Présentation du Solver SMT "Z3"

Nicolas Lhost

Wim Vanhoof

Étienne Payet
Fred Mesnard

Z3

Introduction

Complexité et Problèmes NP

- La théorie de la **complexité** est le domaine des mathématiques et de l'informatique qui étudie formellement le **temps de calcul** requis par un algorithme pour résoudre un problème algorithmique.

  Exemple: Le problème du voyageur de commerce

- Différentes classes: P, NP, EXP, etc.

- NP-Complet: Les plus difficiles de NP

- Les problèmes NPC peuvent être transformés en problème SAT (Théorème de Cook, 1971)

Problème SAT (satisfiabilité booléenne)

- Permet de déterminer si un problème est satisfiable
  - Input: Formule de logique propositionnelle avec k variables
  - Output: Vrai ou Faux

$$((A \wedge B) \vee C) \wedge \neg(B \vee A)) \wedge (B \vee \neg C)$$

$\Rightarrow$ Calculer la table de vérité de la formule pour voir si on a un 1 pour la globalité (se calcule en temps 2^k)

Par force brute, ça va prendre énormément de temps. On va donc utiliser des solvers SAT pour les problèmes de décision

Solver SAT

- Conversion en forme normale conjonctive (CNF)
  - Conjonction de clauses
  - Une clause est une disjonction de littéraux
  - Un littéral est une variable propositionnelle soit un négation

$$(x_1 \vee x_3 \vee \neg x_2) \wedge x_4 \wedge (\neg x_2 \vee \neg x_4)$$

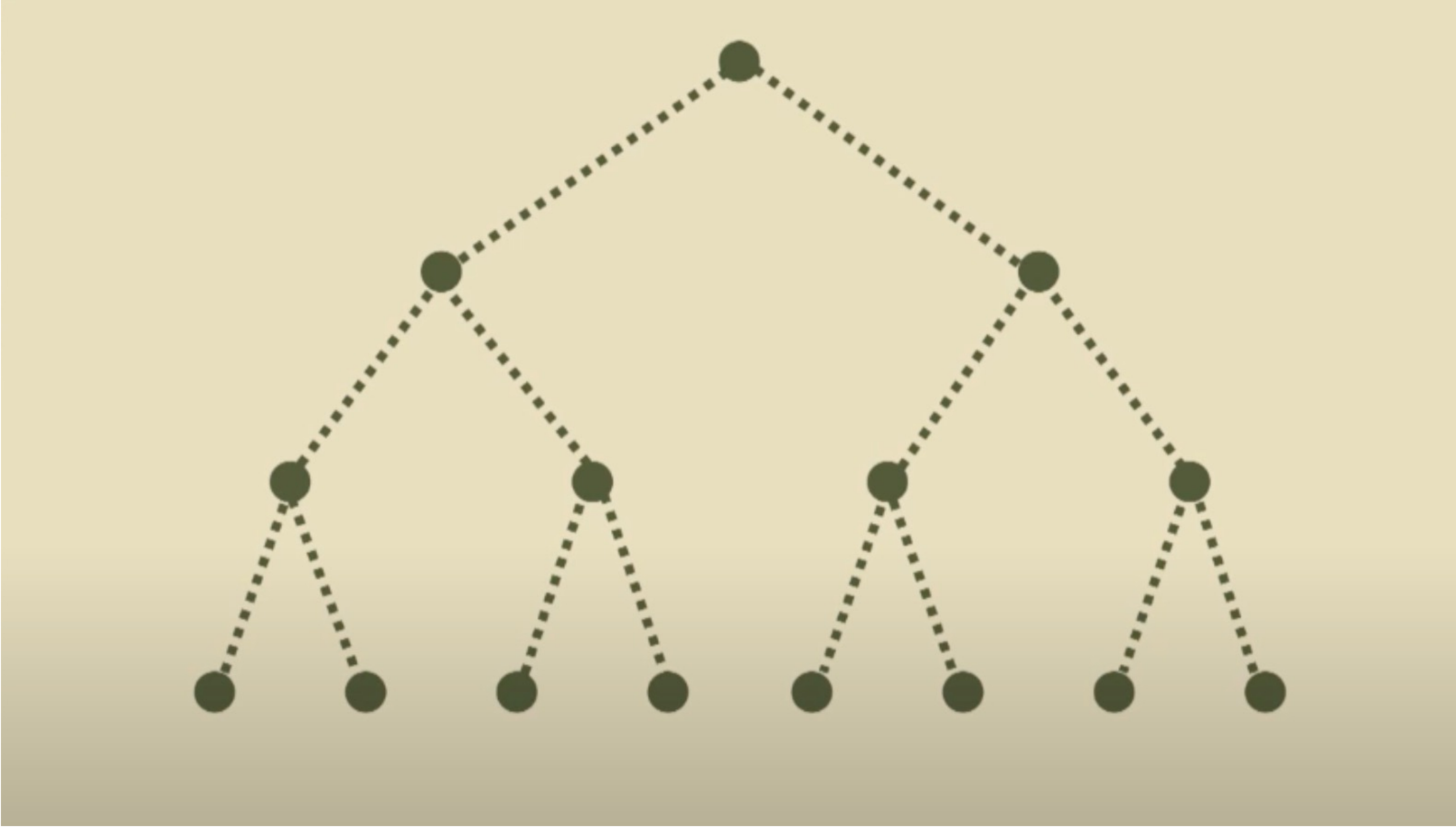
- On peut écrire n'importe quelle formule propositionnelle en CNF

$$A \Rightarrow B \sim \neg A \vee B$$

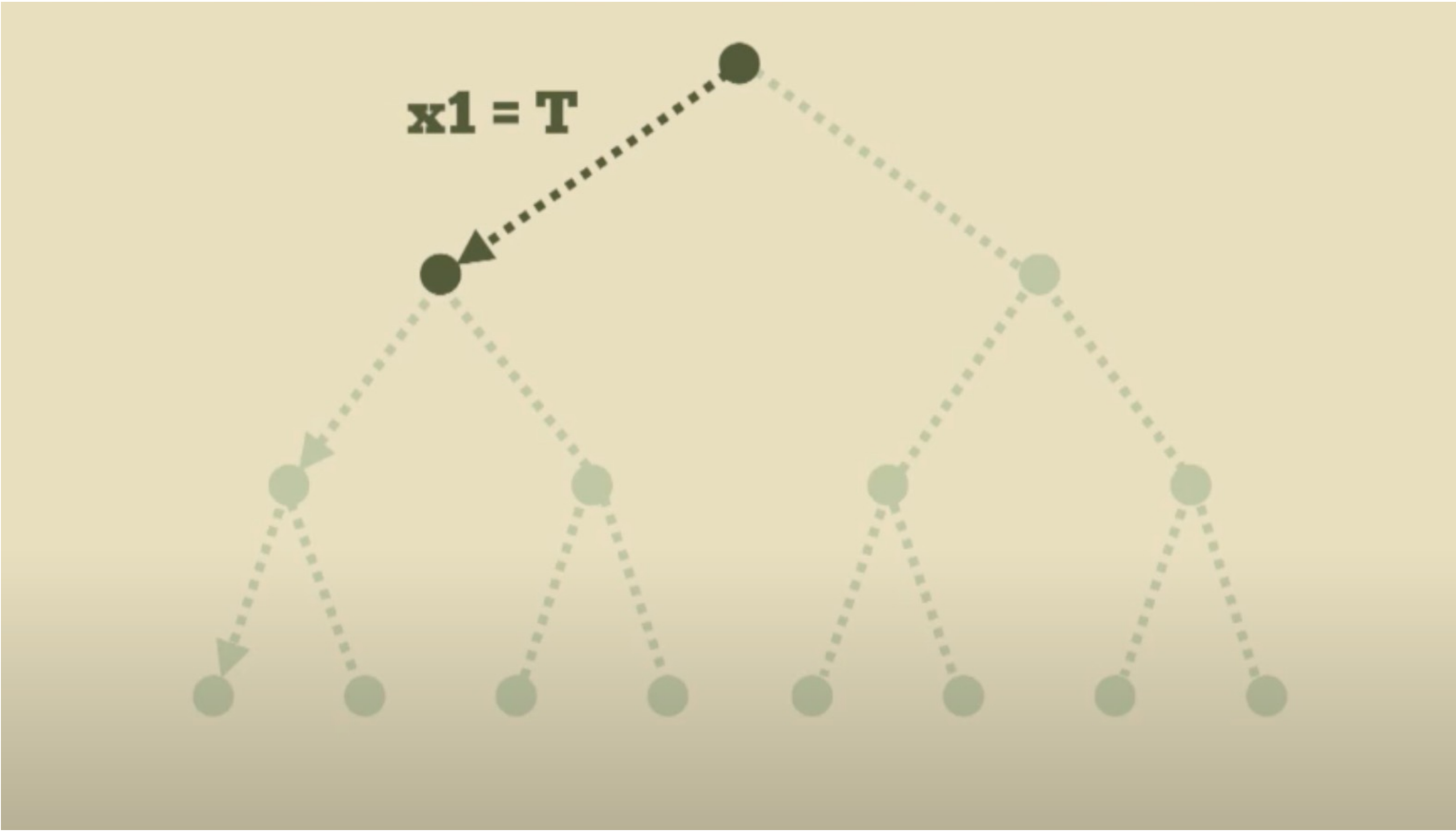
$$(B \vee A) \Rightarrow C \sim (\neg B \vee C) \wedge (\neg A \vee C)$$

$$A \Rightarrow (B \wedge C) \sim (\neg A \vee B) \wedge (\neg A \vee C)$$

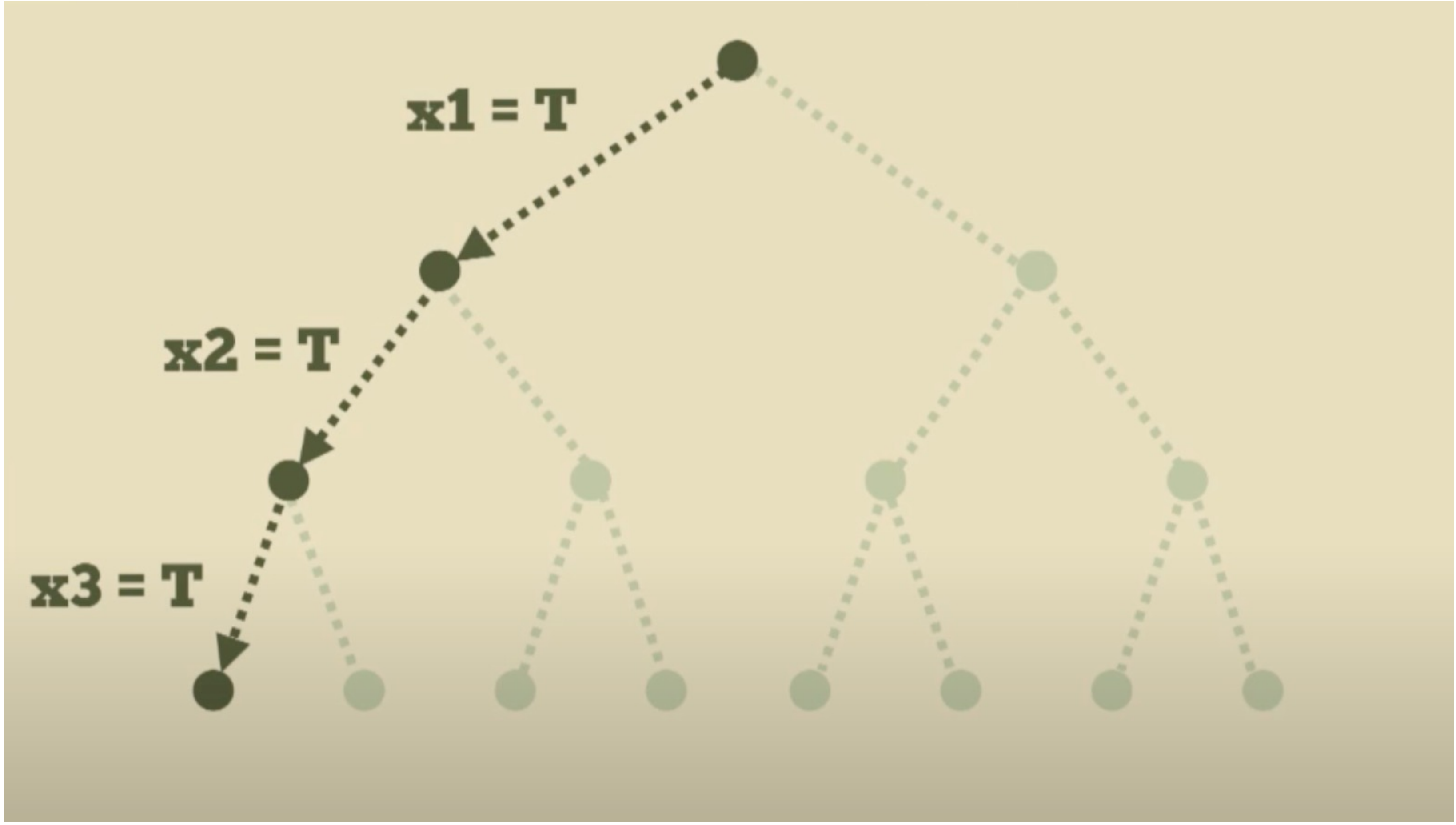
Algorithme Deepfirst



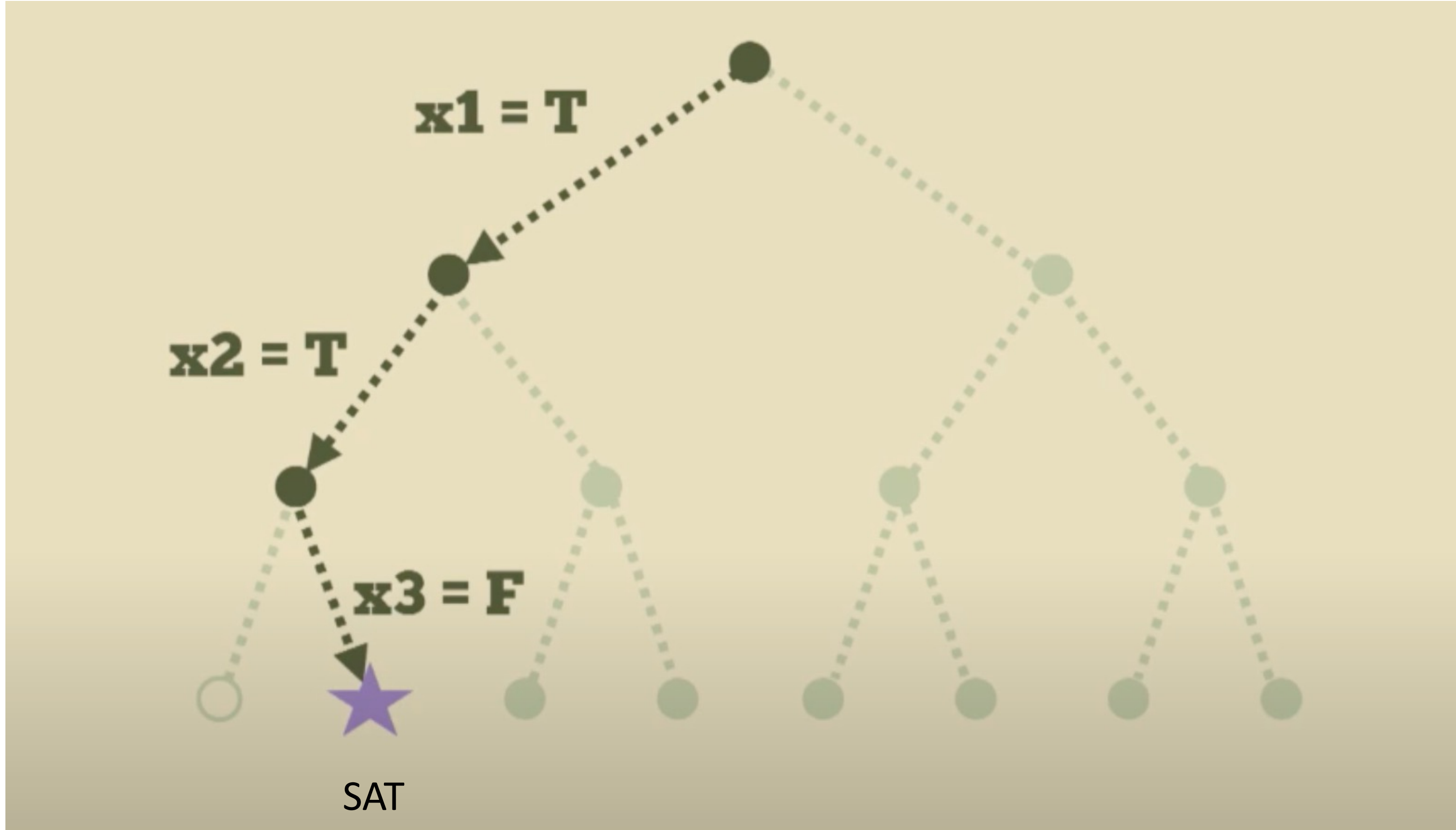
Algorithme Deepfirst



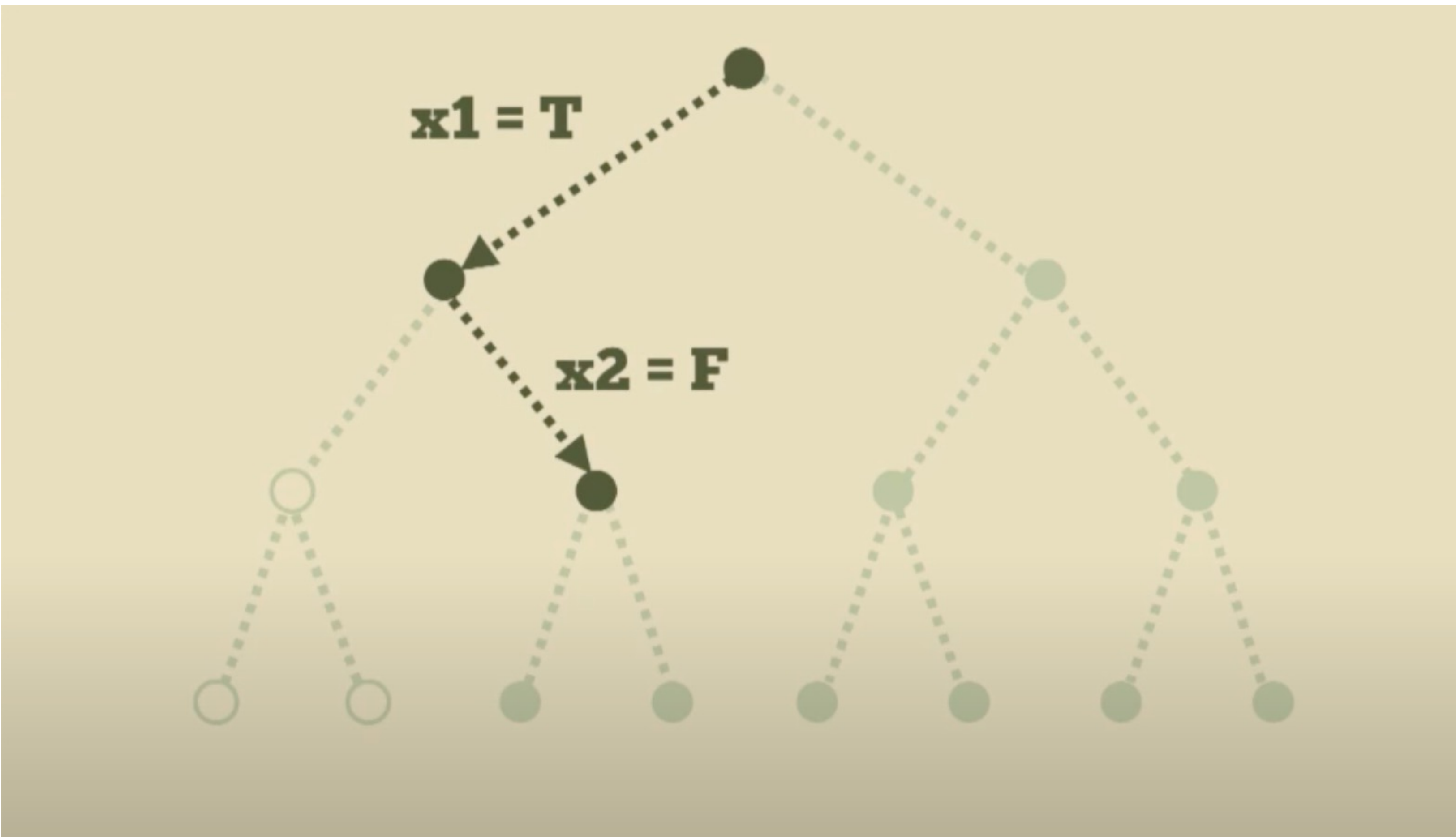
Algorithme Deepfirst



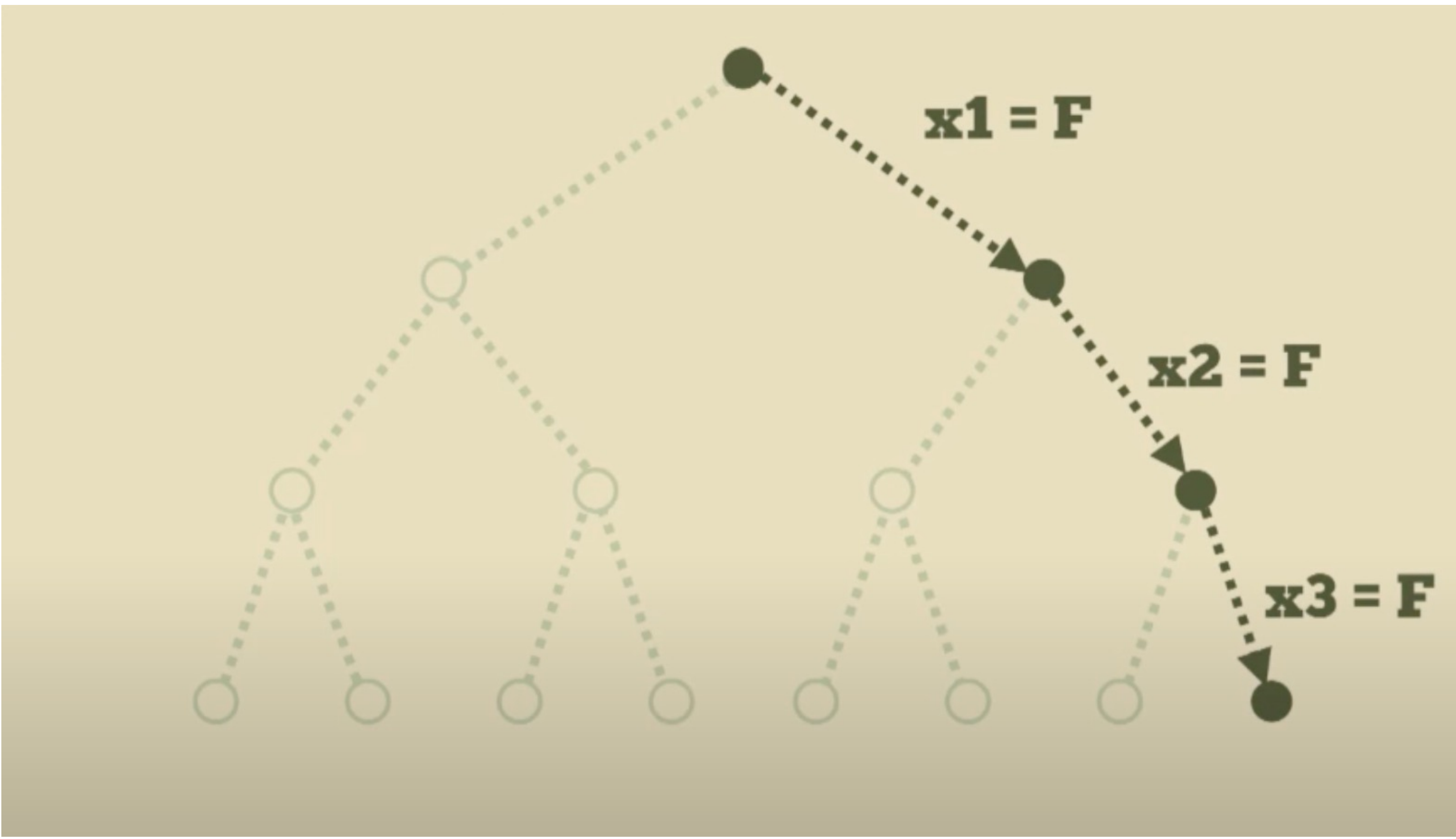
Algorithme Deepfirst



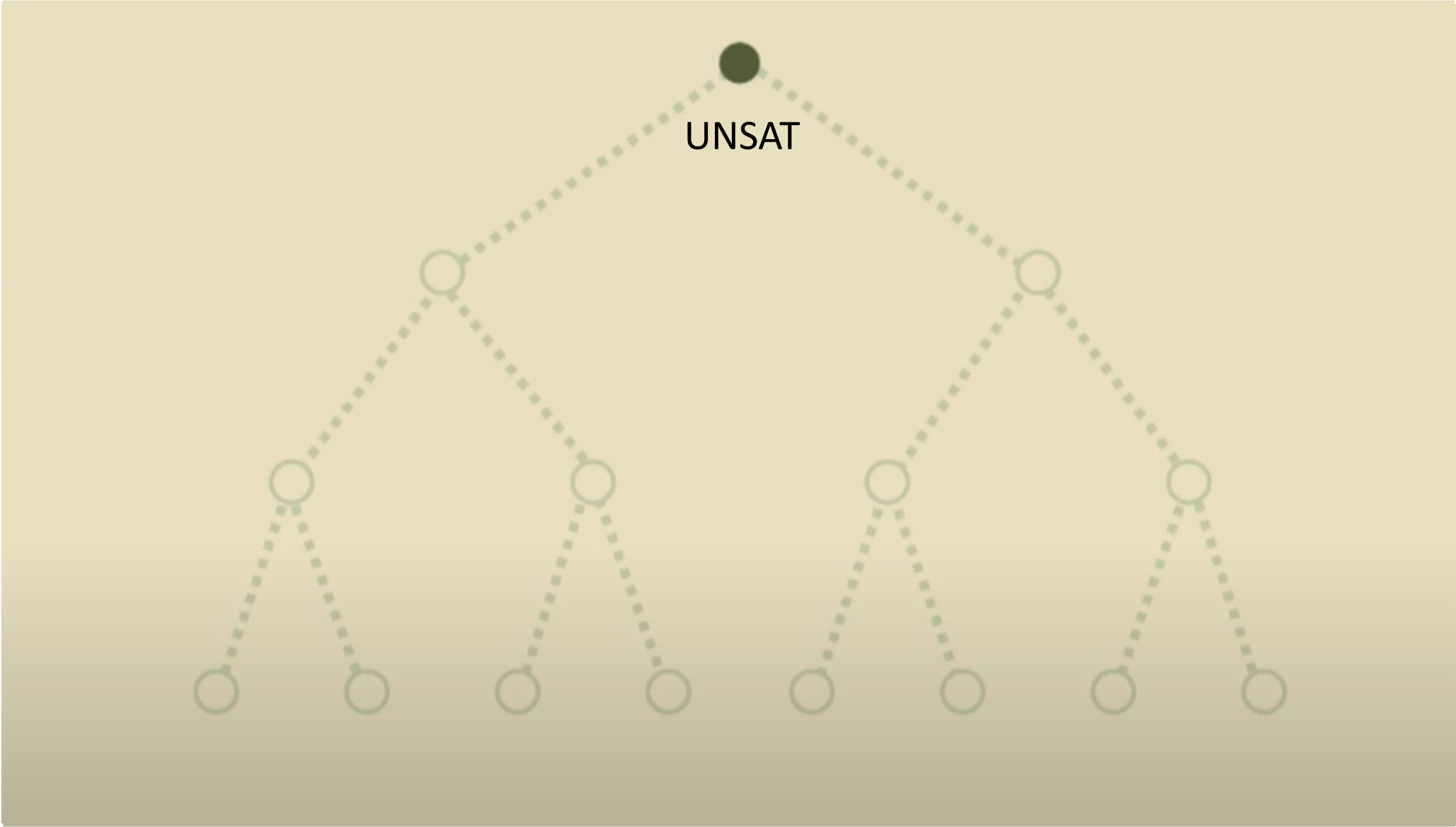
Algorithme Deepfirst



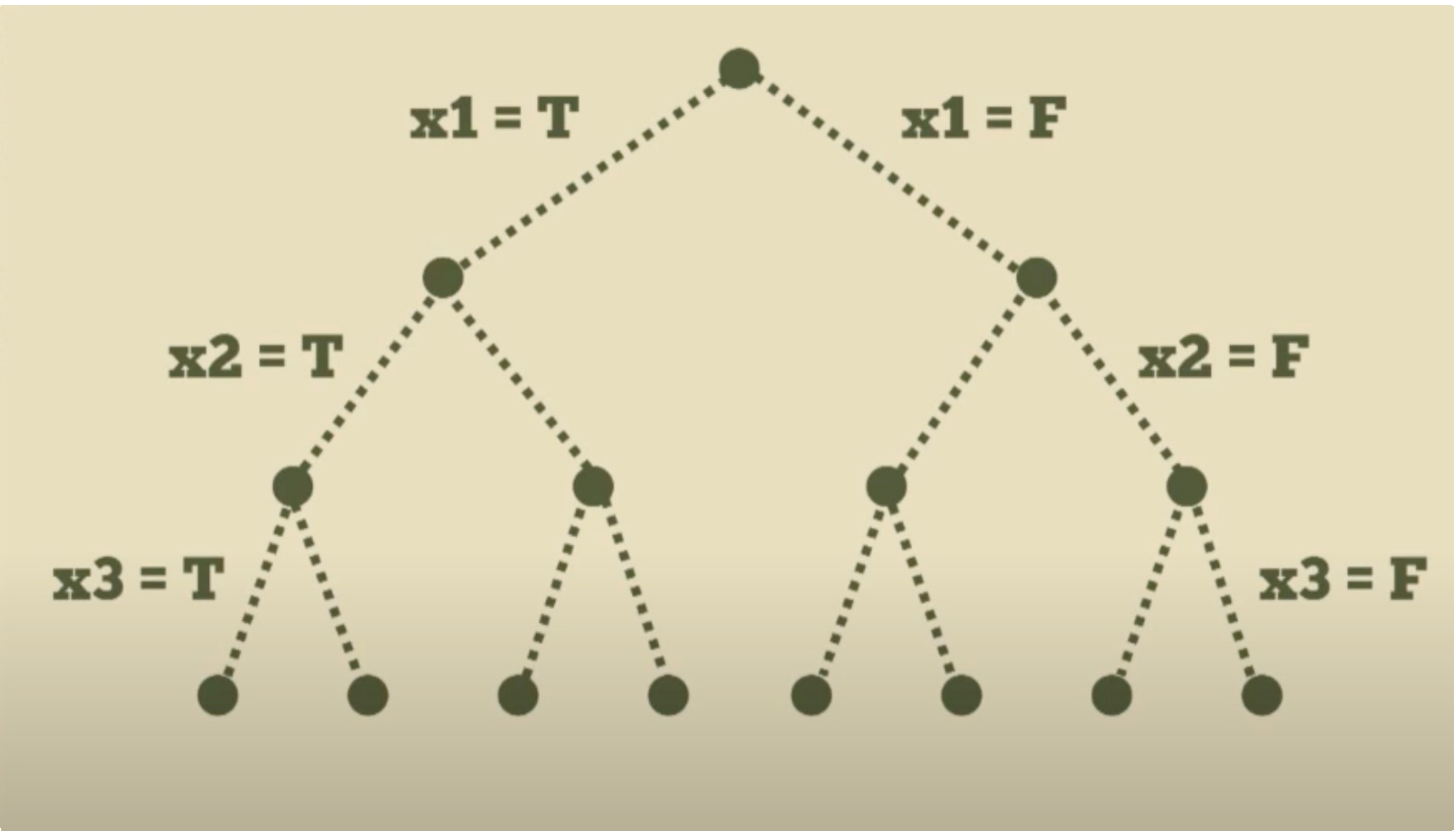
Algorithme Deepfirst



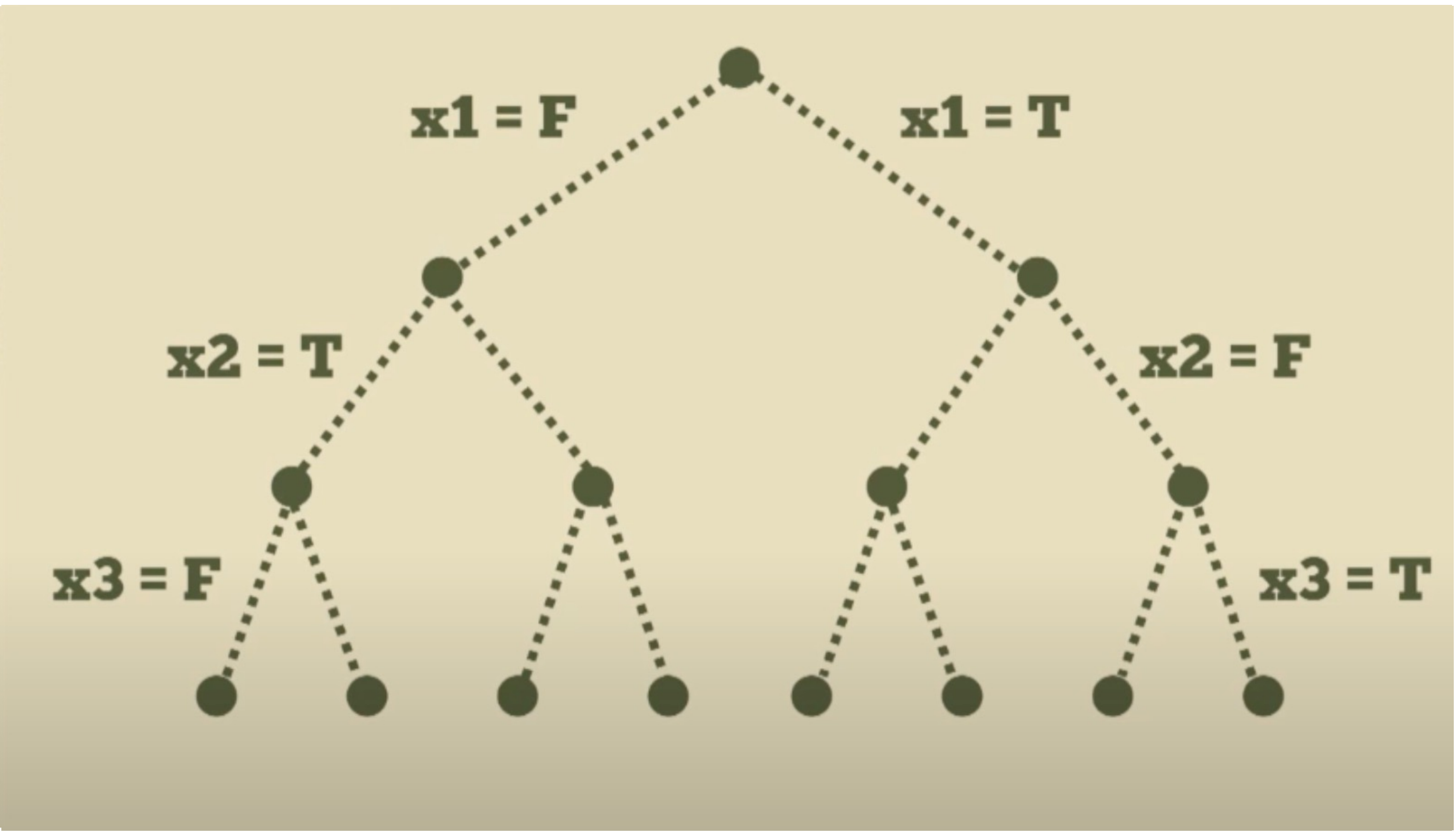
Algorithme Deepfirst



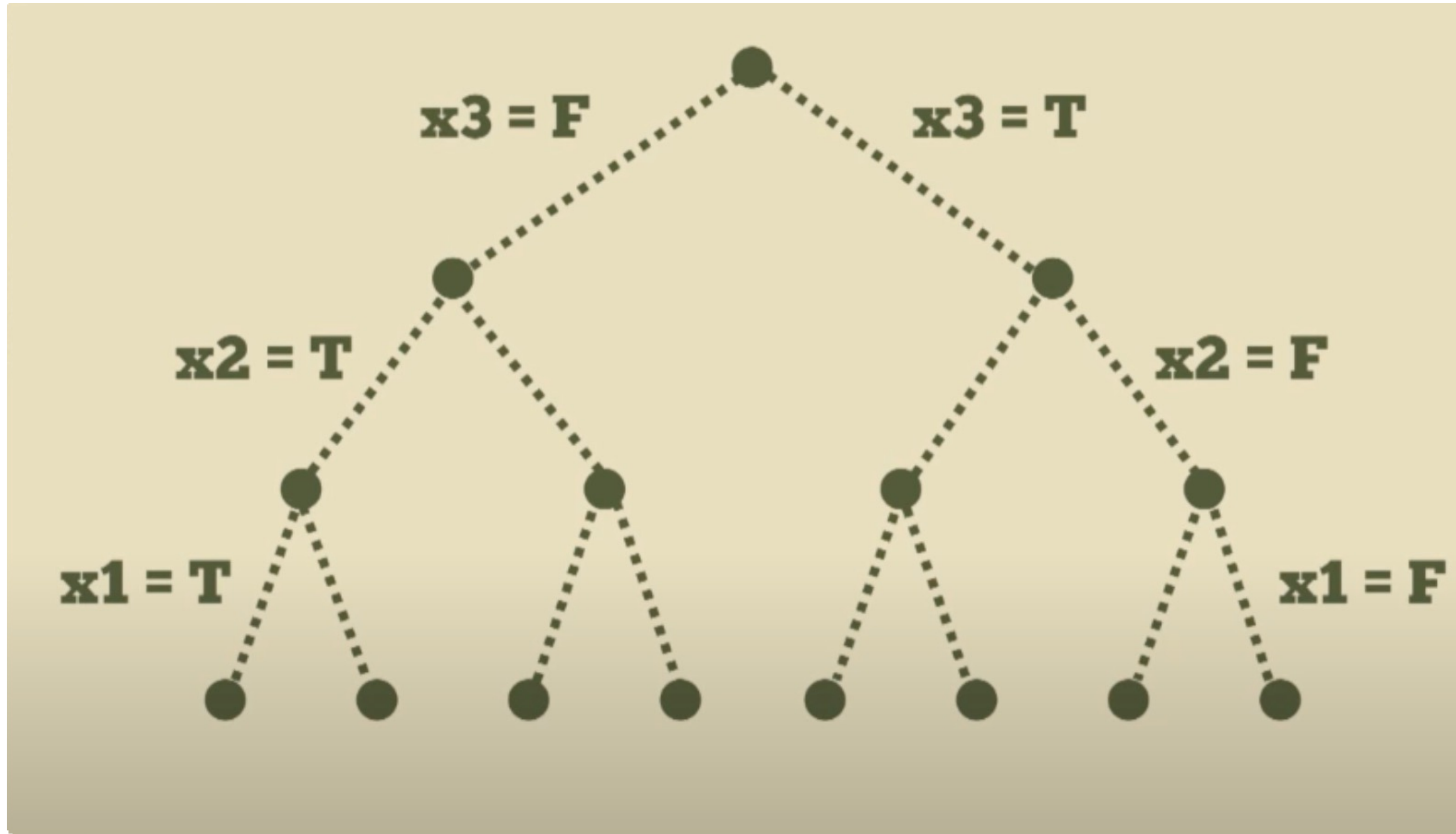
Algorithme Deepfirst



Algorithme Deepfirst



Algorithme Deepfirst



Algorithmes SAT

- Le brute force ne fonctionnera pas, il faut choisir intelligemment les littéraux et clauses

- Unit propagation

```
;; x1 = false
;; x2 = false
(or x1 x2 x3)

;; x3 => true
```

```
(and (or x4)
     (or (not x4) x5))

;; x4 => true
;; x5 => true
```

		5		8		3	9	
				6				8
8	4	6	7	3				2
3		7		5			2	
1	5						4	9
	9			2		7		3
6				1	5	9	3	7
5				9				
	8	3		4		1		

Chercher des infos faciles et trouver des conflits

- Algorithme DPLL, CDCL, etc.

Z3

Satisfiability Modulo Theory (SMT)

Solver SMT

« Problème de décision pour des formules de **logique du premier ordre** avec égalité (sans quantificateurs), combinées à des **théories** dans lesquelles sont exprimées certains symboles de prédicat et/ou certaines fonctions » - Wikipedia

$\Rightarrow$ Théories: Arithmétique (réels, integers), Bitvectors, Arrays, etc.

$\Rightarrow$ Logique du premier ordre:

- Variables/constantes: x, y
- Fonctions: foo(x)
- Prédicats (ou relations) sur les éléments: x > y
- Connecteurs logiques: &&, ||, =>, not
- Quantificateurs: ∃, ∀

Fonctionnement de base d'un solveur SMT

Deux cœurs principaux : Solver SAT et une ou plusieurs procédures de décision de la théorie

$$(x < 0 \vee x > 1) \wedge (x = y + 5) \wedge (y > 0)$$

(a V b) ^ (c) ^ (d)

⇒ Si pas de modèle SAT, la formule n’est pas satisfiable

⇒ Si modèle SAT, vérification de la cohérence avec des procédures de décision de la théorie

- ⇒Si cohérence, la formule est satisfiable
- ⇒Si pas cohérence, on cherche un autre modèle

SMT-LIB2

- Créé en 2003. Initiative internationale pour faciliter la recherche et le développement
- Langage de spécification standard pour les solveurs SMT. Il en existe un grand nombre qui se basent sur ce standard, dont Z3.
- Syntaxe proche du LISP

```
(assert (= (+ (* 6 x) (* 2 y) (* 12 z)) 30))
(assert (= (+ (* 3 x) (* 6 y) (* 3 z)) 12))
```

- Bibliothèque de Théories et de Logiques
- Vérification de programme, sécurité informatique, planification, synthèse de circuits, etc.

Z3

- Créé en 2012 par Microsoft Research.

$\Rightarrow$ Open Source en 2015

- Le format d'entrée par défaut est SMT-LIB2

- Variété d'interfaces pour la spécification et la résolution de problèmes SMT. API en C++, en Java, en Python, en .NET, etc.

$\Rightarrow$ Intégration possible directement dans le code grâce aux API

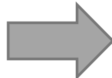
Z3

Le langage SMT-LIB2 et Z3

Commandes de base

- Affichage de texte dans l’output `(echo "starting Z3...")`
- Déclaration de constantes et de fonctions
```
(declare-const a Int)
(declare-fun f (Int Bool) Int)
```
- Assertion de formules
```
(assert (> a 10))
(assert (< (f a true) 100))
```
- Vérification de satisfiabilité: sat, unsat ou unknown `(check-sat)`
- Instanciation des variables et fonctions qui rendent le model sat `(get-model)`

```
(echo "starting Z3...")
(declare-const a Int)
(declare-fun f (Int Bool) Int)
(assert (> a 10))
(assert (< (f a true) 100))
(check-sat)
(get-model)
```



```
starting Z3...
sat
(model
  (define-fun a () Int
    11)
  (define-fun f ((x!1 Int) (x!2 Bool)) Int
    (ite (and (= x!1 11) (= x!2 true)) 0
      0))
)
```

Constantes et fonctions non interprétées

- Z3 permet de définir des constantes (sucre syntaxique) mais dans le standard SMT-LIB tout est fonction.
  - $\Rightarrow$ Une constante est une fonction nullaire (sans argument)

```
(declare-fun a () Int) = (declare-const a Int)
```

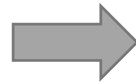
- Les fonctions en logique du premier ordre n'ont pas d'effets secondaires et sont totales -> Renvoie toujours quelque chose

- Les fonctions et constantes en logique pure du premier ordre sont non interprétés/libres

Constantes et fonctions non interprétées

- Illustration:

```
(declare-sort A)
(declare-const x A)
(declare-const y A)
(declare-fun f (A) A)
(assert (= (f (f x)) x))
(assert (= (f x) y))
(assert (not (= x y)))
(check-sat)
(get-model)
```

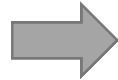


```
sat
(model
  ;; universe for A:
  ;;   A!val!1 A!val!0
  ;; -----------
  ;; definitions for universe elements:
  (declare-fun A!val!1 () A)
  (declare-fun A!val!0 () A)
  ;; cardinality constraint:
  (forall ((x A)) (or (= x A!val!1) (= x A!val!0)))
  ;; -----------
  (define-fun y () A
    A!val!1)
  (define-fun x () A
    A!val!0)
  (define-fun f ((x!1 A)) A
    (ite (= x!1 A!val!0) A!val!1
    (ite (= x!1 A!val!1) A!val!0
      A!val!1)))
)
```

Scopes

- Z3 maintient un stack/une pile des déclarations et assertions
  - Push: Crée un nouveau scope
  - Pop: Retire le scope dernier scope

```
(declare-const x Int)
(declare-const y Int)
(push)
(assert (= (+ x (* 2 y)) 20))
(get-assertions)
(check-sat)
(pop)
(push)
(declare-const p Bool)
(assert (and (= (+ (* 2 x) (* 2 y)) 21) (= p true)))
(get-assertions)
(check-sat)
(pop)
(assert (and (= (+ (* 2 x) (* 2 y)) 21) (= p true)))
```



```
((= (+ x (* 2 y)) 20))
sat
((and (= (+ (* 2 x) (* 2 y)) 21) (= p true)))
unsat
(error "line 15 column 43: unknown constant p")
```

Configuration

- Pour demander les assertions, il faut être en mode interactif

```
(set-option :interactive-mode true)
```

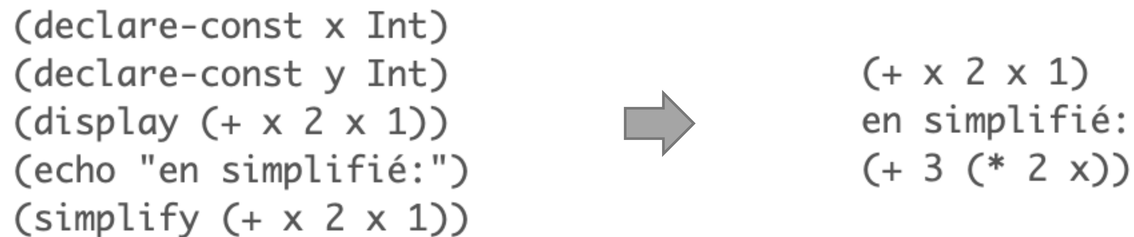
- Set-option permet de configurer Z3

```
(set-option :timeout 2000)

(set-option :produce-models true)

(set-option :print-success true)
```

- Les options sont utilisables dans certains états du solver
  => Reset per revient à un état de base `(reset)`

Commandes additionnelles

- Display: Affiche une formule
- Simplify: Permet de simplifier une formule



- Define-sort: crée un nouveau symbole sort (type) qui est une abréviation d'expression de sort

```
(define-sort List-Set (T) (Array (List T) Bool))
(declare-const s2 (List-Set Int))
```

Z3

Les théories

La théorie derrière les théories

- Z3 permet d'utiliser les théories de base de SMT-LIB:

Logique propositionnelle, Arithmétique (Entiers et Réels), Bitvectors, Arrays, DataType, Floating Points, String

- Les **Logiques** sont des sous/mélanges de théories

  $\Rightarrow$ appliquer des techniques de satisfiabilité spécialisées et plus efficaces

  ABV, ALL, AUFBV, AUFLIA, AUFNIRA, BV, HORN, LIA, NRA, QF_AUFLIA, QF_AUFLIRA, QF_BV, QF_BVFP, QF_FD, QF_FP, QF_FP, QF_FPA, QF_FPBV, QF_IDL, QF_LIA, QF_LRA, QF_NIA, QF_NRA, QF_NRA, QF_RDL, QF_S, QF_UF, QF_UFBV, QF_UFLIA, UFBV, UFLIA.

QF_LRA: Arithmétique linéaire réelle non quantifiée. Combinaisons booléennes d'inéquations entre polynômes linéaires sur des variables réelles.

AUFLIA: Arithmétique linéaire des nombres entiers et des tableaux qui renvoient n'importe quel sort(type) et fonction; les indices de ces tableaux sont limités aux valeurs entières.

Logique propositionnelle

- Z3 supporte les valeurs booléennes (Bool)
- Opérateurs and, or, xor, not, => (implication), ite (if-then-else) et = (équivalent à double implication)

```
(declare-const p Bool)
(declare-const q Bool)
(declare-const r Bool)
(define-fun conjecture () Bool
        (=> (and (=> p q) (=> q r))
                (=> p r)))
(assert (not conjecture))
(check-sat)
```

$\Rightarrow$ unsat

Une formule F est **valide** si F est toujours évaluée comme vraie pour chaque assignation possible.
On cherche une **preuve.**

Une formule F est **satisfiable** si F est évaluée comme vraie pour au moins une assignation.
On cherche une **solution.**

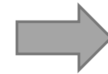
Arithmétique

- Supporte les entiers et les réels (vision mathématique)
- Opérateurs arithmétiques: +, -, <, etc.
- Transformation des entiers en réels avec to_real `(+ (to_real c) 0.5)`
- Arithmétique non linéaire possible

Arithmétique non linéaire

Plus compliqué pour Z3 car indécidable -> peut retourner des unknown

```
(declare-const a Int)
(assert (> (* a a) 3))
(check-sat)
(get-model)

(declare-const b Real)
(declare-const c Real)
(assert (= (+ (* b b b) (* b c)) 3.0))
(check-sat)

(reset)

(declare-const b Real)
(declare-const c Real)
(assert (= (+ (* b b b) (* b c)) 3.0))
(check-sat)
(get-model)
```



```
sat
(model
  (define-fun a () Int
    (- 8))
)

unknown

sat
(model
  (define-fun b () Real
    (/ 1.0 8.0))
  (define-fun c () Real
    (/ 1535.0 64.0))
)
```

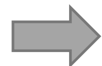
Arithmétique: La division

- Z3 supporte la division, le modulo et l’opérateur de reste
- Z3 autorise la division par 0 -> les fonctions sont totales

  Le résultat ne sera cependant pas spécifié

```
(div a 4)
(mod a 4)
(rem a 4)
```

```
(declare-const a Real)
(declare-const b Real)
(assert (= (/ a 0.0) 10.0))
(assert (= (/ b 2.0) 10.0))
(check-sat)
(get-model)
(assert (= (/ a 0.0) 2.0))
(check-sat)
```



```
sat
(model
  (define-fun b () Real
    20.0)
)
unsat
```

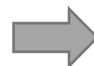
Possibilité d’empêcher la division par 0 avec un ite

```
(define-fun mydiv ((x Real) (y Real)) Real
  (if (not (= y 0.0))
      (/ x y)
      0.0))
(declare-const a Real)
(declare-const b Real)
(assert (>= (mydiv a b) 1.0))
(assert (= b 0.0))
(check-sat)
```

Bitvectors

- Les CPU et principaux langages de programmation utilisent l'arithmétique sur des vecteurs de bits de taille fixe.
- Bitvectors signés ou non signés
- Supporte les Bitvectors de taille arbitraire:
  - Binaire: `#b0100 ;bitvector de taille 4 représentant 4`
  - Hexadécimal: `#x0a0 ;bitvector de taille 12 représentant 160`
  - Taille fixée : `(_ bv10 32) ;bitvector de taille 32 représentant 10`
- Choisir l'affichage avec option

```
(display (_ bv10 32))
(set-option :pp.bv-literals false)
(display #x0a0)
(display #b0100)
```



```
#x0000000a

(_ bv160 12)
(_ bv4 4)
```

Bitvectors

Opérations arithmétiques

```
(bvadd #x07 #x03) ; addition
(bvsub #x07 #x03) ; subtraction
(bvneg #x07) ; unary minus
(bvmul #x07 #x03) ; multiplication
(bvurem #x07 #x03) ; unsigned remainder
(bvsrem #x07 #x03) ; signed remainder
(bvsmod #x07 #x03) ; signed modulo
(bvshl #x07 #x03) ; shift left
(bvlshr #xf0 #x03) ; unsigned (logical) shift right
(bvashr #xf0 #x03) ; signed (arithmetical) shift right
```

deMorgan en Bitvectors

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert (not (= (bvand (bvnot x) (bvnot y)) (bvnot (bvor x y)))))
(check-sat)
```

Opérateurs Bitwise

```
(bvor #x6 #x3)   ; bitwise or
(bvand #x6 #x3)  ; bitwise and
(bvnot #x6) ; bitwise not
(bvnand #x6 #x3) ; bitwise nand
(bvnor #x6 #x3) ; bitwise nor
(bvxnor #x6 #x3) ; bitwise xnor
```

Prédicats sur les bitvecteurs

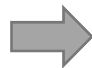
```
(bvule #x0a #xf0)  ; unsigned less or equal
(bvult #x0a #xf0)  ; unsigned less than
(bvuge #x0a #xf0)  ; unsigned greater or equal
(bvugt #x0a #xf0)  ; unsigned greater than
(bvsle #x0a #xf0)  ; signed less or equal
(bvslt #x0a #xf0)  ; signed less than
(bvsge #x0a #xf0)  ; signed greater or equal
(bvsgt #x0a #xf0)  ; signed greater than
```

Arrays

- Théorie basique des tableaux:
  - Select: `(select a i)` retourne la valeur du tableau a stockée à l'indice i
  - Store: `(store a i v)` retourne un tableau identique à a contenant v à l'indice i
  - Déclaration: `(declare-const a1 (Array Int Int))` peut prendre n'importe quel sort (type)

- Exemple:

```
(declare-const x Int)
(declare-const y Int)
(declare-const a1 (Array Int Int))
(assert (= (select a1 x) x))
(assert (= (store a1 x y) a1))
(check-sat)
(get-model)
```

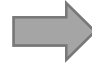


```
sat
(model
  (define-fun y () Int
    1)
  (define-fun a1 () (Array Int Int)
    (_ as-array k!0))
  (define-fun x () Int
    1)
  (define-fun k!0 ((x!1 Int)) Int
    (ite (= x!1 1) 1
      0))
)
```

Arrays

Tableau de constantes

```
(declare-const all1 (Array Int Int))
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(declare-const i Int)
(assert (= all1 ((as const (Array Int Int)) 1)))
(assert (= a (select all1 0)))
(assert (= b (select all1 1)))
(assert (= c (select all1 2)))
(check-sat)
(get-model)

(declare-const d Int)
(declare-const j Int)
(assert (not (= d 1)))
(assert (= d (select all1 j)))
(check-sat)
```

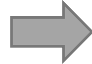


```
sat
(model
  (define-fun all1 () (Array Int Int)
    (_ as-array k!0))
  (define-fun b () Int
    1)
  (define-fun a () Int
    1)
  (define-fun c () Int
    1)
  (define-fun k!0 ((x!1 Int)) Int
    (ite (= x!1 2) 1
    (ite (= x!1 1) 1
    (ite (= x!1 0) 1
      1))))
)


unsat
```

Arrays

Mapper des fonctions sur tableau

```
(declare-const a (Array Int Int))
(assert (= (select a 0) 1))
(assert (= (select a 1) 2))
(assert (= (select a 2) 3))
(assert (= (select a 3) 4))

(declare-fun pair (Int) Int)
(assert (= (pair 1) 1))
(assert (= (pair 2) 0))
(assert (= (pair 3) 1))
(assert (= (pair 4) 0))

(declare-const b (Array Int Int))
(assert (= b ((_ map pair) a)))
```



```
sat
(model
  (define-fun b () (Array Int Int)
    (_ as-array k!1))
  (define-fun a () (Array Int Int)
    (_ as-array k!0))
  (define-fun k!0 ((x!1 Int)) Int
    (ite (= x!1 2) 3
    (ite (= x!1 3) 4
    (ite (= x!1 1) 2
    (ite (= x!1 0) 1
      5)))))
  (define-fun k!1 ((x!1 Int)) Int
    (ite (= x!1 2) 1
    (ite (= x!1 3) 0
    (ite (= x!1 1) 0
    (ite (= x!1 0) 1
      6)))))
  (define-fun pair ((x!1 Int)) Int
    (ite (= x!1 1) 1
    (ite (= x!1 2) 0
    (ite (= x!1 3) 1
    (ite (= x!1 4) 0
    (ite (= x!1 5) 6
      1))))))
)
```

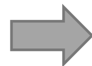
DataTypes

- Permet de spécifier des structures de données:
    - Enregistrements
    - Tuples
    - Enumérations
    - Structures récursives
        - Listes
        - Arbres

DataTypes: Enregistrements

- Permet de spécifier des structures de données en utilisant un constructeur, ici mk-Pair pour l'exemple

```
(declare-datatypes (T1 T2) ((Pair (mk-pair (first T1) (second T2)))))
(declare-const p1 (Pair Int Int))
(declare-const p2 (Pair Int Int))
(assert (= p1 p2))
(assert (> (second p1) 20))
(check-sat)
(get-model)
(assert (not (= (first p1) (first p2))))
(check-sat)
```

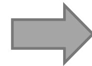


```
sat
(model
  (define-fun p1 () (Pair Int Int)
    (mk-pair 0 21))
  (define-fun p2 () (Pair Int Int)
    (mk-pair 0 21))
)
unsat
```

DataTypes: Enumération

- Permet d’énumérer des constantes distinctes

```
(declare-datatypes () ((S Rouge Bleu Vert)))
(declare-const x S)
(declare-const y S)
(declare-const z S)
(declare-const u S)
(assert (distinct x y z))
(check-sat)
(get-model)
(assert (distinct x y z u))
(check-sat)
```



```
sat
(model
  (define-fun z () S
    Rouge)
  (define-fun y () S
    Bleu)
  (define-fun x () S
    Vert)
)
unsat
```

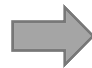
DataTypes: Listes

- Définition d’un liste

```
(declare-datatypes (T) ((Lst nil (cons (hd T) (tl Lst)))))
(declare-const l1 (Lst Int))
(declare-const l2 (Lst Bool))
```

- Exemple d’utilisation

```
(declare-const l1 (List Int))
(declare-const l2 (List Int))
(declare-const l3 (List Int))
(declare-const x Int)
(assert (not (= l1 nil)))
(assert (not (= l2 nil)))
(assert (= (head l1) (head l2)))
(assert (not (= l1 l2)))
(assert (= l3 (insert x l2)))
(assert (> x 100))
(check-sat)
(get-model)
```



```
sat
(model
  (define-fun l3 () (List Int)
    (insert 101 (insert 0 (insert 1 nil))))
  (define-fun x () Int
    101)
  (define-fun l1 () (List Int)
    (insert 0 nil))
  (define-fun l2 () (List Int)
    (insert 0 (insert 1 nil)))
)
```

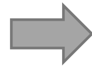
DataTypes: Arbre

- Définition d’un Arbre

```
(declare-datatypes (T) ((Tree leaf (node (value T) (children TreeList)))
                        (TreeList nil (cons (car Tree) (cdr TreeList)))))
```

- Exemple d’utilisation

```
(declare-const t1 (Tree Int))
(declare-const t2 (Tree Bool))
(assert (not (= t1 (as leaf (Tree Int)))))
(assert (> (value t1) 20))
(assert (not (is-leaf t2)))
(assert (not (value t2)))
(check-sat)
(get-model)
```



```
sat
(model
  (define-fun t2 () (Tree Bool)
    (node false (as nil (TreeList Bool))))
  (define-fun t1 () (Tree Int)
    (node 21 (as nil (TreeList Int))))
)
```

```
(assert (= t3 (node 3 (cons (node 4 nil) (cons (node 5 nil) nil)))))
```

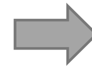
Z3

Pour aller plus loin

Quantifiers

- Utilisation de quantifiers possibles: ∀ (forall) et ∃ (exisits)

```
(declare-fun f (Int Int) Int)
(assert (forall ((x Int)) (exists ((y Int)) (= (f x y) 0))))
(check-sat)
(get-model)
```



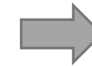
```
sat
(model
  (define-fun f ((x!1 Int) (x!2 Int)) Int
    0)
  (define-fun y!0 ((x!1 Int)) Int
    0)
)
```

- Pas de procédure de décision complète pour la logique du premier ordre avec quantifiers => Pas parfait

Quantifiers

- Utilisation de « pattern » pour améliorer la recherche de solution

```
(assert (exists ((x Int)) (= (+ (* 2 x) 1) 5) :pattern (* 2 x)))
(check-sat)
(get-model)
```



```
sat
(model
  (define-fun x () Int
    2)
)
```

- E-matching en cas de Matching loop

```
(assert (forall ((x Int)) (exists ((y Int)) (= (f x) y))))
```

```
(assert (= (f x!) y!))
```

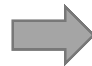
Stratégie & Goals

- Diviser le raisonnement en étapes plus simples, les « tactiques »
- Les tactiques sont composées à l'aide de « combinateurs tactiques »

$\Rightarrow$ Prennent une ou plusieurs tactiques comme entrée et en créent une nouvelle

- Les tactiques prennent en entrée un ensemble de formules logiques appelé « Goal ». 4 résultats possibles

```
(declare-const x Real)
(declare-const y Real)
(assert (> x 0.0))
(assert (> y 0.0))
(assert (= x (+ y 2.0)))
(apply (then simplify solve-eqs))
```

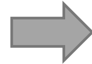


```
(goals
(goal
  (not (<= y (- 2.0)))
  (not (<= y 0.0))
  :precision precise :depth 2)
)
```

Stratégie & Goals

- Division de clause en différents goals

```
(declare-const x Real)
(declare-const y Real)
(assert (or (< x 0.0) (> x 0.0)))
(assert (= x (+ y 1.0)))
(assert (< y 0.0))
(apply split-clause)
```



```
(goals
(goal
  (< x 0.0)
  (= x (+ y 1.0))
  (< y 0.0)
  :precision precise :depth 1)
(goal
  (> x 0.0)
  (= x (+ y 1.0))
  (< y 0.0)
  :precision precise :depth 1)
)
```

- Différentes tactiques
  - (then t s)
  - (par-then t s)
  - (or-else t s)
  - (par-or t s)
  - (repeat t)
  - (repeat t n)

Stratégie & Goals

- Exemple

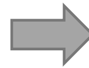
```
(assert (or (= x 0.0) (= x 1.0)))
(assert (or (= y 0.0) (= y 1.0)))
(assert (or (= z 0.0) (= z 1.0)))
(assert (> (+ x y z) 2.0))

(echo "split all...")
(apply (repeat (or-else split-clause skip)))
```

```
split all...
(goals
(goal
  (= x 0.0)
  (= y 0.0)
  (= z 0.0)
  (> (+ x y z) 2.0)
  :precision precise :depth 3)
(goal
  (= x 0.0)
  (= y 0.0)
  (= z 1.0)
  (> (+ x y z) 2.0)
  :precision precise :depth 3)
(goal
  (= x 0.0)
  (= y 1.0)
  (= z 0.0)
  (> (+ x y z) 2.0)
  :precision precise :depth 3)
(goal
  (= x 0.0)
  (= y 1.0)
  (= z 1.0)
  (> (+ x y z) 2.0)
  :precision precise :depth 3)
(goal
  (= x 1.0)
  (= y 0.0)
  (= z 0.0)
  (> (+ x y z) 2.0)
  :precision precise :depth 3)
(goal
  (= x 1.0)
  (= y 0.0)
  (= z 1.0)
  (> (+ x y z) 2.0)
  :precision precise :depth 3)
(goal
  (= x 1.0)
  (= y 1.0)
  (= z 0.0)
  (> (+ x y z) 2.0)
  :precision precise :depth 3)
(goal
  (= x 1.0)
  (= y 1.0)
  (= z 1.0)
  (> (+ x y z) 2.0)
  :precision precise :depth 3)
```

Stratégie & Goals

- Probes: Évaluations sur les goals

```
(declare-const x Real)
(declare-const y Real)
(declare-const z Real)

(push)
(assert (> (+ x y z) 0.0))
(apply (echo "num consts: " num-consts))
(apply (fail-if (> num-consts 2)))
(pop)

(echo "trying again...")
(assert (> (+ x y) 0.0))
(apply (fail-if (> num-consts 2)))
```



```
num consts: (goals
(goal
  (> (+ x y z) 0.0)
  :precision precise :depth 0)
)
(error "tactic failed: fail-if tactic")
trying again...
(goals
(goal
  (> (+ x y) 0.0)
  :precision precise :depth 0)
)
```

```
(apply (if (> num-consts 2) simplify factor))
```

Optimisation

- Optimisation arithmétique: Maximisation et minimisation

```
(declare-const x Int)
(declare-const y Int)
(assert (< x 2))
(assert (< (- y x) 1))
(maximize (+ x y))
(check-sat)
```

➡

```
(+ x y) |-> 2
sat
```

```
(declare-const x Int)
(declare-const y Int)
(assert (< x 4))
(assert (< (- y x) 1))
(assert (> y 1))
(minimize (+ x y))
(check-sat)
```

➡

```
(+ x y) |-> 4
sat
```

- Si pas borné: `(+ x y) |-> oo`

Optimisation

- Contraintes souples: **assert-soft** *formula* **:weight** *numeral*)

1 par défaut

```
(assert (= a3 a1))
(assert (or a3 a2))
(assert-soft a3         :weight 3)
(assert-soft (not a3)   :weight 5)
(assert-soft (not a1)   :weight 10)
```

$\Rightarrow$ Résoudre en minimisant le poids

- Si plusieurs optimisation à faire

```
(assert (< x z))
(assert (< y z))
(assert (< z 5))
(assert (not (= x y)))
(maximize x)
(maximize y)
```

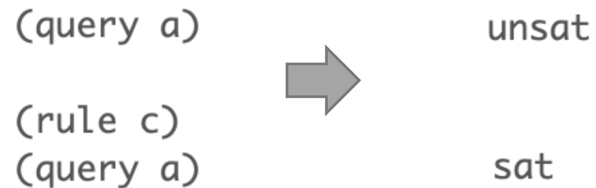
$\Rightarrow$ Priorité en fonction de l'ordre des déclarations

Fixed Points

- Extension μZ
  - Moteur basique: bottom-up Datalog

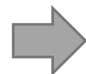
- On a des relations, des règles et des queries

```
(declare-rel a ())
(declare-rel b ())
(declare-rel c ())
(rule (=> b a))
(rule (=> c b))
```



Fixed Points: Relations avec arguments

```
(declare-rel edge (Int Int))
(declare-rel path (Int Int))
(declare-var a Int)
(declare-var b Int)
(declare-var c Int)

(rule (=> (edge a b) (path a b)))
(rule (=> (and (path a b) (path b c)) (path a c)))

(rule (edge 1 2))
(rule (edge 1 3))
(rule (edge 2 4))

(declare-rel q1 ())
(declare-rel q2 ())
(declare-rel q3 (Int))
(rule (=> (path 1 4) q1))
(rule (=> (path 3 4) q2))
(rule (=> (path 1 b) (q3 b)))

(query q1)
(query q2)
(query (q3 4))
```



```
sat
unsat
sat
```

Fixed Points: Property Directed Reachability

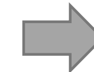
- Algorithme SPACER pour PDR
  - Utilisé sur les Integer, Réels et Datatypes Algébriques

- Exemple avec la fonction 91 de McCarthy

$$M(n) = \begin{cases} n - 10, & \text{if } n > 100 \\ M(M(n+11)), & \text{if } n \leq 100 \end{cases}$$

Fixed Points: McCarthy’s 91 function

```
(declare-rel mc (Int Int))
(declare-var n Int)
(declare-var m Int)
(declare-var p Int)

(rule (=> (> m 100) (mc m (- m 10))))
(rule (=> (and (<= m 100) (mc (+ m 11) p) (mc p n)) (mc m n)))

(declare-rel q1 (Int Int))
(rule (=> (and (mc m n) (< n 91)) (q1 m n)))
(query (q1 m n))                                              unsat

(declare-rel q2 (Int Int))
(rule (=> (and (mc m n) (not (= n 91)) (<= m 101)) (q2 m n)))
(query (q2 m n))
```



```
                                                              unsat

(declare-rel q3 (Int Int))
(rule (=> (and (mc m n) (< n 92)) (q3 m n)))
(query (q3 m n))                                              sat
```

Z3

Récapitulatif de Z3

Récapitulatif

- Puissant et complet
  - Couvre énormément de théories
  - Modulable (configurations, stratégies, choix de logiques)
- Utilisé dans l'industrie et la recherche dans différents domaines:
  - Sécurité, Fiabilité, Planification, Vérification, etc.
- Langage pas très user-friendly
  - Utilisation d'API (python, c, java, etc.)
  - Interfaçage avec d'autres langages => Sujet du stage: Interfaçage avec Prolog

Avez-vous des questions?