LDD2 PROJECT

BOOLEAN CIRCUITS

ESTEBAN CASTRO, based on the course by RENAUD VILMART

2023/2024

Project organisation

Organisation

- *Tutorials : Esteban Castro*
  - *esteban.castro-ruiz@universite-paris-saclay.fr*
  - *office 67, building 650*
- *Tutorials (TD)*
- *documents posted on the course website project carried out in*
- *project realised in pairs/groups of three*
- *marking :*
  - *project progress (50%) see next slide*
  - *final project status (50%)*

PROJECT PROGRESS

- *project in TDs, objectives to achieve, to finish afterwards if needed*
- ***PROJECT**: tutorials depend on previous ones ⇒ make sure you finish them as homework*
- *project progress mark: 2 tutorials selected, to be handed in before the next session.*
- *ask me questions if you get stuck.*

The following will be taken into account:

- *Code readability*
- *The use of comments*
- *The presence of tests when requested*

Objectives

- *Become more comfortable with programming*
- *Know how to work on a (relatively) long project*
- *Know how to use tools dedicated to the project*
- *Acquire a certain rigour*
- *Learn about graphs and boolean circuits*

Presentation of the project

TECHNOLOGIES USED AND EXPECTATIONS

Python 3

The following environments:

- *git (github, gitlab)*
- 1. *text editor + terminal or*
  2. *IDE (Spyder3, PyCharm, etc.)*

TECHNOLOGIES USED AND EXPECTATIONS

Python 3

- *The following environments:*
- *git (github, gitlab)*
    1. *text editor + terminal or*
    2. *IDE (Spyder3, PyCharm, etc.)*

Insist:

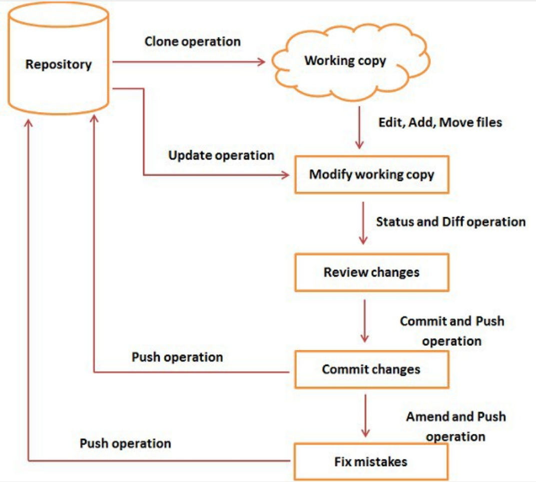
- *clear, readable code*
    - *meaningful variable/function names...*
- *comments*
    - *parameters and function/method outputs*
    - *explanation of non-intuitive code (imagine that someone who hasn't participated at all in the project has to appropriate the code)*
- *architecture and compartmentalisation (breaking up big problems into smaller ones)*

Architecture

on github/lab

```
Project-LDD2-S4
├─ worksheet.py
├─ modules
│   ├─ concept1.py
│   └─ conceptn.py
└─ tests
    ├─ concept1_test.py
    └─ conceptn_test.py
```

USING GIT



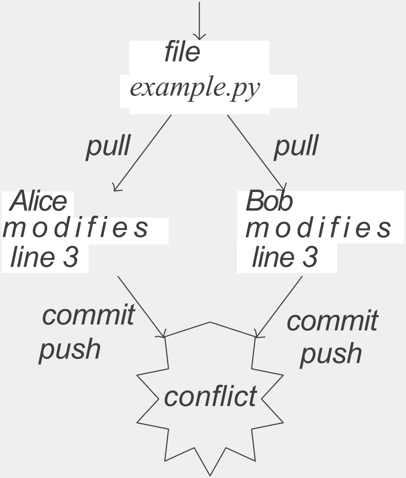
Create a private repo (per group) on GitHub/GitLab. Invite other members of the group.

```
git clone <url from git repo> # creates a local version
                              # of the repo
git pull # updates the local version
git diff # displays what hasn't been committed yet
git add <files> # tells git which
                # commiter files
git commit -m "<message>" # commit changes
git log # list of commits
git amend # changes the last commit
git push origin master # apply changes
                       # of the commit
```

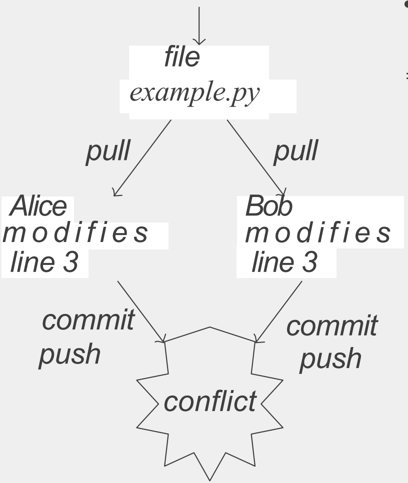
Git is more complete than that (see branches). Doc : https://git-scm.com/docs


KEEP
CALM
AND
CRUSH
CONFLICTS

Conflicts in Git



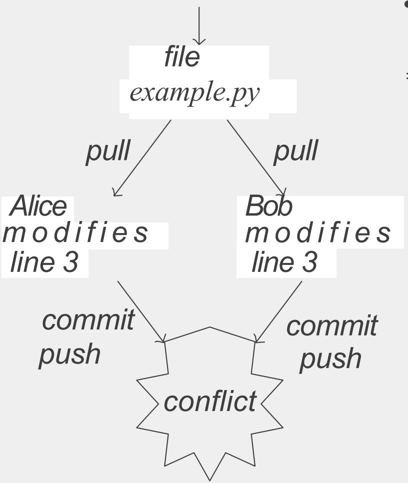
Conflicts in Git



- A *conflict* arises when two people have modified the same piece of code "at the same time".

$\Rightarrow$ *Git is unable to determine which version to keep.*

CONFLICTS IN GIT

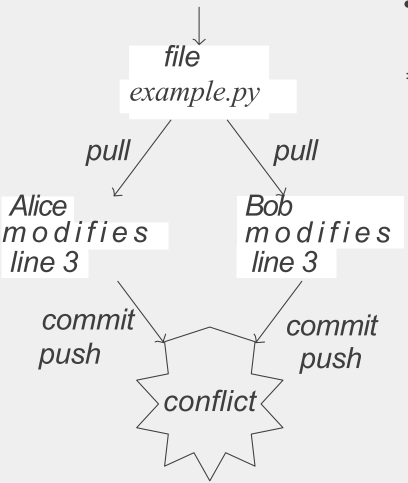


- A *conflict* arises when two people have modified the same piece of code "at the same time".

$\Rightarrow$ *Git is unable to determine which version to keep.*

```
CONFLICT (content): merge conflict in example.py
```

Conflicts in Git



- A *conflict* arises when two people have modified the same piece of code "at the same time".

⇒ *Git is unable to determine which version to keep.*

```
CONFLICT (content): merge conflict in example.py
```

conflicting lines in the :

```
<<<<<<< HEAD
master branch line(s)
=======
line(s) of the conflicting branch
>>>>>>> conflict_branch
```

Once the conflicts have been resolved, you need to re-commit.

KEY CONCEPTS FOR THE PROJECT

- *Logic can be programmed*

  ↓ *can be physically implemented by*

- *Boolean circuits*

  ↓ *represented by*

- *Directed, open and acyclic graphs*

PROPOSITIONAL LOGIC

LOGIC IN GENERAL

- *a research field in its own right, at the intersection of maths and computing*
there are many different ways of thinking

LOGIC IN GENERAL

- *a research field in its own right, at the intersection of maths and computing*

there are many different ways of thinking

- *to express mathematical problems, theorems and reasoning in a way that is easy to understand.*

LOGIC IN GENERAL

- *a research field in its own right, at the intersection of maths and computing*

there are many different ways of thinking

- *to express mathematical problems, theorems and reasoning in a way that is easy to understand.*
- *very formal $\Rightarrow$ "computerisation" of maths*

LOGIC IN GENERAL

- *a research field in its own right, at the intersection of maths and computing*

there are many different ways of thinking

- *to express mathematical problems, theorems and reasoning in a way that is easy to understand.*
- *very formal $\Rightarrow$ "computerisation" of maths*
  - *you can let a computer check that a line of reasoning is true*

LOGIC IN GENERAL

- *a research field in its own right, at the intersection of maths and computing*

there are many different ways of thinking

- *to express mathematical problems, theorems and reasoning in a way that is easy to understand.*
- *very formal $\Rightarrow$ "computerisation" of maths*
  - *you can let a computer check that a line of reasoning is true*
  - *the computer can be made to search for a mathematical reasoning that shows P knowing Q*

LOGIC IN GENERAL

- *a research field in its own right, at the intersection of maths and computing*

there are many different ways of thinking

- *to express mathematical problems, theorems and reasoning in a way that is easy to understand.*
- *very formal* $\Rightarrow$ *"computerisation" of maths*
  - *you can let a computer check that a line of reasoning is true*
  - *the computer can be made to search for a mathematical reasoning that shows* P *knowing* Q
  - *mathematical applications*
    - *connect 4 theorem*
    - *4 colour theorem*
    - *classification of pentagonal tessellations*
    - …

LOGIC IN GENERAL

- *a research field in its own right, at the intersection of maths and computing*

there are many different ways of thinking

- *to express mathematical problems, theorems and reasoning in a way that is easy to understand.*
- *very formal* $\Rightarrow$ *"computerisation" of maths*
  - *you can let a computer check that a line of reasoning is true*
  - *the computer can be made to search for a mathematical reasoning that shows* P *knowing* Q
  - *mathematical applications*
    - *connect 4 theorem*
    - *4 colour theorem*
    - *classification of pentagonal tessellations*
    - $\ldots$
  - *industrial applications*
    - *verification of embedded software (aircraft, certain metro lines, etc.)*
    - *verification of compilers*
    - *...*

LOGIC IN GENERAL

- *a research field in its own right, at the intersection of maths and computing*

there are many different ways of thinking

- *to express mathematical problems, theorems and reasoning in a way that is easy to understand.*
- *very formal $\Rightarrow$ "computerisation" of maths*
  - *you can let a computer check that a line of reasoning is true*
  - *the computer can be made to search for a mathematical reasoning that shows P knowing Q*
  - *mathematical applications*
    - *connect 4 theorem*
    - *4 colour theorem*
    - *classification of pentagonal tessellations*
    - …
  - *industrial applications*
    - *verification of embedded software (aircraft, certain metro lines, etc.)*
    - *verification of compilers*
    - ...
- *provides a good model for the software/hardware interface*

PROPOSITIONAL FORMULAS

- *2 constants 0 and 1 (or False and True, $\perp$ and T, ...) of propositional variables*

PROPOSITIONAL FORMULAS

- *2 constants 0 and 1 (or False and True, $\perp$ and T, ...) of propositional variables*
- x_0 , x_1 , …

PROPOSITIONAL FORMULAS

- *2 constants 0 and 1 (or False and True, $\perp$ and T, ...) of propositional variables*
- x_0 , x_1 , ...
- *logical connectors ~, &, |*

PROPOSITIONAL FORMULAS

- *2 constants 0 and 1 (or False and True, $\perp$ and T, ...) of propositional variables*
- x_0 , x_1 , ...
- *logical connectors ~, &, |*
- *propositional formulae (PF) :*

PROPOSITIONAL FORMULAS

- 2 constants 0 and 1 (or False and True, $\perp$ and T, ...) of propositional variables
- x_0 , x_1 , ...
- logical connectors ~, &, |
- propositional formulae (PF) :
  - 0 and 1 are PF
  - propositional variables x_i are PFs
  - if P and Q are PFs, then :

PROPOSITIONAL FORMULAS

- *2 constants 0 and 1 (or False and True, $\perp$ and T, ...) of propositional variables*
- x_0 , x_1 , …
- *logical connectors ~, &, |*
- *propositional formulae (PF) :*
  - *0 and 1 are PF*
  - *propositional variables x_i are PFs*
  - *if P and Q are PFs, then :*
    - $\sim P$ *is an FP (named "not P")*
    - $P \& Q$ *is an FP (named "P and Q")*
    - $P \mid Q$ is an IF (called "P or Q")

PROPOSITIONAL FORMULAS

- *2 constants 0 and 1 (or False and True, $\perp$ and T, ...) of propositional variables*
- x_0 , x_1 , …
- *logical connectors ~, &, |*
- *propositional formulae (PF) :*
  - *0 and 1 are PF*
  - *propositional variables x_i are PFs*
  - *if P and Q are PFs, then :*
    - *$\sim P$ is an PF (named “not P")*
    - *$P \& Q$ is an PF (named "P and Q")*
    - $P \mid Q$ is an PF (called "P or Q")

Examples: Propositional formulae				
1	x_0	$\sim x_0$	$1\&(\sim x_0)$	$(x_0 \& x_1 \mid (\sim (x_1 \mid x_2))$

evaluation OF PROPOSITIONAL FORMULAE

we denote Var(P) the propositional variables in the formula P

Example

Var $((x_0 \mathbin{\&} x_1) | (\sim (x_1 \mid x_2)) = \{x_0, x_1, x_2\}$

evaluation OF PROPOSITIONAL FORMULAE

we denote Var(P) the propositional variables in the formula P

Example

Var $((x_0 \,\&\, x_1)|(\sim (x_1 \mid x_2)) = \{x_0, x_1, x_2\}$

An evaluation ρ of P is a function ρ : Var(P) → {0, 1}. It induces an evaluation of P.

Example

With ρ : $\begin{cases} x_0 \mapsto 1 \\ x_1 \mapsto 1 \\ x_2 \mapsto 0 \end{cases}$ *the previous formula becomes (1&1)|(~ (1 | 0)).*

evaluation OF PROPOSITIONAL FORMULAE

we denote Var(P) the propositional variables in the formula P

Example

Var ((x_0 & x_1)|(~ (x_1 | x_2)) = {x_0 , x_1 , x_2}

An evaluation ρ of P is a function ρ : Var(P) → {0, 1}. It induces an evaluation of P.

Example

With ρ : $\begin{cases} x_0 \mapsto 1 \\ x_1 \mapsto 1 \\ x_2 \mapsto 0 \end{cases}$ *the previous formula becomes (1&1)|(~ (1 | 0)).*

With n propositional variables, we have 2^n possible evaluations.

EVALUATION AND TRUTH TABLES

P	$\sim P$
0	1
1	0

P	Q	$P \& Q$
0	0	0
0	1	0
1	0	0
1	1	1

P	Q	$P \mid Q$
0	0	0
0	1	1
1	0	1
1	1	1

EVALUATION AND TRUTH TABLES

P	$\sim P$
0	1
1	0

P	Q	$P \& Q$
0	0	0
0	1	0
1	0	0
1	1	1

P	Q	$P \mid Q$
0	0	0
0	1	1
1	0	1
1	1	1

Example

$(1\&1)|(\sim (1 \mid 0)) = 1|(\sim 1) = 1|0 = 1$

EVALUATION AND TRUTH TABLES

P	$\sim P$
0	1
1	0

P	Q	$P\&Q$
0	0	0
0	1	0
1	0	0
1	1	1

P	Q	$P \mid Q$
0	0	0
0	1	1
1	0	1
1	1	1

Example

(1&1)|(~ (1 | 0)) = 1|(~ 1) = 1|0 = 1

Can you do more than "and", "or", "not"?

EVALUATION AND TRUTH TABLES

P	$\sim P$
0	1
1	0

P	Q	$P\&Q$
0	0	0
0	1	0
1	0	0
1	1	1

P	Q	$P \mid Q$
0	0	0
0	1	1
1	0	1
1	1	1

Example

(1&1)|(~ (1 | 0)) = 1|(~ 1) = 1|0 = 1

Can you do more than "and", "or", "not"?

Example

$(\sim P) \mid Q$ and the logical implication $P \to Q$ have the same evaluation whatever the evaluations of P and Q.

Aside: Satisfiability

Satisfiability (SAT)

Given P a PF, find an evaluation of Var(P) which evaluates P to 1.

Aside: Satisfiability

Satisfiability (SAT)

Given P a PF, find an evaluation of Var(P) which evaluates P to 1.

- *complicated problem*

ASIDE: SATISFIABILITY

Satisfiability (SAT)

Given P a PF, find an evaluation of Var(P) which evaluates P to 1.

- *complicated problem*
- *find an "efficient" algorithm (polynomial complexity) to solve it $\Rightarrow$ $1 million*

ASIDE: SATISFIABILITY

Satisfiability (SAT)

Given P a PF, find an evaluation of Var(P) which evaluates P to 1.

- *complicated problem*
- *find an "efficient" algorithm (polynomial complexity) to solve it $\Rightarrow$ \$1 million*
- *solves a huge number of problems (e.g. Sudoku, Hamiltonian path in a graph, etc.)*

Aside: Satisfiability

Satisfiability (SAT)

Given P a PF, find an evaluation of Var(P) which evaluates P to 1.

- *complicated problem*
- *find an "efficient" algorithm (polynomial complexity) to solve it $\Rightarrow$ \$1 million*
- *solves a huge number of problems (e.g. Sudoku, Hamiltonian path in a graph, etc.)*
- *applications in the automation of evidence*

Aside: Satisfiability

Satisfiability (SAT)

Given P a PF, find an evaluation of Var(P) which evaluates P to 1.

- *complicated problem*
- *find an "efficient" algorithm (polynomial complexity) to solve it $\Rightarrow$ \$1 million*
- *solves a huge number of problems (e.g. Sudoku, Hamiltonian path in a graph, etc.)*
- *applications in the automation of evidence*
- *major research on the subject*

ASIDE: SATISFIABILITY

Satisfiability (SAT)

Given P a PF, find an evaluation of Var(P) which evaluates P to 1.

- *complicated problem*
- *find an "efficient" algorithm (polynomial complexity) to solve it $\Rightarrow$ \$1 million*
- *solves a huge number of problems (e.g. Sudoku, Hamiltonian path in a graph, etc.)*
- *applications in the automation of evidence*
- *major research on the subject*
- *brute-force: exponential algorithm (tests all possible evaluations) e.g. Sudoku: 324 propositional variables $\Rightarrow 2^{324} \approx$ 3, 4.10^{97} evaluations*

Equivalence of propositional formulas

Equivalence

Two FPs P and Q which have the same variables are equivalent ($P \equiv Q$) if any evaluation of the variables gives the same result in P and Q, i.e. if the truth tables of P and Q are the same.

EQUIVALENCE OF PROPOSITIONAL FORMULAS

Equivalence

Two FPs P and Q which have the same variables are equivalent ($P \equiv Q$) if any evaluation of the variables gives the same result in P and Q, i.e. if the truth tables of P and Q are the same.

Examples

- $\sim (P \& Q) \equiv (\sim P) \mid (\sim Q)$
- $P \& (Q \mid R) \equiv (P \& Q) \mid (P \& R)$

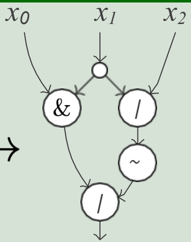
Boolean circuits

Boolean circuits, for example

Informally: representation of a propositional formula.

Example

$(x_0 \,\&\, x_1) \,|\, (\sim (x_1 \,/\, x_2)) \quad \mapsto$

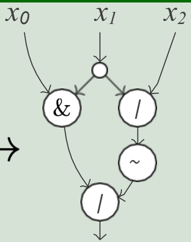


Boolean circuits, for example

Informally: representation of a propositional formula.

Example

$(x_0 \,\&\, x_1 \,|)(\sim (x_1 \mid x_2))$ $\mapsto$

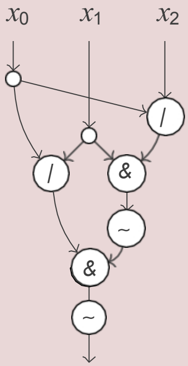


Boolean circuits are open directed and acyclic graphs.

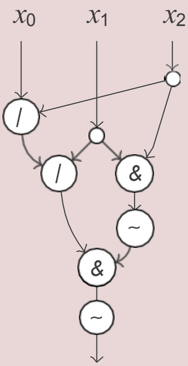
Boolean CIRCUIT

Exercise

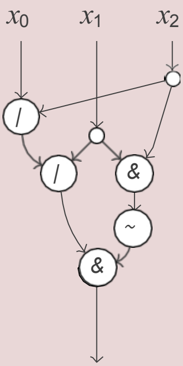
What Boolean circuit is obtained from $((x_0 \mid x_2) \mid x_1) \& (\sim (x_1 \& x_2))$?



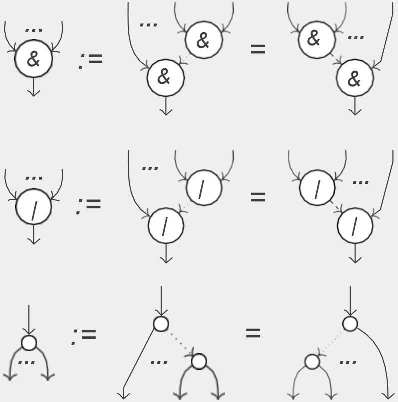
or



or

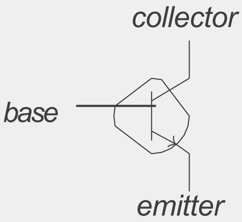


Since &, | and copy are all associative, we can represent :

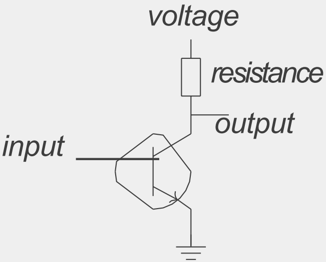


The benefits of Boolean circuits

- *Logic gates that can be implemented using transistors:*



For example, the "not" gate (~) :



- *Detailed study of the complexity of algorithms*

Graphs

Graph

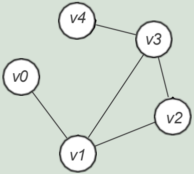
- *Informally: set of nodes (or vertices) connected by edges Mathematically:* $G = (V, E)$ *with* V *the set of vertices,* $E \subseteq V \times V$ *the set of edges*

Graph

- *Informally: set of nodes (or vertices) connected by edges Mathematically:* $G = (V, E)$ *with* V *the set of vertices,* $E \subseteq V \times V$ *the set of edges*
- *(in a simple non-oriented graph, at most one edge between two vertices. Convention to choose:* $(u, v) \in E \Rightarrow (v, u) \notin E$ *or conversely* $(u, v) \in E \Rightarrow (v, u) \in E$*)*

- *Informally: set of nodes (or vertices) connected by edges Mathematically:* $G = (V, E)$ *with* V *the set of vertices,* $E \subseteq V \times V$ *the set of edges*
- *(in a simple non-oriented graph, at most one edge between two vertices. Convention to choose:* $(u, v) \in E \Rightarrow (v, u) \notin E$ *or conversely* $(u, v) \in E \Rightarrow (v, u) \in E$*)*

Example



is represented by

$$(\{v_0, v_1, v_2, v_3, v_4\}, \{(v_0, v_1), (v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_4)\})$$

Exercise

Draw the graph described by :

$$\Big(\{v_0, v_1, v_2, v_3, v_4, v_5\},$$
$$\{(v_0,v_3), (v_0,v_4), (v_0,v_5), (v_1,v_3), (v_1,v_4), (v_1,v_5), (v_2,v_3), (v_2,v_4), (v_2,v_5)\}\Big)$$

DIRECTED GRAPH

In an undirected graph, (u, v) and (v, u) are interchangeable.

DIRECTED GRAPH

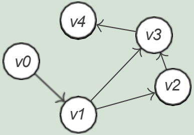
In an undirected graph, (u, v) and (v, u) are interchangeable.
To have a directed graph, all we need to do is consider the edge (u, v) as directed from u to v.

DIRECTED GRAPH

In an undirected graph, (u, v) and (v, u) are interchangeable.
To have a directed graph, all we need to do is consider the edge (u, v) as directed from u to v.

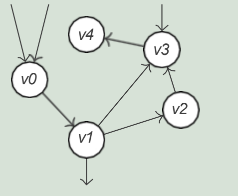
Example

$(\{v_0, v_1, v_2, v_3, v_4\}, \{(v_0, v_1), (v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_4)\})$:



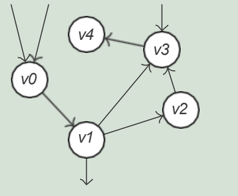
Open directed graph

Example



OPEN DIRECTED GRAPH

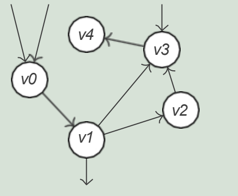
Example



Mathematically, we add two lists (sequences) I and O which contain the input (or output) nodes

OPEN DIRECTED GRAPH

Example



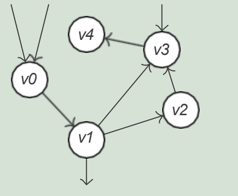
Mathematically, we add two lists (sequences) I and O which contain the input (or output) nodes

Example

$G = (V, I, O, E)$ where $I = [v_0, v_0, v_3]$ and $O = [v]_1$

OPEN DIRECTED GRAPH

Example



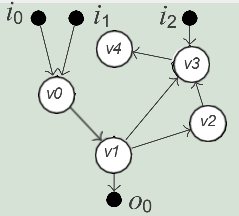
Mathematically, we add two lists (sequences) I and O which contain the input (or output) nodes

Example

$G = (V, I, O, E)$ where $I = [v_0\,,\, v_0\,,\, v_3\,]$ and $O = [v\,]_1$

Computationally, we can do exactly the same, or treat the ends of the 'half-edges' as nodes in their own right, with constraints (a single connection, incoming if it's an output, outgoing if it's an input), and with one order for inputs and another for outputs.

Example



$i_0 < i_1 < i_2$

DEGREES OF A NODE IN A GRAPH

Informally, the incoming (resp. outgoing) degree of a node is the number of edges arriving at (resp. departing from) the node.

DEGREES OF A NODE IN A GRAPH

Informally, the incoming (resp. outgoing) degree of a node is the number of edges arriving at (resp. departing from) the node.

The degree of a node is the sum of its incoming and outgoing degrees.

DEGREES OF A NODE IN A GRAPH

Informally, the incoming (resp. outgoing) degree of a node is the number of edges arriving at (resp. departing from) the node.

The degree of a node is the sum of its incoming and outgoing degrees. In $G = (V, I, O, E)$, with $u \in V$:

DEGREES OF A NODE IN A GRAPH

Informally, the incoming (resp. outgoing) degree of a node is the number of edges arriving at (resp. departing from) the node.

The degree of a node is the sum of its incoming and outgoing degrees. In $G = (V, I, O, E)$, *with* $u \in V$ *:*

- $deg^{+}(u) = |\{ (x, u) \in E \mid x \in V \}| + |\{ x \mid I[x] = u \}|$

DEGREES OF A NODE IN A GRAPH

Informally, the incoming (resp. outgoing) degree of a node is the number of edges arriving at (resp. departing from) the node.

The degree of a node is the sum of its incoming and outgoing degrees. In $G = (V, I, O, E)$, *with* $u \in V$ *:*

- $deg^{+}(u) = |\{ (x, u) \in E \mid x \in V \}| + |\{ x \mid I[x] = u \}|$
- $deg^{-}(u) = |\{ (u, x) \in E \mid x \in V \}| + |\{ x \mid O[x] = u \}|$

DEGREES OF A NODE IN A GRAPH

Informally, the incoming (resp. outgoing) degree of a node is the number of edges arriving at (resp. departing from) the node.

The degree of a node is the sum of its incoming and outgoing degrees. In $G = (V, I, O, E)$, with $u \in V$:

- $deg^+(u) = |\{ (x, u) \in E \mid x \in V\}| + |\{ x \mid I[x] = u\}|$
- $deg^-(u) = |\{ (u, x) \in E \mid x \in V\}| + |\{ x \mid O[x] = u\}|$
- $deg(u) = deg^+(u) + deg^-(u)$

DEGREES OF A NODE IN A GRAPH

Informally, the incoming (resp. outgoing) degree of a node is the number of edges arriving at (resp. departing from) the node.

The degree of a node is the sum of its incoming and outgoing degrees. In $G = (V, I, O, E)$, with $u \in V$:

- $deg^{+}(u) = |\{ (x, u) \in E \mid x \in V\}| + |\{ x \mid I[x] = u\}|$
- $deg^{-}(u) = |\{ (u, x) \in E \mid x \in V\}| + |\{ x \mid O[x] = u\}|$
- $deg(u) = deg^{+}(u) + deg^{-}(u)$

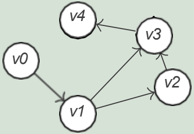
Maximum $\Delta(G)$ and minimum $\delta(G)$ degrees of a graph G :

- $\Delta(G) = max(\{deg(u) \mid u \in V\})$
- $\delta(G) = \min(\{\deg(u) \mid u \in V\})$

We can also define Δ^{+}, δ^{+}, Δ^{-} and δ^{-}.

PATHS ON AN (OPEN) DIRECTED GRAPH

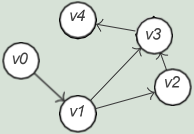
Example

In 

$[(v_0, v_1), (v_1, v_3), (v_3, v_4)]$ *is a path from* v_0 *to* v_4
.

PATHS ON AN (OPEN) DIRECTED GRAPH

Example

In 

$[(v_0, v_1), (v_1, v_3), (v_3, v_4)]$ *is a path from* v_0 *to* v_4 .

Let $G = (V, I, O, E)$ be a directed graph (open or not). A path from $u \in V$ to $v \in V$ is a sequence of edges $[e]_{i0 \leq i < n}$ such that :

- $e_0[0] = u$ and $e_{n-1}[1] = v$
- $e_i[1] = e_{i+1}[0]$ (for $0 \leq i < n - 1$)

We call n the length of the path.

Connectedness

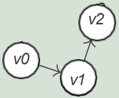
Informally, two nodes in a graph are related if they are not separable. Formally, u and v are related if there is a path between u and v in the induced undirected graph.

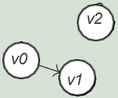
Connectedness

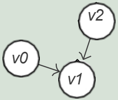
Informally, two nodes in a graph are related if they are not separable. Formally, u and v are related if there is a path between u and v in the induced undirected graph.

Example

In the following graphs, v_0 and v_2 are ...





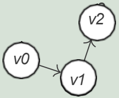


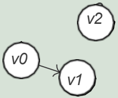
Connectedness

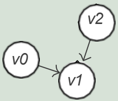
Informally, two nodes in a graph are related if they are not separable. Formally, u and v are related if there is a path between u and v in the induced undirected graph.

Example

In the following graphs, v_0 and v_2 are ...







If u_0 and u_1 are connected, then connected $(u_0, v) \iff \mathtt{connected}(u_1, v)$.

$\mathit{Partitions}$ the nodes of a graph (into what are known as related components).

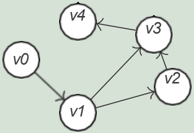
Cycles on a directed graph

A cycle is a path that loops back on itself.

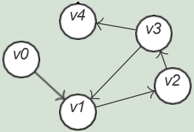
Cycles on a Directed Graph

A cycle is a path that loops back on itself.

Example



has no cycle, but

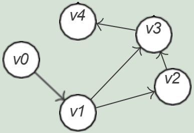


has one: $[(v_1, v_2), (v_2, v_3), (v_3, v_1)]$

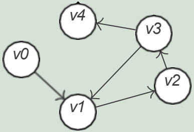
CYCLES ON A DIRECTED GRAPH

A cycle is a path that loops back on itself.

Example



has no cycle, but



has one: $[(v_1, v_2), (v_2, v_3), (v_3, v_1)]$

Their study is important in graph theory.

Examples: 1) the length of a path in a graph that contains cycles is not necessarily bounded

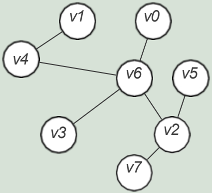
2) infinite paths between v_0 and v_4 above.

Acyclicity

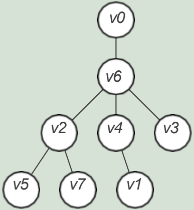
A graph that has no cycle is called acyclic.

An acyclic undirected connected graph is sometimes called a tree (more precisely, the choice of an order on the nodes fixes a tree).

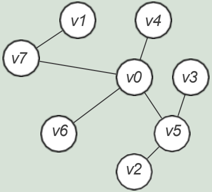
Example



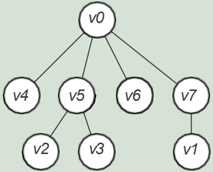
~



and



~



ACYCLICITY TEST

Let $G = (V, E)$ be a directed graph. If it is acyclic, it necessarily has a node that is not the source of any edge 

. Such a node is called a sink.

ACYCLICITY TEST

Let G = (V, E) be a directed graph. If it is acyclic, it necessarily has a node that is not the source of any edge 

. Such a node is called a sink.

A graph with a sink removed is acyclic if the starting graph is also acyclic.

ACYCLICITY TEST

Let $G = (V, E)$ be a directed graph. If it is acyclic, it necessarily has a node that is not the source of any edge 

. Such a node is called a sink.

A graph with a sink removed is acyclic if the starting graph is also acyclic.

A simple algo :

- If G has no nodes, it is acyclic
- We look for a sink of G
  - if there is none, G is cyclic
  - else, remove the sink and start again from the beginning

ACYCLICITY TEST

Let $G = (V, E)$ be a directed graph. If it is acyclic, it necessarily has a node that is not the source of any edge 

. Such a node is called a sink.

A graph with a sink removed is acyclic if the starting graph is also acyclic.

A simple algo :

- If G has no nodes, it is acyclic
- We look for a sink of G
  - if there is none, G is cyclic
  - else, remove the sink and start again from the beginning

This algorithm terminates because each time it is called, the number of nodes is reduced.

ACYCLICITY TEST

Let $G = (V, E)$ be a directed graph. If it is acyclic, it necessarily has a node that is not the source of any edge  *. Such a node is called a sink.*

A graph with a sink removed is acyclic if the starting graph is also acyclic.

A simple algo :

- If G has no nodes, it is acyclic
- We look for a sink of G
  - if there is none, G is cyclic
  - else, remove the sink and start again from the beginning

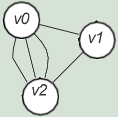
This algorithm terminates because each time it is called, the number of nodes is reduced.

Exercise :

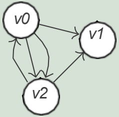
Is there anything more efficient than going through the graph again looking for a leaf each time you call?

A multigraph is a graph that can have several edges between the same two nodes.

Example



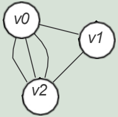
or



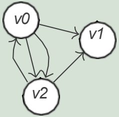
Multi-sets and Multi-graphs

A multigraph is a graph that can have several edges between the same two nodes.

Example



or



Problem: E is a set $\Rightarrow$ no duplicates. Solution: make E a multiset (simply a set where you can have the same element several times).

Example (with G above)

$G = (\{v_0, v_1, v_2\}, \{\{(v_0, v_1), (v_0, v_2), (v_0, v_2), (v_2, v_0), (v_2, v_1)\}\})$

COMPUTATIONAL REPRESENTATION

There are lots of different approaches depending on what you want to optimise.

Computational representation

There are lots of different approaches depending on what you want to optimise.

Here, using sets / dictionaries → access time on average very good, in the worst case bad, and space usage not great.

COMPUTATIONAL REPRESENTATION

There are lots of different approaches depending on what you want to optimise.

Here, using sets / dictionaries → access time on average very good, in the worst case bad, and space usage not great.

In the following, an object approach, structure with constant-time access to any node, doubly chained.

COMPUTATIONAL REPRESENTATION

There are lots of different approaches depending on what you want to optimise.

Here, using sets / dictionaries → access time on average very good, in the worst case bad, and space usage not great.

In the following, an object approach, structure with constant-time access to any node, doubly chained.

A class for nodes, and one for graphs (open, directed).

THE NODE CLASS

```
class node:
    def __init__(self, identity, label, parents, children):
        '''
        identity: int; its unique id in the
graph label: string;
        parents: int->int dict; maps a parent node's id to its
multiplicity children: int->int dict; maps a child node's id to its
multiplicity
        '''
        self.id = identity
        self.label= label
        self.parents =  parents
        self.children = children
```

The (open directed) graph class

```python
class open_digraph: # for open directed graph

    def __init__(self, inputs, outputs, nodes):
        '''
        inputs: int list; the ids of the input nodes
outputs: int list; the ids of the output nodes
        nodes: node  iter;
        '''
        self.inputs = inputs
self.outputs = outputs
        self.nodes = {node.id:node for node in nodes}
        # self.nodes: <int,node> dict
```

THE (OPEN DIRECTED) GRAPH CLASS

```
class open_digraph: # for open directed graph

    def __init__(self, inputs, outputs, nodes):
        '''
        inputs: int list; the ids of the input nodes
outputs: int list; the ids of the output nodes
        nodes: node iter;
        '''
        self.inputs = inputs
self.outputs = outputs
        self.nodes = {node.id:node for node in nodes}
        # self.nodes: <int,node> dict
```

In `self.nodes` *we use :*

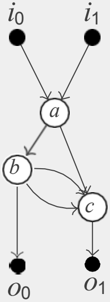
a "dict" dictionary: a dynamic, mutable data structure with

- ▶ *access/addition/withdrawal generally in constant time*
  ▶ *fairly poor use of space*

an understanding

-

EXAMPLE OF A CIRCUIT DAWS THIS implémewtatiow




```
n0 = node(0, 'a', {3:1, 4:1}, {1:1, 2:1})
n1 = node(1, 'b', {0:1}, {2:2, 5:1})
n2 = node(2, 'c', {0:1, 1:2}, {6:1})
i0 = node(3, 'i0', {}, {0:1})
i1 = node(4, 'i1', {}, {0:1})
o0 = node(5, 'o0', {1:1}, {})
o1 = node(6, 'o1', {2:1}, {})

G = open_digraph([3,4], [5,6], [n0,n1,n2,i0,i1,o0,o1])
```

! *For a graph to be well formed, each edge must be found in both the* `children` *of the source node, and in the* `parents` *of the target node.*

To be done quickly

- *Form a pair or a team of three, and send me an e-mail to let me know.*
- *Send me an email if you can't find a partner/three-person team.*
- *1 person per group: try to create a repo on GitHub/GitLab (and invite the other people in the group).*