Additionneur ternaire

Sommaire

Binaire / Ternaire

Définitions

Système de portes logiques : ensemble de portes logiques

Système complet : système de portes logiques générant toute porte logique

ET	1	0
1	1	0
0	0	0

OU	1	0
1	1	1
0	1	0

NON	
1	0
0	1

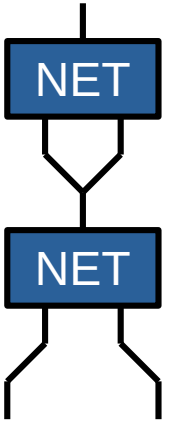
Exemple

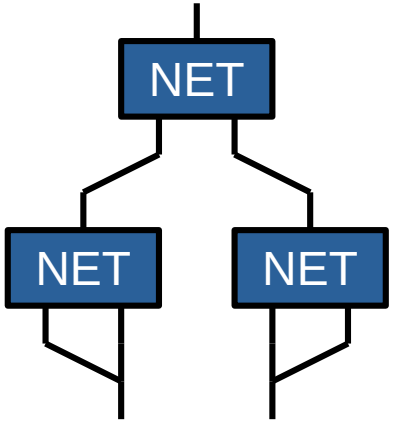
NET	1	0
1	0	1
0	1	1

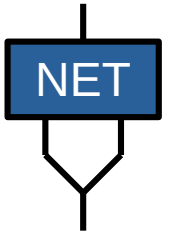
ET	1	0
1	1	0
0	0	0

OU	1	0
1	1	1
0	1	0

NON	
1	0
0	1







Binaire / Ternaire

NON	
+	-
-	+

OU	+	-
+	+	+
-	+	-

ET	+	-
+	+	-
-	-	-

NON	
+	-
0	0
-	+

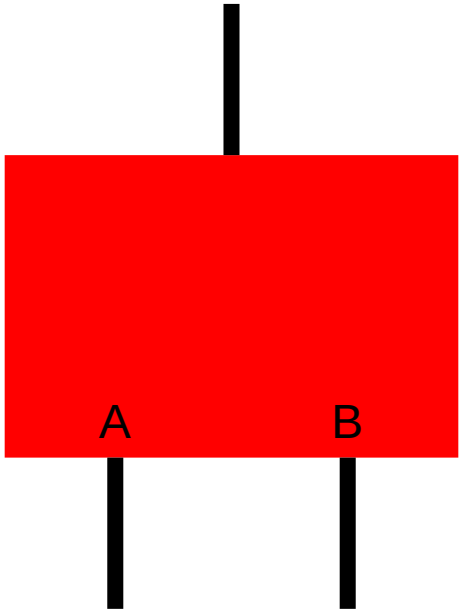
OU	+	0	-
+	+	+	+
0	+	0	0
-	+	0	-

ET	+	0	-
+	+	0	-
0	0	0	-
-	-	-	-

$2^4 = 16$

$3^9 = 19683$

Conventions



	B		
A			

	+	0	-
+	t_8	t_7	t_6
0	t_5	t_4	t_3
-	t_2	t_1	t_0

-	0
0	1
+	2

$$id = \sum_{i=0}^{8} t_i \times 3^i$$

Système complet minimal

Minimisation

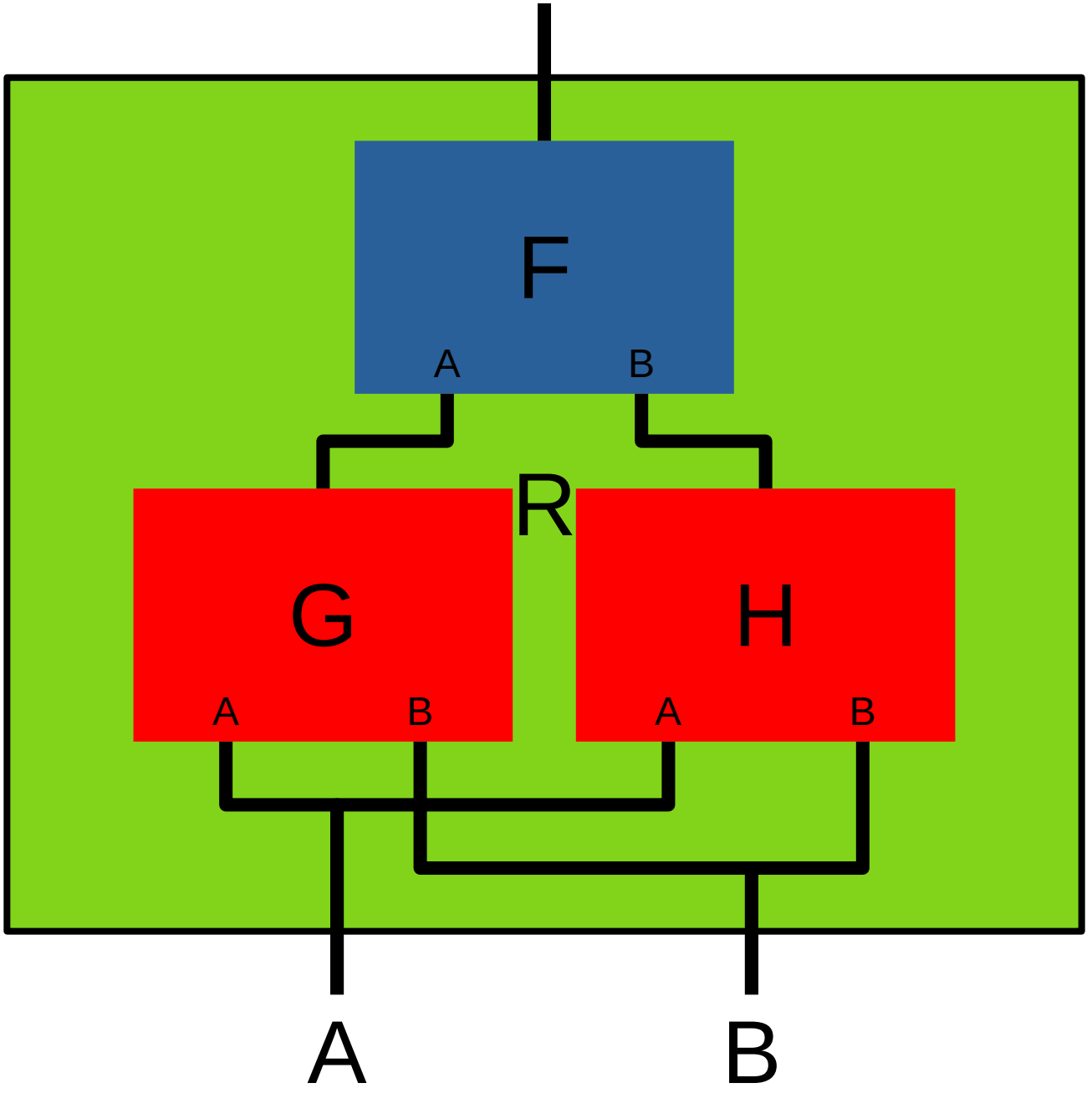
Système complet minimal : système complet de taille minimale

NET	1	0
1	0	1
0	1	1

NOU	1	0
1	0	0
0	0	1

N : taille des systèmes complets minimaux

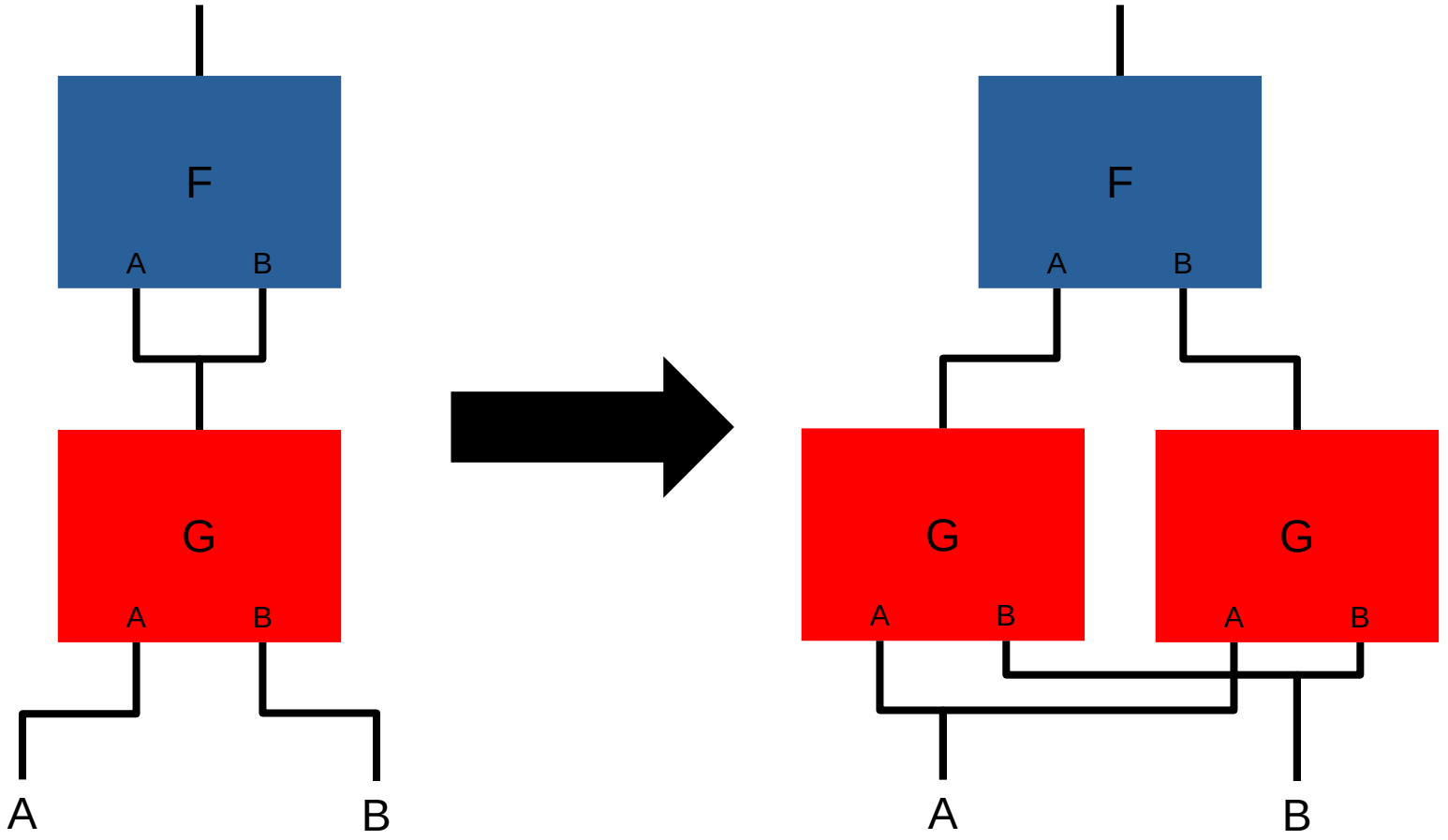
Jointure



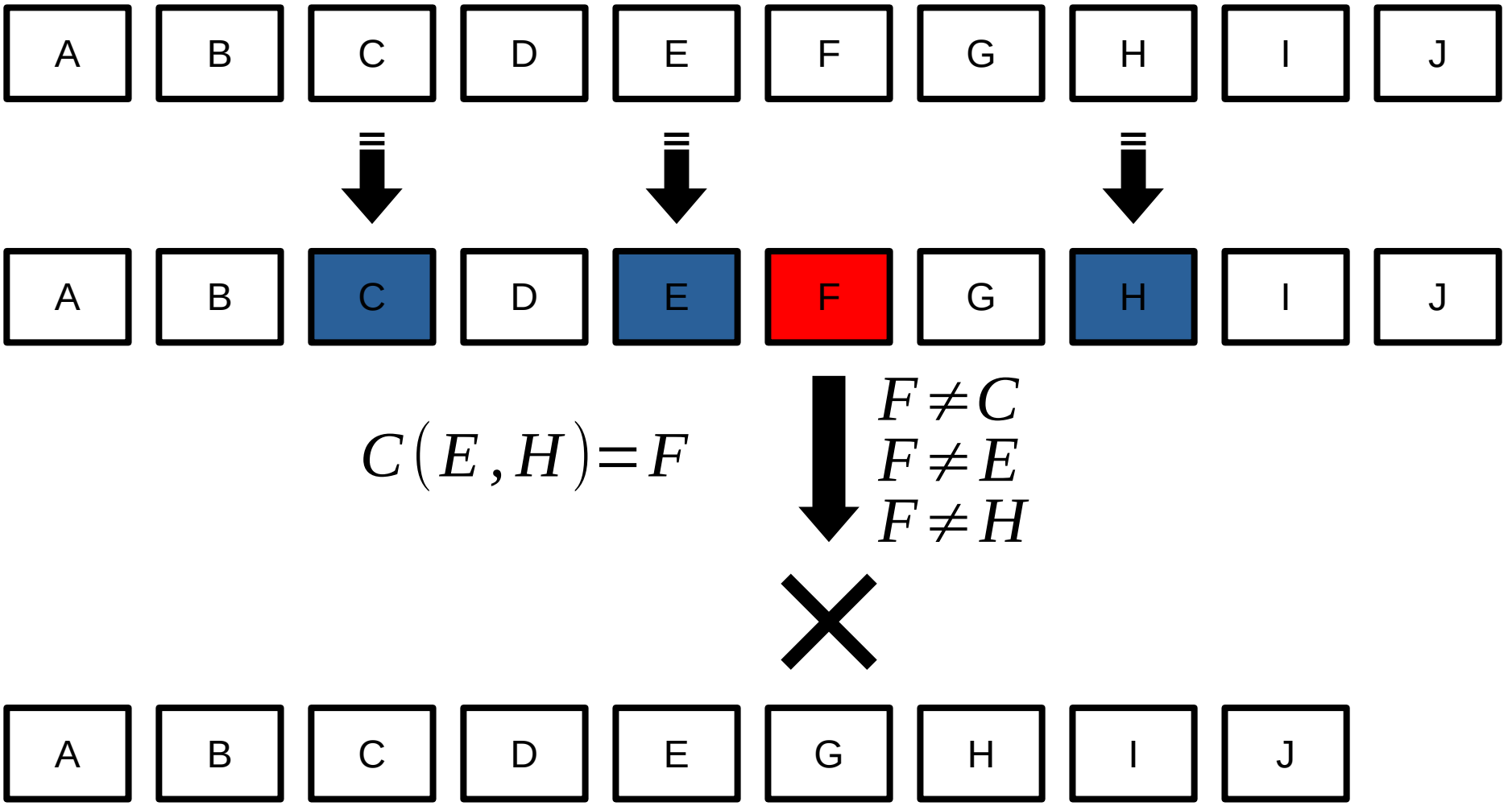
$$R = F(G, H)$$

$$\forall a, b \in S, R(a,b) = F(G(a,b), H(a,b))$$

Duplication



Réduction



Version 1

Systèmes de taille 20 à 30

142	1139	1343	3089	3395	3600	4928
5065	5177	7081	8489	9690	10554	10820
11540	11814	11892	12482	13714	14493	15101

(21)

$N \leq 20$

1.5 secondes

Version 2

Entrée

142 1139 1343 3089 3395 3600 4928

5065 5177 7081 8489 9690 10554 10820 (21)

11540 11814 11892 12482 13714 14493 15101

V2.0

1139 1343 3395 3600 4928 7081 8489 (13)

11540 13714 15101 2805 3076 10788

V2.2

3395 7081 11540 11892 50 106 1871 2034 (8)

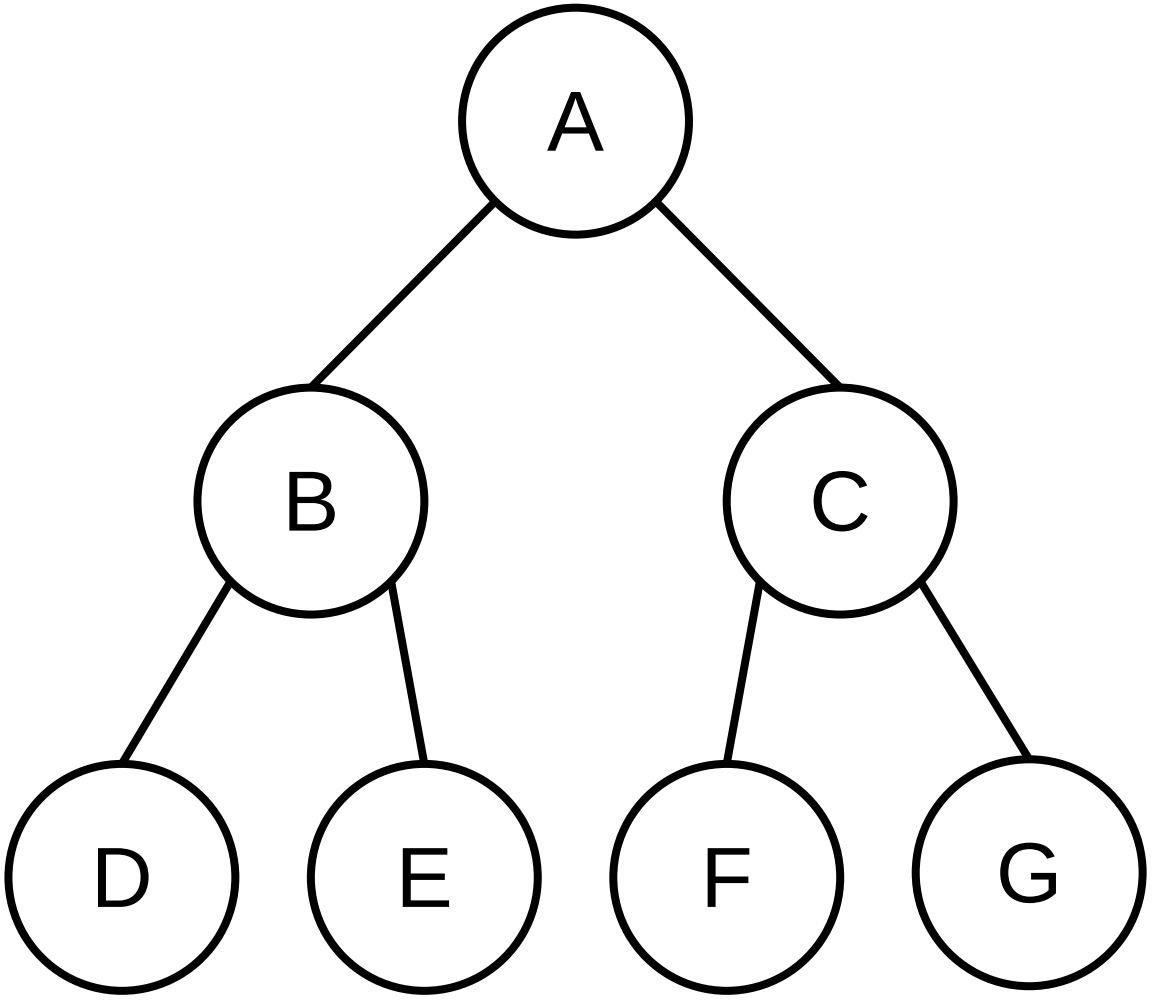
V2.3

3395 9690 11814 777 2079 5361 16194 (7)

$N \leq 7$

0.2 secondes

Génération



$$A(B(D,E),C(F,G))$$

Essais

3395 | 9690 | 11814 | 777 | 2079 | 5361 | 16194 (7)

3395 | 11814 | 777 (3)

3395	+	0	-
+	-	0	0
0	0	+	+
-	+	-	+

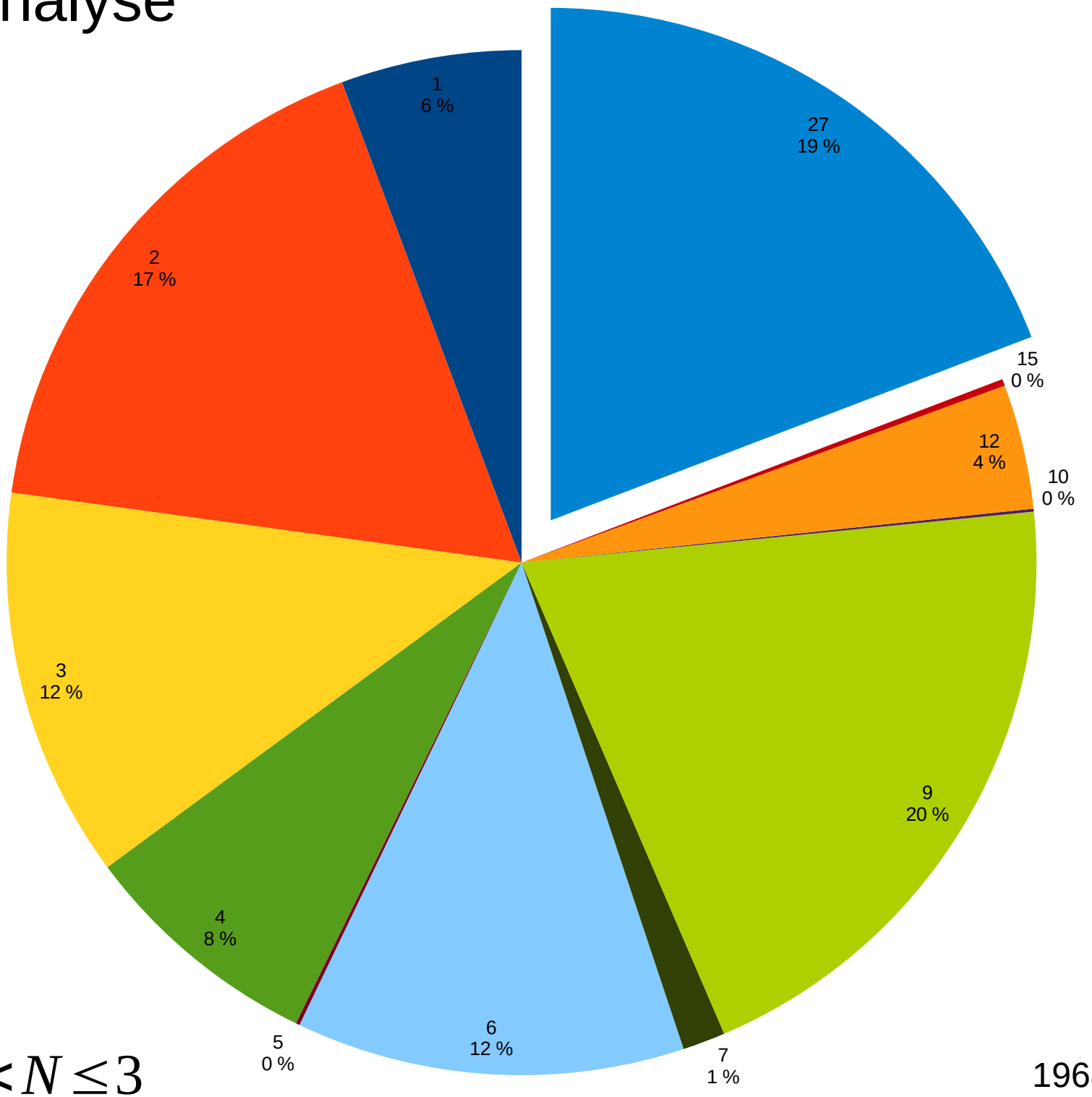
11814	+	0	-
+	0	+	0
0	-	0	+
-	0	+	-

777	+	0	-
+	-	-	0
0	-	-	0
-	+	0	-

$N \leq 3$

Analyse

Taille système : 1



$1 < N \leq 3$

19683 portes, 0.5 secondes

Résultat

7957	+	0	-
+	0	-	0
0	+	+	-
-	+	-	0

3913	+	0	-
+	-	0	+
0	0	-	-
-	+	+	0

$N = 2$

Algorithme 1 : 2’ 29’’
Algorithme 2 : 3’ 25’’

Pratique & Optimisation

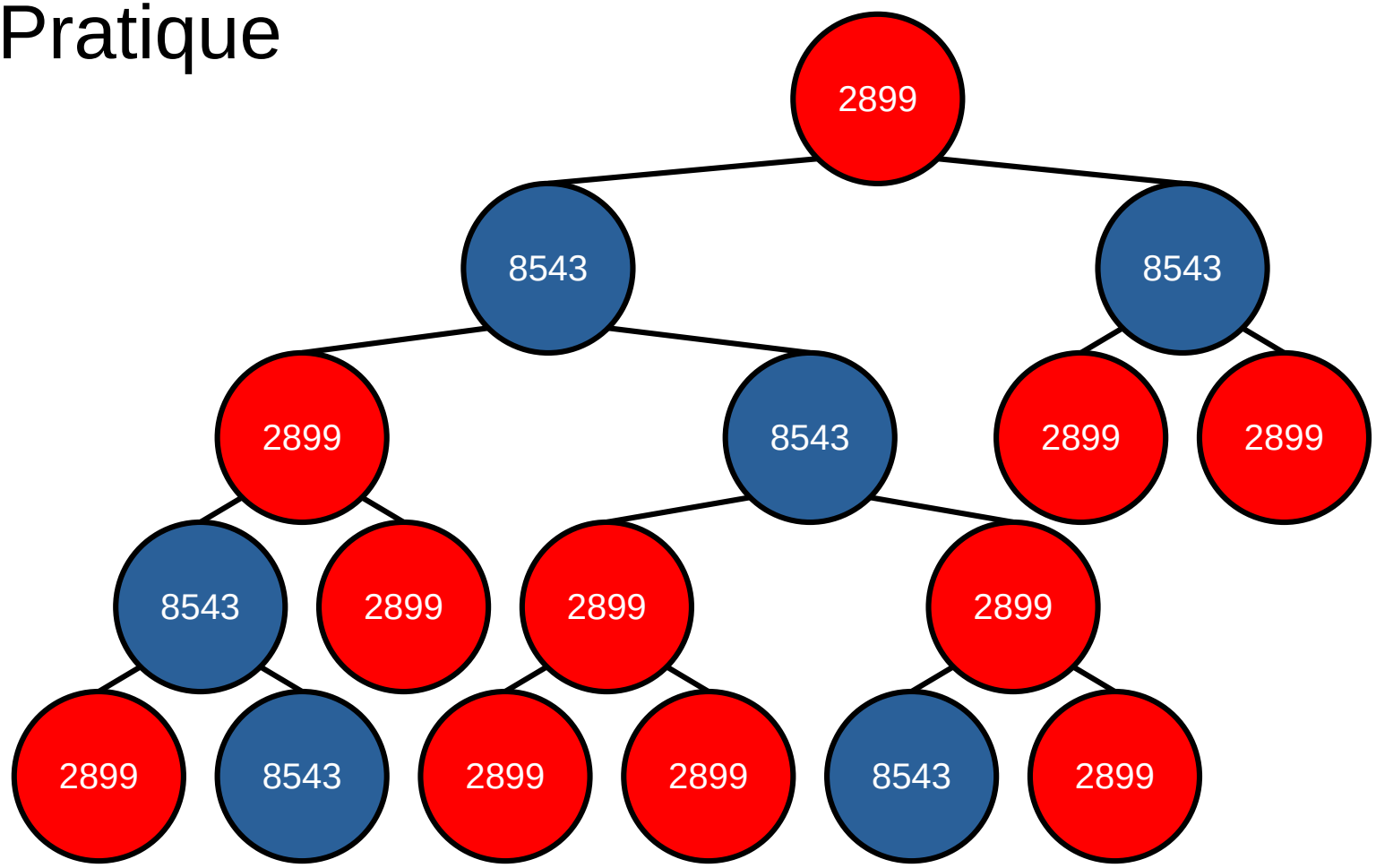
Système

2899	+	0	-
+	-	0	-
0	+	+	+
-	0	-	0

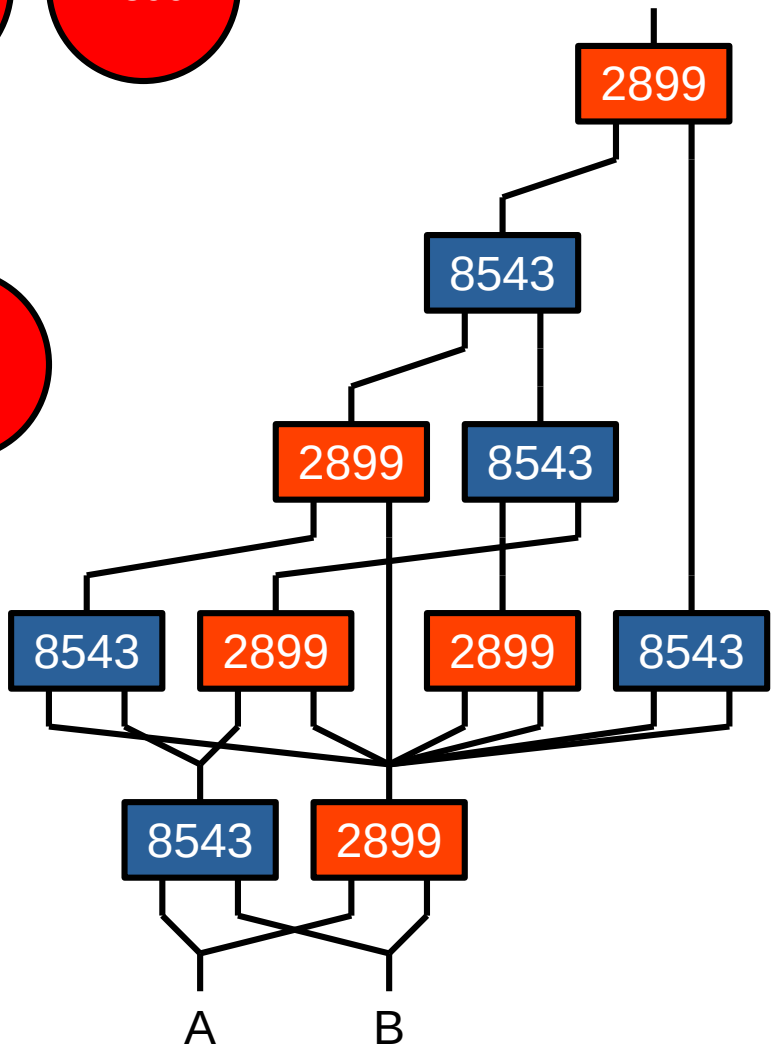
8543	+	0	-
+	0	-	+
0	+	-	0
-	0	-	+

Pratique

Algo 1 : 2’ 17”

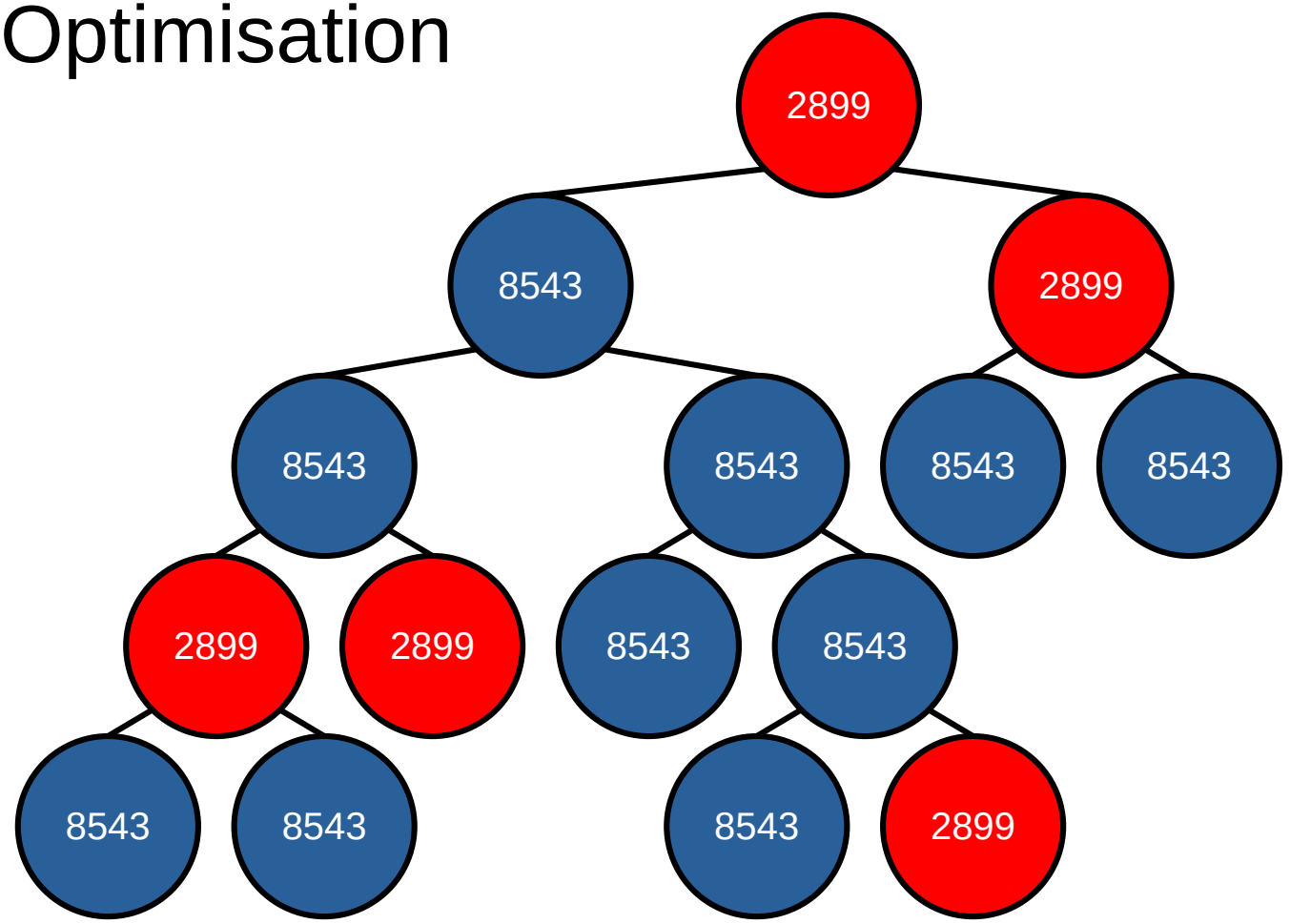


5681	+	0	-
+	-	+	0
0	+	0	-
-	0	-	+

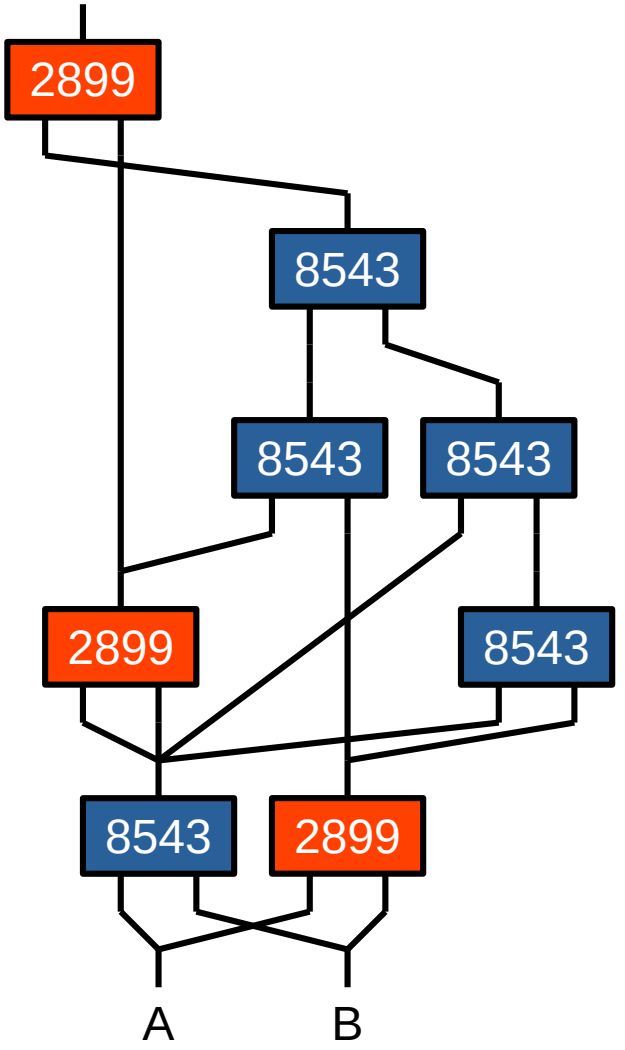


Optimisation

Algo 2 : 7’ 6”



5681	+	0	-
+	-	+	0
0	+	0	-
-	0	-	+



Additionneur ternaire

Ternaire équilibré

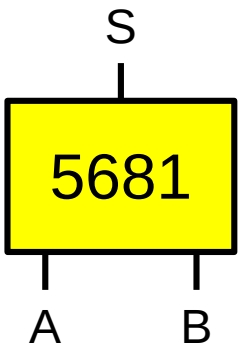
+	+1
0	0
-	-1

$$\sum_{k=0}^{n} d_k \times 3^k$$

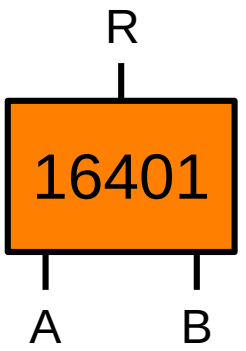
Nombre	Positif	Négatif
0	0	0
1	+	−
2	+−	−+
3	+0	−0
4	++	−−
5	+−−	−++
6	+−0	−+0
7	+−+	−+−
8	+0−	−0+
9	+00	−00
10	+0+	−0−
11	++−	−−+
12	++0	−−0
13	+++	−−−

Implémentation

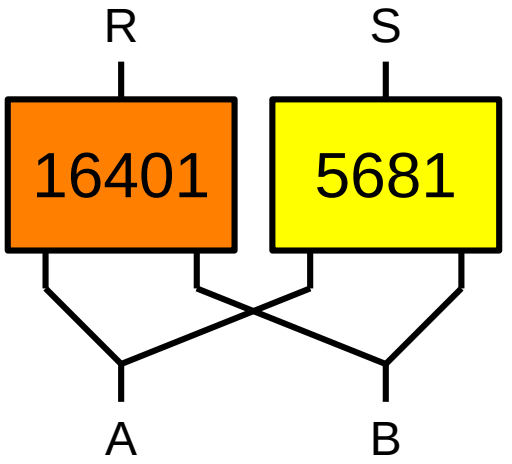
5681	+	0	-
+	-	+	0
0	+	0	-
-	0	-	+



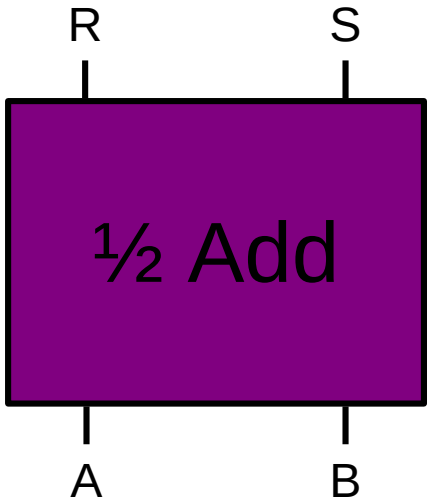
16401	+	0	-
+	+	0	0
0	0	0	0
-	0	0	-

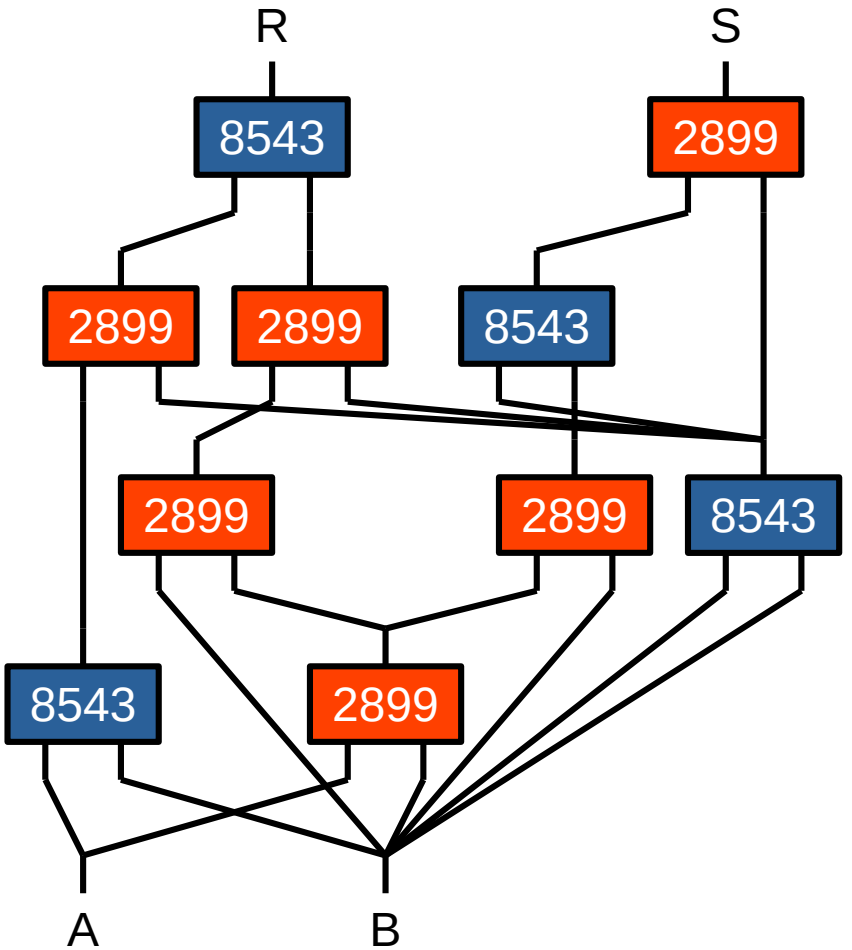


½ Additionneur

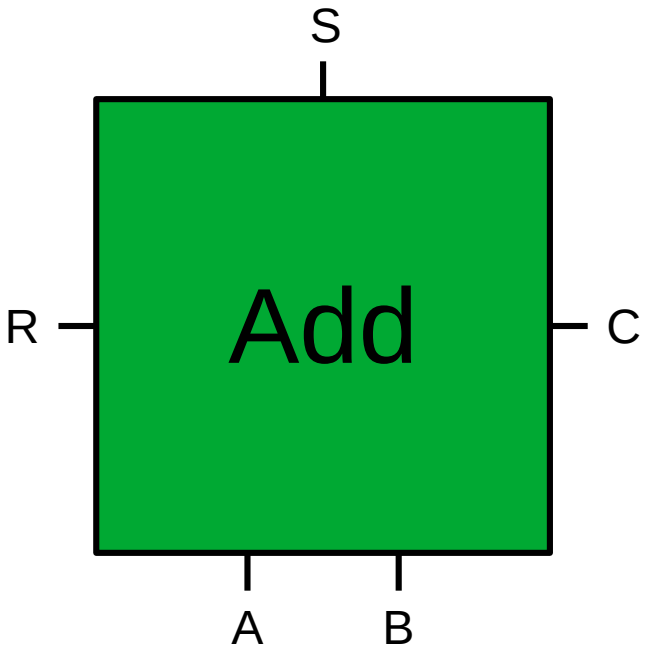


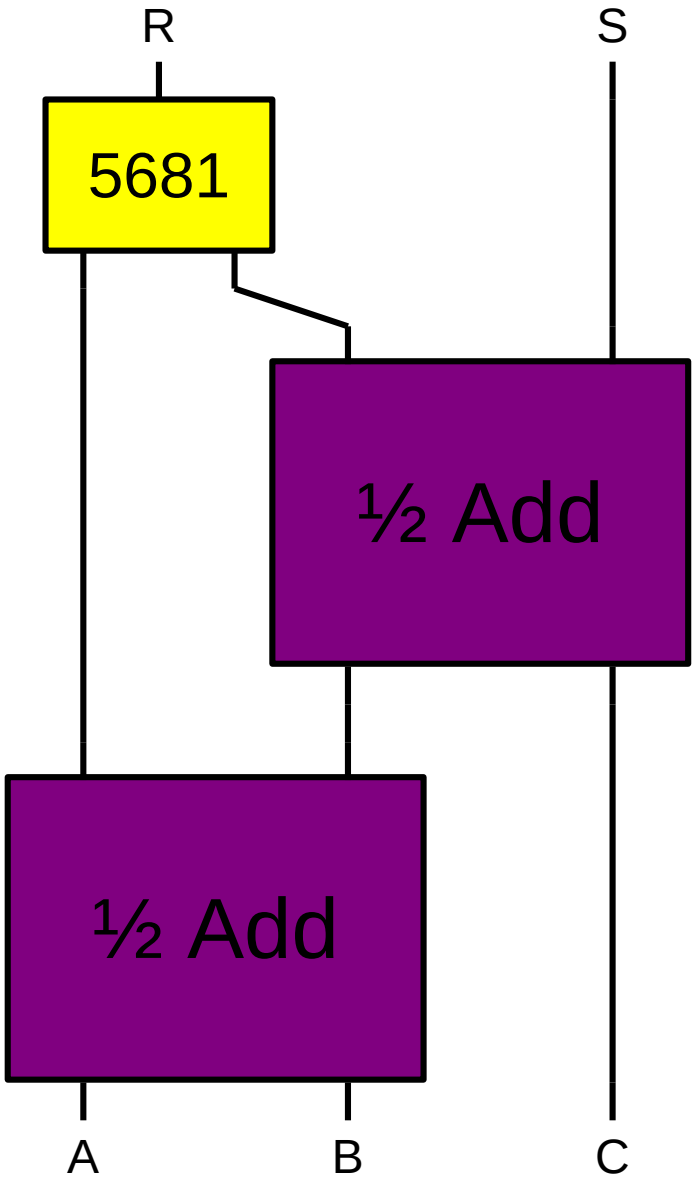
A	B	R	S
+	+	+	-
+	0	0	+
+	-	0	0
0	+	0	+
0	0	0	0
0	-	0	-
-	+	0	0
-	0	0	-
-	-	-	+



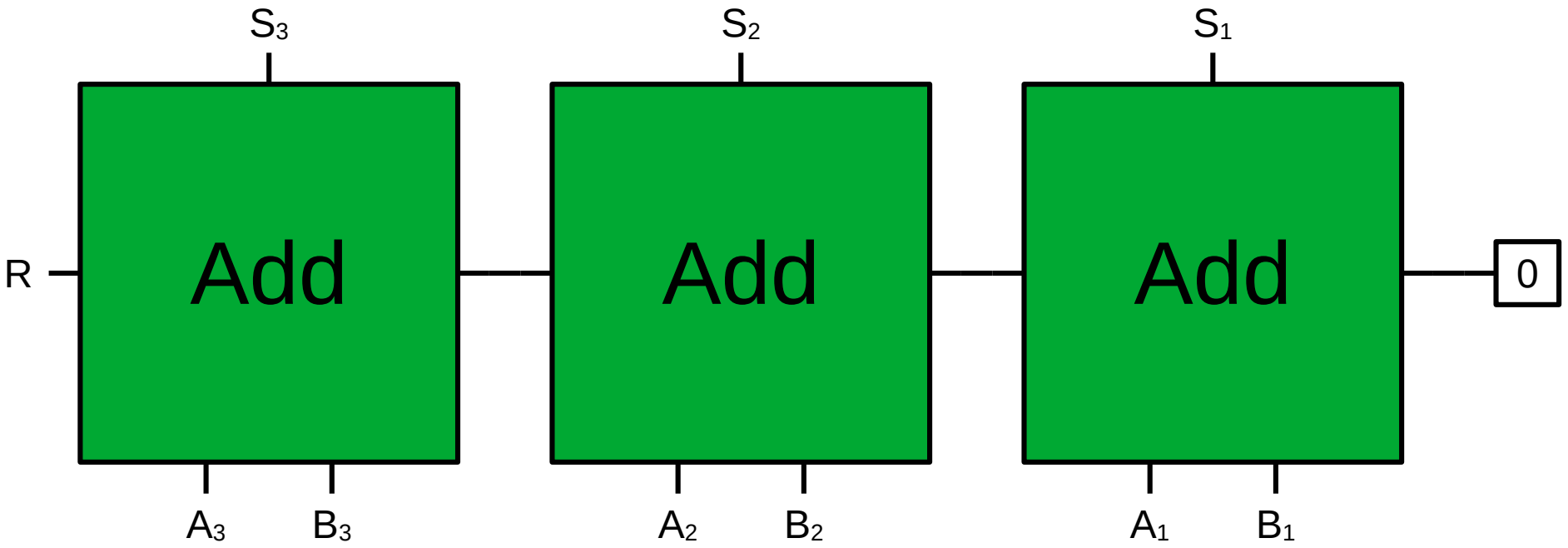


Additionneur 1 trit





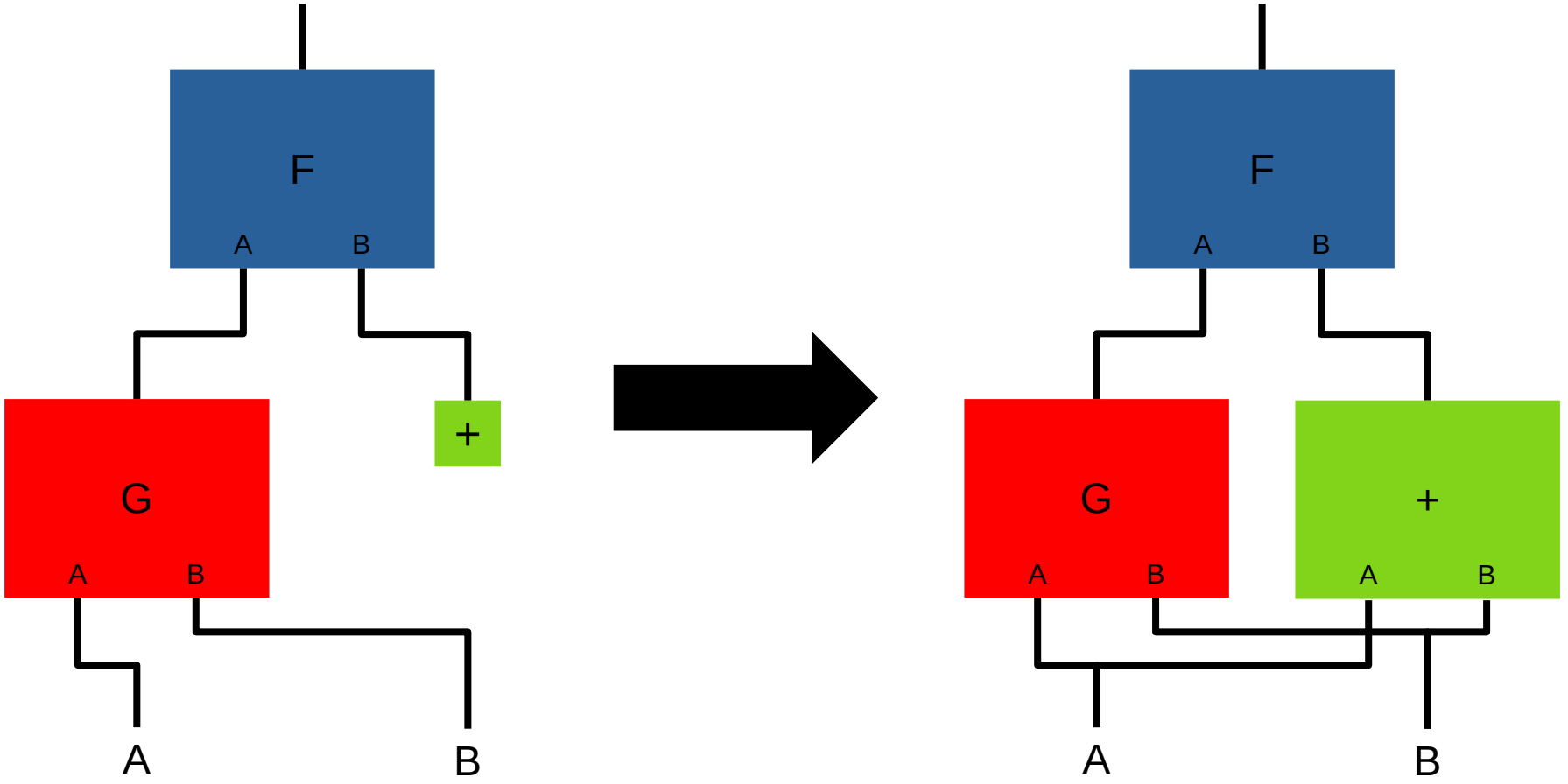
Additionneur 3 trits



Conclusion

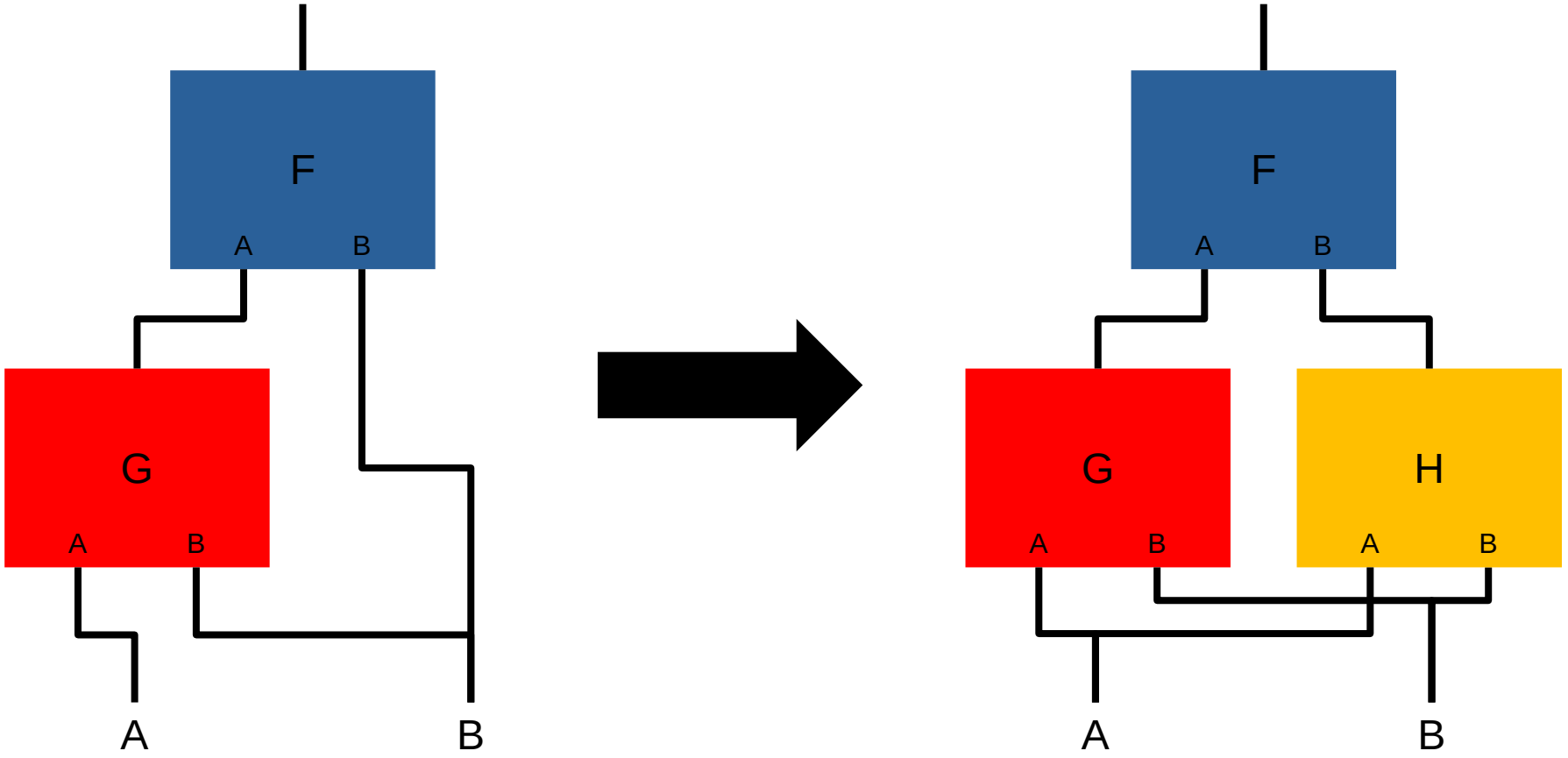
Annexes

Constante



+	+	0	-
+	+	+	+
0	+	+	+
-	+	+	+

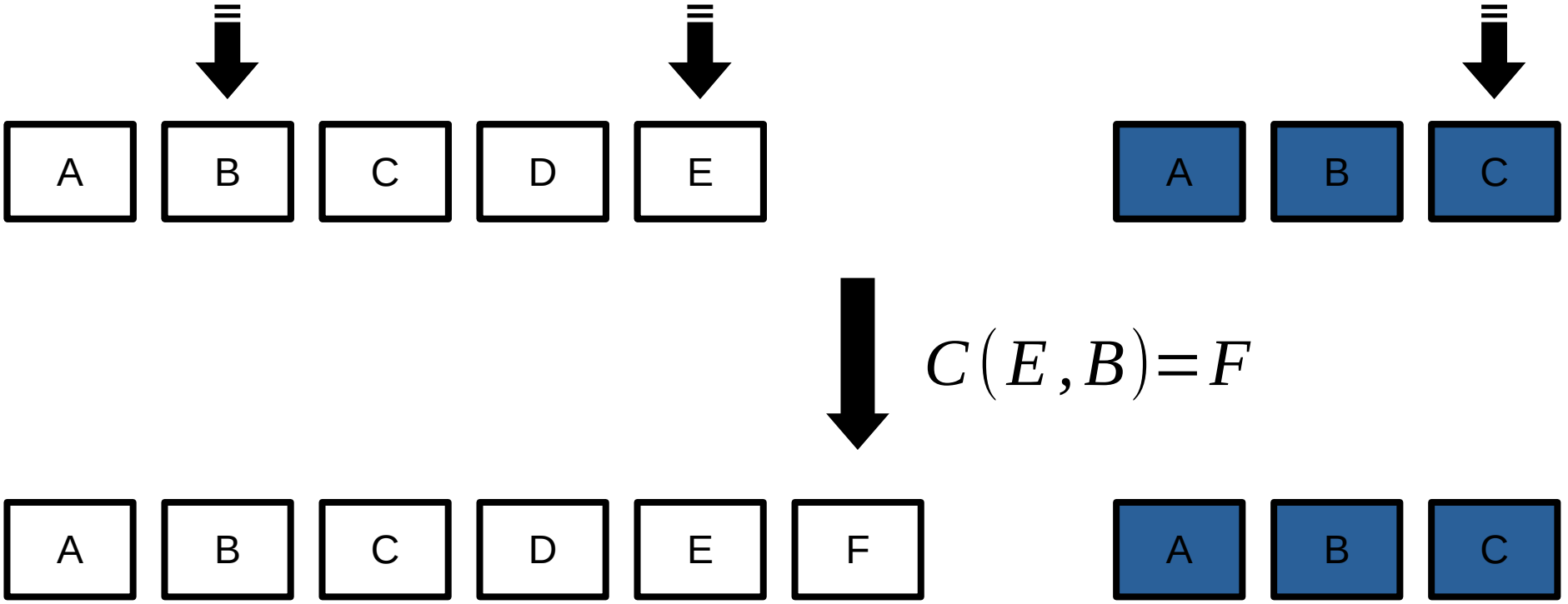
Branche directe



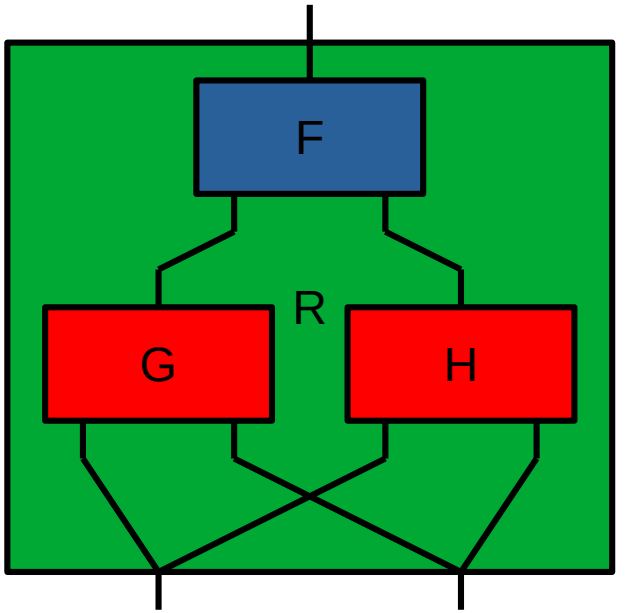
$$\forall a,b \in S, H(a,b) = b$$

H	+	0	-
+	+	0	-
0	+	0	-
-	+	0	-

Algorithme 1



Jointure



H	+	0	-
+	-	0	+
0	+	+	+
-	+	-	-

G	+	0	-
+	+	+	-
0	+	-	0
-	-	0	+

F	+	0	-
+	0	-	+
0	+	+	0
-	-	+	-

R	+	0	-
+	+	-	-
0	0	-	+
-	-	0	+

Algorithme 2

H	+	0	-
+	-	0	+
0	+	+	+
-			

G	+	0	-
+	+	+	-
0	+	-	0
-			

F	+	0	-
+	0	-	+
0	+	+	0
-	-	+	-

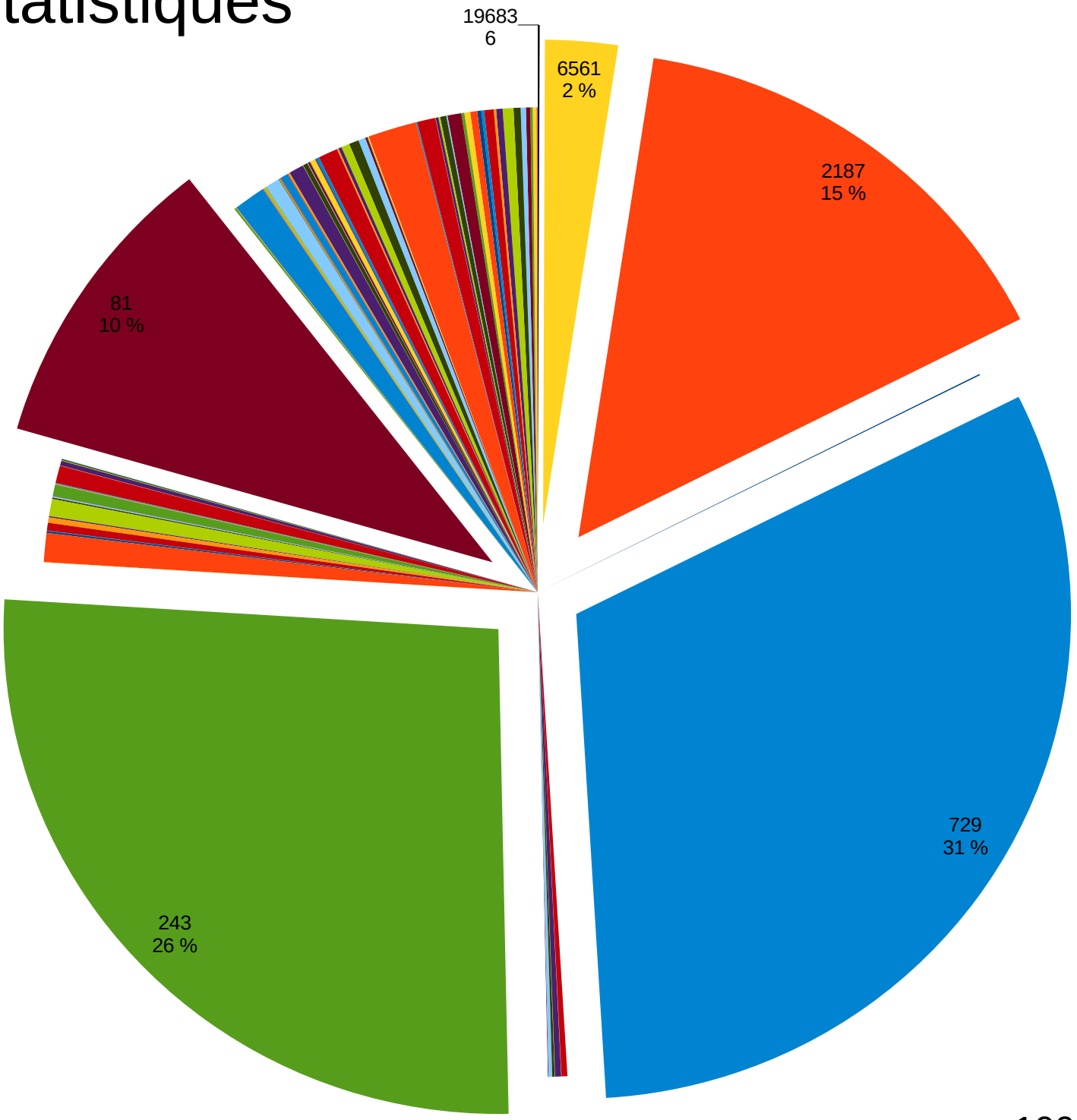
R	+	0	-
+	+	-	-
0	0	-	+
-	0	+	+

Performances

Système	Complet	Algorithme 1	Algorithme 2
3395, 9690, 11814, 777, 2079, 5361, 16194 (7)	Oui	7’ 39’’	4’ 15’’
3395, 11814, 777 (3)	Oui	4’ 1’’	3’ 8’’
11814, 777, 2079 (3)	Non	23’’	6’ 48’’
2899, 8543 (2)	Oui	2’ 8’’	2’ 55’’
3395, 9690 (2)	Non	2’’	5’ 56’’

Statistiques

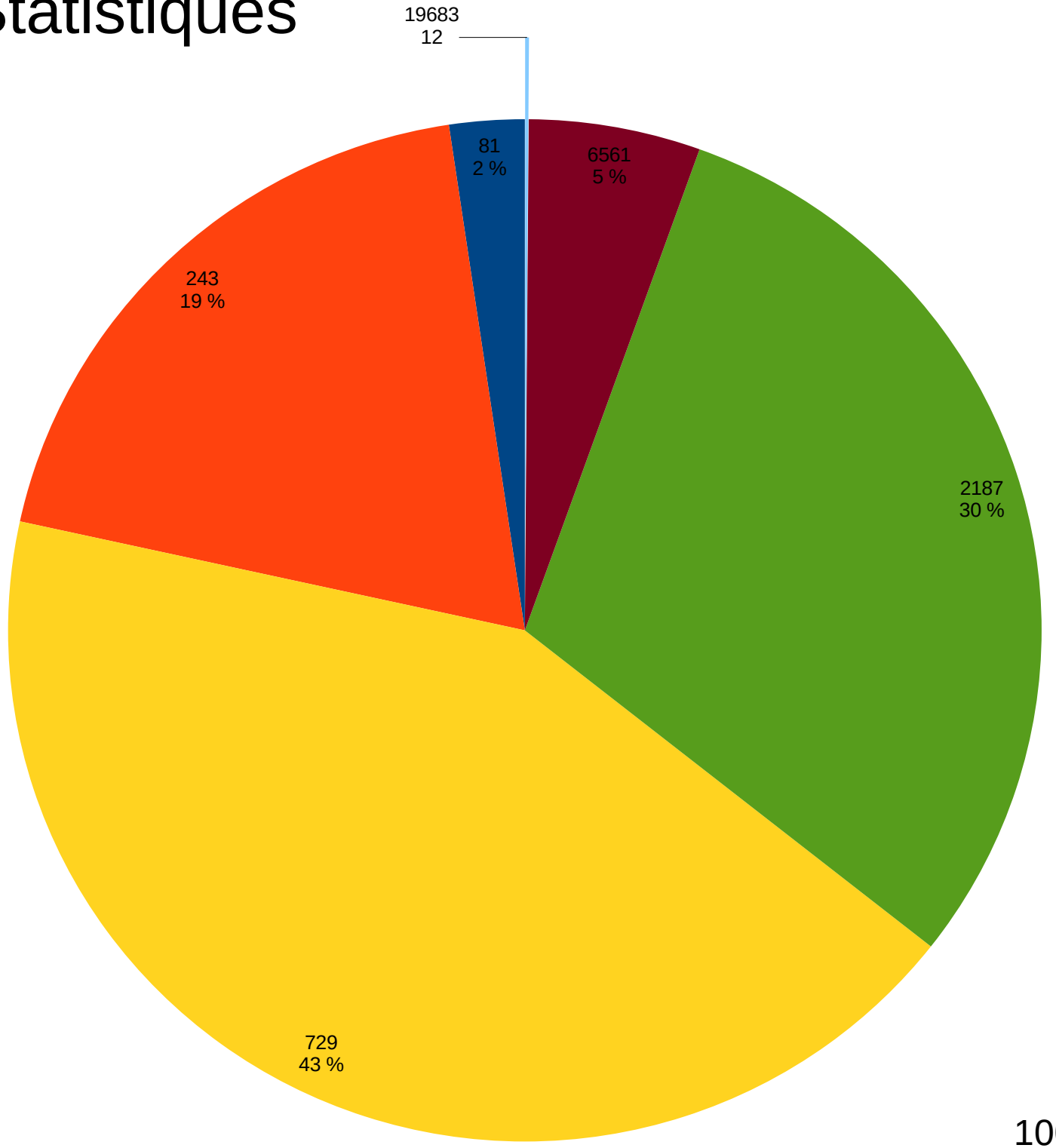
Taille système : 2



10000 échantillons, 144 minutes

Statistiques

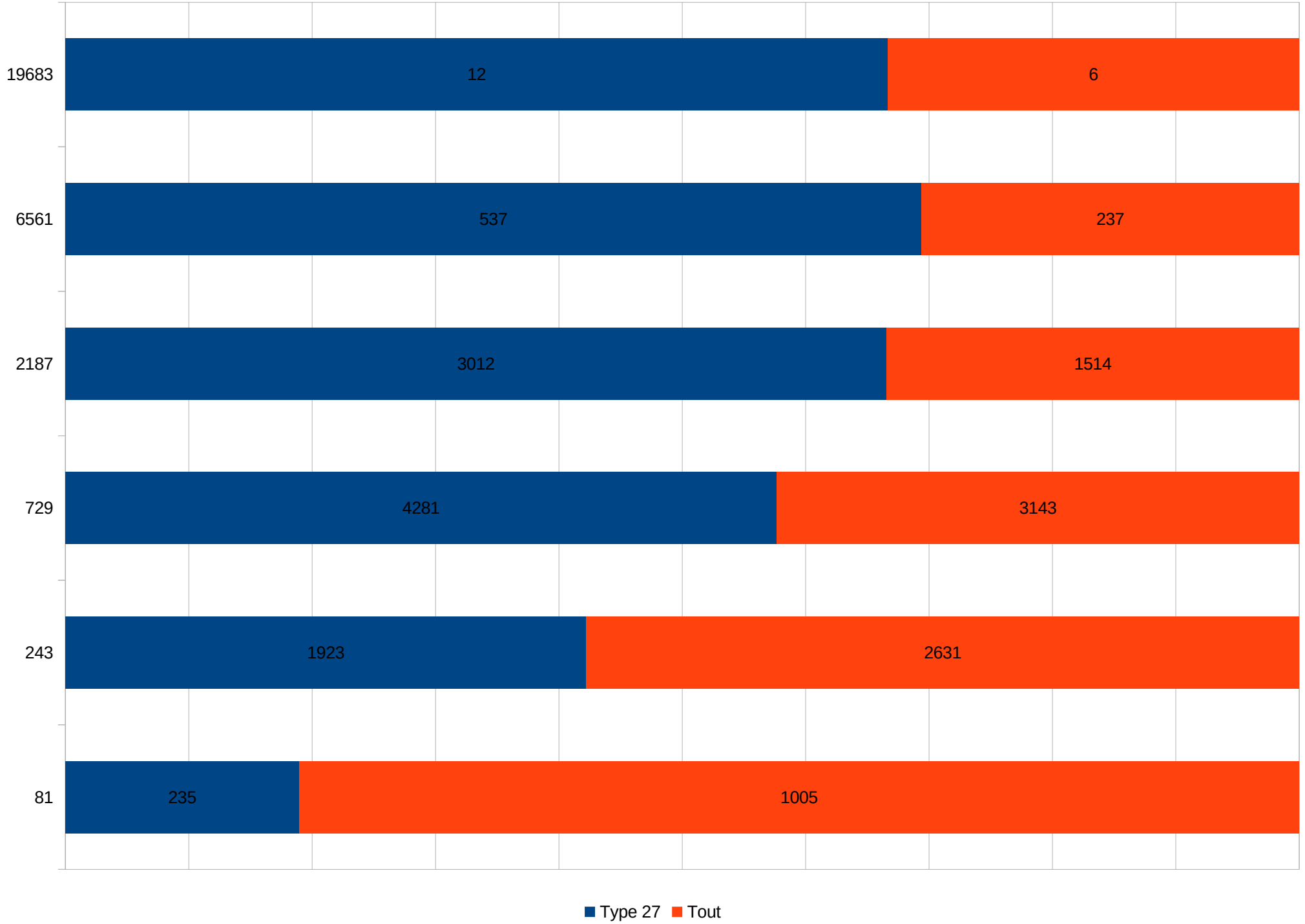
Taille système : 2, Type 27



10000 échantillons, 278 minutes

Statistiques

Taille système : 2

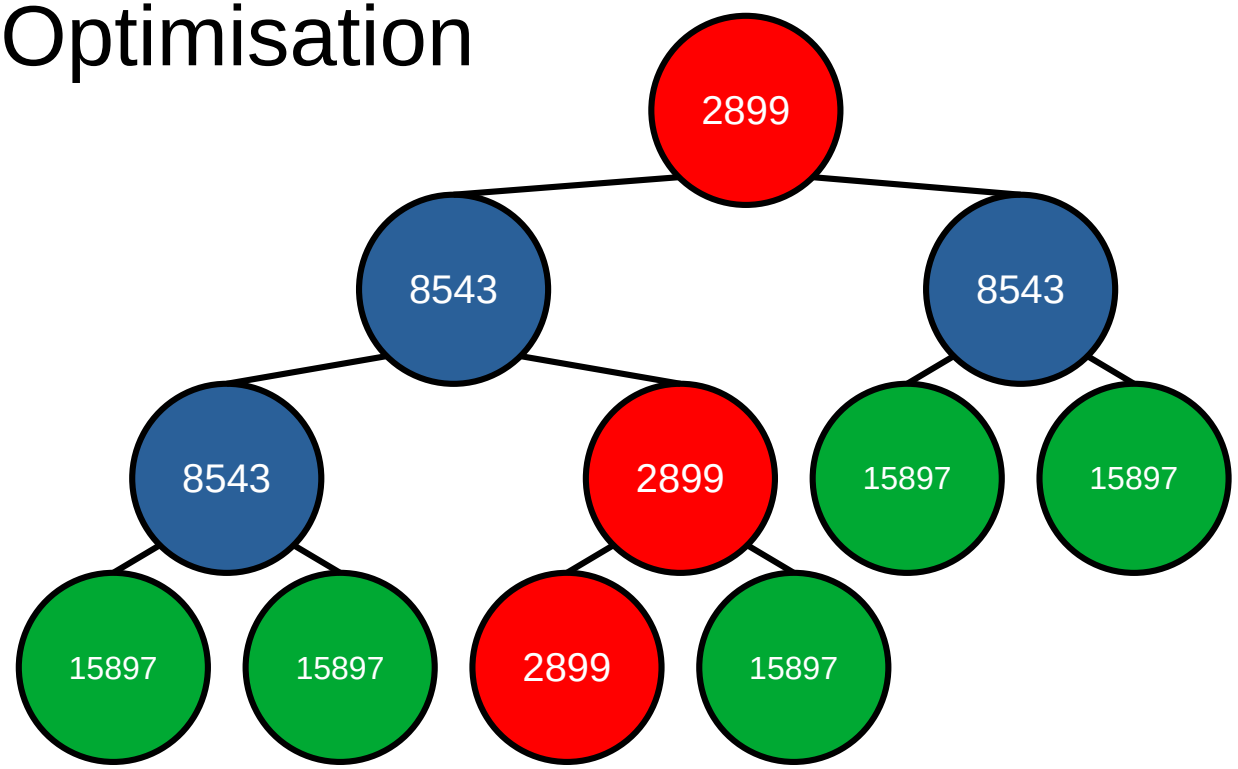


Remarque

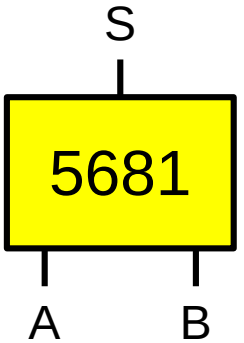
4669	13273
403	**3829**
1877	**8569**
475	11229
9307	16713
18075	**3425**

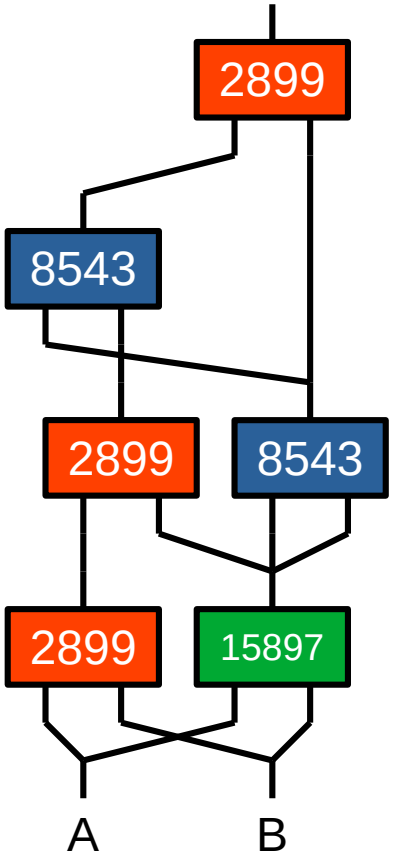
Optimisation

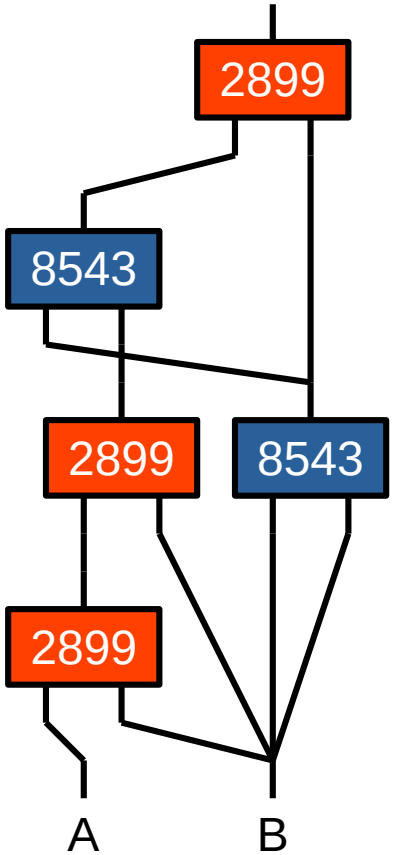
Algo 2 : 7’ 31”



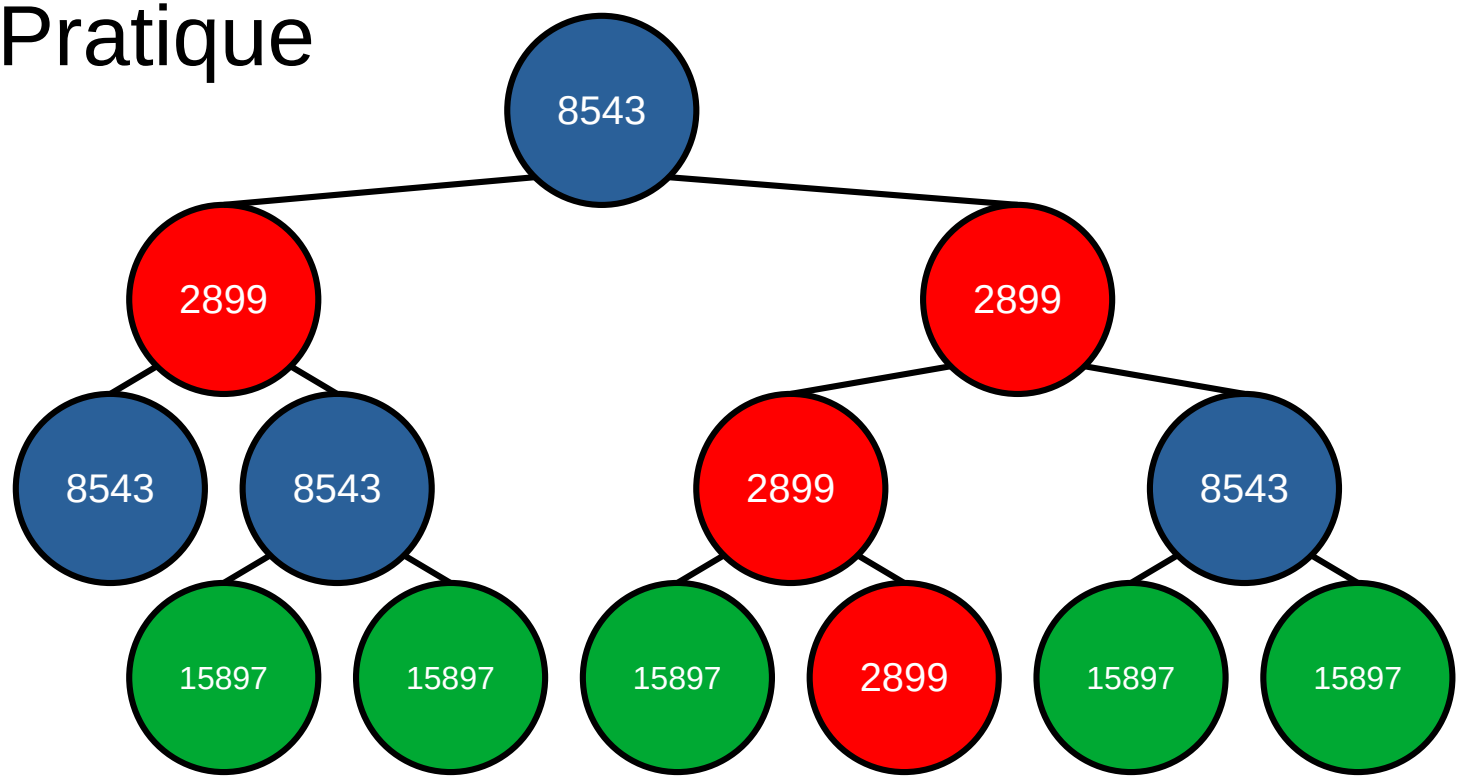
5681	+	0	-
+	-	+	0
0	+	0	-
-	0	-	+



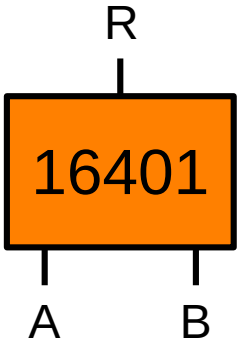


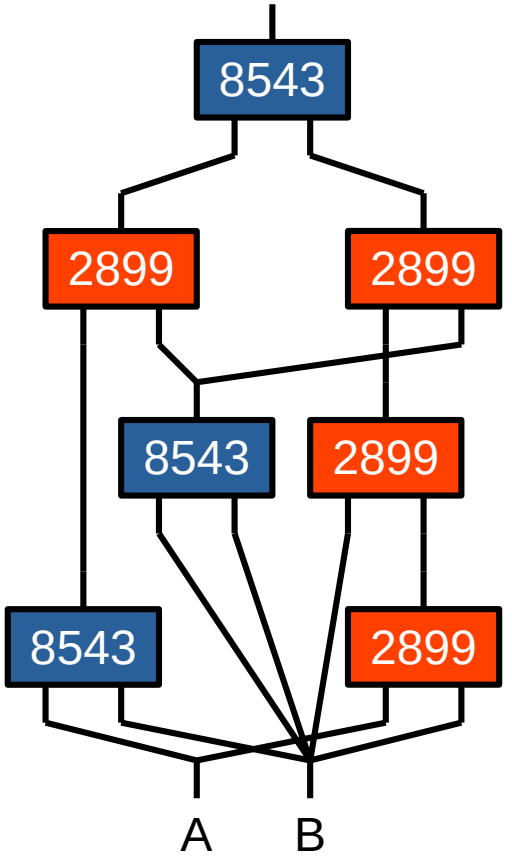


Pratique



16401	+	0	-
+	+	0	0
0	0	0	0
-	0	0	-





Code

lib.rs

```rust
use std::{
    array,
    cmp::max,
    collections::{BTreeMap, BTreeSet, HashMap, HashSet},
    convert::identity,
    fmt,
    iter::zip,
    ops::{Index, IndexMut},
    usize,
};

pub const NUM_TRITS: usize = 3;

#[derive(Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash, Debug, Default)]
pub enum Trit {
    Plus = 2,
    #[default]
    Zero = 1,
    Minus = 0,
}

impl From<usize> for Trit {
    fn from(value: usize) -> Self {
        match value {
            2 => Self::Plus,
            1 => Self::Zero,
            0 => Self::Minus,
            _ => unreachable!(),
        }
    }
}

impl From<Trit> for usize {
    fn from(value: Trit) -> Self {
        value as Self
    }
}

impl<T> Index<Trit> for [T; NUM_TRITS] {
    type Output = T;

    fn index(&self, index: Trit) -> &Self::Output {
        &self[usize::from(index)]
    }
}

impl<T> IndexMut<Trit> for [T; NUM_TRITS] {
    fn index_mut(&mut self, index: Trit) -> &mut Self::Output {
        &mut self[usize::from(index)]
    }
}

impl fmt::Display for Trit {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(
            f,
            "{}",
            match self {
                Trit::Plus => '+',
                Trit::Zero => '0',
                Trit::Minus => '-',
            }
        )
    }
}

pub const NUM_GATES_CELL: usize = NUM_TRITS * NUM_TRITS;
pub const NUM_GATES: usize = NUM_TRITS.pow((NUM_GATES_CELL) as u32);

#[derive(Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash, Debug)]
pub struct Gate(pub usize);

impl Gate {
    fn brute_eval(&self, x: usize, y: usize) -> usize {
        let mut v = self.0;
        for _ in 0..x {
            v /= NUM_TRITS.pow(NUM_TRITS as u32);
        }
        for _ in 0..y {
            v /= NUM_TRITS;
        }
        v % NUM_TRITS
    }

    pub fn eval(&self, x: Trit, y: Trit) -> Trit {
        self.brute_eval(x.into(), y.into()).into()
    }

    pub fn join(&self, &a: &Self, &b: &Self) -> Self {
        let mut v = [0; NUM_GATES_CELL];

        let x: [usize; NUM_GATES_CELL] = a.into();
        let y: [usize; NUM_GATES_CELL] = b.into();

        for (v, (a, b)) in v.iter_mut().zip(zip(x, y)) {
            *v = self.brute_eval(a, b);
        }

        v.into()
    }

    pub fn reverse_search(&self) -> [Vec<(Trit, Trit)>; NUM_TRITS] {
        let mut trits = array::from_fn(|_| Vec::new());

        let a: [usize; NUM_GATES_CELL] = (*self).into();

        for (i, v) in a.into_iter().enumerate() {
            trits[v].push(((i / NUM_TRITS).into(), (i % NUM_TRITS).into()));
        }

        trits
    }
}

impl From<usize> for Gate {
    fn from(value: usize) -> Self {
        Self(value)
    }
}

impl From<Gate> for usize {
    fn from(value: Gate) -> Self {
        value.0
    }
}

impl<T> Index<Gate> for [T; NUM_GATES] {
    type Output = T;

    fn index(&self, index: Gate) -> &Self::Output {
        &self[usize::from(index)]
```

```rust
    }
}

impl<T> IndexMut<Gate> for [T; NUM_GATES] {
    fn index_mut(&mut self, index: Gate) -> &mut Self::Output {
        &mut self[usize::from(index)]
    }
}

// + 0 -
// + 8 7 6
// 0 5 4 3
// - 2 1 0
impl From<[usize; NUM_GATES_CELL]> for Gate {
    fn from(value: [usize; NUM_GATES_CELL]) -> Self {
        Self(value.iter().rev().fold(0, |acc, e| acc * NUM_TRITS + e))
    }
}

impl From<Gate> for [usize; NUM_GATES_CELL] {
    fn from(value: Gate) -> Self {
        let mut v = value.0;

        array::from_fn(|_| {
            let r = v % NUM_TRITS;
            v /= NUM_TRITS;
            r
        })
    }
}

impl From<[Trit; NUM_GATES_CELL]> for Gate {
    fn from(value: [Trit; NUM_GATES_CELL]) -> Self {
        let array = value.map(usize::from);
        array.into()
    }
}

impl From<Gate> for [Trit; NUM_GATES_CELL] {
    fn from(value: Gate) -> Self {
        let array: [usize; NUM_GATES_CELL] = value.into();
        array.map(Trit::from)
    }
}

impl fmt::Display for Gate {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(
            f,
            " {} {} {}\n{} {} {} {}\n{} {} {} {}\n{} {} {} {}",
            Trit::Plus,
            Trit::Zero,
            Trit::Minus,
            Trit::Plus,
            self.eval(Trit::Plus, Trit::Plus),
            self.eval(Trit::Plus, Trit::Zero),
            self.eval(Trit::Plus, Trit::Minus),
            Trit::Zero,
            self.eval(Trit::Zero, Trit::Plus),
            self.eval(Trit::Zero, Trit::Zero),
            self.eval(Trit::Zero, Trit::Minus),
            Trit::Minus,
            self.eval(Trit::Minus, Trit::Plus),
            self.eval(Trit::Minus, Trit::Zero),
            self.eval(Trit::Minus, Trit::Minus),
        )
    }
}

pub fn generated_gates_depth(system: &[Gate], max: usize) -> (Vec<Gate>, usize) {
    let mut seen_gates: HashSet<_> = system.iter().copied().collect();
    let mut all_gates: Vec<_> = system.into();

    let mut new_gates = Vec::from(system);
    let mut found_gates = Vec::new();

    for depth in 1..max {
        for a in &all_gates {
            for b in &new_gates {
                for (x, y) in [(a, b), (b, a)] {
                    for z in system {
                        let gate = z.join(x, y);

                        if seen_gates.insert(gate) {
                            found_gates.push(gate);
                        }
                    }
                }
            }
        }

        if found_gates.is_empty() {
            return (all_gates, depth);
        }

        new_gates.clear();
        all_gates.extend(&found_gates);

        (found_gates, new_gates) = (new_gates, found_gates);

        eprintln!("Depth: {}", depth);
    }

    (all_gates, max)
}

pub fn generated_gates(system: &[Gate]) -> Vec<Gate> {
    generated_gates_depth(system, usize::MAX).0
}

pub fn generated_gates_tree(system: &[Gate]) -> Vec<(Gate, Gate, Gate)> {
    let mut seen_gates: HashMap<_, _> = system.iter().map(|&g| (g, None)).collect();
    let mut all_gates: Vec<_> = system.into();

    let mut new_gates = Vec::from(system);
    let mut found_gates = Vec::new();

    loop {
        for a in &all_gates {
            for b in &new_gates {
                for (x, y) in [(a, b), (b, a)] {
                    for z in system {
                        let gate = z.join(x, y);

                        if !seen_gates.contains_key(&gate) {
                            seen_gates.insert(gate, Some((*z, *x, *y)));
                            found_gates.push(gate);
                        }
                    }
                }
            }
        }

        if found_gates.is_empty() {
```

```rust
            return seen_gates.into_values().filter_map(identity).collect();
        }

        new_gates.clear();
        all_gates.extend(&found_gates);

        (found_gates, new_gates) = (new_gates, found_gates);
    }
}

const MIN_TRIT: Trit = Trit::Minus;
const MAX_TRIT: Trit = Trit::Plus;

pub fn revere_generated_gates_depth(system: &[Gate], max: usize) -> (Vec<Gate>, usize)
{
    let reverse_system: Vec<_> = system.iter().map(Gate::reverse_search).collect();

    let mut gates_set: BTreeSet<_> = system.iter().copied().collect();

    let mut todo_gates: Vec<_> = (0..NUM_GATES)
        .map(Gate)
        .filter(|g| !gates_set.contains(g))
        .collect();

    let mut found_gates = Vec::new();

    let mut stack = Vec::with_capacity(NUM_GATES_CELL);
    let mut a_gate = Vec::with_capacity(NUM_GATES_CELL);
    let mut b_gate = Vec::with_capacity(NUM_GATES_CELL);

    for depth in 1..max {
        todo_gates.retain(|&gate| {
            eprintln!("{:?}", gate);

            let gate_values: [Trit; NUM_GATES_CELL] = gate.into();

            for reverse_gate in &reverse_system {
                let possibles_values = gate_values.map(|trit| &reverse_gate[trit]);

                stack.push(possibles_values.first().unwrap().iter().copied());

                while let Some(iter) = stack.last_mut() {
                    if let Some((a, b)) = iter.next() {
                        a_gate.push(a);
                        b_gate.push(b);

                        if gates_set
                            .range(
                                Gate::from(array::from_fn(|i| {
                                    a_gate.get(i).copied().unwrap_or(MIN_TRIT)
                                }))
                                    ..=Gate::from(array::from_fn(|i| {
                                        a_gate.get(i).copied().unwrap_or(MAX_TRIT)
                                    })),
                            )
                            .next()
                            .is_some()
                            && gates_set
                                .range(
                                    Gate::from(array::from_fn(|i| {
                                        b_gate.get(i).copied().unwrap_or(MIN_TRIT)
                                    }))
                                        ..=Gate::from(array::from_fn(|i| {
                                            b_gate.get(i).copied().unwrap_or(MAX_TRIT)
                                        })),
                                )
                                .next()
                                .is_some()
                        {
                            let i = stack.len();
                            if i == NUM_GATES_CELL {
                                found_gates.push(gate);

                                stack.clear();
                                a_gate.clear();
                                b_gate.clear();

                                return false;
                            }

                            stack.push(possibles_values[i].iter().copied());
                        } else {
                            a_gate.pop();
                            b_gate.pop();
                        }
                    } else {
                        stack.pop();

                        a_gate.pop();
                        b_gate.pop();
                    }
                }
            }

            return true;
        });

        if found_gates.is_empty() {
            return (gates_set.into_iter().collect(), depth);
        }

        gates_set.extend(&found_gates);
        found_gates.clear();

        eprintln!("Depth: {}", depth);
    }

    (gates_set.into_iter().collect(), max)
}

pub fn revere_generated_gates(system: &[Gate]) -> Vec<Gate> {
    revere_generated_gates_depth(system, usize::MAX).0
}

fn build_tree(
    set: &mut HashSet<Gate>,
    tree: &BTreeMap<Gate, Option<(Gate, Gate, Gate)>>,
    gate: &Gate,
) -> usize {
    set.insert(*gate);
    match tree[gate] {
        Some((_, x, y)) => build_tree(set, tree, &x) + build_tree(set, tree, &y) + 1,
        None => 0,
    }
}

fn tree_stats(
    tree: &BTreeMap<Gate, Option<(Gate, Gate, Gate)>>,
    gate_left: &Gate,
    gate_right: &Gate,
) -> (usize, usize) {
    let mut set = HashSet::new();
    let s = build_tree(&mut set, tree, gate_left) + build_tree(&mut set, tree,
gate_right);
```

```rust
    (set.len(), s)
}

pub fn revere_generated_gates_tree(system: &[Gate]) -> Vec<(Gate, Gate, Gate)> {
    let reverse_system: Vec<_> = system
        .iter()
        .map(|gate| (gate, gate.reverse_search()))
        .collect();

    let mut gates_set: BTreeMap<_, _> = system.iter().map(|&g| (g, None)).collect();

    let mut todo_gates: Vec<_> = (0..NUM_GATES)
        .map(Gate)
        .filter(|g| !gates_set.contains_key(g))
        .collect();

    let mut found_gates = Vec::new();

    let mut stack = Vec::with_capacity(NUM_GATES_CELL);
    let mut a_gate = Vec::with_capacity(NUM_GATES_CELL);
    let mut b_gate = Vec::with_capacity(NUM_GATES_CELL);

    loop {
        todo_gates.retain(|&gate| {
            eprintln!("{:?}", gate);

            let mut best_gate = None;

            let gate_values: [Trit; NUM_GATES_CELL] = gate.into();

            for (&sys_gate, reverse_gate) in &reverse_system {
                let possibles_values = gate_values.map(|trit| &reverse_gate[trit]);

                stack.push(possibles_values.first().unwrap().iter().copied());

                while let Some(iter) = stack.last_mut() {
                    if let Some((a, b)) = iter.next() {
                        a_gate.push(a);
                        b_gate.push(b);

                        if gates_set
                            .range(
                                Gate::from(array::from_fn(|i| {
                                    a_gate.get(i).copied().unwrap_or(MIN_TRIT)
                                }))
                                    ..=Gate::from(array::from_fn(|i| {
                                        a_gate.get(i).copied().unwrap_or(MAX_TRIT)
                                    })),
                            )
                            .next()
                            .is_some()
                            && gates_set
                                .range(
                                    Gate::from(array::from_fn(|i| {
                                        b_gate.get(i).copied().unwrap_or(MIN_TRIT)
                                    }))
                                        ..=Gate::from(array::from_fn(|i| {
                                            b_gate.get(i).copied().unwrap_or(MAX_TRIT)
                                        })),
                                )
                                .next()
                                .is_some()
                        {
                            let i = stack.len();
                            if i == NUM_GATES_CELL {
                                let x =
                                    Gate::from(array::from_fn(|i|
a_gate.get(i).copied().unwrap()));
                                let y =
                                    Gate::from(array::from_fn(|i|
b_gate.get(i).copied().unwrap()));

                                if let Some((_, a, b)) = best_gate {
                                    if tree_stats(&gates_set, &x, &y)
                                        < tree_stats(&gates_set, &a, &b)
                                    {
                                        best_gate = Some((sys_gate, x, y));
                                    }
                                } else {
                                    best_gate = Some((sys_gate, x, y));
                                }

                                a_gate.pop();
                                b_gate.pop();
                            } else {
                                stack.push(possibles_values[i].iter().copied());
                            }
                        } else {
                            a_gate.pop();
                            b_gate.pop();
                        }
                    } else {
                        stack.pop();

                        a_gate.pop();
                        b_gate.pop();
                    }
                }
            }

            if best_gate.is_some() {
                found_gates.push((gate, best_gate));
                return false;
            }

            return true;
        });

        if found_gates.is_empty() {
            return gates_set.into_values().filter_map(identity).collect();
        }

        gates_set.extend(found_gates.iter().copied());
        found_gates.clear();
    }
}
```

test_system.rs

```rust
use std::usize;

use tipe::{generated_gates_depth, revere_generated_gates_depth, Gate, NUM_GATES};

fn main() {
    let (g, d) = generated_gates_depth(&[Gate(2899), Gate(8543)], usize::MAX);

    println!("{:?}", g);

    println!("{} {}", g.len(), d);

    if NUM_GATES == g.len() {
        println!("Found system !");
    }
}
```

gates_tree.rs

```rust
use tipe::{revere_generated_gates_tree, Gate};

fn main() {
    let tree = revere_generated_gates_tree(&[Gate(2899), Gate(8543), Gate(19305), Gate(15897)]);
    for (z, x, y) in tree {
        let g = z.join(&x, &y);
        println!("{} {} {} {}", g.0, z.0, x.0, y.0);
    }
}
```

gates_stats.rs

```rust
use std::collections::HashMap;

use rand::seq::SliceRandom;
use tipe::{generated_gates, Gate, NUM_GATES};

fn main() {
    let mut gates = Vec::new();
    for id in 0..NUM_GATES {
        let g = Gate(id);
        if generated_gates(&[g]).len() == 27 {
            gates.push(g);
        }
    }

    let mut rng = rand::thread_rng();

    let mut map: HashMap<_, usize> = HashMap::new();

    loop {
        let &a = gates.choose(&mut rng).unwrap();
        let &b = gates.choose(&mut rng).unwrap();

        if a == b {
            continue;
        }

        let l = generated_gates(&[a, b]).len();

        *map.entry(l).or_default() += 1;

        println!("{:?} {:?} {}", a, b, l);
        println!("{:?}", map);
    }
}
```

lib_tests.rs

```rust
use std::collections::BTreeSet;

use tipe::{generated_gates, Gate, Trit};

// + 0 -
// + 0 0 0
// 0 0 0 0
// - 0 0 0
const ZERO: Gate = Gate(9841);

// + 0 -
// + + + +
// 0 0 0 0
// - - - -
const ID: Gate = Gate(19305);

// + 0 -
// + + - -
// 0 0 - +
// - 0 + 0
const G: Gate = Gate(13435);

// + 0 -
// + + - -
// 0 0 - +
// - 0 + 0
const H: Gate = Gate(18473);

// + 0 -
// + 0 - 0
// 0 + 0 0
// - - + +
const RAND: Gate = Gate(7892);

#[test]
fn eval_gate() {
    assert_eq!(Trit::Plus, G.eval(Trit::Plus, Trit::Plus));
    assert_eq!(Trit::Minus, G.eval(Trit::Zero, Trit::Zero));
    assert_eq!(Trit::Zero, G.eval(Trit::Minus, Trit::Minus));

    assert_eq!(Trit::Zero, G.eval(Trit::Zero, Trit::Plus));
    assert_eq!(Trit::Minus, G.eval(Trit::Plus, Trit::Zero));

    assert_eq!(Trit::Zero, G.eval(Trit::Minus, Trit::Plus));
    assert_eq!(Trit::Plus, G.eval(Trit::Zero, Trit::Minus));
}

#[test]
fn join_gate() {
    assert_eq!(ZERO, ZERO.join(&ZERO, &ZERO));
    assert_eq!(ZERO, ZERO.join(&G, &H));

    assert_eq!(ZERO, ID.join(&ZERO, &RAND));
    assert_eq!(RAND, ID.join(&RAND, &ZERO));
    assert_eq!(G, ID.join(&G, &H));
    assert_eq!(H, ID.join(&H, &G));

    assert_eq!(ZERO, RAND.join(&ZERO, &ZERO));
    assert_eq!(Gate(8462), RAND.join(&G, &H));
}

#[test]
fn reverse_search() {
    let rev = ZERO.reverse_search();
    assert!(rev[2].is_empty());
    assert_eq!(
        BTreeSet::from([
            (Trit::Plus, Trit::Plus),
            (Trit::Plus, Trit::Zero),
            (Trit::Plus, Trit::Minus),
            (Trit::Zero, Trit::Plus),
            (Trit::Zero, Trit::Zero),
            (Trit::Zero, Trit::Minus),
            (Trit::Minus, Trit::Plus),
            (Trit::Minus, Trit::Zero),
            (Trit::Minus, Trit::Minus)
        ]),
        rev[1].iter().copied().collect()
    );
    assert!(rev[0].is_empty());

    let rev = ID.reverse_search();
    assert_eq!(
        BTreeSet::from([
            (Trit::Plus, Trit::Plus),
            (Trit::Plus, Trit::Zero),
            (Trit::Plus, Trit::Minus),
        ]),
        rev[2].iter().copied().collect()
    );
    assert_eq!(
        BTreeSet::from([
            (Trit::Zero, Trit::Plus),
            (Trit::Zero, Trit::Zero),
            (Trit::Zero, Trit::Minus),
        ]),
        rev[1].iter().copied().collect()
    );
    assert_eq!(
        BTreeSet::from([
            (Trit::Minus, Trit::Plus),
            (Trit::Minus, Trit::Zero),
            (Trit::Minus, Trit::Minus),
        ]),
        rev[0].iter().copied().collect()
    );

    let rev = RAND.reverse_search();
    assert_eq!(
        BTreeSet::from([
            (Trit::Zero, Trit::Plus),
            (Trit::Minus, Trit::Zero),
            (Trit::Minus, Trit::Minus),
        ]),
        rev[2].iter().copied().collect()
    );
    assert_eq!(
        BTreeSet::from([
            (Trit::Plus, Trit::Plus),
            (Trit::Plus, Trit::Minus),
            (Trit::Zero, Trit::Zero),
            (Trit::Zero, Trit::Minus),
        ]),
        rev[1].iter().copied().collect()
    );
    assert_eq!(
        BTreeSet::from([(Trit::Plus, Trit::Zero), (Trit::Minus, Trit::Plus),]),
        rev[0].iter().copied().collect()
    );
}
```

```rust
#[test]
fn generate_gates() {
    assert_eq!(vec![ZERO], generated_gates(&[ZERO]));

    assert_eq!(vec![ID], generated_gates(&[ID]));

    assert_eq!(
        BTreeSet::from([ZERO, ID]),
        generated_gates(&[ZERO, ID]).iter().copied().collect()
    );

    assert_eq!(
        BTreeSet::from([ZERO, ID]),
        generated_gates(&[ID, ZERO]).iter().copied().collect()
    );

    let set = BTreeSet::from([
        RAND,
        Gate(9841),
        Gate(9594),
        Gate(7645),
        Gate(7398),
        Gate(12284),
        Gate(12037),
        Gate(11790),
        Gate(10088),
    ]);
    assert_eq!(set, generated_gates(&[RAND]).iter().copied().collect());

    assert_eq!(
        set,
        generated_gates(&[RAND, RAND]).iter().copied().collect()
    );

    let gates: BTreeSet<_> = generated_gates(&[RAND, ID]).iter().copied().collect();
    assert_eq!(
        gates,
        generated_gates(&[ID, RAND]).iter().copied().collect()
    );

    assert!(gates.is_superset(&set));

    assert!(gates.contains(&ID.join(&RAND, &RAND)));
    assert!(gates.contains(&ID.join(&ID, &RAND)));
    assert!(gates.contains(&RAND.join(&RAND, &ID)));
    assert!(gates.contains(&RAND.join(&ID.join(&ID, &RAND), &ID)));
    assert!(gates.contains(&RAND.join(&RAND.join(&RAND, &ID), &RAND.join(&RAND,
&ID))));
    assert!(gates.contains(&RAND.join(&RAND.join(&RAND.join(&RAND, &ID), &RAND),
&RAND)));
}
```

old/v2.3.rs

```rust
use rand::Rng;
use tipe::Gate;

// |
// C
// |
// / \
// A B
// | |

fn main() {
    let seed = [
        142, 1139, 1343, 3089, 3395, 3600, 4928, 5065, 5177, 7081, 8489, 9690, 10554, 10820, 11540,
        11814, 11892, 12482, 13714, 14493, 15101,
    ];

    let mut input = Vec::from_iter(seed.map(Gate::from));

    for n_i in 0..19683 {
        let n = Gate::from(n_i);

        let mut v = input.clone();

        'main: loop {
            for mut a in v.iter() {
                for mut b in v.iter() {
                    let mut c = &n;
                    for _ in 0..3 {
                        let g = c.join(a, b);

                        if &g != a && &g != b && &g != c {
                            if let Some(index) = v.iter().position(|x| x == &g) {
                                v.remove(index);
                                continue 'main;
                            }
                        }

                        (a, b, c) = (b, c, a);
                    }
                }
            }

            break;
        }

        v.push(n);

        if v.len() < input.len() {
            input = v;
        }
    }

    println!("{:?}", input);
}
```

old/v2.2.rs

```rust
use rand::Rng;
use tipe::Gate;

// |
// C
// |
// / \
// A B
// | |

fn main() {
    let seed = [
        142, 1139, 1343, 3089, 3395, 3600, 4928, 5065, 5177, 7081, 8489, 9690, 10554,
10820, 11540,
        11814, 11892, 12482, 13714, 14493, 15101,
    ];

    let mut input = Vec::from_iter(seed.map(Gate::from));

    for n_i in 0..19683 {
        let n = Gate::from(n_i);

        let mut v = input.clone();
        v.push(n);

        'main: loop {
            for c in v.iter() {
                for b in v.iter() {
                    for a in v.iter() {
                        let g = c.join(a, b);

                        if &g != a && &g != b && &g != c {
                            if let Some(index) = v.iter().position(|x| x == &g) {
                                v.remove(index);
                                continue 'main;
                            }
                        }
                    }
                }
            }

            break;
        }

        if v.len() < input.len() {
            input = v;
        }
    }

    println!("{:?}", input);
}
```

old/v2.0.rs

```rust
use rand::Rng;
use tipe::Gate;

// |
// C
// |
// / \
// A B
// | |

fn main() {
    let seed = [
        142, 1139, 1343, 3089, 3395, 3600, 4928, 5065, 5177, 7081, 8489, 9690, 10554, 10820, 11540,
        11814, 11892, 12482, 13714, 14493, 15101,
    ];

    let mut input = Vec::from_iter(seed.map(Gate::from));

    for c_i in 0..19683 {
        let c = Gate::from(c_i);

        let mut v = input.clone();

        'main: loop {
            for a in v.iter() {
                for b in v.iter() {
                    let g = c.join(a, b);

                    if &g != a && &g != b && g != c {
                        if let Some(index) = v.iter().position(|x| x == &g) {
                            v.remove(index);
                            continue 'main;
                        }
                    }
                }
            }

            break;
        }

        v.push(c);

        if v.len() < input.len() {
            input = v;
        }
    }

    println!("{:?}", input);
}
```

old/v1.0.rs

```rust
use rand::Rng;
use tipe::Gate;

// |
// C
// |
// / \
// A B
// | |

fn main() {
    let seed = 0..19683;
    let mut input = Vec::from_iter(seed.map(Gate::from));

    let mut rng = rand::thread_rng();

    let mut c = 0;
    let mut p = 0;

    loop {
        let l = input.len();
        if p == l {
            c += 1;
            if c > 19683 {
                break;
            }
        } else {
            p = l;
            c = 0;
        }

        let a = &input[rng.gen_range(0..l)];
        let b = &input[rng.gen_range(0..l)];
        let c = &input[rng.gen_range(0..l)];
        let g = c.join(a, b);

        if &g != a && &g != b && &g != c {
            if let Some(index) = input.iter().position(|x| x == &g) {
                input.remove(index);
            }
        }
    }

    println!("{:?}", input);
}
```