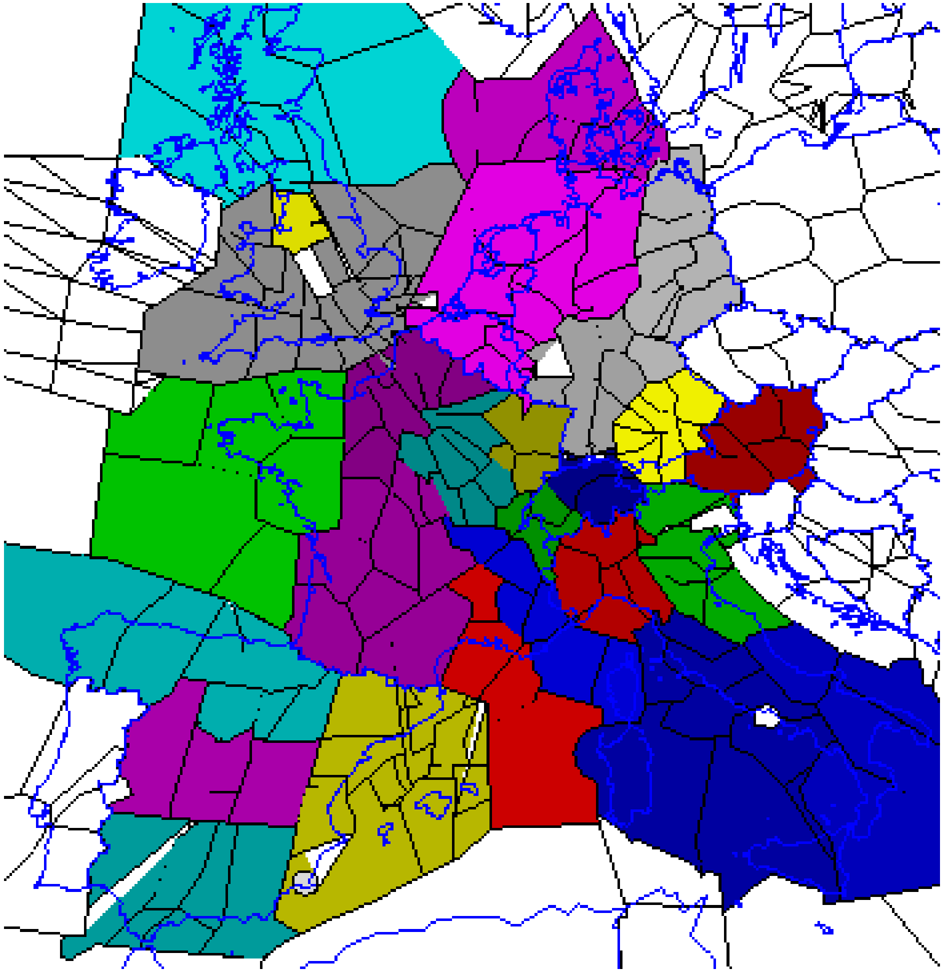
- Air traffic control
  - Minimize screen swaps and equilibrate controllers' amount of work

- VLSI circuits optimization
  - Partition large circuits into smaller ones and re-compute their layout

- Model ?
- Model properties to focus ? What is the optimization criteria ?
- Which algorithm is adapted, how to use it ?

Air traffic control : model

- Given flight level
- Standard altitude for an airplane.
- FL260 is the Flight Level 260
- (26 000 feet, 7 900m approx.).

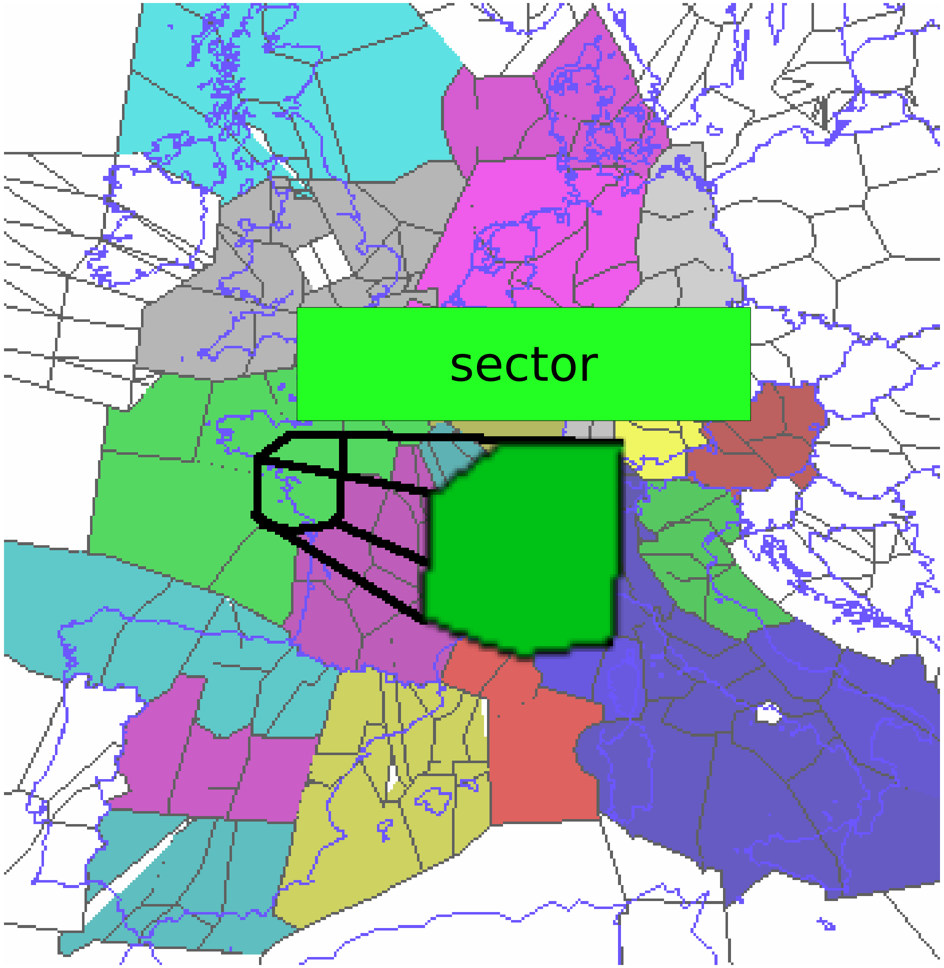


http://www.emse.fr/spip/IMG/pdf/Bichot-11-05-07.pdf

Air traffic control : problem

- Sector

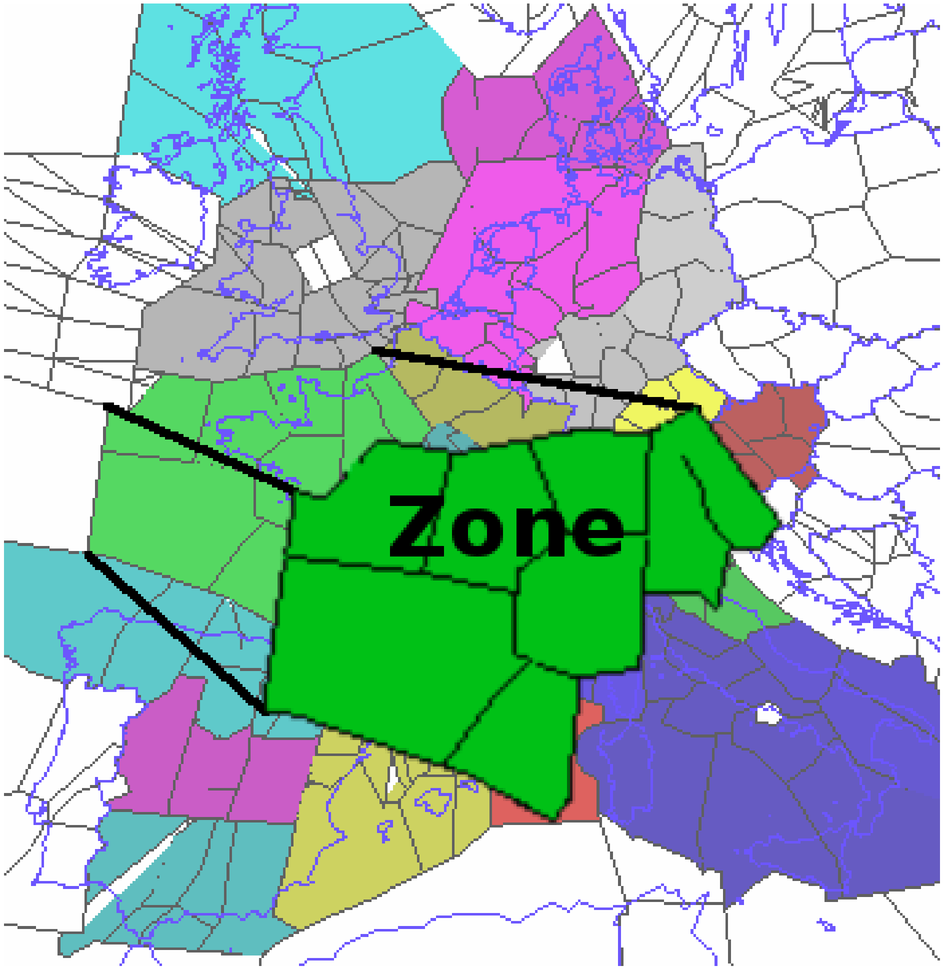
Geographic area under the responsability of an air traffic controller



Air traffic control : problem

- Qualification zone

A set of sectors for which an air traffic controller is qualified



Air traffic control : goals

Improving safety, fluidity and capacity of the european sky

- Decrease the amount of work of controllers
  - Traffic monitoring
  - Conflicts management (predictive/actual)
  - Sectors Coordination.

Space partitioning

- Find a partition $P_k = \{S_1, . . . , S_k\}$ of the nodes of a graph G(V, E) :
  - a part = a qualification zone
  - a node = a sector
- There are streams of airplanes between the differents sectors : G is connected and weighted
- k, # parts = number of qualification zones
- European sky area :
  k = 32 parts, 762 nodes, 10 328 edges.

Outline

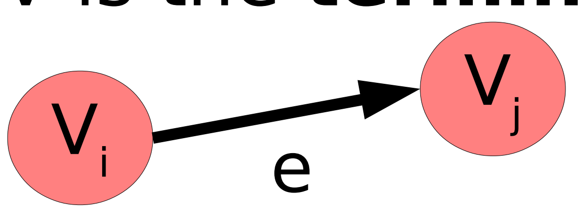
- Relations
- Graph basics
  - Definitions
  - Implementation
  - Traversal
- Optimization algorithms
  - Paths, trees, flow graphs
- Scheduling
  - Definitions
  - PERT and MPM methods

Bibliography

- web
- Algorithmic graph theory
  J. A. McHugh, Prentice Hall, 1990

Graphs
Terminology

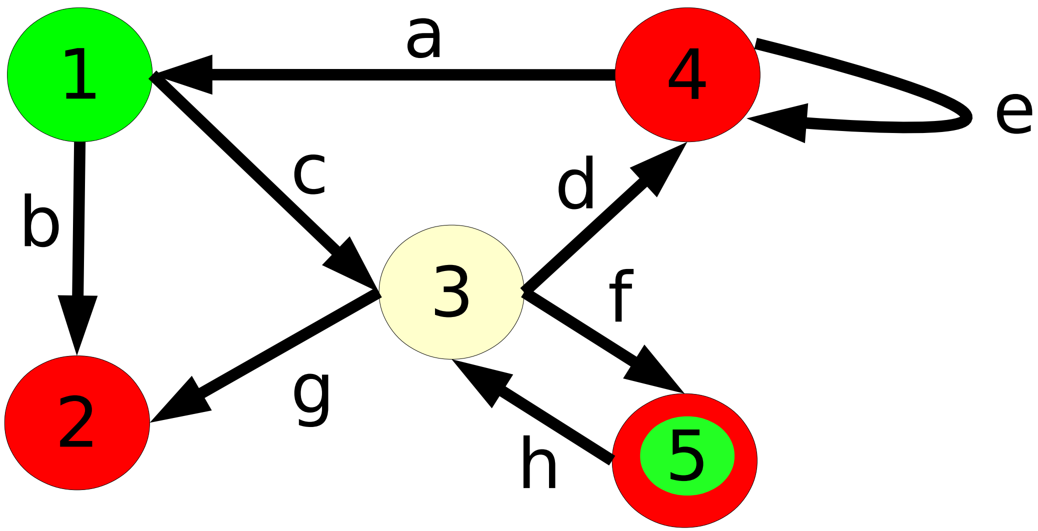
- **Directed Graph** (**Digraph**) G : 2 sets
  - V : set of **vertices (nodes)** of G
  - E : set of **edges (arcs)** of G
- A directed edge $e \in E$ is an ordered couple (V_i, V_j) of endpoints
  - $V_i \in V$ is the **initial** node (**tail**) of e
  - $V_j \in V$ is the **terminal** node (**head**) of e



E represents a relation between V_i and V_j

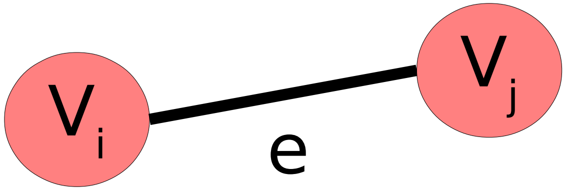
Graphs
Terminology

- **order** of G = (V, E) : number of nodes |X|
- **p-graph** : $\forall\ V_i, V_j \in V$, each edge set $(V_i \rightarrow V_j)$ has a size $\leq$ p
- If $V_i \rightarrow V_j \in E$, V_j is a **direct successor** of V_i and V_i is a **direct predecessor** of V_j



Graphs
Terminology

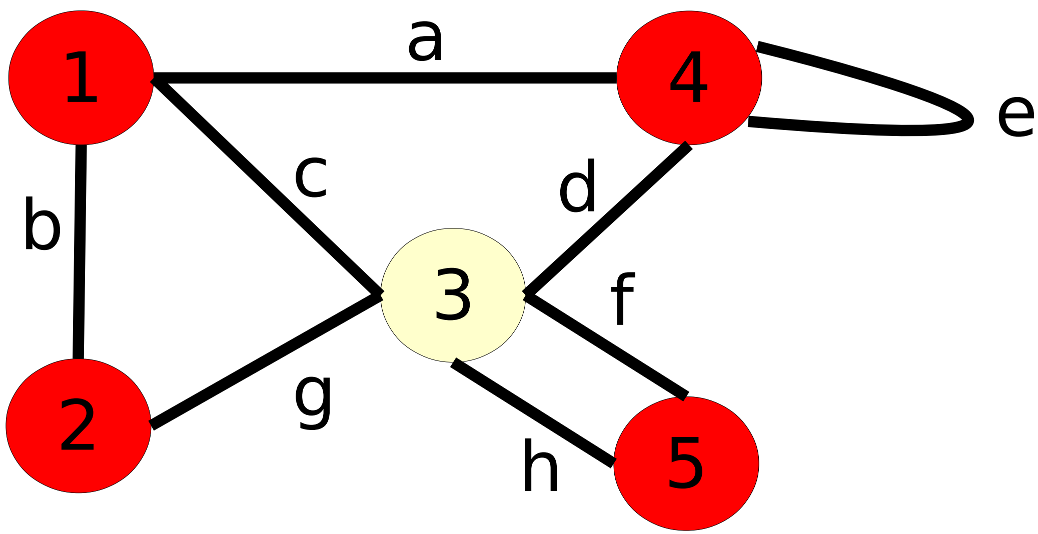
- **Unoriented Graph** $G = (V, E)$
  - V : **edge** set of G
- An edge $e \in E$ is an unoriented couple (S_i, S_j)
  - $V_{i,}$ V_j $\in$ V are **adjacents**



e represents a symetric relation between S_i et S_j

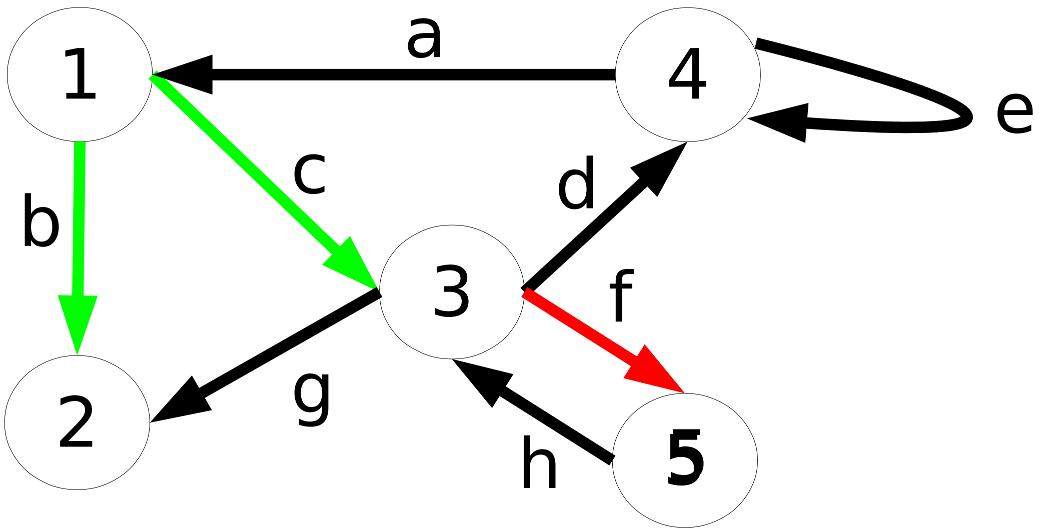
Graphs
Terminology

- **multigraph**, more than one edge between at least a couple $V_{i,}$ V_j
- **simple** graph if at most one edge for each couple $V_{i,}$ V_j and no loop



Graphs
Terminology

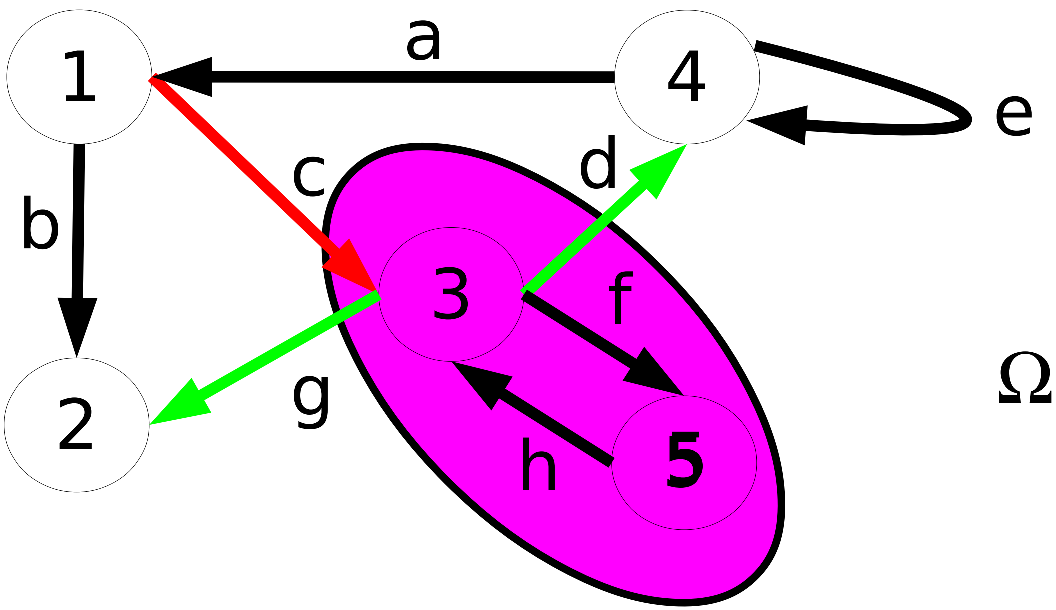
- **degree, half-degrees**, $d^+(S_i) + d^-(S_i) = d(S_i)$
  # leaving (resp. entering) edges :
  out-degree (resp. in-degree) of V_i



- $d^+(S_1) = 2$; $d^-(S_5) = 1$

Graphs
Terminology

- **Positive Cocycles of** $A \subseteq V$
  - $\Omega^+(A) = \{ e \in E \mid tail(u) \in A \wedge head(u) \notin A \}$
  - $\Omega^-(A) = \{ e \in E \mid tail(u) \notin A \wedge head(u) \in A \}$



$\Omega(A) = \Omega^+(A) \cup \Omega^-(A)$

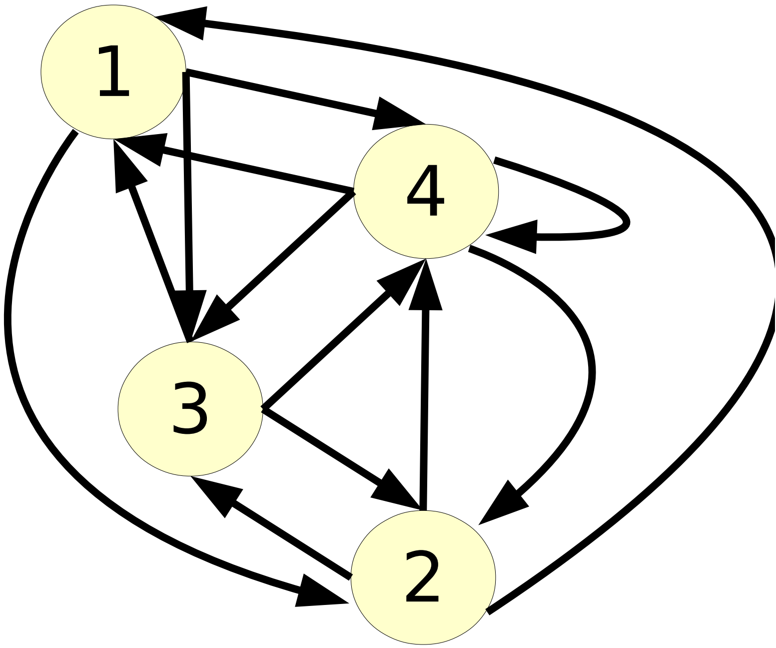
- $\Omega^+(A) = \{d, g\}$; $\Omega^-(A) = \{c\}$

Graphs
Terminology

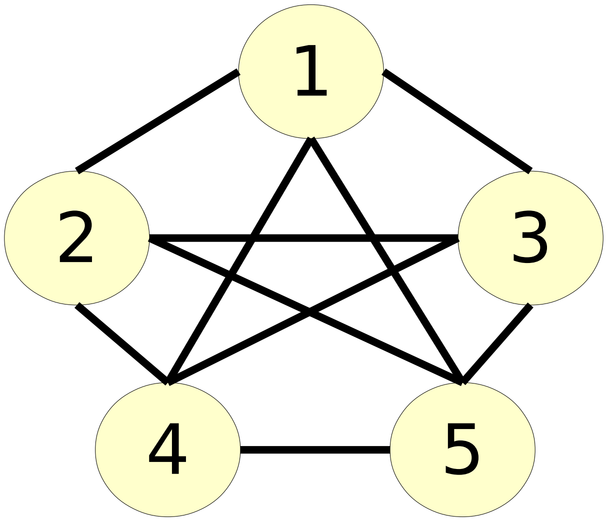
- **Complete graph, clique**
  - G = (V, E) is complete iff

    $\forall\ V_i, V_j \in V^2$, $i \neq j$,

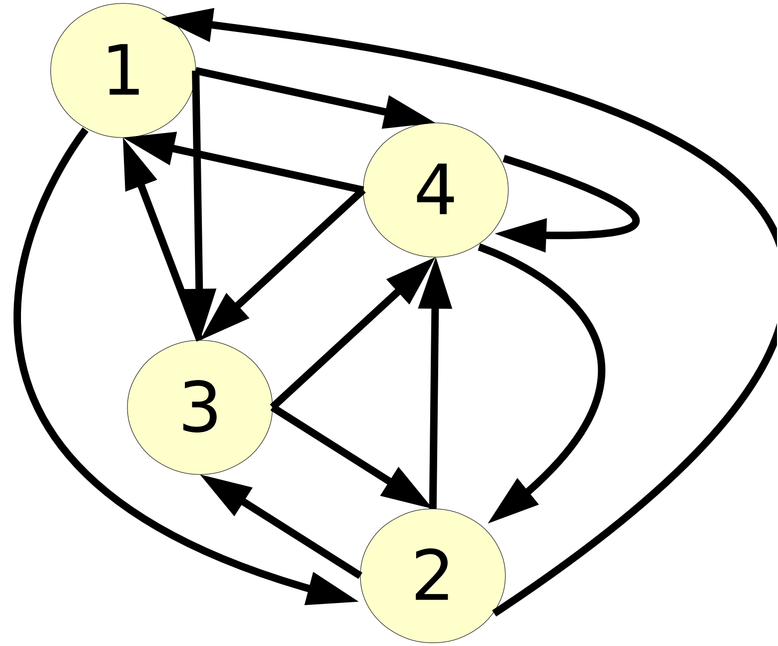
    $\exists$ edge $e = (V_i \rightarrow V_j) \in E$



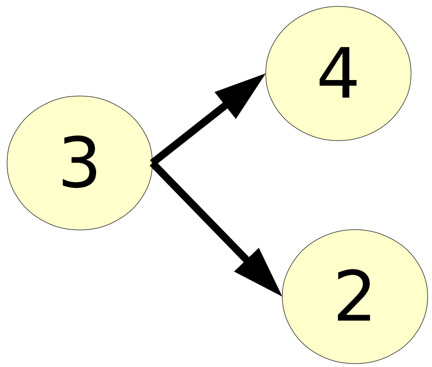
- **Clique** for unoriented graphs



Graphs Terminology

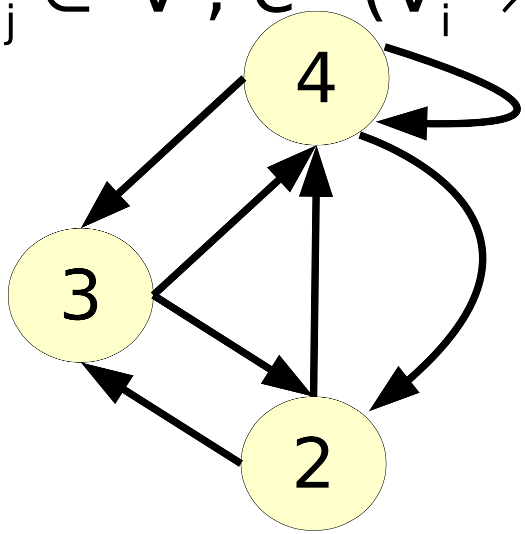


- **Subgraph**
  G' = (V', E') is a subgraph of de G= (V,E) iff
  $V' \subseteq V \wedge E' \subseteq E$



- Subgraph G'=(V', E') **generated by** $V' \subseteq V$:
  $E' = \{e \in E \mid \exists V_i, V_j \in V', e=(V_i \rightarrow V_j) \textit{ or } e=(V_j \rightarrow V_i)\}$

  $V' = \{V_2, V_3, V_4\}$



Graphes
Terminology

- **Directed path in G = (V, E)**
  $C = e_1, e_2, \ldots, e_m$ with

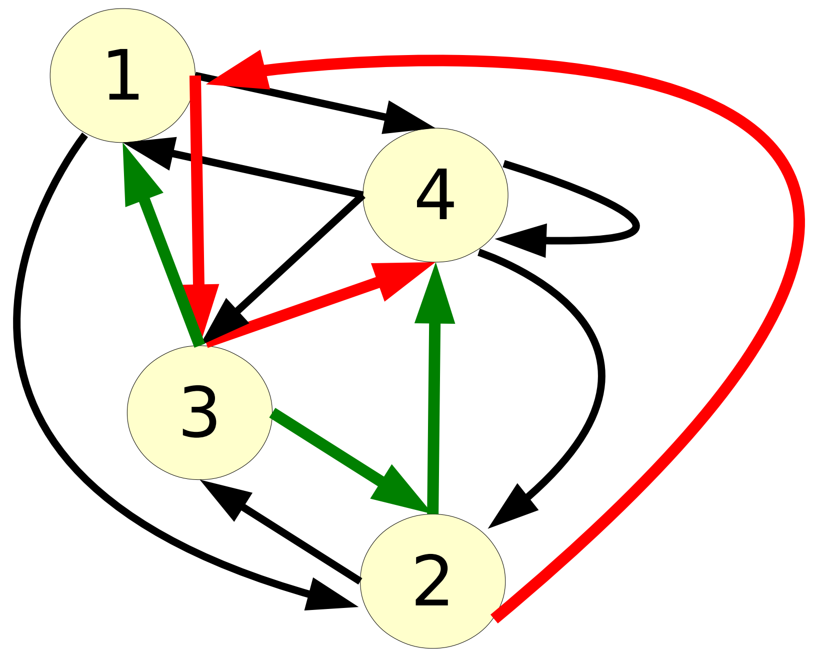
  $e_1, e_2, \ldots, e_m \in E$
  is a directed path *iff*

  $\forall a \in]1..m]$,
  $tail(e_a) = head(e_{a-1})$

- **Path in G = (X, U)**
  $C = e_1, e_2, \ldots, e_m$ with

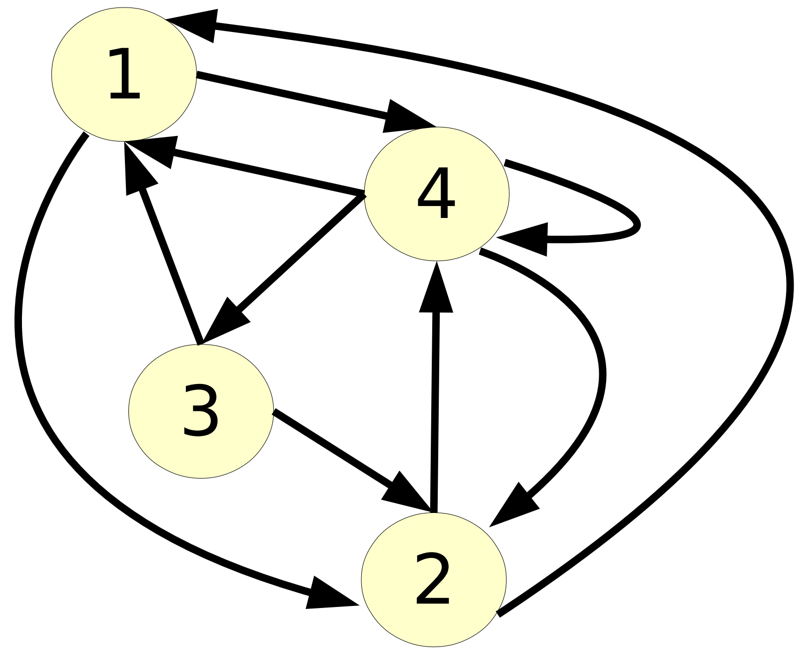
  $e_1, e_2, \ldots, e_m \in E$
  is a path *iff*

  $\forall a \in]1..m]$, e_a and e_{a-1} are adjacents



Graphs Implementation

- G = (V, E) in computer memory
- Matrix
  - Node-arc incidence
  - Node-edge incidence
  - adjacency
- Lists
  - Adjacency
  - Incidency
  - Cocycles



	S_1	S_2	S_3	S_4
S_1	1	0	0	1
S_2	0	1	1	0
S_3	0	0	1	1
S_4	1	1	0	0

	1	2	3	4
A[]	1	3	5	7

	1	2	3	4	5	6	7
B[]	1	3	1	3	2	1	–

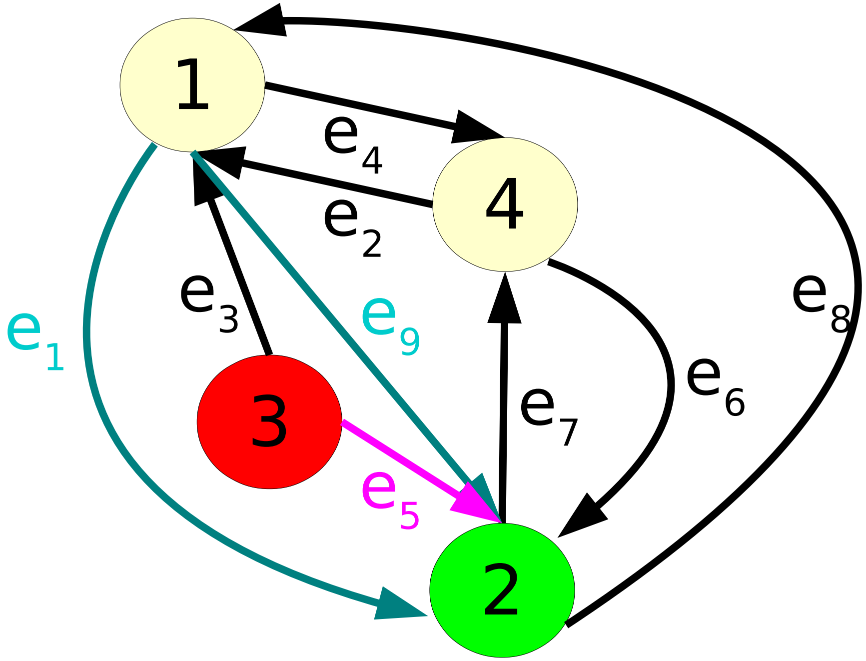
Digraph implementation
Node-arc Matrix

- G = (V, E)

  $\forall\ e = (V_i \rightarrow V_j) \in E$

  $\begin{cases} A_{ie} = 1 \\ A_{je} = -1 \\ \forall\ x \neq i \text{ and } j,\ A_{xe} = 0 \end{cases}$

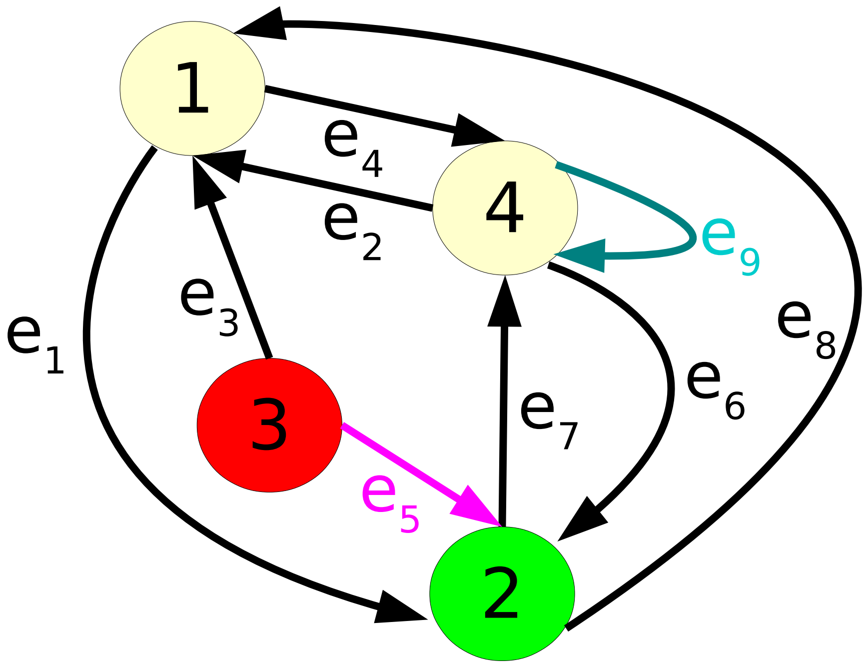
- Loops ?
- Memory footprint
  |X| x |U|
- Node-edge matrix for graphs
  $A_{ie} = A_{je} = 1$



	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
V_1	1	−1	−1	1	0	0	0	−1	1
V2	−1	0	0	0	−1	−1	1	1	−1
V_3	0	0	1	0	1	0	0	0	0
V_4	0	1	0	−1	0	1	−1	0	0

Digraph implementation
Node-node matrix

- $G = (V, E)$
  $\forall\, e = (V_i \rightarrow V_j)$
  $$\begin{cases} e \in E \Rightarrow A_{ij} = 1 \\ e \notin E \Rightarrow A_{ij} = 0 \end{cases}$$



	V_1	V_2	V_3	V_4
V_1	0	1	0	1
V_2	1	0	0	1
V_3	1	1	0	0
V_4	1	1	0	1

- Memory footprint $|X|^2$
- Multi-graphs ?

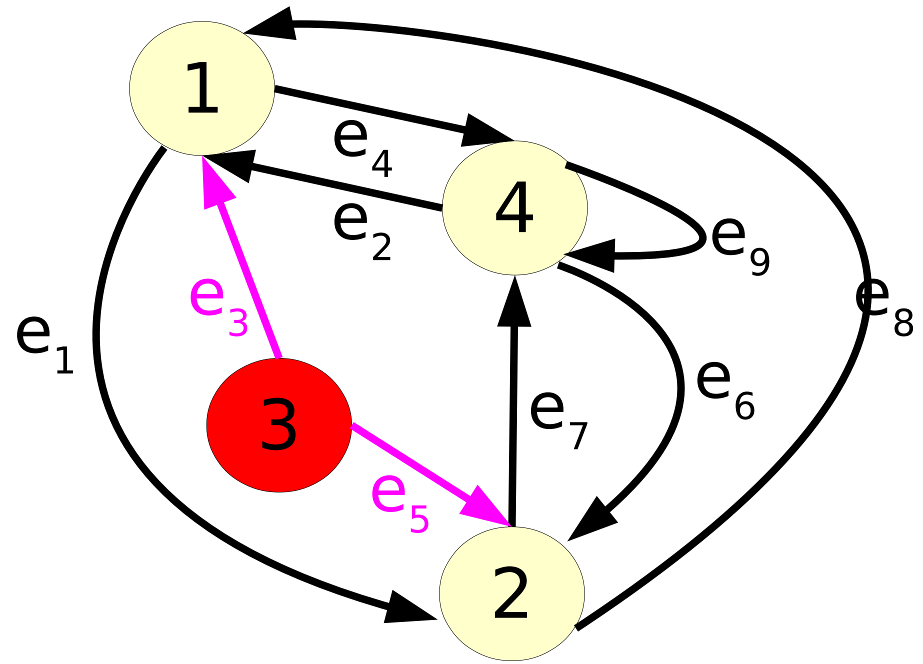
Implementation
Adjacency list

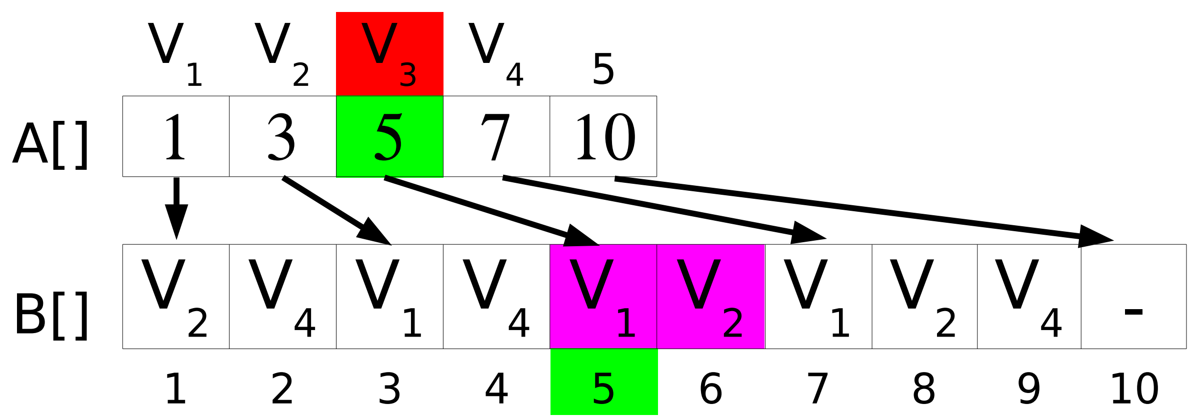
- $G = (V, E)$ (1-graph)
  - 2 arrays A and B
- B[] contains, starting at A[i] index, the list of nodes adjacents to V_i
- Properties :

$d^+(V_i) = A[i+1]-A[i]$

$A[i] = \sum_{j=1}^{i-1} d^+(V_j) + 1$

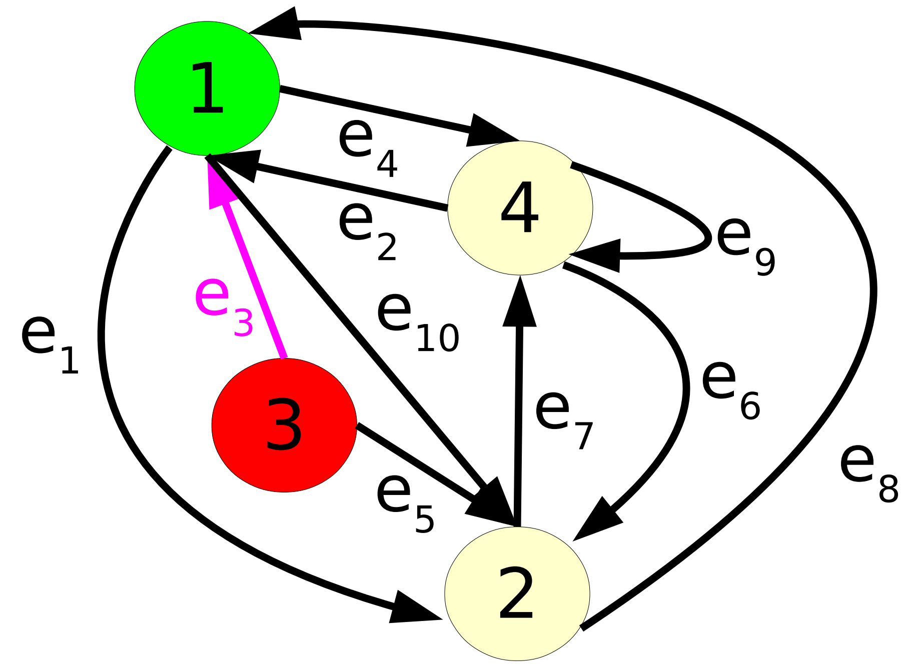
- Memory footprint |X|+|U|





Implantation
Arc List

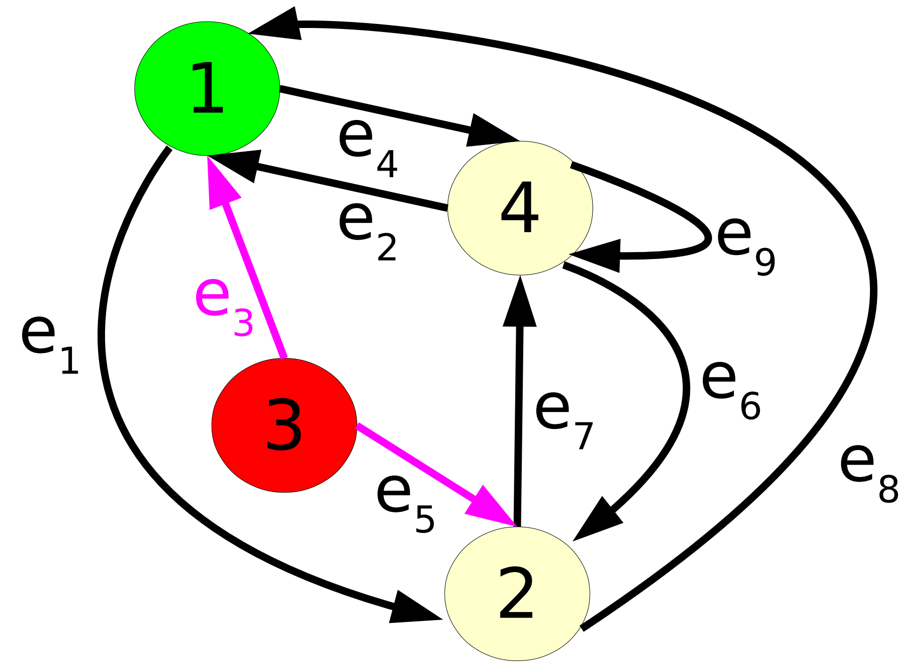
- G = (V, E)
  - 2 arrays T[] and H[]
- T[e] contains the tail of edge e, H[e] the head node of edge e
- Any kind of graph
- Footprint
  2*|E|

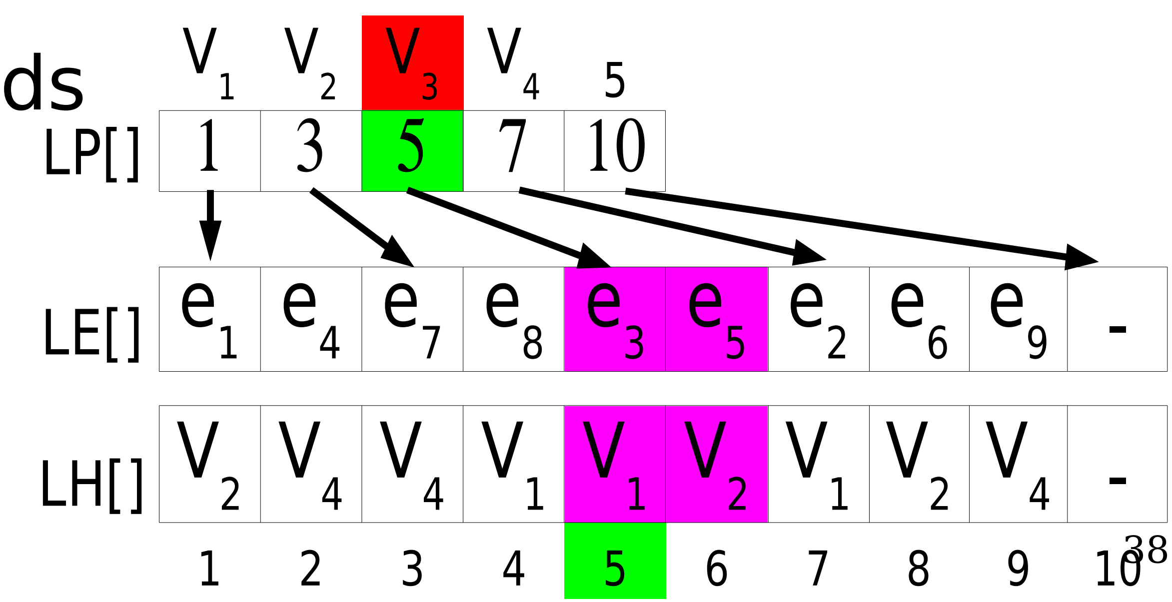


	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}
H[]	1	4	3	1	3	4	2	2	4	2
T[]	2	1	1	4	2	2	4	1	4	1

Implantation
Cocycles list

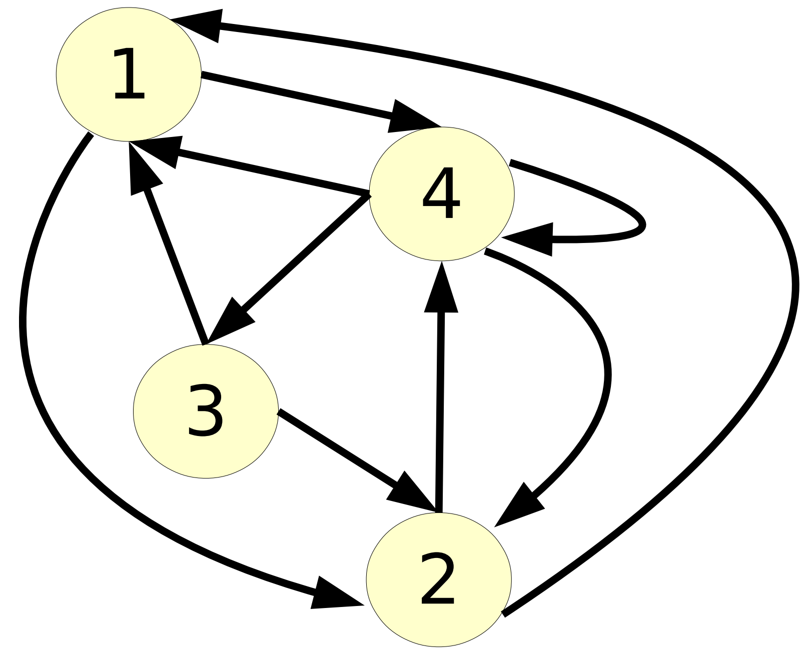
- G = (V, E)
  - 2 arrays LP[] et LE[]
- LE[] contains, starting at index LP[i], the list of edges leaving node V_i (positive cocycle $\Omega^+(V_i)$)
- LH[] contains the associated list of heads
- Footprint
  |X|+2*|U|





Graphes
Design choice

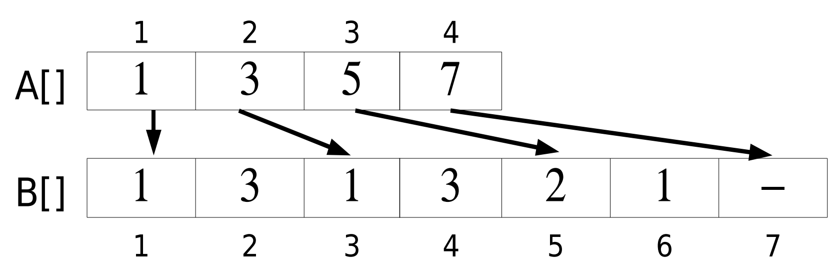
- G = (V, E)
  - Kind of graph
  - Kind of application



- Matrix
  - Huge memory
  - Pre-computed results

	S_1	S_2	S_3	S_4
S_1	1	0	0	1
S_2	0	1	1	0
S_3	0	0	1	1
S_4	1	1	0	0

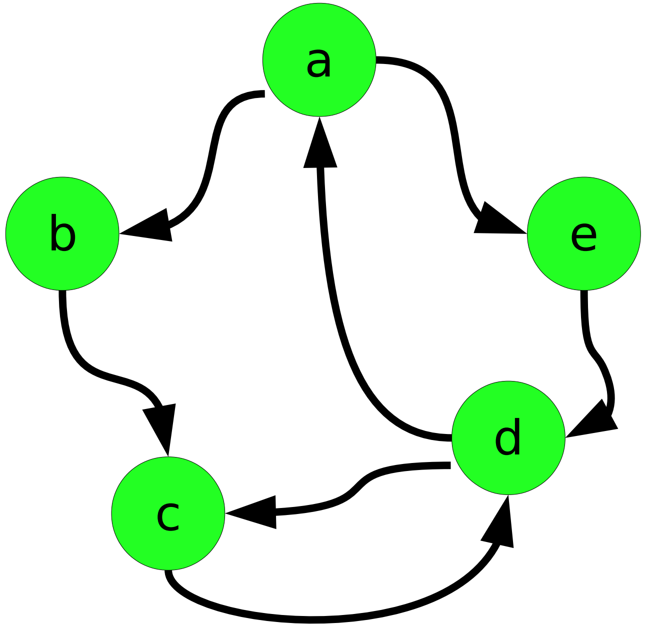
- Lists
  - More computations needed
  - Compact footprints



Graph search

- Data : some graph G = (V, E)
- Goal : How to visit all of the graph's nodes once ?
- Applications : find a data, print the nodes ...

- Example : is there a node named c ?



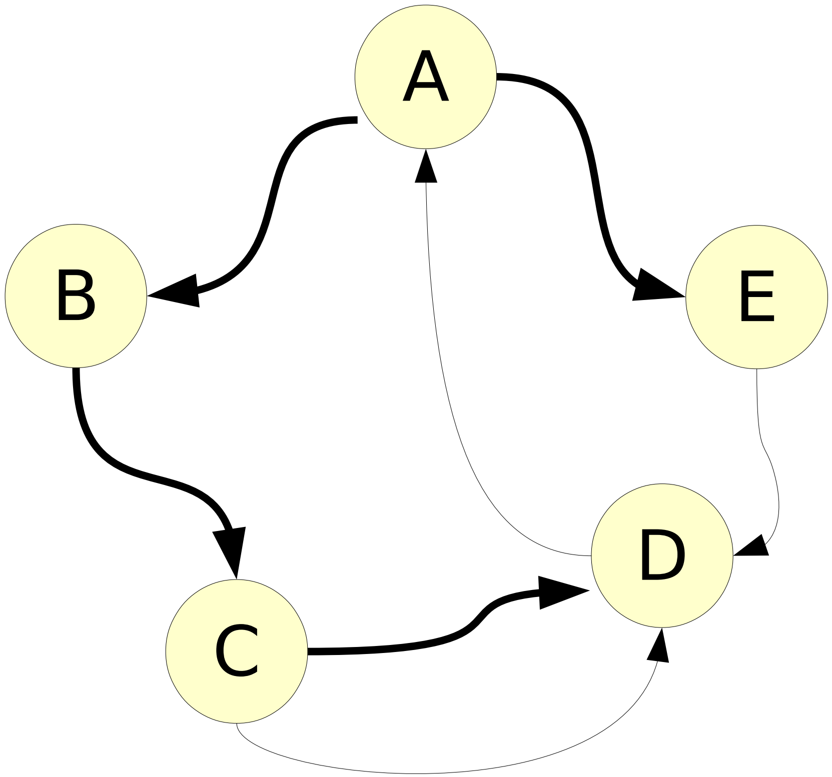
Depth First Search Algorithm

```
DFS(Graph G;  Node V)( )
    Node N
    List adjList
    Si  Open( V, G ) =  true
        Visit( V, G )
        adjList ← ListOfAdjacentNodes( V, G )
        Foreach N in adjList
        Si  Open( N, G ) =  true
            dfs( G, N )
```

Depth First Search Algorithm

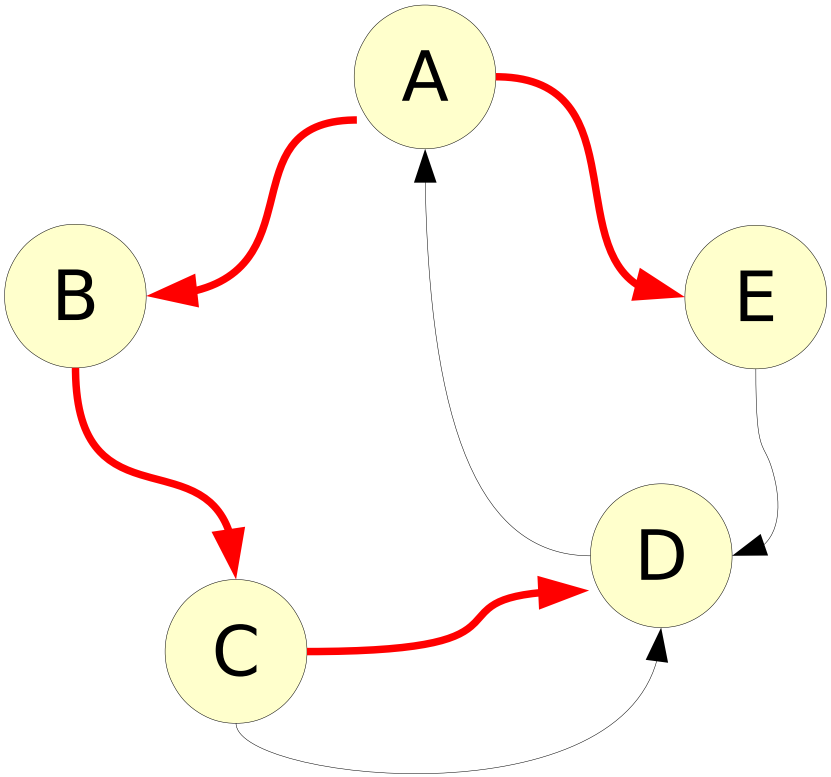
- With subfunctions :
  - Visit(Node V; Graph G)
    does some operation on V and closes it.
  - Open(Node V; Graph G) (boolean)
    returns true if V is open and false if it is closed.

Example



#	Ancestors & current node	adjacent nodes

Example

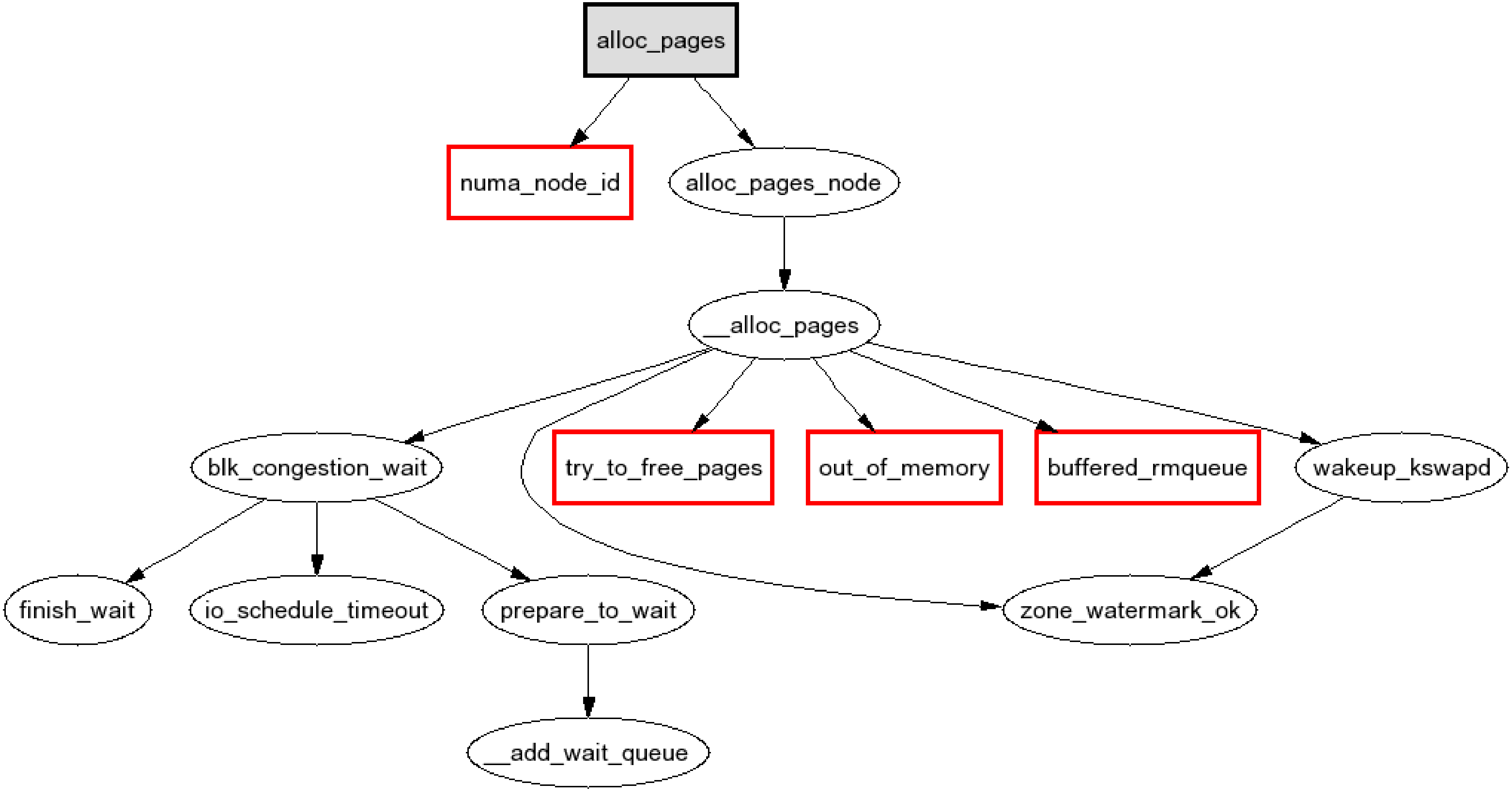


#	Ancetors & current node	adjacent nodes
1	A	B E
2	A B	C
3	A B C	D
4	A B C D	~~A~~
5	A B C	~~D~~
6	A B	~~C~~
7	A	~~B~~ E
8	AE	~~D~~
9	A	~~BE~~
10	-	~~-~~


Spanning tree

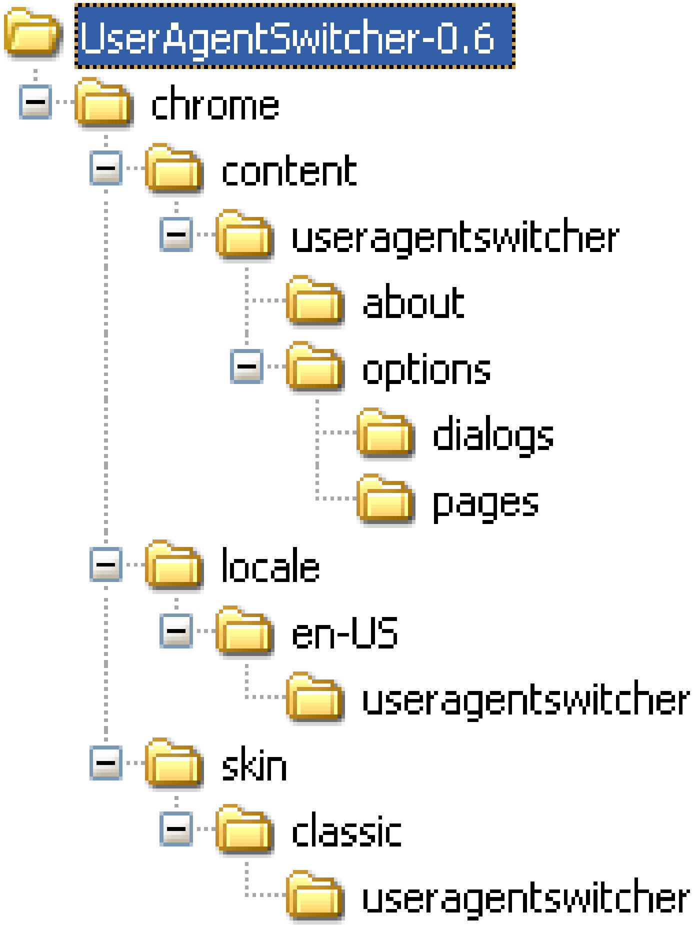
Applications

Is finish_wait called by alloc_pages ?



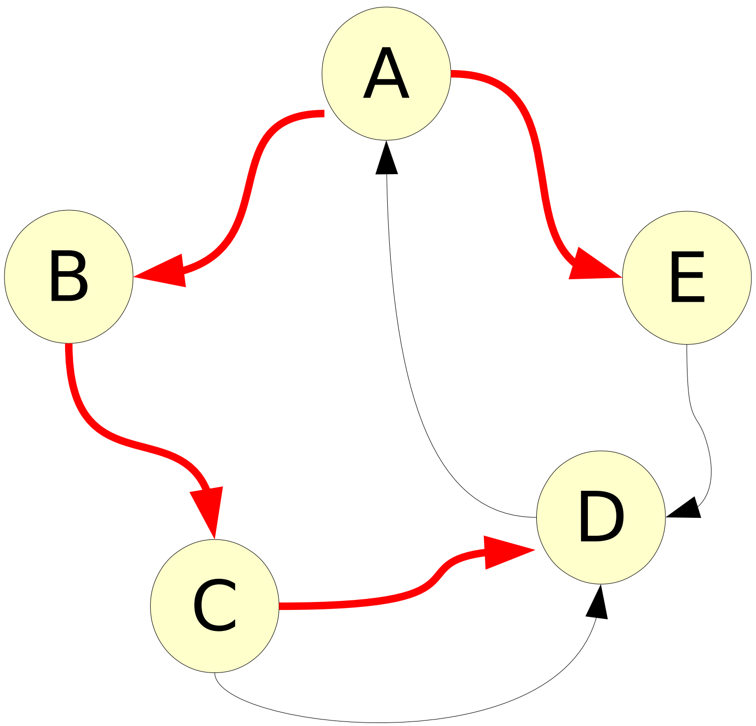
http://www.csn.ul.ie/~mel/projects/codeviz/alloc_pages.png

Applications



Which directories need to be backed up ?

Remarks



- The choice between open nodes is free
  $\rightarrow$ variable visiting order

- If there is no path $V_0 \rightarrow V$, then V is not visited
  $\rightarrow$ connected component

- Spanning tree 

- Exploring adjacent nodes before successor nodes
  $\rightarrow$ breadth first walk

Breadth first search

- Same node marking principle as for depth first algorithm
- You visit V_0 then all of its adjacent open nodes. And next the open nodes of these, *etc*.
- Applications : data mining, nodes printing ...

- Stack for yet unexplorated nodes

Breadth first Search algorithm

BFS(Graph G; Node V_0)( )

Node N

Stack S

List adjList

push(V_0, S)

visit(V_0, G)

While not empty(S)

pop(N, S)

adjList ← listOfAdjNodes(N, G)

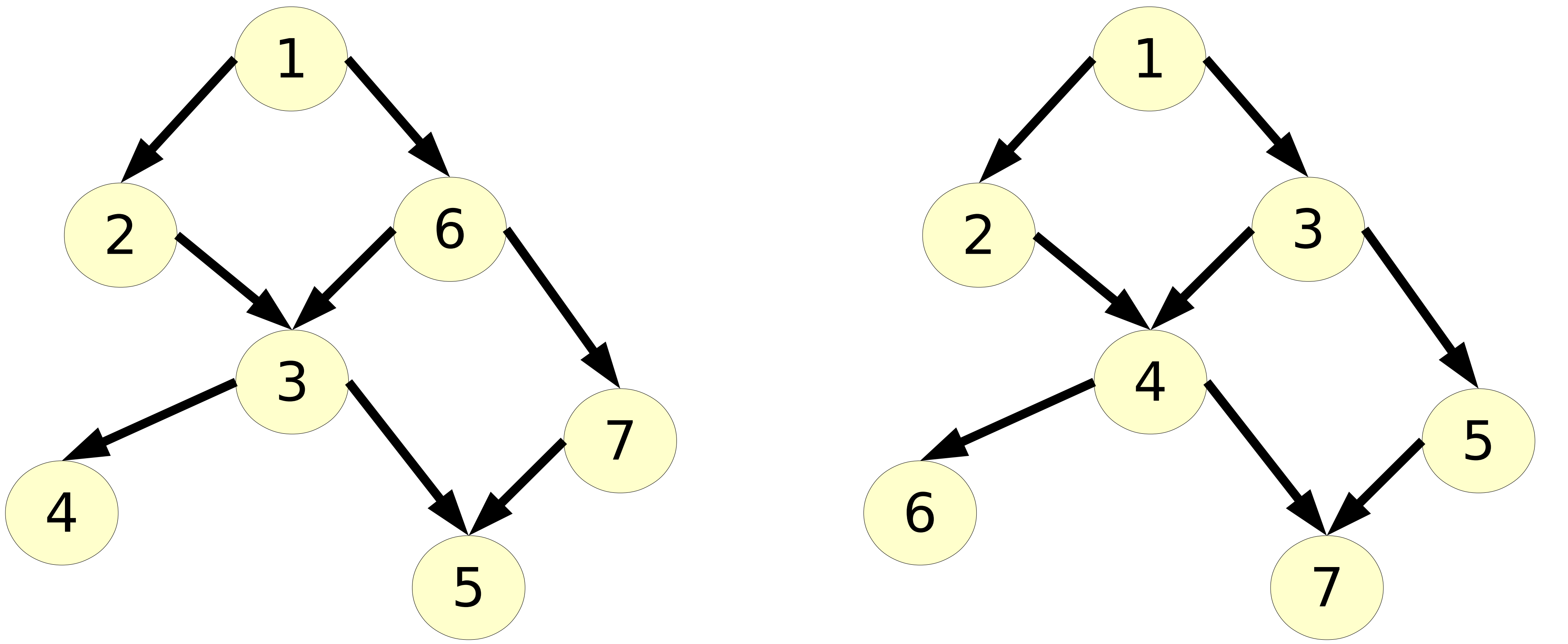
Foreach N in adjList

If open(N, G) = true

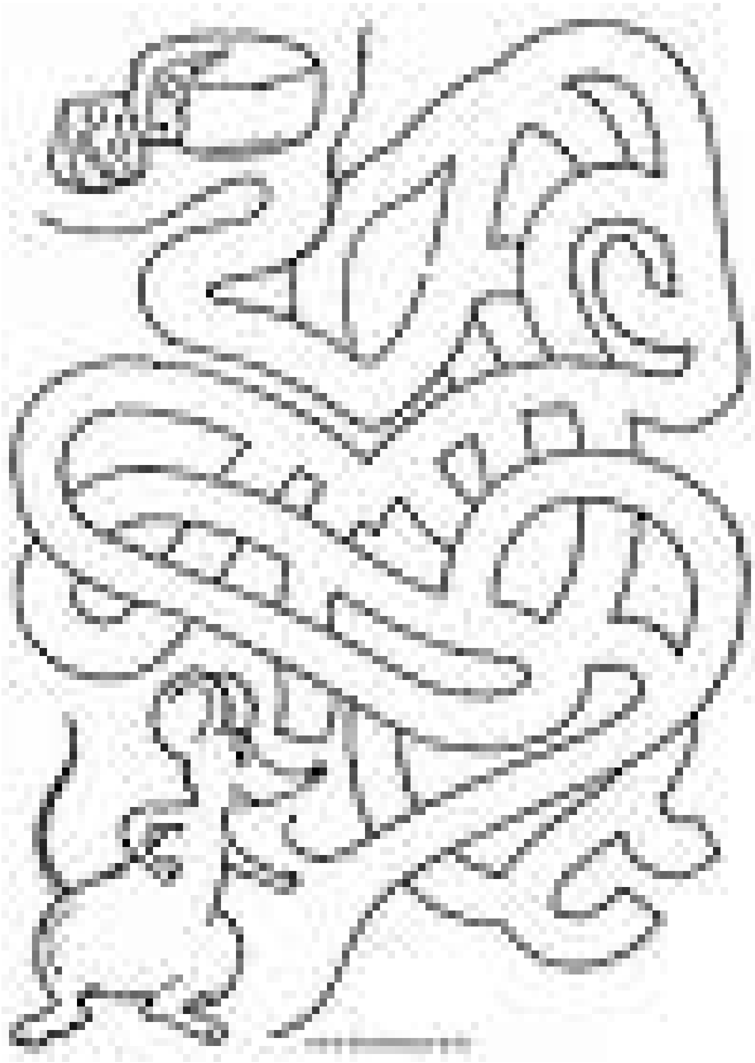
visit(N, G)

push(N, S)

Breadth *vs*. Depth First Search



Connected components



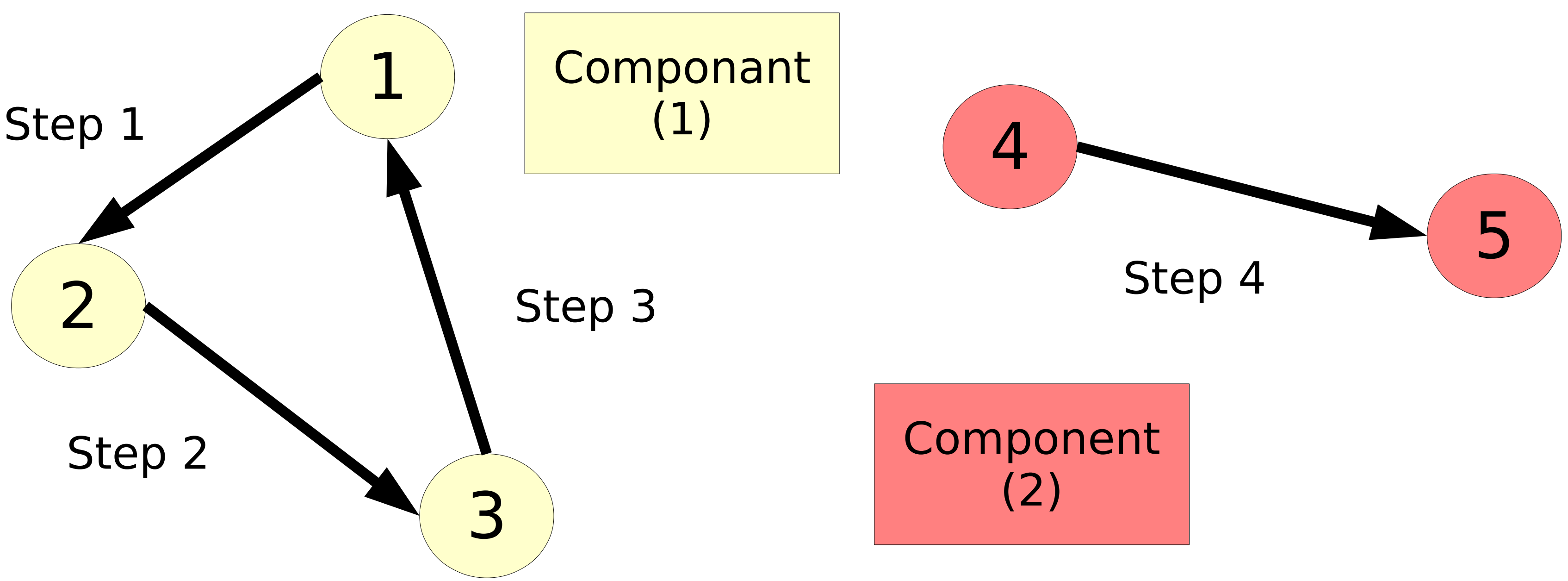
www.atelier-duotang.com

- Is there a path between i and j ?
  $\rightarrow$ i and j are in the same *connected component*

- All nodes are connected
  $\rightarrow$ *connected graph*

Connected components
Principle

1. DFS starting at S_0
2. If still unvisited nodes, DFS again starting at one of these nodes

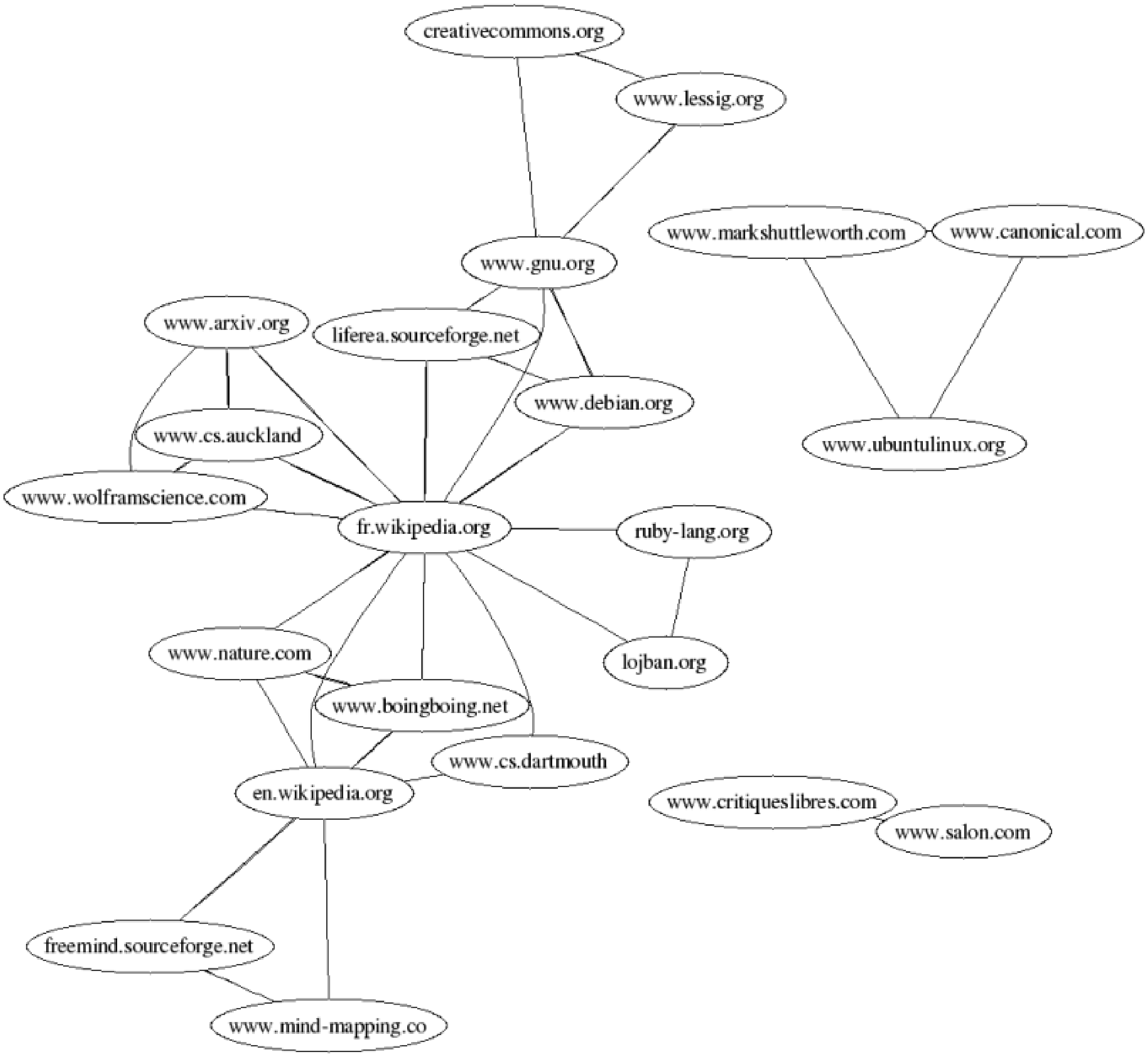


Connected components - Algorithm

```
Connex(Graph G)(node List)
    Node N
    List L ← nodes(G)
    NC ← 0
    Foreach N in L
        If open(N, G) = true
        NC ← NC+1
        DFS2(G, N, NC)
    return L
```

```
DFS2(Graph G;  Node V0 ,
  integer NC)
    ...
    Visit(N, G, NC)
    ...
        DFS2(G, N, NC)
```

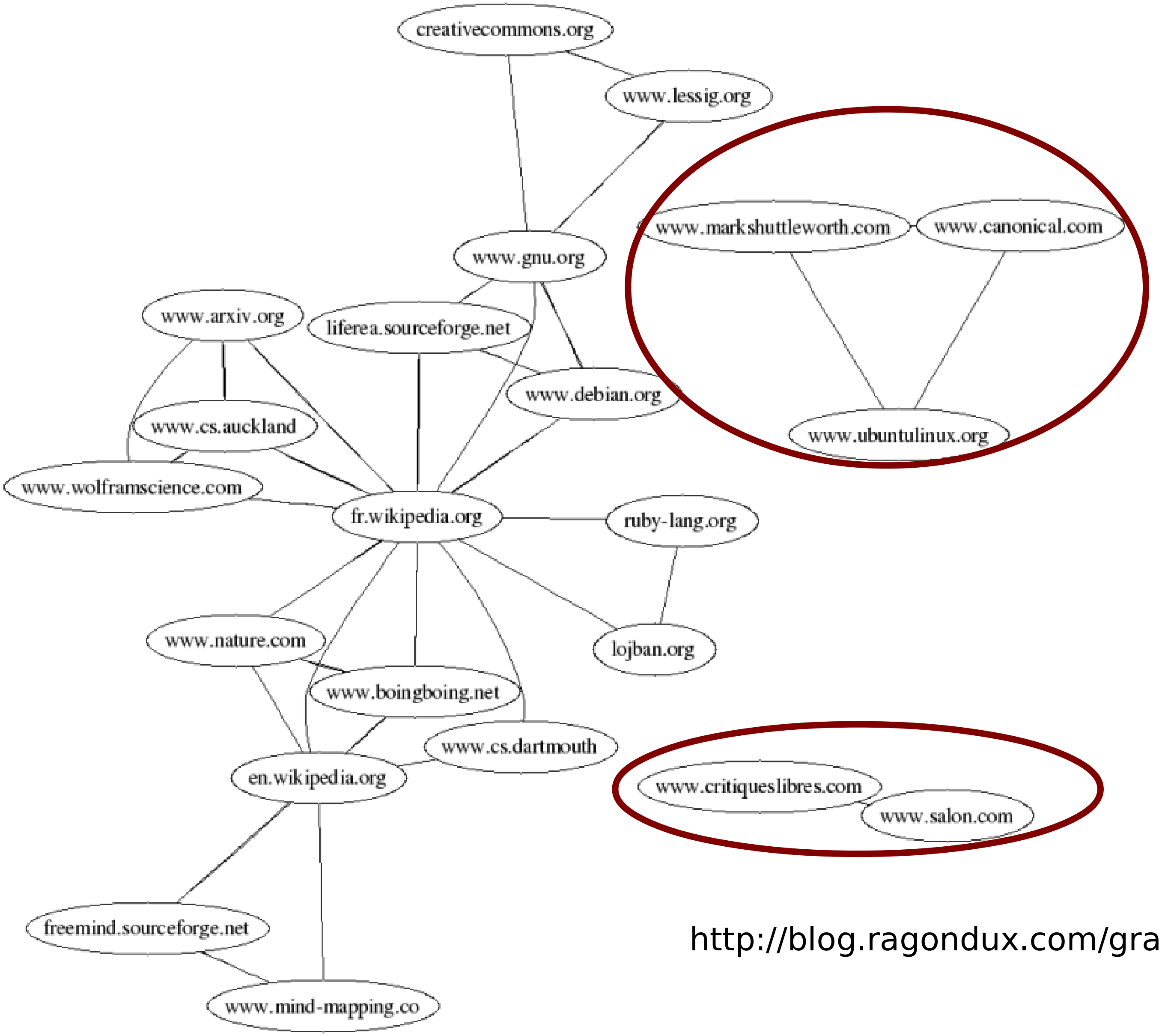
Exercice



http://blog.ragondux.com/graphes

Exercice

- The graph isn't connected



http://blog.ragondux.com/graphes

Strongly connected components

- **Strongly** connected component C $\Leftrightarrow$
  $\forall$i, j $\in$ C, $\exists$ **directed** path (i$\rightarrow$j)

- $\neq$ connected component $\Leftrightarrow$
  $\forall$i, j $\in$ C, $\exists$ path (i$\leftrightarrow$j) (chain, undirected)

- For a given node V_0 d, the associated strongly connected component is defined by :
  X set s.t.
  - (1) $\forall$V $\in$ X, directedPath($V_0 \rightarrow$V)
  - (2) $\forall$V $\in$ X, directedPath(V$\rightarrow V_0$)

Strongly CC - Algorithm

- Directed path search algorithms :
  - (1) ⇒ algorithm based on **successors**
  - (2) ⇒ algorithme based on **predecessors**

(1)/(2)

DFSSuccs(Graph G; Node V_0)

... adjacents ← DirectSucessors(V, G) ...

- Algorithm :

StrongCC(Graph G, Node V_0)(List)

$X_1 \leftarrow$ DFSSuccs(G, V_0)

$X_2 \leftarrow$ DFSPreds(G, V_0)

return $X_1 \cap X_2$

Strongly CC - Example

1. DFSSuccs(G, V_1)

   $\rightarrow X_1 = \{V_1, V_2, V_3, V_4, V_5\}$

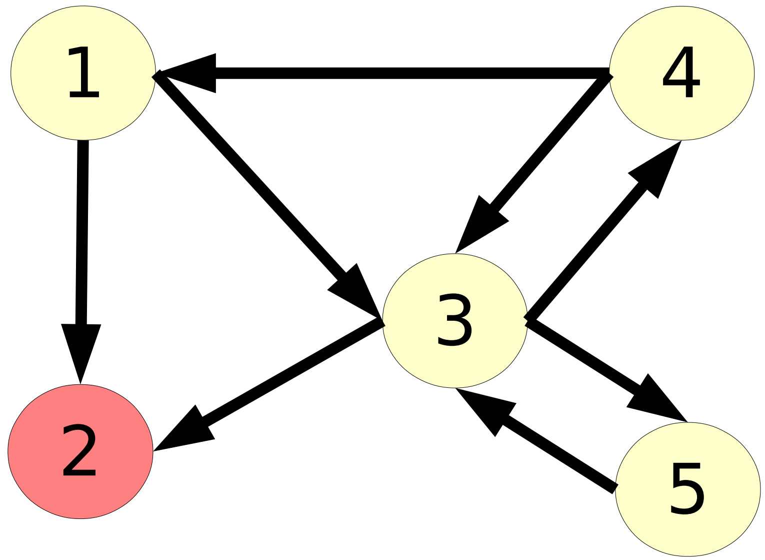
2. DFSPreds(G, V_1)

   $\rightarrow X_2 = \{V_1, V_3, V_4, V_5\}$

Strongly CC

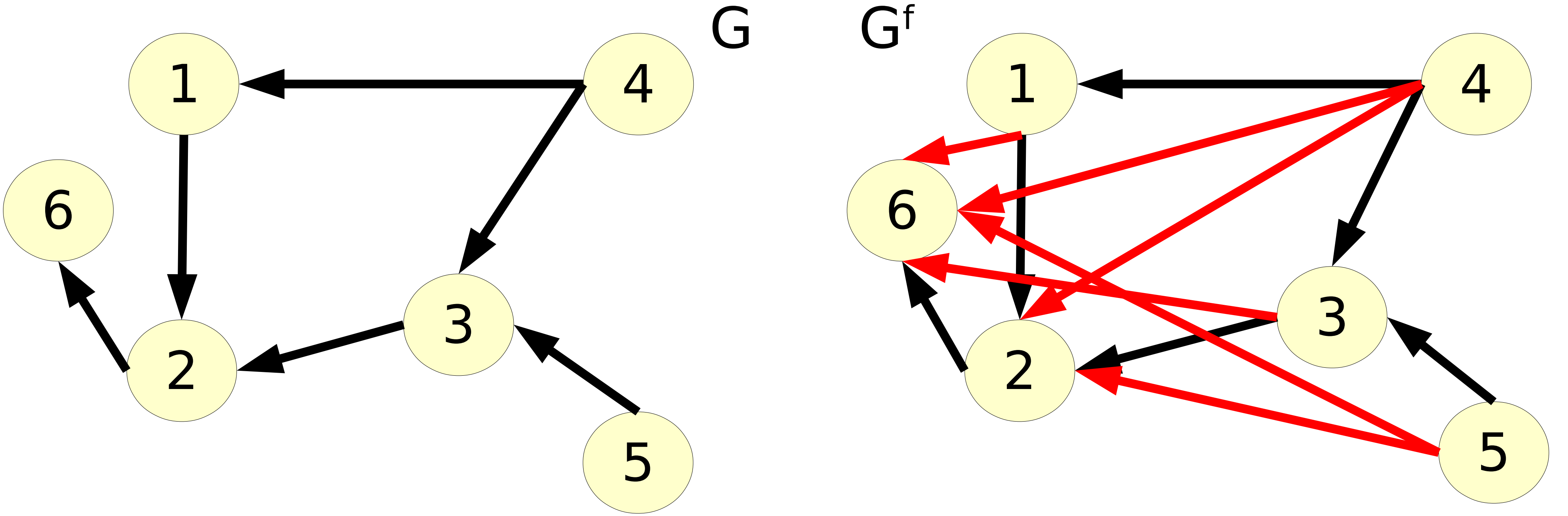
associated to V_1

$X_1 \cap X_2 = \{V_1, V_3, V_4, V_5\}$



Transitive closure

- For graph G=(V, E), graph G^f = (V, E') s.t .
  $E' = \{e'=(S_i \rightarrow S_j) \in E' \mid \exists$ dipath $V_i \rightarrow V_j$ in G$\}$



- If the graph is strongly connected, its transitive closure is a complete graph

Transitive closure
Roy-Warshall algorithm

- Idea : from G = {V, E}, add iteratively edges $V_i \rightarrow V_j$, if $V_i \rightarrow V_k$ and $V_k \rightarrow V_j$ exist
- Works with the adjacency matrix

RoyWarshall(Graph G=(V, E))(graph G_f)

1- Init

$M[i, j] \leftarrow$ true *iff* edge $V_i \rightarrow V_j \in G$

2- Step

$\forall\ V_i \in V, \forall\ V_j \in V, \forall\ V_k \in V$

$M[i, j] \leftarrow M[i, j] \vee (M[i, k] \wedge M[k, j])$

Exercice

- Can we reach each station despite the work in progress ?