Automatisation du Test Logiciel

Sébastien Bardin

CEA-LIST, Laboratoire de Sûreté Logicielle

sebastien.bardin@cea.fr
http://sebastien.bardin.free.fr

Objectifs du cours

Compétences à aquérir

- connaître les principes généraux du test logiciel
- comprendre l'état de la technique en automatisation du test logiciel
- avoir un aperçu des tendances et de l'état de l'art académique

Pour quelles industries ?

- bagage général de l'honnête informaticien [Agile Programming, etc.]
- systèmes critiques
- systèmes "de qualité" (sécurité, etc.)

Rappel : difficultés du test

- ✗ choisir les cas de test / données de test à exécuter
- ✗ estimer le résultat attendu [oracle]
- ✓ exécuter le programme sur les données de test
  - . attention : systèmes embarqués / cyber-physiques
  - . attention : niveau unitaire, code incomplet ! [stubs, mocks]
- ✓ comparer le résultat obtenu à l'oracle
  - . attention : systèmes embarqués / cyber-physiques
- ✓ a-t-on assez de tests ? si oui stop, sinon goto 1
  - . attention : calcul couverture ✓ , choix du critère de couverture ●
- ✓ rejouer les tests à chaque changement [test de régression]
  - . attention : rejeu ✓ , maintenance / optimisation ●

pour les ✓ , des solutions standard existent, doivent être appliquées ! !

Plan

- Introduction au test logiciel
- Exécution symbolique pour automatiser la génération de tests
- Exécution symbolique et critères de test avancés

Automatisation de la génération de tests

Méthodes basées sur les interfaces : quickcheck, fuzzing

- assez simple à mettre en oeuvre, des outils existent [à utiliser !]
- problème de l'oracle
- problème de la couverture des "corner cases"

Méthodes basées sur les comportements du programme

- plus amont, techniques prometteuses
- niveau code : "smart fuzzing"
  - exploration systématique des chemins du programme
  - trouve mieux les corner cases
  - attention : problème de l'oracle
- niveau modèle : model-based testing
  - tests de conformité aux spécifications
  - génère les entrées et l'oracle ! !
  - attention : problème du passage CT $\mapsto$ DT
- point central : exécution symbolique

Dans la suite

On se concentre dans cette partie sur la génération de données de test à partir du code

L'oracle est vu comme un problème orthogonal

On suppose qu'on dispose d'un oracle automatisé

- oracle exact dans certains cas (test dos à dos)
- oracle partiel sinon : assertions, contrats (JML, Spec#)

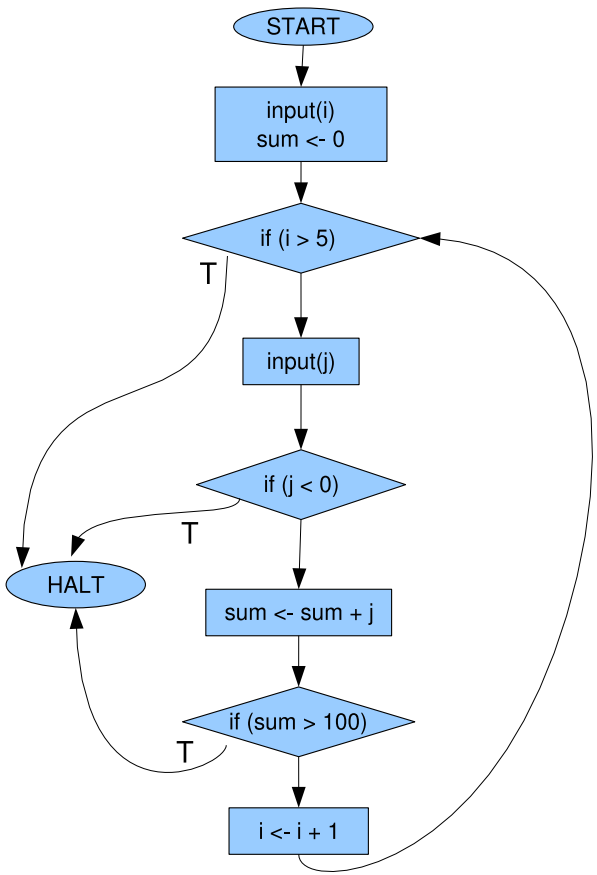
Plan

- Automatisation de la génération de tests
  - Prédicat de chemins
  - Exécution symbolique
  - Limites
  - Exécution dynamique symbolique
  - Optimisations
  - En pratique
  - Discussion
  - Complément : Aspects logiques
  - Complément : Optimisations

Rappel : Control-Flow Graph (CFG)

Le graphe de flot de contrôle d'un programme est défini par :

- un noeud pour chaque instruction, plus un noeud final de sortie
- pour chaque instruction du programme, le CFG comporte un arc reliant le noeud de l'instruction au noeud de l'instruction suivante (ou au noeud final si pas de suivant)



Notion centrale : le prédicat de chemin

- π un chemin (fini) du programme P
  - c-à-d $\pi \in L(P)$, si P vu comme automate
- D l'espace des entrées du programme (arguments, variables volatiles, etc.)
  - ex : $D = \mathbb{N} \times \mathbb{N}$ pour `void foo(int a, int b)`
- $V \in D$ une entrée du programme
- On note $P(V)$ la trace d'exécution de P lancé sur la donnée d'entrée V
- On note par $\preceq$ la relation de préfixe entre les chemins
  . ($\approx$ préfixes de mots, $ab \preceq abc$)

Notion centrale : le prédicat de chemin (2)

- On se donne une théorie logique T
  - théorie : ensemble d'opérateurs / prédicats permis, et leurs axiomes
  - ex : arithmétique linéaire $[x \leq a + 5 \times b]$
  - ex : théorie des tableaux $[x = load(M', i) \wedge M' = store(M, j, x)\ ]$

Prédicat de chemin

Un prédicat de chemin pour π (dans T) est une formule logique $\varphi_\pi \in T$ interprétée sur D telle que si $V \models \varphi_\pi$ alors l'exécution du programme sur V suit le chemin π, c'est à dire que $\pi \preceq P(V)$.

Permet de lancer une exécution sur le chemin visé !

Construction du prédicat de chemin (1)

Un prédicat de chemin pour π peut se calculer en exécutant symboliquement le chemin

- exécution concrète : màj des valeurs des variables
- exécution symbolique : màj des relations logiques entre variables

Mémoire concrète / symbolique

- état mémoire concret : variable $\mapsto$ valeur concrète
- état mémoire symbolique :
  - . variable $\mapsto$ variables logiques ("noms")
  - . $+$ contraintes sur les variables logiques

Construction du prédicat de chemin (2)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat de chemin (entrées Y_0 et Z_0)

$\top$

Construction du prédicat de chemin (2)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat de chemin (entrées Y_0 et Z_0)

$\top \wedge W_1 = Y_0 + 1$

Construction du prédicat de chemin (2)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat de chemin (entrées Y_0 et Z_0)

$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3$

Construction du prédicat de chemin (2)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat de chemin (entrées Y_0 et Z_0)

$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3 \wedge X_2 < 2 \times Z_0$

Construction du prédicat de chemin (2)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat de chemin (entrées Y_0 et Z_0)

$$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3 \wedge X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$$

Construction du prédicat de chemin (2)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat de chemin (entrées Y_0 et Z_0)

$$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3 \wedge X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$$

Projection sur les entrées $Z_0 \leq Y_0 + 4 < 2 \times Z_0$

- une solution possible : $Y_0 = 0, Z_0 = 3$
- suit bien le chemin voulu

Construction du prédicat de chemin (3)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Remarque : autre construction possible (avec des "let")

$$\begin{array}{l} \text{let } W_1 \triangleq Y_0 + 1 \text{ in} \\ \quad \text{let } X_2 \triangleq W_1 + 3 \text{ in} \\ \quad\quad X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0 \end{array}$$

Point technique

attention : introduire une nouvelle variable logique à chaque nouvelle utilisation d'une variable du programme

- les "variables" du programme C peuvent être modifiées à chaque étape de l'exécution
- les "variables" de la théorie T sont des inconnues, de valeur constante
- le renommage est nécessaire pour prendre en compte la dynamique de l'exécution
  - prédicat de chemin pour `x := x+1` ?
  - $X_{n+1} = X_n + 1$, plutôt que $X = X + 1$

Plan

- Automatisation de la génération de tests
  - Prédicat de chemins
  - Exécution symbolique
  - Exécution dynamique symbolique
  - En pratique
  - Discussion
  - Complément : Aspects logiques
  - Complément : Optimisations

Exécution symbolique

input : a program P
output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

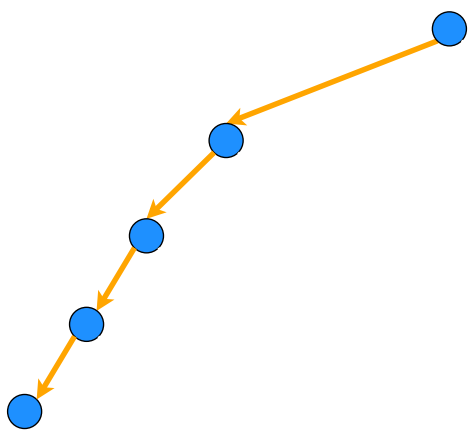
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, $\sigma >$
- loop until no more path to cover

Exécution symbolique

input : a program P

output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

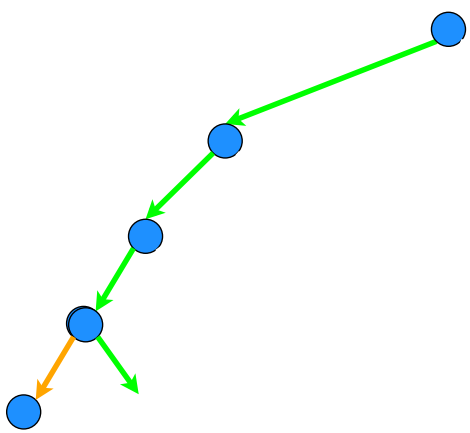
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, $\sigma >$
- loop until no more path to cover



Exécution symbolique

input : a program P
output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, $\sigma >$
- loop until no more path to cover

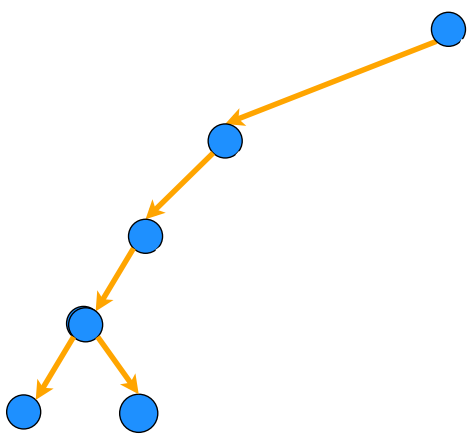


Exécution symbolique

input : a program P

output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, $\sigma >$
- loop until no more path to cover

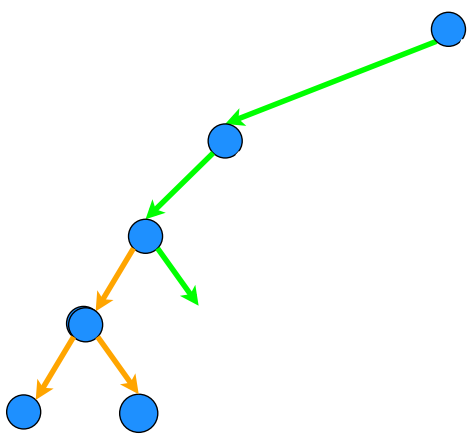


Exécution symbolique

input : a program P

output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, $\sigma >$
- loop until no more path to cover

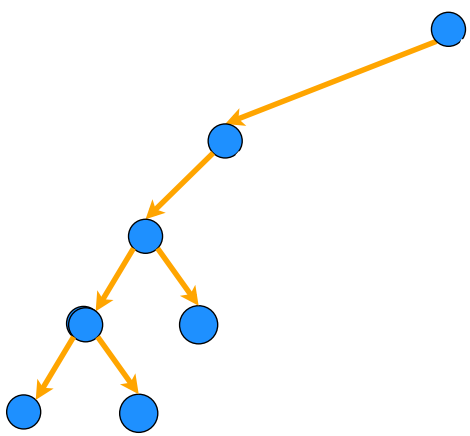


Exécution symbolique

input : a program P

output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, $\sigma >$
- loop until no more path to cover

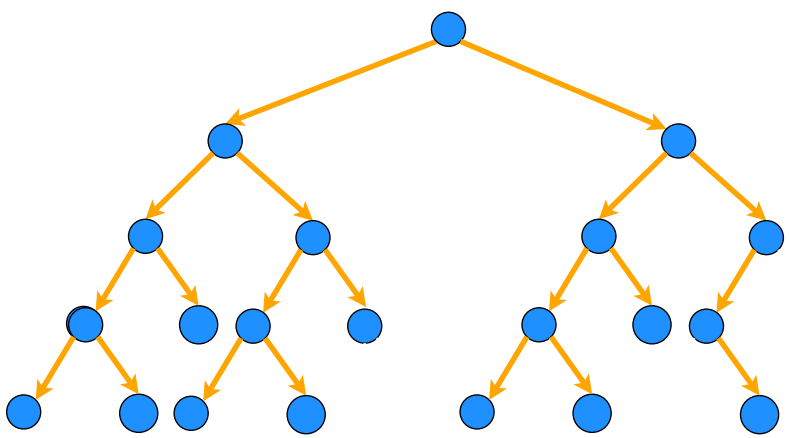


Exécution symbolique

input : a program P

output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ [wpre, spost]
- solve φ_σ for satisfiability [smt solver]
- SAT(s) ? get a new pair $<$ s, $\sigma >$
- loop until no more path to cover



Éléments clés

- création du prédicat de chemin [dans T] ✓
  - [cf après] concrétisation et symbolisation ●
- résolution d'un prédicat de chemin [`solve` : $T \mapsto$ `B`] ✓
  - on peut toujurs faire mieux : flottants ou string ●, $\forall$ ✗
- ordre d'exploration des chemins ●

Aparté : solveurs SMT

Logique considérée

- logique du 1er ordre, sans quantificateur
- structure booléenne arbitraire [$\wedge, \vee, \neg, etc.$]
- combinaison de théories [$x \leq y + load(M, i + j)$]
  - arithmétique linéaire entière ou réelle
  - vecteurs de bits
  - tableaux, mémoire, etc.

solveurs SMT :

- état de l'art de la résolution automatique
- prouve unsat/valid où trouve des solutions / contre-exemples
- moteur : combinaison d'inférence, de backtracking et d'apprentissage

Note : architecture modulaire et efficace [= plein de théories supportées]

- DPLL(T) : solver($T_\wedge$) $\mapsto$ solver($T_{\wedge,\vee}$)
- Nelson-Oppen : solver($T_\wedge$) $\times$ solver($T'_\wedge$) $\mapsto$ solver($(T \otimes T')_\wedge$)

Propriétés : correction et complétude

Le calcul de prédicat de chemin est :

- correct s'il produit un prédicat de chemin plus fort que le prédicat de chemin le plus lâche
- complet s'il produit un prédicat de chemin *équisatisfiable* au prédicat de chemin le plus lâche

Le calcul symbolique est correct (resp. complet) si :

- le calcul de prédicat de chemin est correct (resp. complet)
- le solveur est correct et complet pour la théorie considérée

Propriétés

Correction si le calcul symbolique est correct, alors la procédure est correcte : chaque DT généré suit le chemin prévu

Complétude si le calcul symbolique est complet, alors la procédure est complète : quand la procédure termine, chaque chemin faisable est couvert

Terminaison la procédure termine ssi le nombre de chemins est fini

Intérêt pour le test (1)

La procédure produit des témoins d'accessibilité :

- on peut vérifier le résultat fournit par des outils externes simples (calcul de couverture)
- un couple (DT, π) est plus facile à comprendre humainement que des invariants
- les DT peuvent être exportées vers des outils classiques de gestion de tests (couverture, tests de régression)

Correction : chaque DT généré suit le chemin prévu

- pas de faux positifs ! !
- un bug reporté est un bug trouvé
- la couverture du jeu de tests fourni est effectivement atteinte
- les instructions couvertes lors de l'exécution symboliques sont vraiment atteignables

MAIS : La complétude n'est que rarement obtenue, car le nombre de chemins doit être limité a priori

Intérêt pour le test (2)

- couverture des k-paths
- mieux : active testing : essayer de couvrir les cas d'erreur runtime
  - ajout de branches implicites dans le code
  - `x := *b` $\mapsto$ essayer de résoudre `b == null`
  - `x := a/b` $\mapsto$ essayer de résoudre `b == 0`
  - `x := A[j]` $\mapsto$ essayer de résoudre `j` $>$ `size(A)`
  - `free(x)` $\mapsto$ essayer de résoudre `x==null` $\vee$ `unallocated(*x)`
  - ...
- mieux : prise en compte native des critères de couverture
  - ok pour de nombreux critères [inclu mutations faibles et forme faible de mcdc]
  - attention à l'explosion du nombre de chemins (cf cours 3)

Ajouts classiques à la procédure de base

Ajouts classiques à la procédure

1. borne sur la longueur des chemins
2. time out sur le solveur
3. gestion de couverture (instructions, branches)

Les points 1. et 2. cassent la propriétés de complétude pour assurer terminaison et temps de calcul raisonnable

En pratique, l'hypothèse de calcul symbolique parfait (correct + complet) est difficile à obtenir.

- pour du test, il vaut mieux garder la correction et sacrifier la complétude (cohérent avec la restriction arbitraire du nombre de chemins)
- remarque : dans le cas dynamique symbolique (cf + tard), on peut imaginer se passer dans une certaine mesure de la correction du solveur

Ajouts classiques à la procédure de base (2)

Critère de test naturellement associé : couverture de chemins

Peut être modifié pour d'autres critères (instructions ou branches)

- arrêt lorsque le taux de couverture est suffisant
- guide le choix des chemins

Paramètres principaux de la méthode : théorie logique, énumération de chemins, critère d'arrêt

Focus : Procédure symbolique de base (parcours DFS)

Variable globale **Tests** initialisée à $\emptyset$

Procédure principale : SEARCH(node_init, ε, $\top$)
/ màj Tests, ensemble de paires (TD,π) */*

procedure SEARCH(node, π, Φ)
input : CFG node, path prefix π, path predicate Φ for π
output : no result, update Tests

```
 1: Case node of
 2: | halt →        /* end node */
 3:       try S_p := SOLVE(Φ) ; Tests := Tests + {(S_p, π)} /* new TD */
 4:       with unsat → () ;
 5:       end try
 6: | block i → SEARCH(node.next, π · node, Φ ∧ SYMB(i))
 7: | goto tnode → SEARCH(tnode, π · node, Φ)
 8: | ite(cond,inode,tnode) →     /*branching*/
 9:       SEARCH(inode, π · node,Φ ∧ SYMB(COND)) ;
10:       SEARCH(tnode, π · node,Φ ∧ ¬SYMB(COND))
11: end case
```

Focus : Procédure symbolique de base (2)

Procédure SYMB : Instr $\mapsto T$

- transforme une instruction de base en formule de la théorie T
- exemple : `x:=x+1` $\rightarrow X_1 = X_0 + 1$
- attention : introduire une nouvelle variable logique à chaque nouvelle utilisation d'une variable du programme

Procédure SOLVE : $T \mapsto \{SAT(Sol), UNSAT\}$

- procédure de décision pour la théorie T
- retourne SAT ($+$ une solution) ou UNSAT

Focus : Procédure SYMB, exemple

$expr \ ::= \ | \ V_C \ | \ k \in \mathbb{N} \ | \ expr \ (+,\text{-},*) \ expr$

Expressions de la théorie logique T définies par

$$\text{termF} : := k \in \mathbb{N} \mid V_F \\ \mid \text{termF} +_F \text{termF} \mid \text{termF} -_F \text{termF} \mid \text{termF} \times_F \text{termF}$$

let SYMB e = match e with

- $V_C \rightarrow \alpha(V_c)$ // *fonction de renommage*
- $k \rightarrow k$
- $e_1 \ (+,\text{-},*) \ e_2 \rightarrow \text{SYMB}(e_1) \ (+_F, -_F, \times_F) \ \text{SYMB}(e_2)$

SYMB définit de manière similaire sur les conditions

Focus : Procédure SYMB, exemple (2)

Pourquoi $\alpha(V_c)$:

- les "variables" du programme C peuvent être modifiées à chaque étape de l'exécution
- les "variables" de la théorie T sont des inconnues, de valeur constante
- le renommage est nécessaire pour prendre en compte la dynamique de l'exécution
  - prédicat de chemin pour `x := x+1` ?
  - $X_{n+1} = X_n + 1$, plutôt que $X = X + 1$

Plan

- Automatisation de la génération de tests
  - Prédicat de chemins
  - Exécution symbolique
  - Limites
  - Exécution dynamique symbolique
  - Optimisations
  - En pratique
  - Discussion
  - Complément : Aspects logiques
  - Complément : Optimisations

Problèmes de l'exécution symbolique

PB0 : Performances (cf plus tard dans le cours)

- nombre de chemins
- coût d'un appel au solveur

PB1 : Exploration (inutile) de chemins infaisables

- pas de détection : coûteux en # chemins inutiles explorés
- détection au plus tôt : coûteux en # appels solveurs

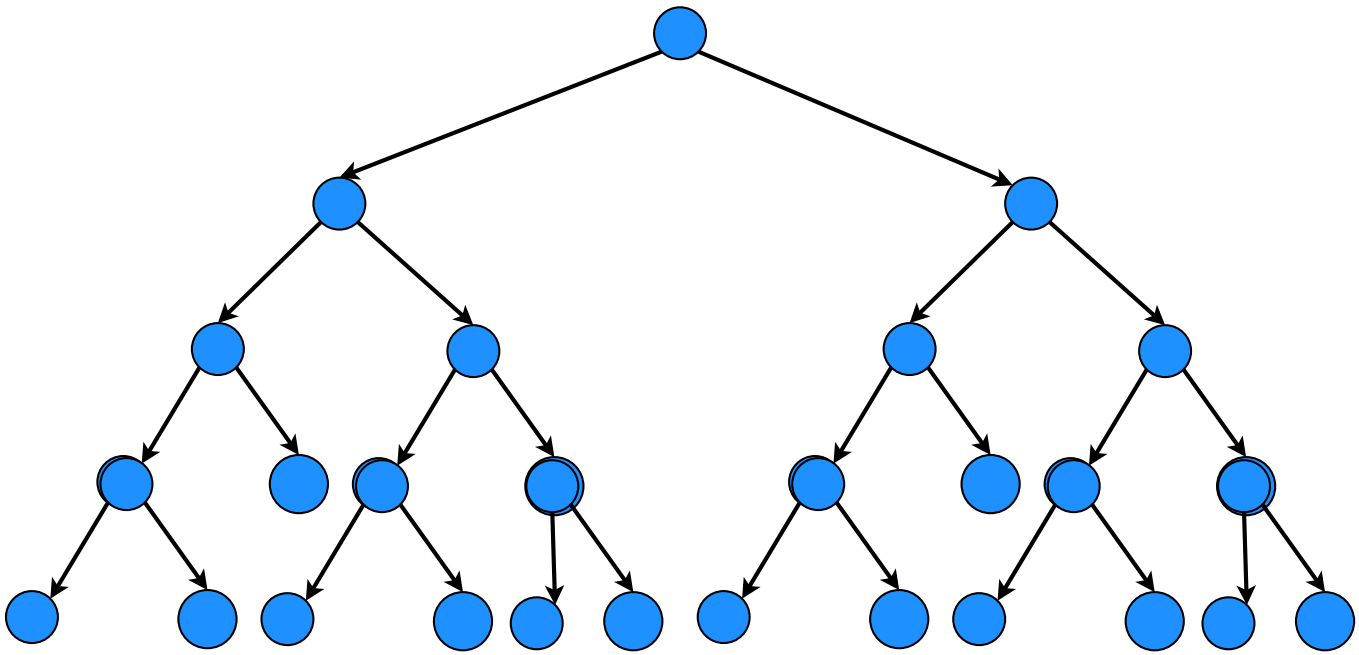
PB2 : Robustesse : Constructions du langage hors de portée de la théorie choisie

- opérations non linéaire
- assembleur incorporé, bibliothèques en code natif

  casse la correction !

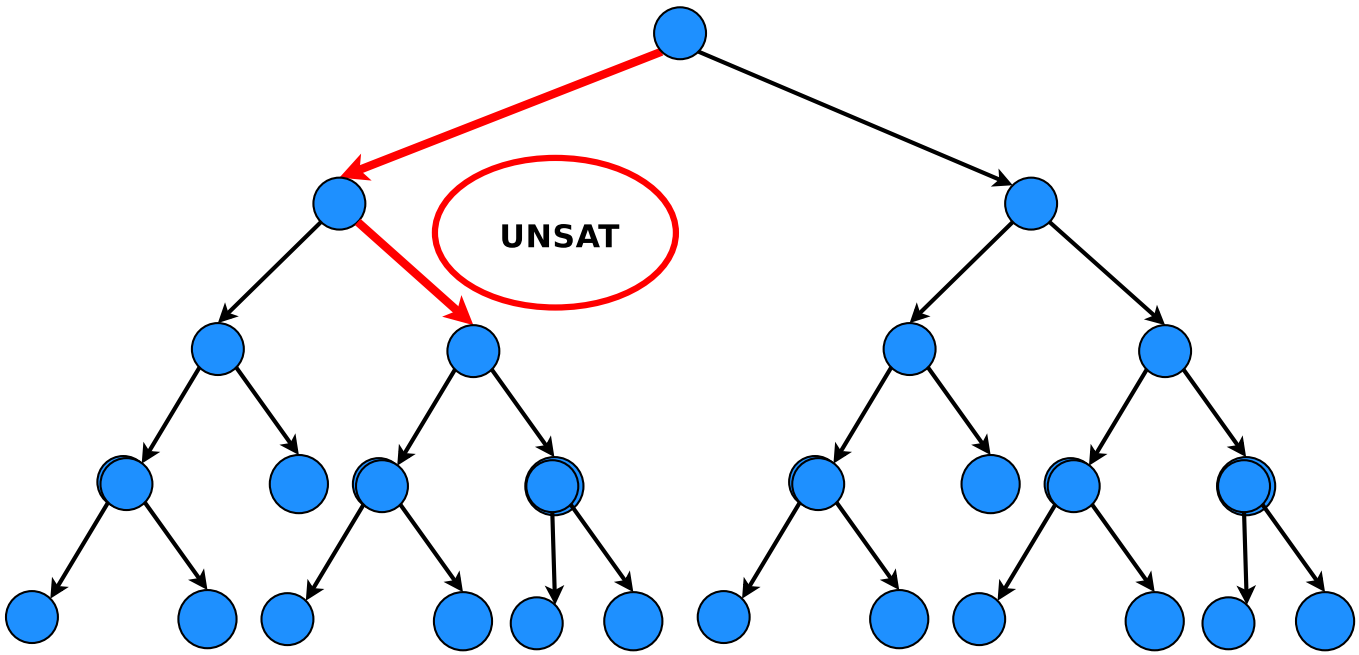
L'exécution dynamique symbolique (DSE) apporte des solutions aux 2 derniers problèmes (cf après)

Problème PB1 : chemins infaisables



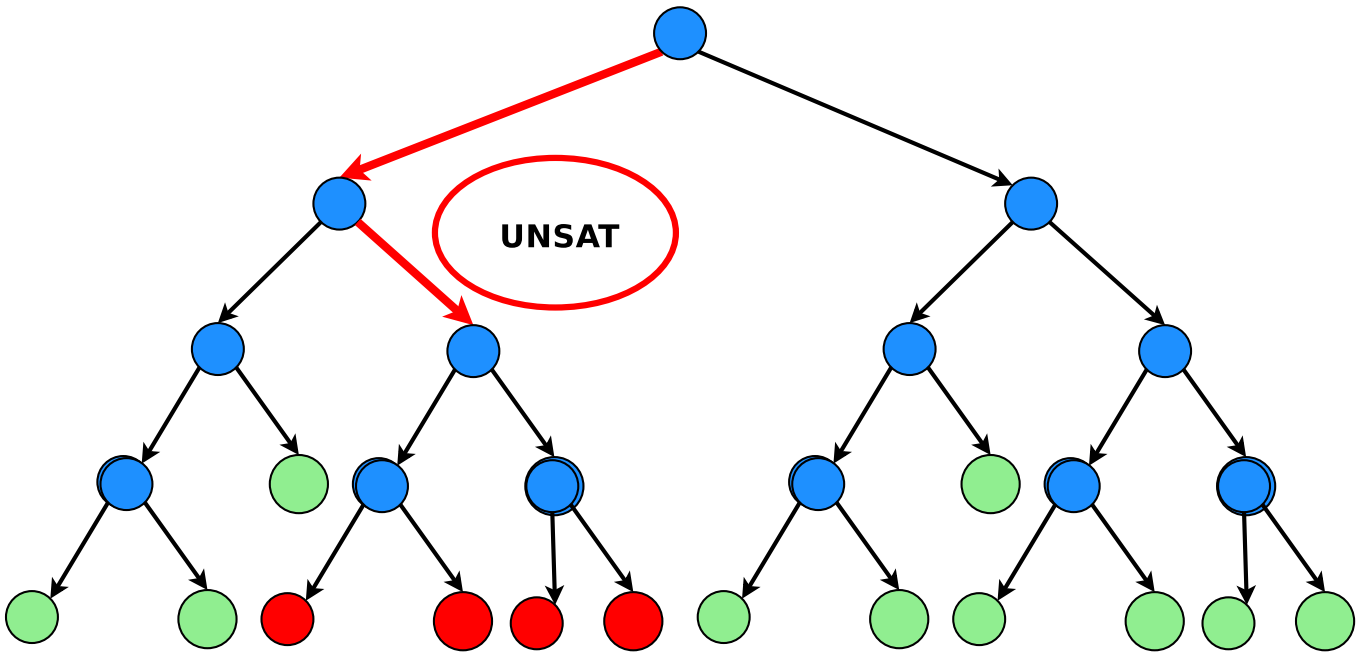
Supposons un chemin infaisable dans l'arbre des exécutions possibles

Problème PB1 : chemins infaisables



Supposons un chemin infaisable dans l'arbre des exécutions possibles

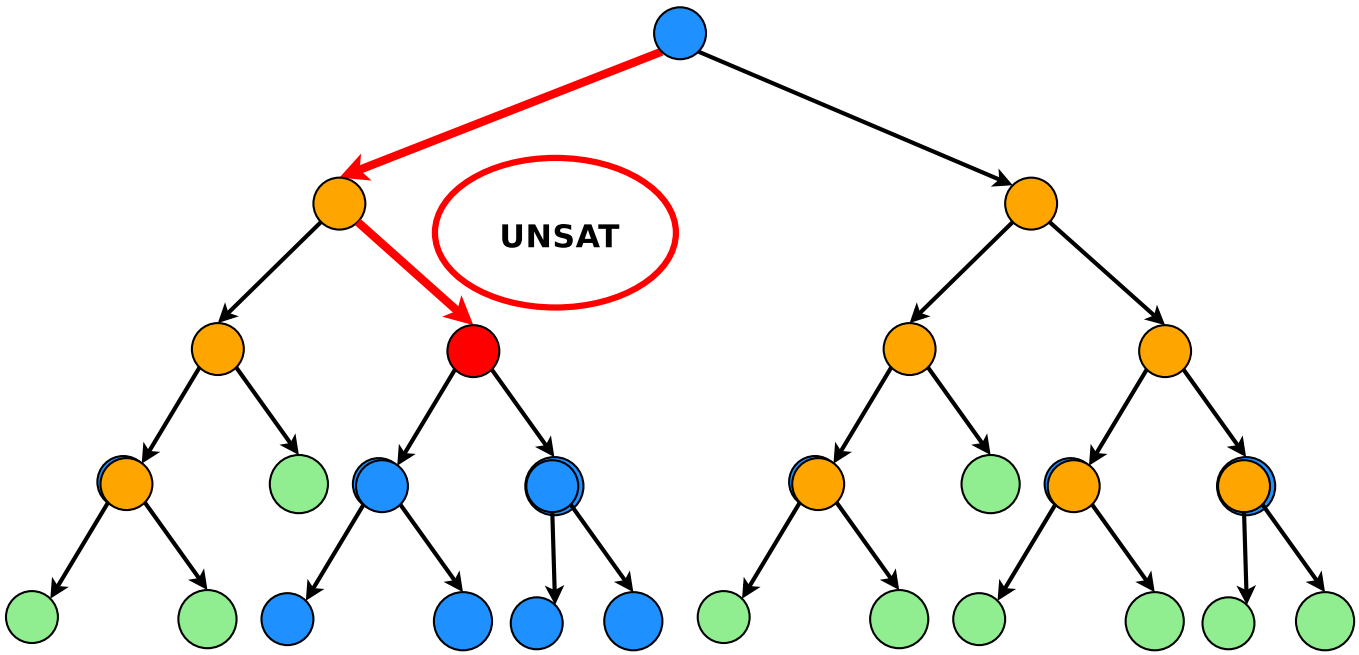
Problème PB1 : chemins infaisables



Méthode usuelle : résoudre le prédicat à la fin du chemin

- + : un appel au solveur par chemin (sur un arbre : 2^N)
- - : on peut continuer la recherche à partir de préfixes UNSAT
- KO sur programmes avec beaucoup de chemins infaisables

Problème PB1 : chemins infaisables



Alternative : résoudre le prédicat à chaque branche

- $+$: détecte UNSAT au plus tôt
- - : un appel au solveur par préfixe de chemin faisable, et un appel au solveur par préfixe minimal infaisable (sur un arbre : $2 * 2^N - 1$)
- KO sur programmes avec peu de chemins infaisables

Problème PB2 : robustesse

Un problème classique : constructions du langage hors de portée de la théorie choisie

---

Générer un test pour `f` atteignant `ERROR` ci-dessous
(hyp : théorie = arithmétique linéaire)

```
f(int x, int y) {z=x*x; if (y == z) {ERROR; }else OK }
```

Problème

- Une exécution symbolique génère une expression symbolique de type $Y = X * X$
- Cette expression n'est pas solvable en arithmétique linéaire

Solutions classiques tirées de l'analyse statique / preuve de programme

- surapproximation, ici : $Y = X * X \mapsto \top$
- PROBLEME : on perd la correction, DT ne suit pas le chemin prévu
  . $Y = 0, X = 3$ est solution, mais n'atteint pas `ERROR`

Problème PB2 : robustesse (2)

Un problème classique : constructions du langage non modélisables

- opérateur hors de portée de la logique choisie [ex : non-linéaire]
- code non disponible [ex : bibliothèque, cots]
- code écrit dans un autre langage, non supporté [ex : asm]
- intéraction avec l'environnement [ex : syscall]

Plan

- Automatisation de la génération de tests
  - Prédicat de chemins
  - Exécution symbolique
  - Limites
  - Exécution dynamique symbolique
  - Optimisations
  - En pratique
  - Discussion
  - Complément : Aspects logiques
  - Complément : Optimisations

Exécution dynamique symbolique

Combinaison d'exécutions symboliques et concrètes

[GKS-05] [SMA-05] [WMM-04]

Exécution concrète : collecte des infos pour aider le raisonnement symbolique

- concrétisation : force une variable symbolique à prendre sa valeur concrète courante

Deux utilisations typiques

- suivre uniquement des chemins faisables à moindre coût
  . toujours suivre une exécution concrète + résoudre au plus tôt
- approximation de constructions du langage "difficiles"
  . concrétisation d'une partie des entrées/sorties
  . approximations correctes

Rend l'exécution symbolique robuste et flexible ! !

(Rappel) Problème 2

Générer un test pour `f` atteignant `ERROR` ci-dessous
(hyp : théorie = arithmétique linéaire)

```
g(int x) {return x*x; }
f(int x, int y) {z=g(x); if (y == z) {ERROR; }else OK }
```

- une exécution symbolique génère une expression symbolique de type $z = x * x$
- non solvable en arithmétique linéaire

Exploiter l'information dynamique

```
g(int x) {return x*x+(x%2); }
f(int x, int y) {z=g(x); if (y == z) {ERROR; }else OK }
```

Exploitation d'une exécution concrète

- première exécution avec comme entrées de `f` : $x = 3, y = 4$
- lors du calcul de prédicat, x*x reconnu non traitable
- l'expression est "concrétisée" à 9, ET ses opérandes (ici x) sont aussi concrétisés (hyp $x = 3$).
- l'exécution aboutit au prédicat de chemin $(y! = 9)$ (branche else du test, hyp $x = 3$)
- un nouveau chemin est obtenu par négation du prédicat, soit $(y = 9)$ (branche then du test, hyp $x = 3$)
- on résoud, on trouve $x = 3, y = 9$, cette entrée atteint bien `ERROR`

Exploiter l'information dynamique

```
g(int x) {return x*x+(x%2); }
f(int x, int y) {z=g(x); if (y == z) {ERROR; }else OK }
```

Exploitation d'une exécution concrète

- première exécution avec comme entrées de `f` : $x = 3, y = 4$
- lors du calcul de prédicat, x*x reconnu non traitable
- l'expression est "concrétisée" à 9, ET ses opérandes (ici x) sont aussi concrétisés (hyp $x = 3$).
- l'exécution aboutit au prédicat de chemin $(y! = 9)$ (branche else du test, hyp $x = 3$)
- un nouveau chemin est obtenu par négation du prédicat, soit $(y = 9)$ (branche then du test, hyp $x = 3$)
- on résoud, on trouve $x = 3, y = 9$, cette entrée atteint bien `ERROR`

Technique correcte et robuste, mais perte de complétude

Concrétisation et correction

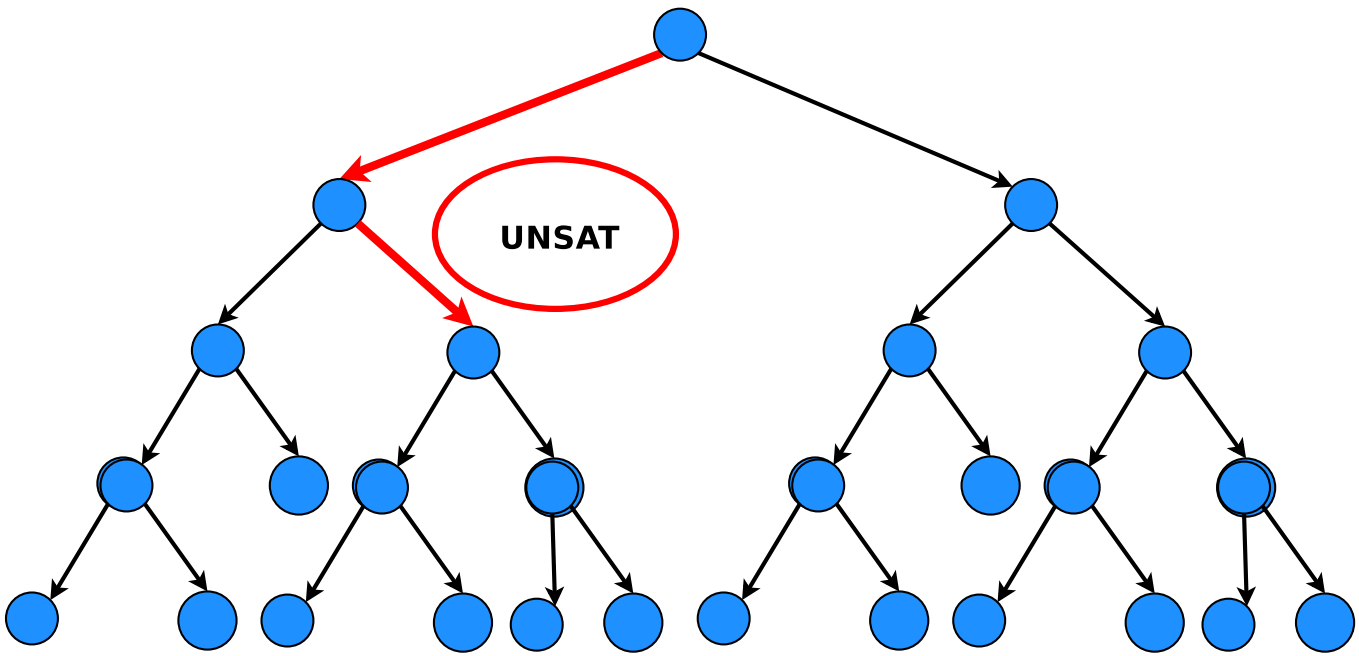
Attention, plusieurs manières de concrétiser

Soit `x := (a+b) * c` [concret : a=5, b=3]

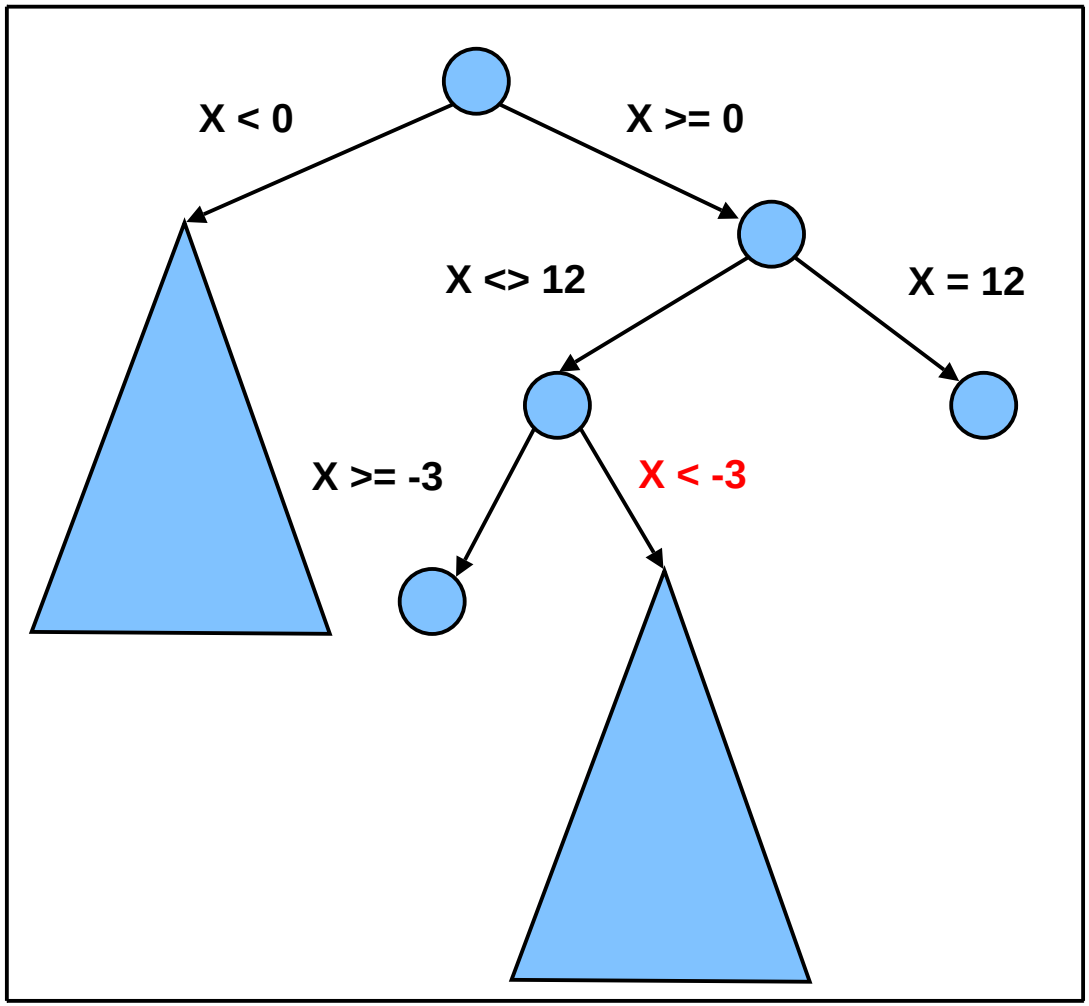
- $X' = 8 \times C \wedge A + B = 8$ correct, minimal
- $X' = 8 \times C \wedge A = 5 \wedge B = 3$ correct, less complete
- $X' = 8 \times C$ incorrect

Remarque : en pratique, on peut tolérer une incorrection "légère"
. facile de rejouer pour évaluer la correction du DT

Rappel Problème 1 : chemins infaisables

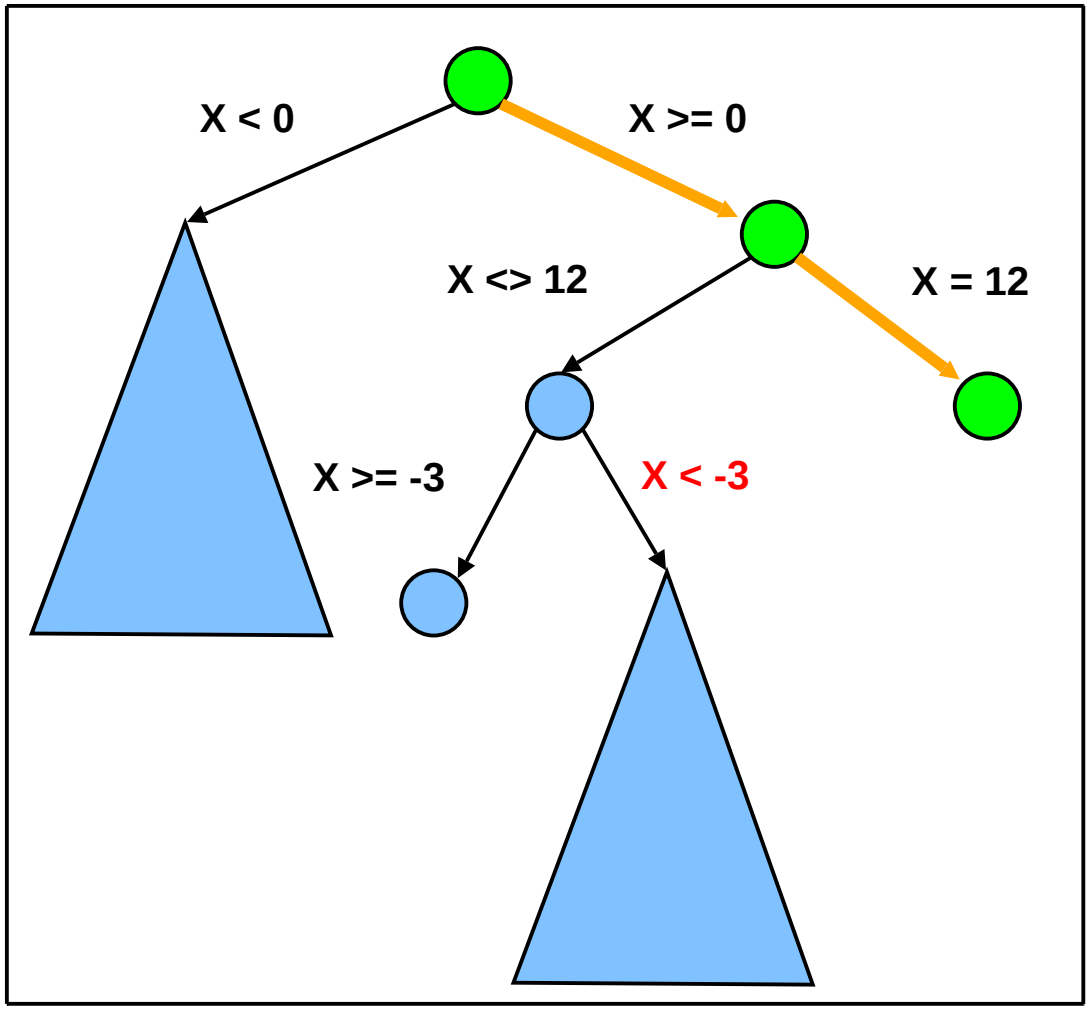


Suivre seulement des chemins faisables



concret : X=12
backtrack + résolution, solution X = 5
concret : X=5
backtrack + résolution, unsat

Suivre seulement des chemins faisables



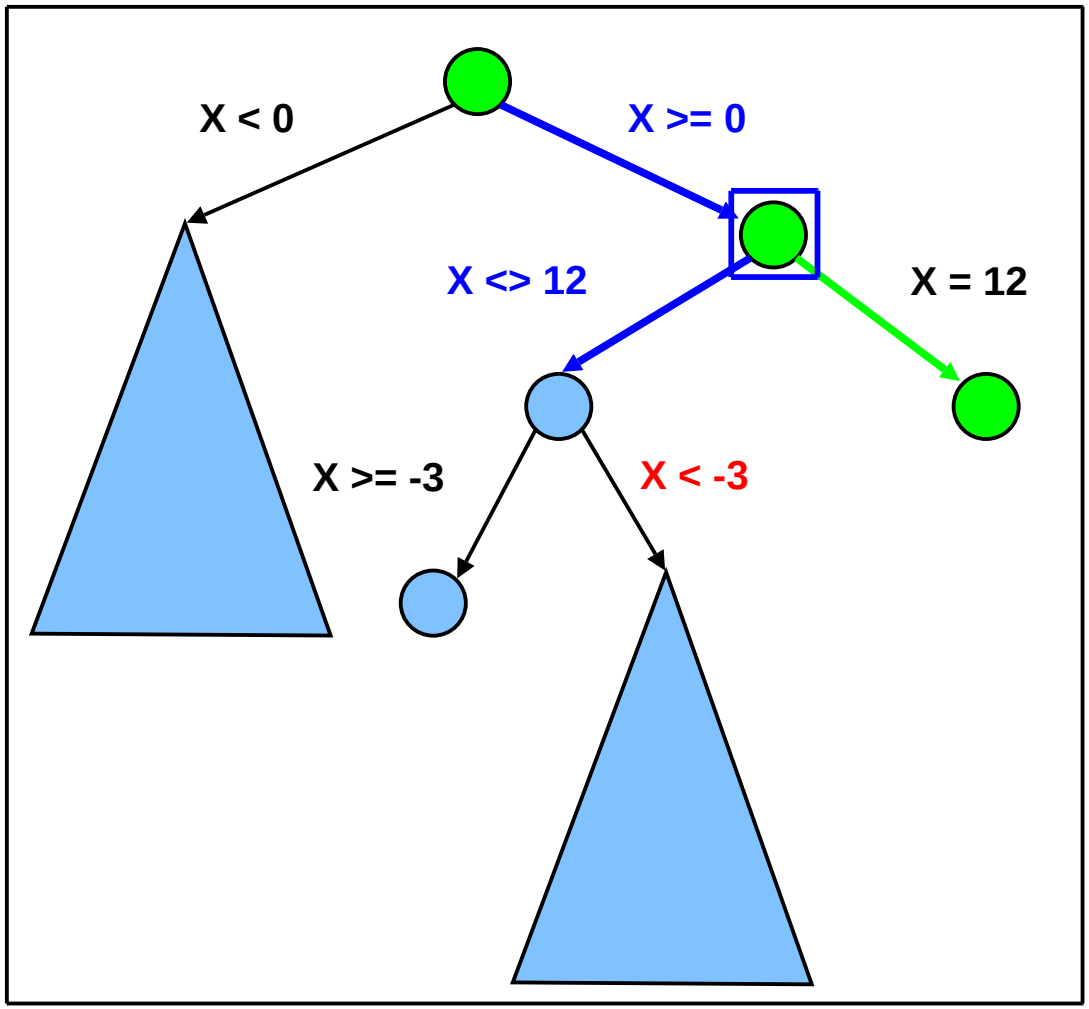
concret : X=12

backtrack + résolution, solution X = 5

concret : X=5

backtrack + résolution, unsat

Suivre seulement des chemins faisables



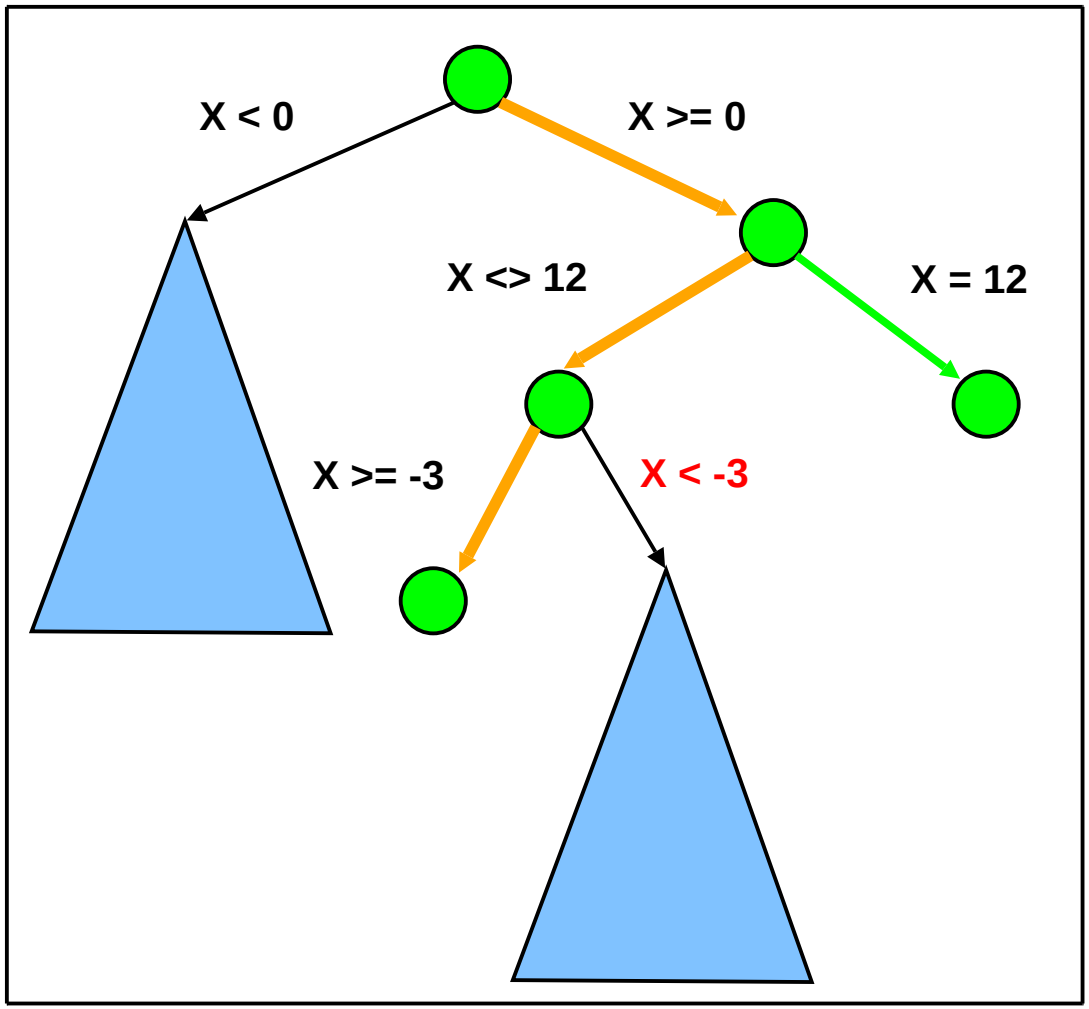
concret : X=12

backtrack + résolution, solution X = 5

concret : X=5

backtrack + résolution, unsat

Suivre seulement des chemins faisables



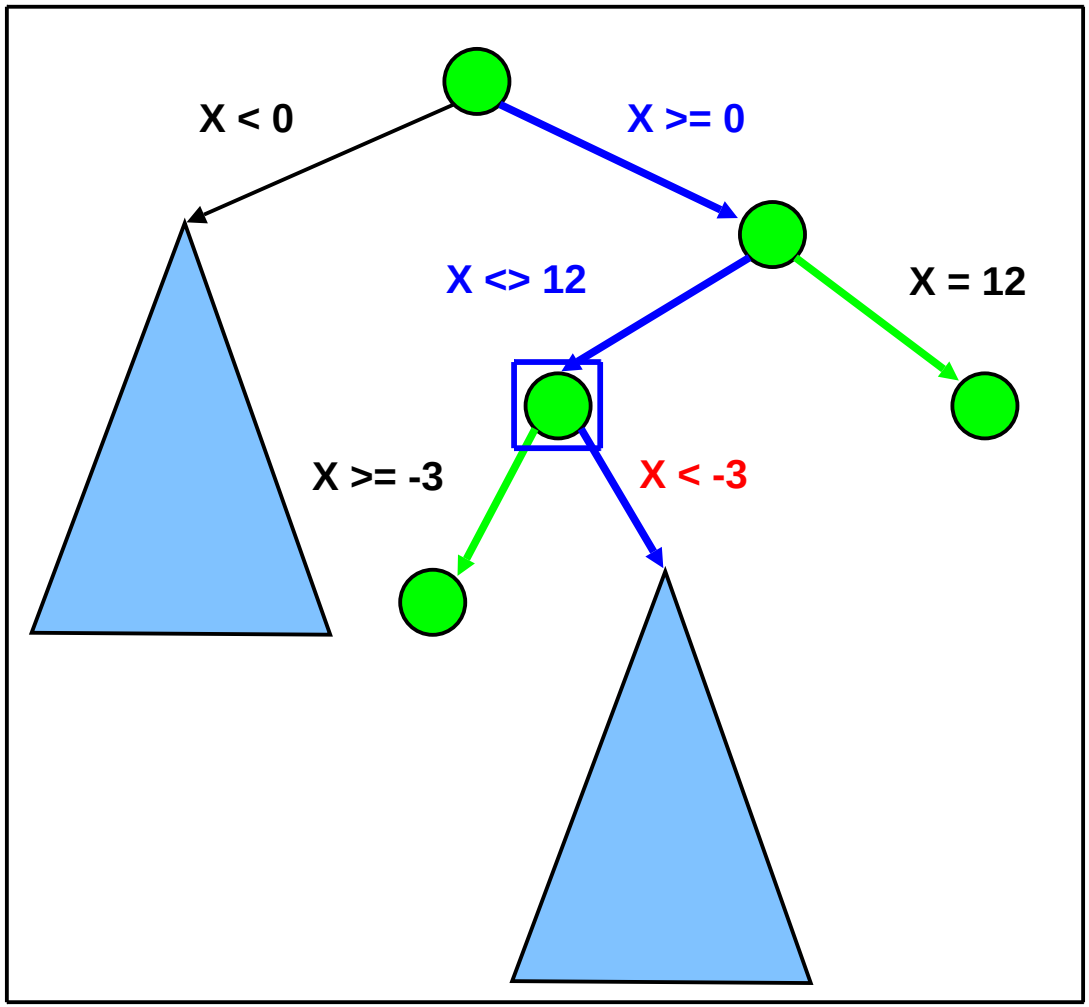
concret : X=12

backtrack + résolution, solution X = 5

concret : X=5

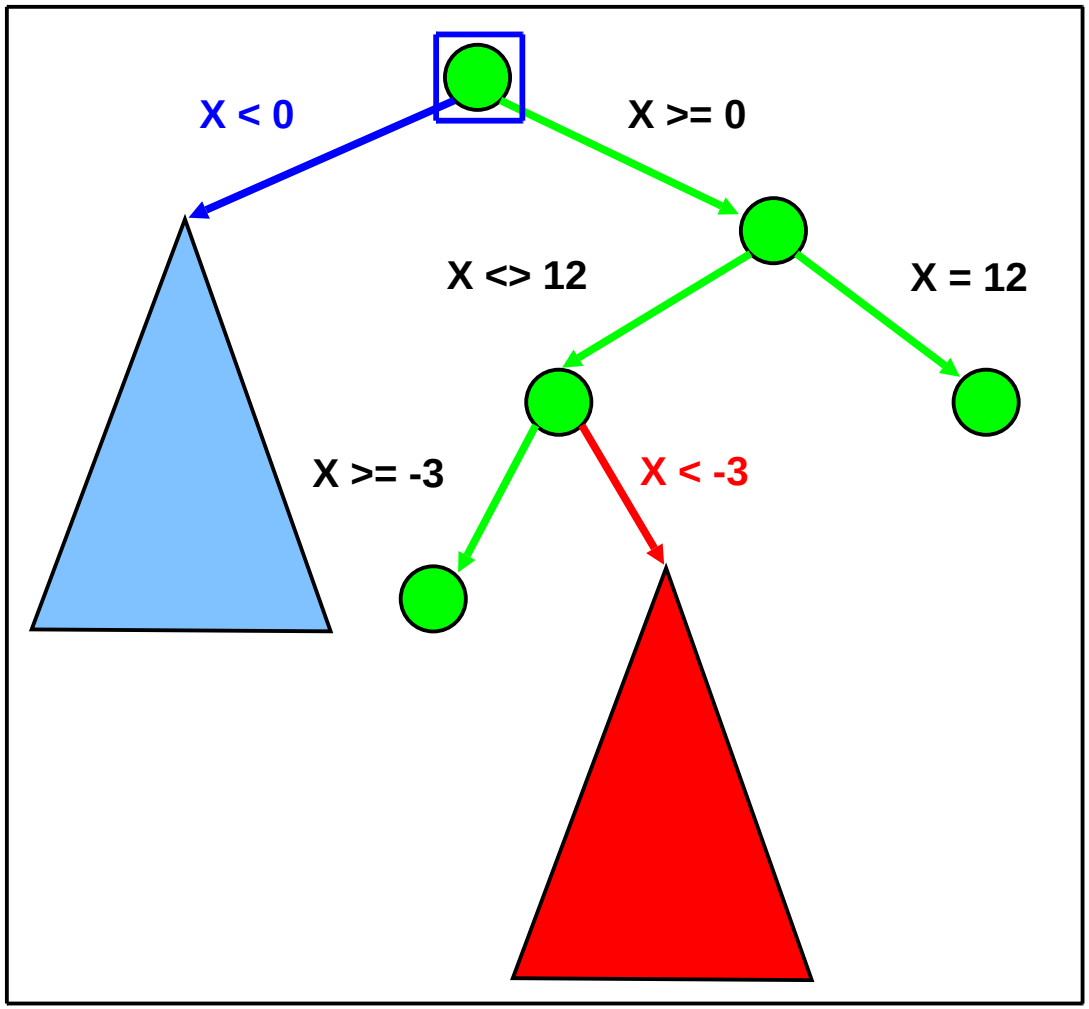
backtrack + résolution, unsat

Suivre seulement des chemins faisables



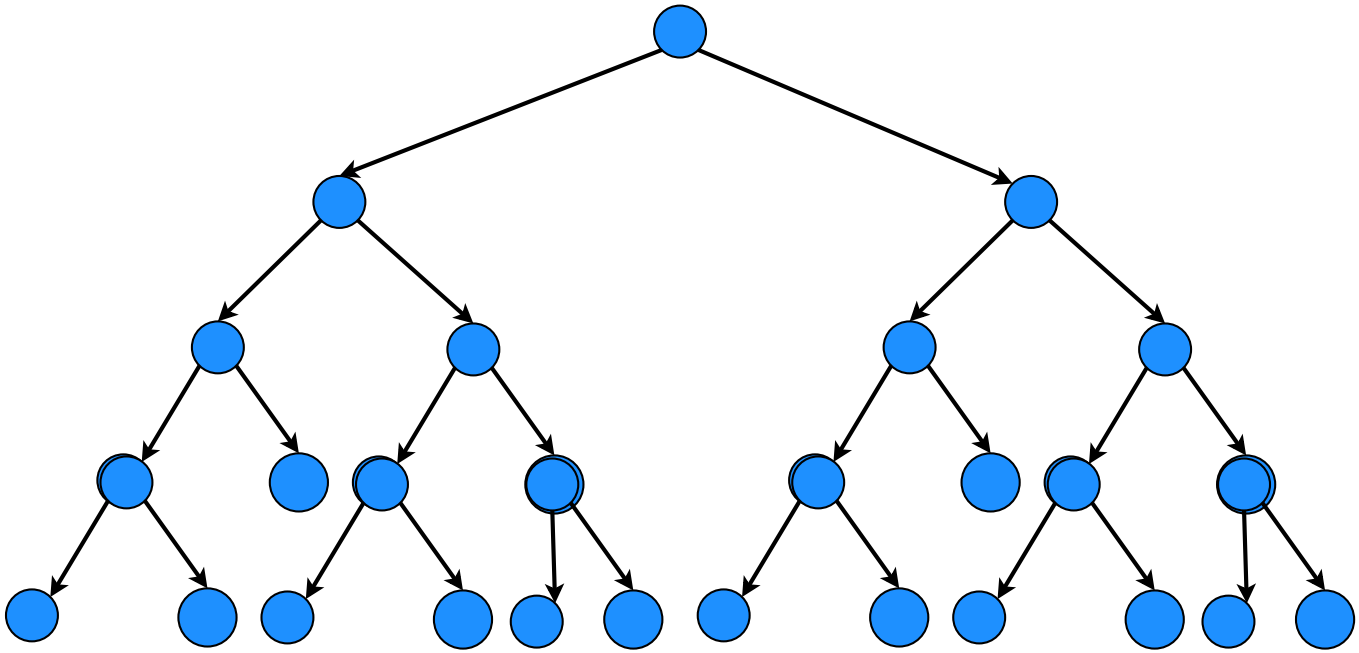
concret : X=12
backtrack + résolution, solution X = 5
concret : X=5
backtrack + résolution, unsat

Suivre seulement des chemins faisables

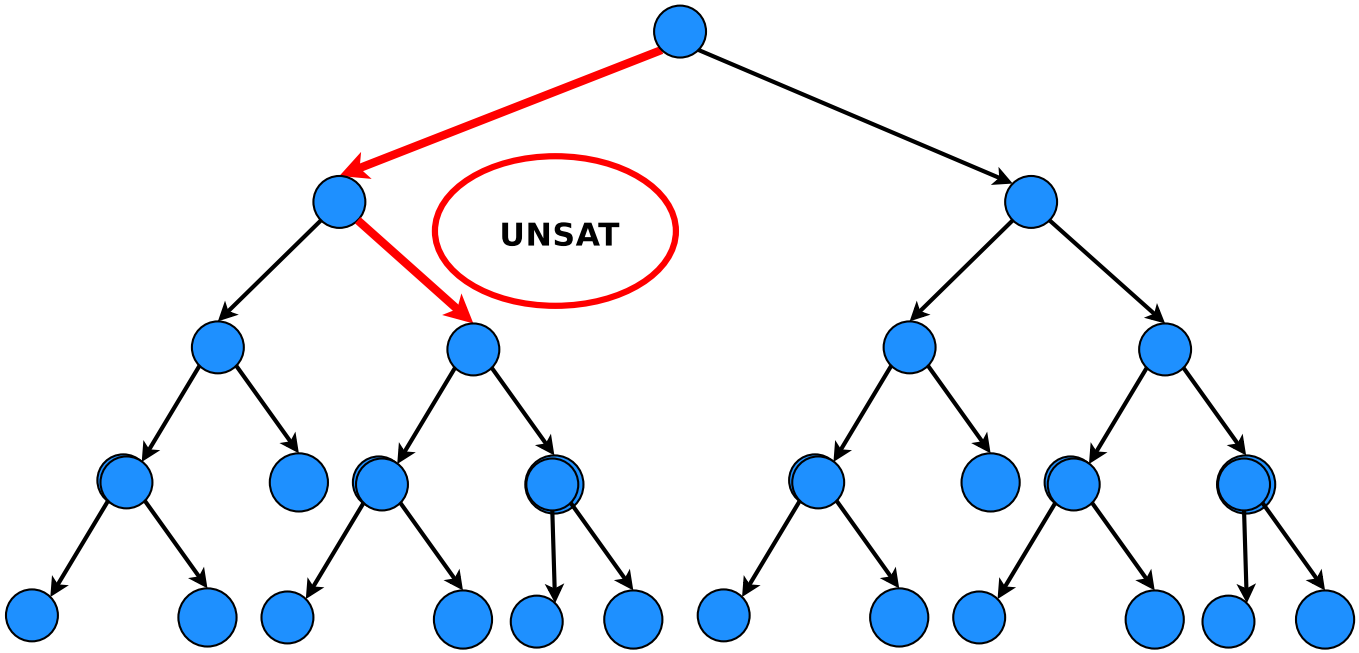


concret : X=12
backtrack + résolution, solution X = 5
concret : X=5
backtrack + résolution, unsat

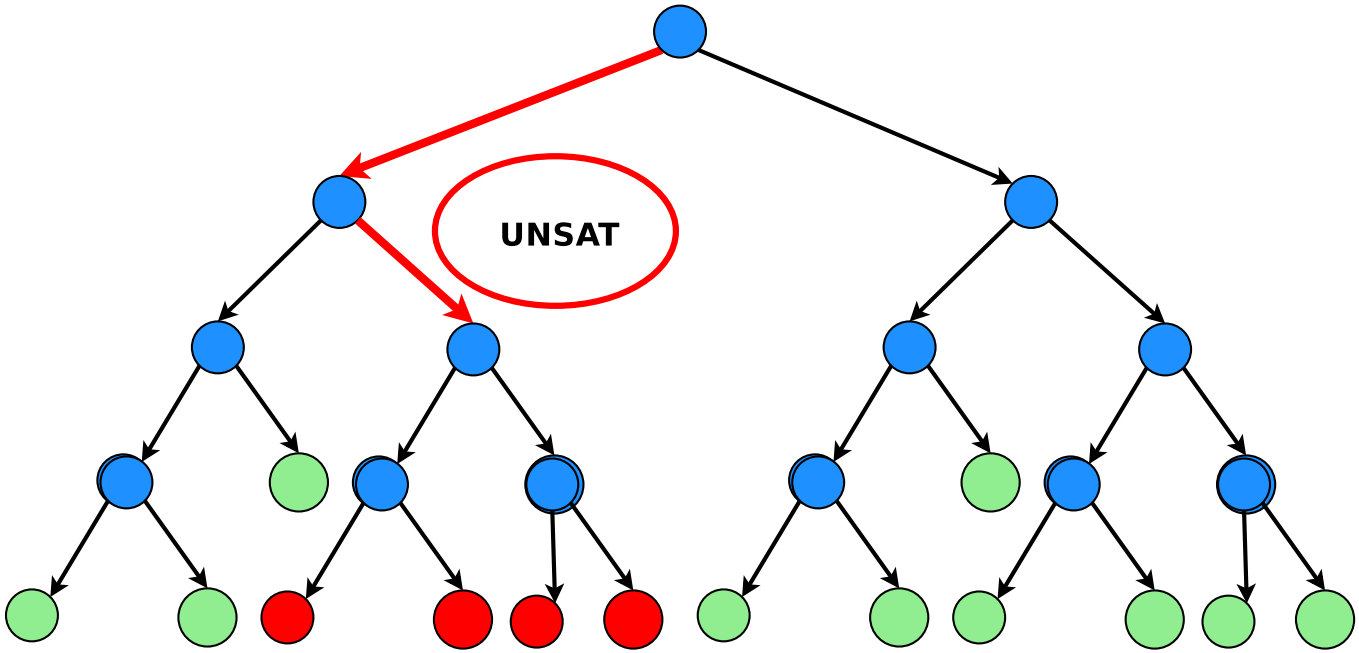
Suivre seulement des chemins faisables (2)



Suivre seulement des chemins faisables (2)



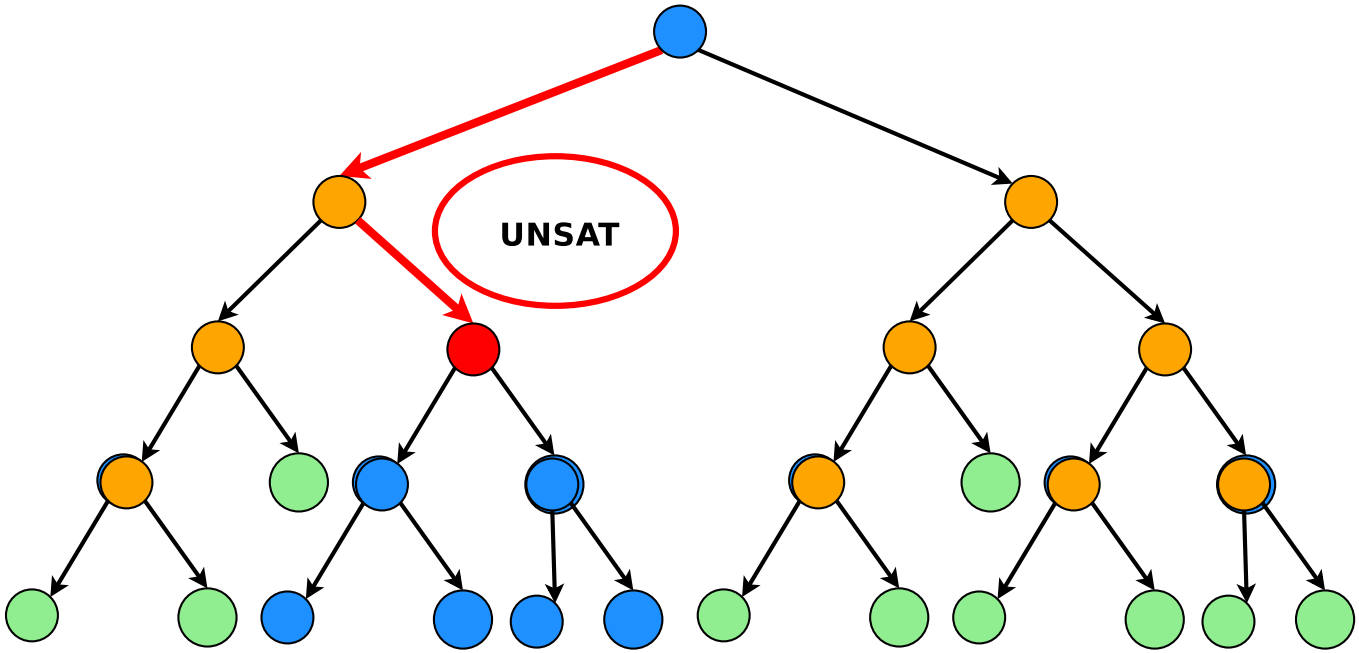
Suivre seulement des chemins faisables (2)



Méthode usuelle : résoudre le prédicat à la fin du chemin

- $+$: un appel au solveur par chemin (sur un arbre : 2^N)
- - : on peut continuer la recherche à partir de préfixes UNSAT
- KO sur programmes avec beaucoup de chemins infaisables

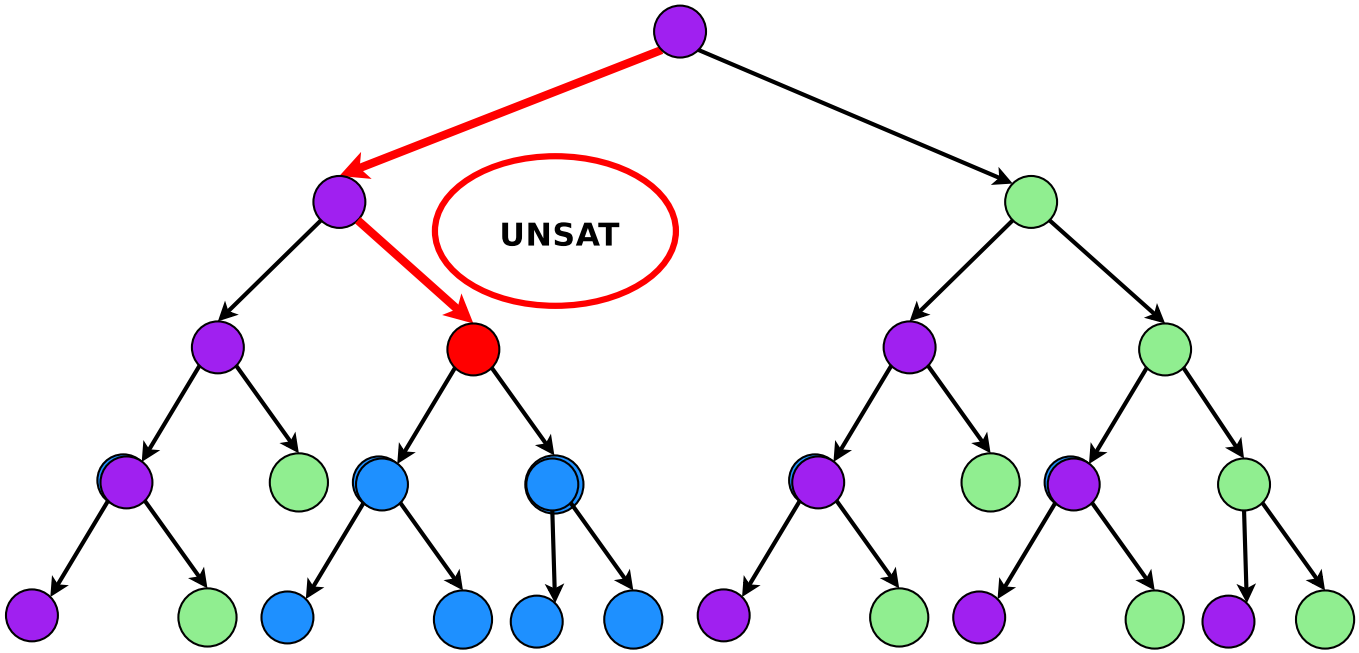
Suivre seulement des chemins faisables (2)



Alternative : résoudre le prédicat à chaque branche

- $+$: détecte UNSAT au plus tôt
- - : un appel au solveur par préfixe de chemin faisable
  et un appel au solveur par préfixe minimal infaisable
  (sur un arbre : $2 * 2^N - 1$)
- KO sur programmes avec peu de chemins infaisables

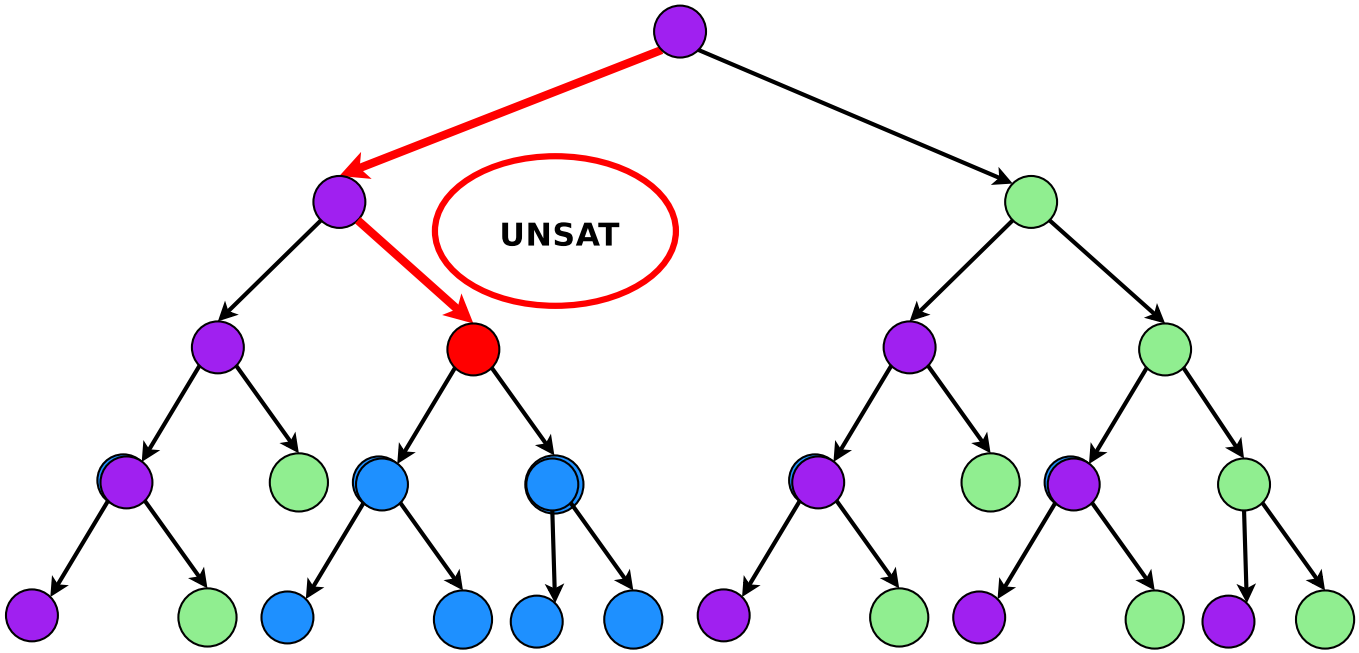
Suivre seulement des chemins faisables (2)



Un exemple possible d'exécution dynamique symbolique

- (hyp : exéc. concrète suit le fils gauche)
- (noeud violet : couvert par exec concrète)

Suivre seulement des chemins faisables (2)



Un exemple possible d'exécution dynamique symbolique

- + : détecte UNSAT au plus tôt
- + : un appel au solveur par chemin (maximal) faisable + un appel par prefixe minimal infaisable
- + : toujours moins d'appels que les deux autres méthodes

Intérêts de la concrétisation

Le mécanisme général de concrétisation peut être utiliser de multiples manières, pour donner des sous-approximations pertinentes

- instructions du programme avec une sémantique hors de T
- instructions du programme hors scope de l'analyseur (ex : asm, sql, librairies en binaire, etc.)
- programmes avec alias et structures complexes : imposer un ensemble fini mais réaliste de relations d'alias entre variables, ou de "formes mémoires"
- multi-thread : imposer un (des) entrelacement(s) réaliste(s) des processus
- contraintes générées trop complexes
  - réduction a priori du nombre de variables
  - élimination de contraintes trop coûteuses [opérateurs non linéaires, read/write de tableaux]

Intérêts de la concrétisation (2)

Le mécanisme de concrétisation est un levier très utile pour adapter la méthode sur des cas difficiles

- compromis d'utilisation
- robustesse à la classe de programmes supportés
- conserve la correction
- ex : pas besoin de gérer parfaitement toutes les constructions d'un langage pour développer une analyse dynamique symbolique *correcte* pour ce langage

Procédure dynamique symbolique basique

Nouvel argument : état mémoire concret C
lancement : SEARCH(node.init, ε ,$\top$, 0)

procedure SEARCH(n, π, Φ, C)

```
 1: Case n of
 2: | halt → ()        /* end node */
 3: | block i → SEARCH(n.next, π · n, Φ ∧ SYMB(i), update(C,i))
 4: | goto n' → SEARCH(n', π · n,Φ, C)
 5: | ite(cond,in,tn) →
 6:       Case eval(cond,C) of     /* follow concrete branch */
 7:          | true →
 8:             SEARCH(in, π · n, Φ ∧ SYMB(cond), C) ;
 9:             try /* solve new branch first */
10:                Sp := SOLVE(Φ ∧ ¬cond) ; Tests := Tests + {(Sp, π.tn)}
11:                C' := UPDATE_C_FOR_BRANCHING(Sp)
12:                SEARCH(tn, π · n,Φ ∧ ¬SYMB(cond),C')  /* branching */
13:             with unsat → ()
14:             end try
15:          | false → .....        /* symmetric case */
16:       end case
17: end case
```

Procédure dynamique symbolique basique (2)

update(C : mem-conc,i : instr) $\rightarrow$ mem-conc : màj de l'état mémoire actuel

eval(cond : predicat,C : mem-conc) $\rightarrow$ bool : évalue la condition cond vis à vis de l'état mémoire actuel

update_C_for_branching(Sp : DT)$\rightarrow$ mem-conc : créer un nouvel état mémoire cohérent avec le préfixe de chemin suivi par DT

(Explication du dernier point : à chaque moment, invariant : Φ et C cohérents avec chemin suivi jusque là. Cet invariant est cassé quand on impose un branchement car l'exécution concrète ne "partait pas dans ce sens là").

Plan

- Automatisation de la génération de tests
  - Prédicat de chemins
  - Exécution symbolique
  - Limites
  - Exécution dynamique symbolique
  - Optimisations
  - En pratique
  - Discussion
  - Complément : Aspects logiques
  - Complément : Optimisations

Optimisations possibles

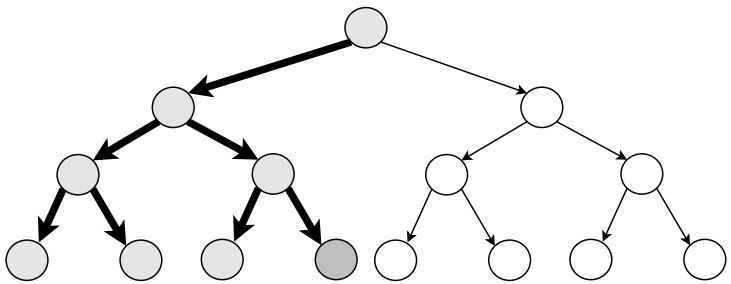
Diminuer le nombre d'appels au solveur

- élaguer les chemins redondants [ne perd rien]
  - redondance par rapport au futur [rien de neuf à couvrir]
  - redondance par rapport au passé [état symbolique déjà visité]
- parcourir plus rapidement [juste une heuristique]
  - dfs est un des pires choix ...
- gestion des appels de fonction ? problème ouvert ...

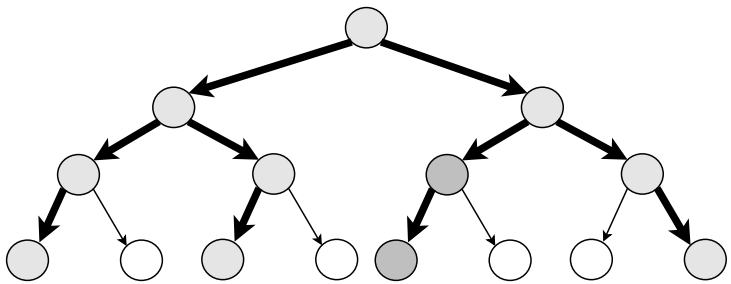
Diminuer le coût d'un appel au solveur

- preprocessing [cst propag, réécriture, slicing, constraint splitting, etc.]
- résolution incrémentale, conservation des "unsat core"
- cache
- réutilisation des solutions précédentes

Exemple : couvrir plus vite



(a) DFS



(b) BFS

Couverture pour 4 tests générés

Exemple : couvrir plus vite (2)

Couverture des chemins par DSE : tous les parcours de chemins se valent

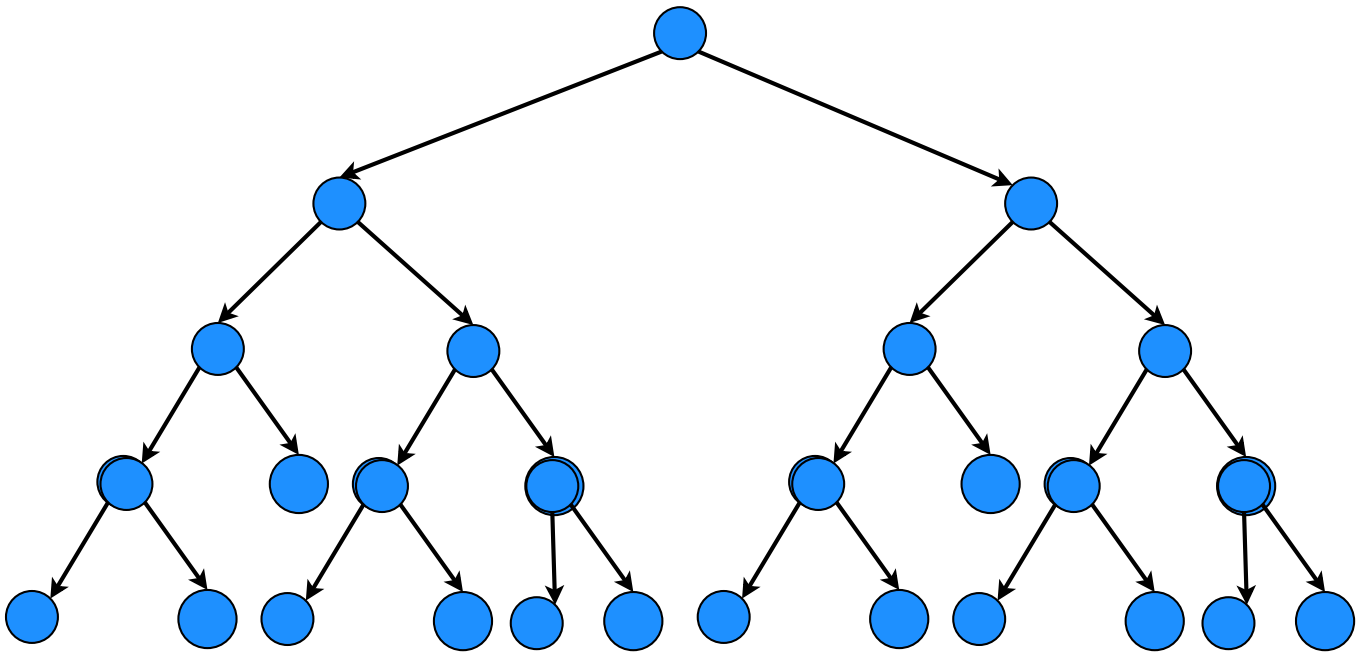
Couverture des branches (budget limité ou non) : tous les parcours de chemins ne se valent pas

- DFS et BFS sont souvent très mauvais

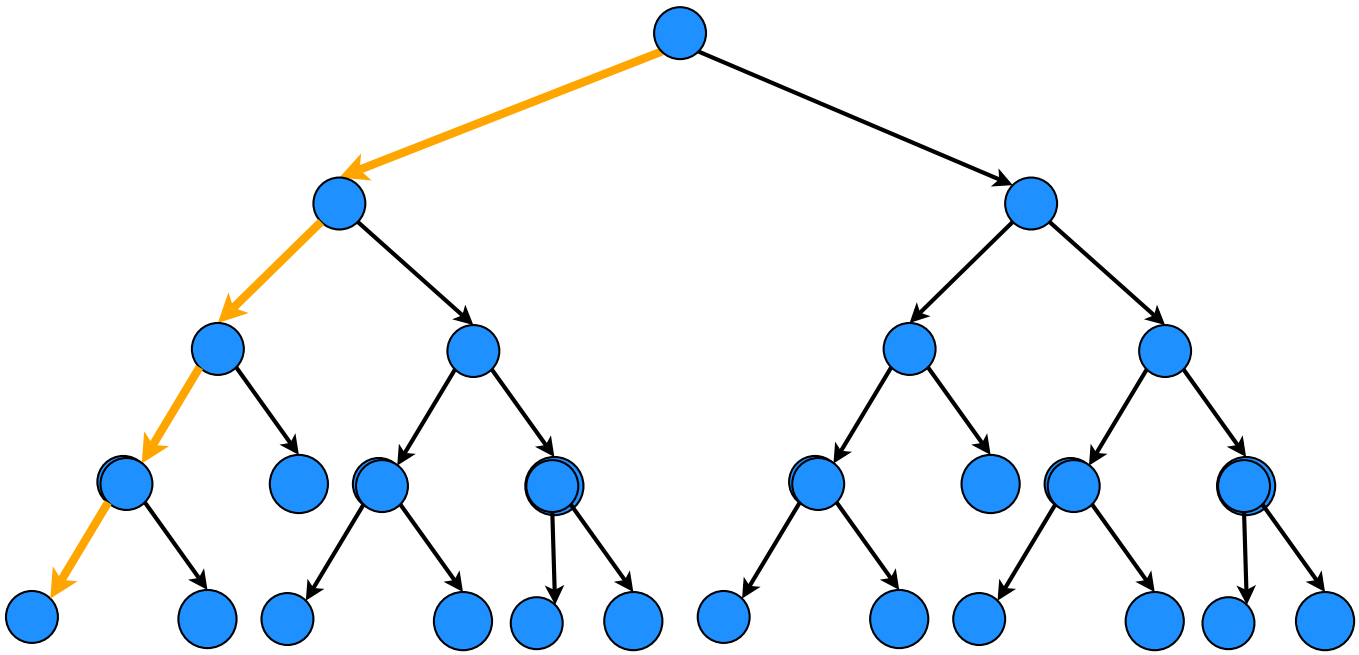
Quelques solutions

- parcours hybride (CUTE) : DFS + aléatoire
  [simple, meilleur que dfs]
- *fitness guided* [EXE, SAGE, PEX] : les préfixes actifs sont évaluées, celui de plus haut score est étendu
  [mécanisme très versatile]

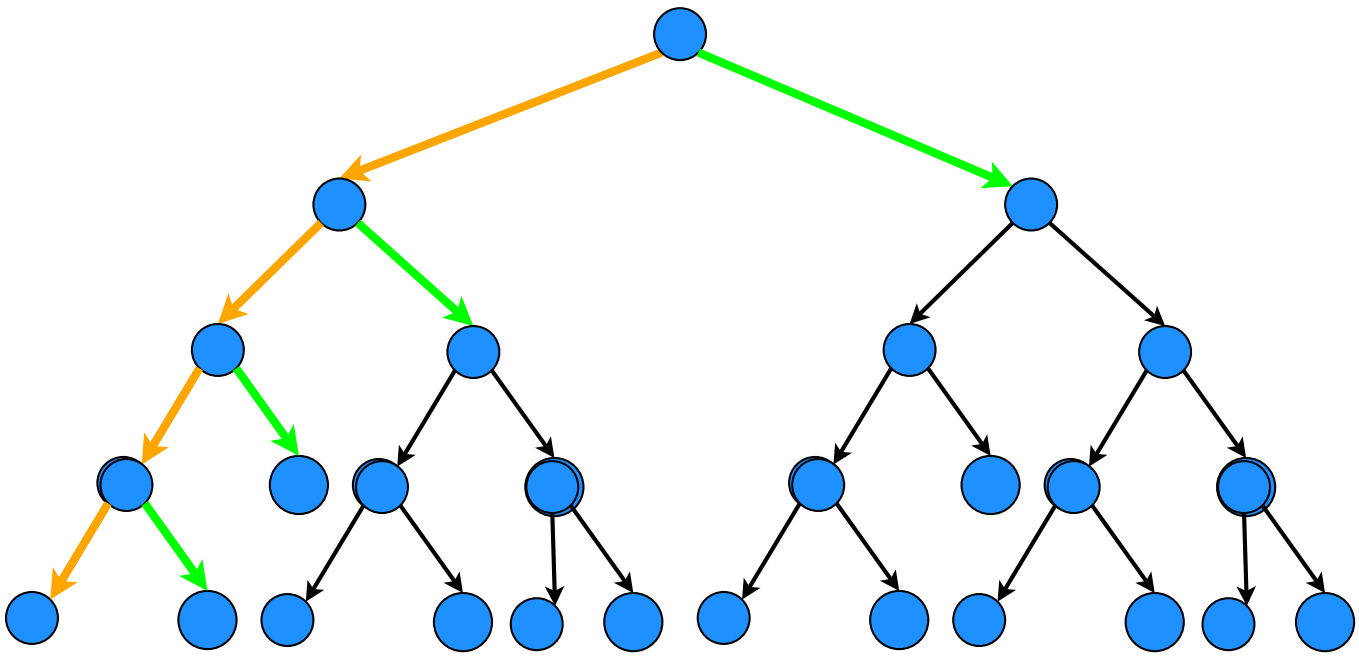
Exemple : couvrir plus vite (3)



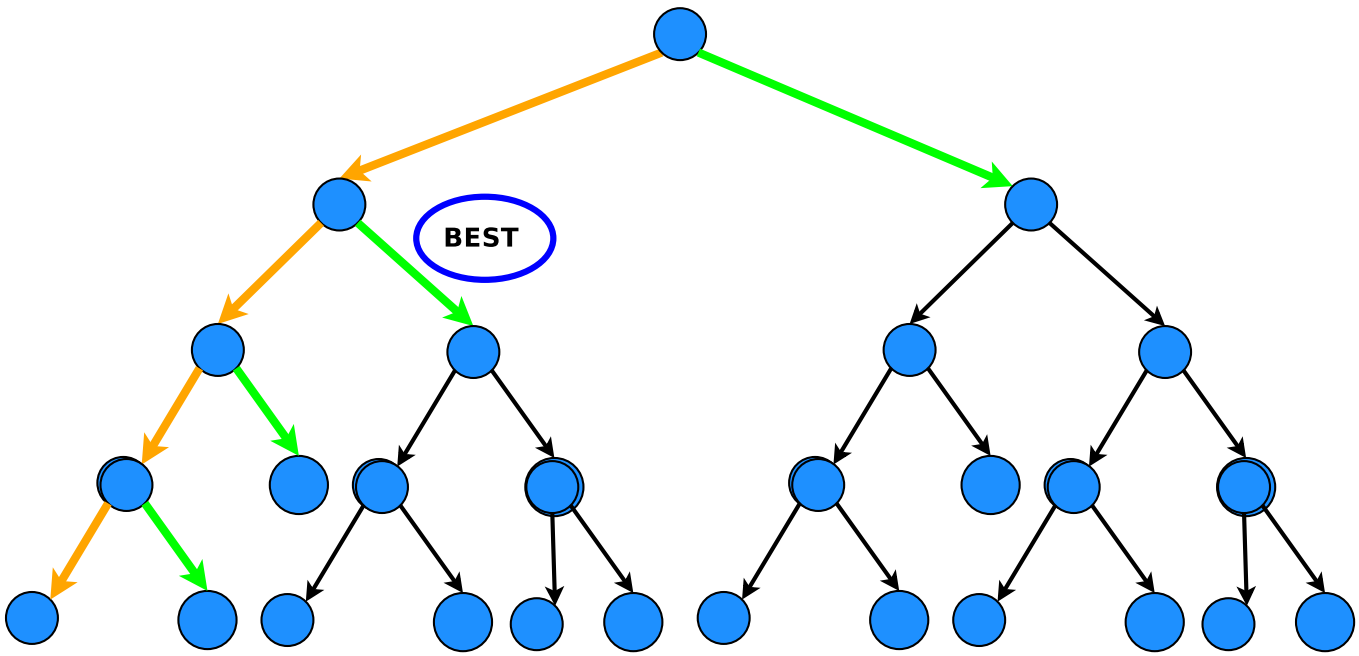
Exemple : couvrir plus vite (3)



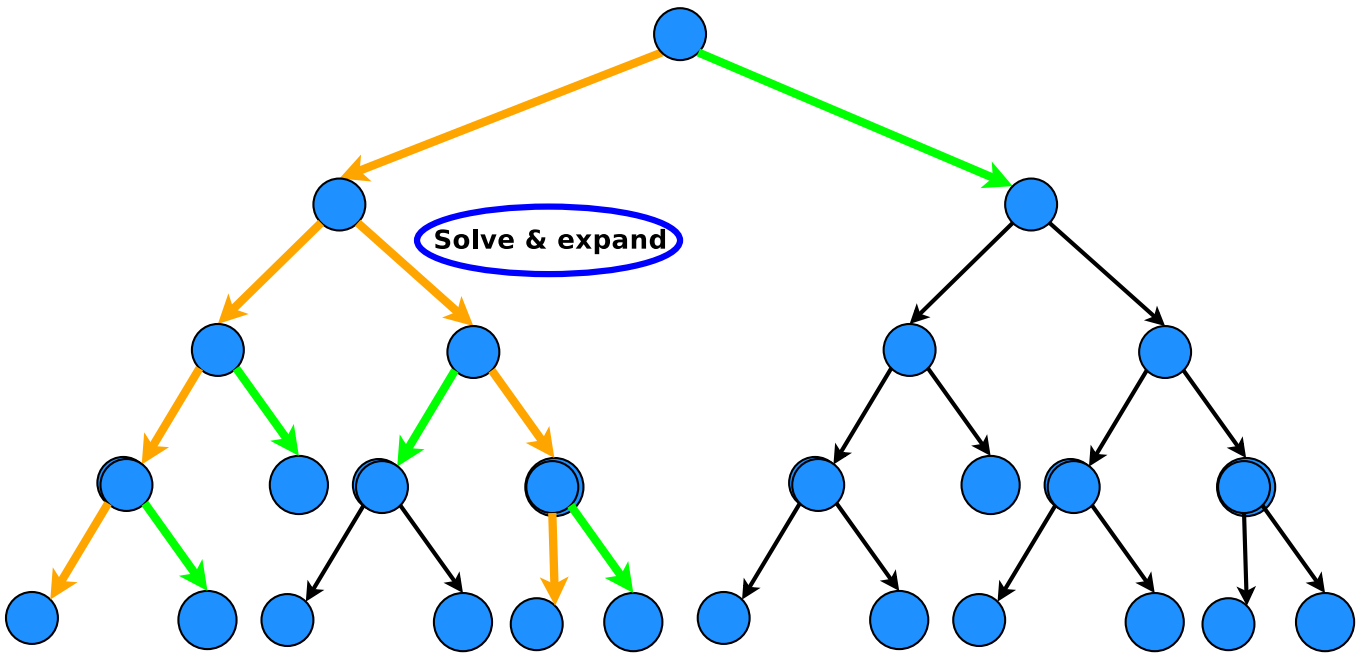
Exemple : couvrir plus vite (3)



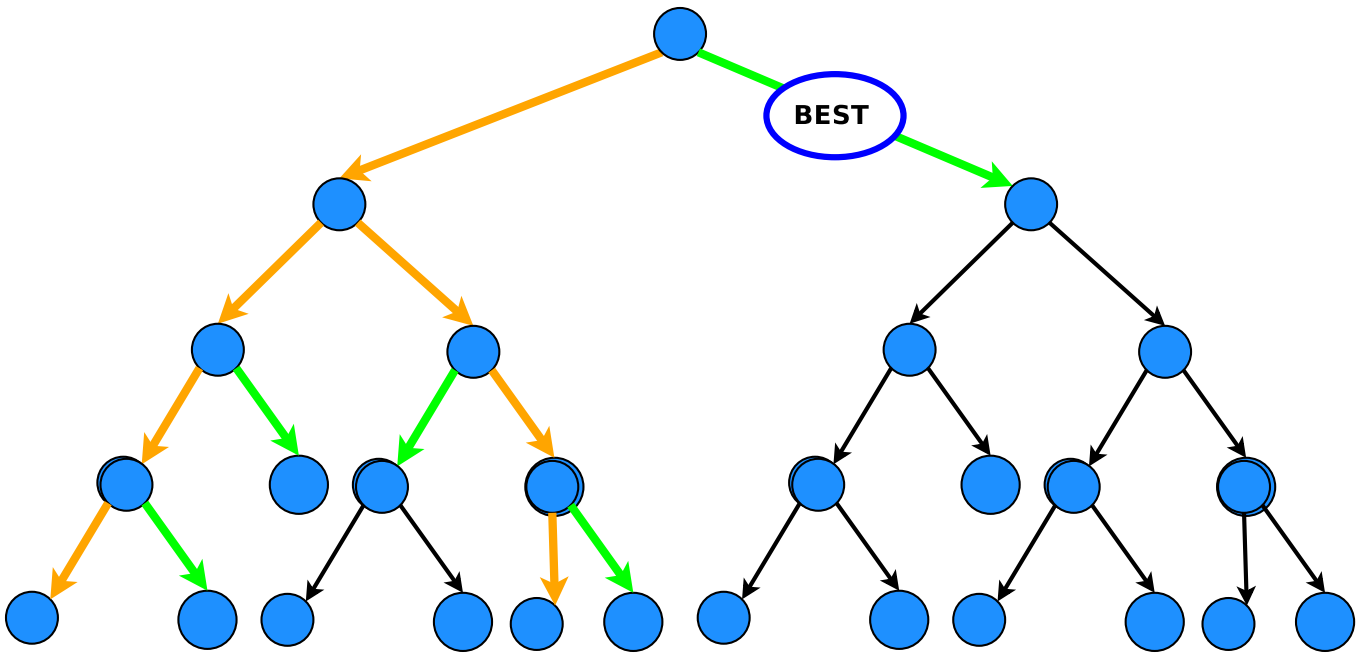
Exemple : couvrir plus vite (3)



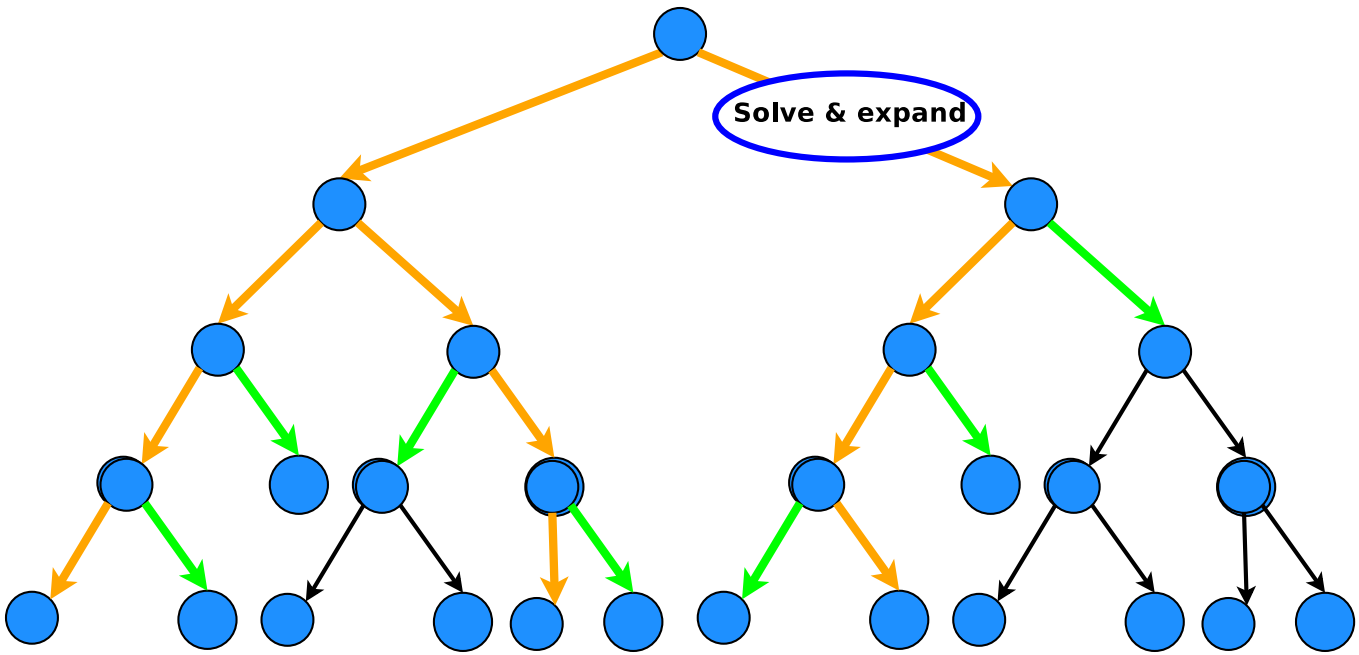
Exemple : couvrir plus vite (3)



Exemple : couvrir plus vite (3)



Exemple : couvrir plus vite (3)



Plan

- Automatisation de la génération de tests
  - Prédicat de chemins
  - Exécution symbolique
  - Limites
  - Exécution dynamique symbolique
  - Optimisations
  - En pratique
  - Discussion
  - Complément : Aspects logiques
  - Complément : Optimisations

Quelques résultats pratiques

PEX livré dans Visual C#

- cible = aide au programmeur

SAGE en production interne chez Microsoft (sécurité)

- service interne de "smart fuzzing"
- le logiciel tourne en boucle sur de gros serveurs
- nombreux bugs trouvés

Études de cas académiques sur des codes type drivers / kernel (Linux, BSD)

- codes souvent déjà bien testés, nombreux bugs trouvés
- ex : Klee : $>$ 95 % de couverture obtenue automatiquement sur les Unix coreutils (make, grep, wc, etc.)

Quelques problèmes

- préconditions complexes
- oracle et détection de bugs
- critères de couverture

Préconditions complexes

Pourquoi les prendre en compte ? éviter des tests non pertinents

- ex : algorithme de recherche dichotomique
- le tableau d'entrée généré doit être trié, sinon le test n'est pas représentatif

Pas toujours simple

- `x` non nul ✓
- le tableau doit être trié ✗ [à cause du $\forall$]

SOLUTION 1 : filtrer a posteriori

- générer les DT comme avant
- puis éliminer toutes les DT ne respectant pas la précondition
- PB : nécessite une précondition exécutable
- PB : ne fonctionne pas si précondition trop contrainte

Préconditions complexes (2)

SOLUTION 2 : générer à coup sûr des DT satisfaisant *Pre*

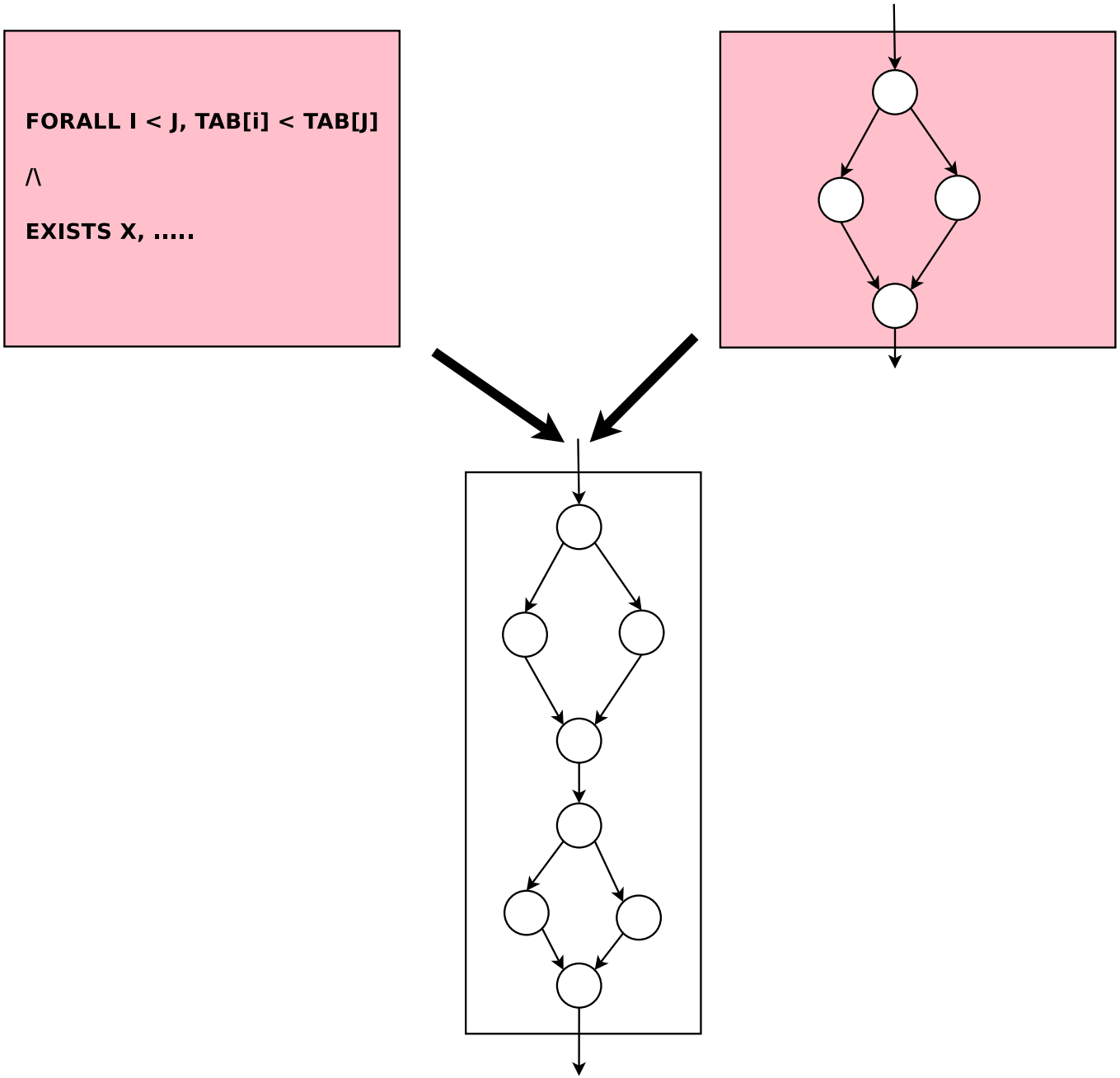
Approche 1 : gestion de la précondition au niveau logique

- on résoud des formules de la forme $\pi \wedge \text{PRECOND}$
- PB : les préconditions élaborées (ex : tableau trié) demandent des quantificateurs $\forall$
  - . pas de synthèse de solution possible pour le moment ...

Approche 2 : gestion de la précondition au niveau du code

- on ajoute au début du programme une fonction `check_precond(args)`
- PB1 : les préconditions élaborées (ex : tableau trié) demandent des boucles (#chemins ++)
- PB2 : nécessite des préconditions exécutables

Préconditions complexes (3)



Oracle

Prise en compte de l'oracle de la fonction à tester

- rapport de tests plus informatifs (dt,chemin,verdict)
- idéal : générer directement des tests fautifs

On retrouve les mêmes problèmes que pour la précondition

- a posteriori : tests générés sans oracle, puis verdict ensuite
- a priori : tests générés *contre* l'oracle

Quel format pour l'oracle ?

- code ou formule : même pb que pour la précondition
- des oracles partiels simples (runtime errors) peuvent être légers et intéressants du point de vue guidage de la génération

Autres

Sortir les tests dans un format réutilisable

- exporter vers autres outils
  - JUnit, calcul de couverture, sélection / minimisation, etc.
- utilisation conjointe de DT issues d'autres méthodes / outils
  - tests issus de méthodes manuelles ou orientées modèles
  - méthodes automatiques simples (random testing, interface-based testing)

Génération en complément de tests existants

- éviter redondance avec tests existants
- génération incrémentale

Plan

- Automatisation de la génération de tests
  - Prédicat de chemins
  - Exécution symbolique
  - Limites
  - Exécution dynamique symbolique
  - Optimisations
  - En pratique
  - Discussion
  - Complément : Aspects logiques
  - Complément : Optimisations

Exécution symbolique

Idée ancienne, mais automatisation complète récente

- concept introduit par King dans les années 1970
- au début : tout à la main, utilisateur se débrouille
- ensuite : on crée φ_π, puis utilisateur se débrouille
- automatisation complète sur des programmes : 2004-2006 (Berkeley, CEA, CMU, Microsoft, Stanford)
  . nécessite deux avancées : solveurs, dynamic symbolic

Réflexions sur dynamique symbolique

Plutôt récent

- déjà dans PathCrawler (CEA, 2004) pour chemins infaisables
- popularisé par DART et CUTE (2005) pour robustesse

Symbolic execution : $\approx$ analyses statiques sur un chemin

- plus facile car juste un chemin, mais incomplet
- garde les défauts des analyses statique pure (enfermé dans une théorie)

Dynamic Symbolic execution : statique + dynamique

- paradigme vraiment différent de statique pure
- grande robustesse
- compromis de mise en œuvre

Quelques prototypes existants

PathCrawler (CEA)	2004
Dart (Bell Labs), Cute (Berkeley)	2005
Exe (Stanford)	2006
Java PathFinder (NASA)	2007
Osmose (CEA), Sage (Microsoft), Pex (Microsoft)	2008

Bilan (subjectif) sur l'approche dynamique symbolique

Points forts pour une utilisation industrielle

- totalement automatisée si oracle automatique
- robuste aux "vrais" programmes
- correcte, résultats facilement vérifiables
- s'insère dans pratiques de test existantes (process, outils, etc.)
- utilisation (naturellement) incrémentale
- gain incrémental
- surpasse assez facilement les pratiques usuelles (ex : fuzzbox testing)

Puissance maximale si couplée avec un langage de contrat

Quels domaines d'utilisation ?

- pb pour la certification : traçabilite DT - exigences
- pb si l'on veut absolument 100% de couverture, même sur des codes de taille petite / moyenne
- mais ok pour le débogage intensif

Autres utilisations de la DSE

- back-to-back testing
- conformance testing
- binary-level testing
- exploit generation
- exploit hardening
- deobfuscation
- malware exploration
- *proof of correctness for certain compiler optimizations (peephole)*

Quelques challenges

Passage à l'échelle (# chemins)

- quelle notion de résumé de fonction / boucle ?
- comment "bouchonner" facilement un morceau de code ?

Prise en compte de préconditions complexes en cas de structures dynamiques

- quantificateurs, axiomes

Au niveau solveurs :

- chaines de caractères, flottants

Plan

- Automatisation de la génération de tests
  - Prédicat de chemins
  - Exécution symbolique
  - Limites
  - Exécution dynamique symbolique
  - Optimisations
  - En pratique
  - Discussion
  - Complément : Aspects logiques
  - Complément : Optimisations

À propos des théories utilisées

Chaque instruction doit être traduite vers une formule

- le langage des formules peut être très riche

Les contraintes doivent être résolues automatiquement

- satisfiabilité décidable + résolution efficace *en pratique*
- beaucoup moins de liberté ! !

Remarques

- compromis expressivité VS décidabilité / complexité
- si théorie pas assez expressive : approximations [concrétisation]

Quelques définitions

Une théorie logique T

- un ensemble prédéfini de symboles de fonctions et prédicats [en général au moins $=$]
- une sémantique implicite [domaine des variables, "sens" des fonctions et prédicats]
- des axiomes imposant la sémantique implicite
- on note $T \models \varphi$ pour dire que la formule φ est valide dans la théorie T

Un fragment logique

- on limite les connecteurs logiques
- typiquement : pas de quantificateur, pas de $\vee$

À propos des théories utilisées (2)

Avantage d'être sur un chemin

- fragment simple : pas de quantificateur, seulement des conjonctions
- beaucoup de classes décidables, voir solubles efficacement

Théories pour les types de base

- $(\mathbb{N}, x - y\#k)$ (logique de différence, P)
- $(\mathbb{R}, +, \times k)$ (arithmétique linéaire, P [Simplex (non polynomial)])
- $(\mathbb{N}, +, \times k)$ (arithmétique linéaire, NP-complet [Omega test])
- $\mathcal{B}$ (booléens, NP-complet [DPLL ou BDDs])
- $(\mathbb{N}_{\leq}, +, \times)$ (arithmétique bornée non linéaire, NP-complet [CP(FD)])
- BV (bitvecteurs, NP-complet [bitblasting])
- $FLOAT$ (arithmétique flottante, NP-complet)
- types algébriques (algèbre libre, P [unification])

Exemple de théorie : EUF

Théorie des fonctions non interprétées (EUF)

signature : $\langle =, \neq, x_1, \ldots, x_n, f_1(\ldots), \ldots, f_m(\ldots) \rangle$

axiomatique : $(FC)\ x = y \Rightarrow f(x) = f(y)$

Utilité : pratique pour relier des éléments entre eux de manière implicite

- `&x` en C devient $addr(X)$
- `x` une structure avec deux champs `num` et `flag` : $num(X)$ et $flag(X)$

Résolution très efficace

- algorithme de *congruence closure* (cf Wikipedia)
- polynomial

Variantes

- (AC) axiomes d'associativité / commutativité sur certains symboles de fonction
- (free-algebra) algèbre de types libres

Exemple de théorie : Arrays

Théorie des tableaux

signature : $\langle ARRAY, I, E, =_I, \neq_I, =_E, \neq_E, load, store \rangle$

sémantique :

- $load : ARRAY \times I \mapsto E$
- $store : ARRAY \times I \times E \mapsto ARRAY$
- axiomes : FC pour $load/store$, plus
  $(RoW1)\ i = j \Rightarrow load(store(A, i, v), j) = v$
  $(RoW2)\ i \neq j \Rightarrow load(store(A, i, v), j) = load(A, j)$

Utilité : tableaux bien sûrs, mais aussi map, vectors, etc.

Résolution : EUF + case-split, problème NP-complet

Quelles théories en pratique ?

expressivité $\nearrow$: moins d'échecs mais résolution sur un chemin + chère

- ex : BV + Array (NP-complet)

expressivité $\searrow$: risque plus d'échecs (concrétisation), mais résolution sur un chemin - chère

- ex : Difference + EUF (Polynomial)

Compromis idéal ? ?

Observation 2004-2011 : théories de + en + puissantes

Deux technologies de solveurs

- SMT : schéma très intéressant de combinaison de solveurs (Nelson-Oppen) intégré à une gestion efficace des booléens
- (plus confidentiel) Constraint Programming :
  - pour les variables à domaines finis
  - des approches intéressantes pour FLOAT, BV, $(\mathbb{N}_{\leq}, +, \times)$

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 1 :

$X_{n+1} = A_n + B_n$

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 1 :

$X_{n+1} = A_n + B_n$

Modèle mémoire sous-jacent : ensemble de variables $\{A, B, X, \ldots\}$

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 1 :

$X_{n+1} = A_n + B_n$

Modèle mémoire sous-jacent : ensemble de variables $\{A, B, X, \ldots\}$

Bien mais ne pourra prendre en compte les pointeurs

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 2 : ajout d'un état mémoire M

$store(M, addr(X), load(M, addr(A)) + load(M, addr(B)))$

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 2 : ajout d'un état mémoire M

$store(M, addr(X), load(M, addr(A)) + load(M, addr(B)))$

Modèle mémoire sous-jacent : map $\{Addr_1 \mapsto A, Addr_2 \mapsto B, \ldots\}$

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 2 : ajout d'un état mémoire M

$store(M, addr(X), load(M, addr(A)) + load(M, addr(B)))$

Modèle mémoire sous-jacent : map $\{Addr_1 \mapsto A, Addr_2 \mapsto B, \ldots\}$

ok pour les pointeurs, mais on ne peut écrire au milieu de `x` (respect du typage, modèle mémoire à la Java)

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 3 : encodage de M au niveau octet (ici : 3 octets)

let tmpA = load(M,addr(A)) @ load(M,addr(A)+1) @ load(M,addr(A)+2)
and tmpB = load(M,addr(B)) @ load(M,addr(B)+1) @ load(M,addr(B)+2)
in
let nX = tmpA+tmpB
in

$$store(store(store(M, addr(X), nX[0]), addr(X)+1, nX[1]), addr(X)+2, nX[2])$$

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 3 : encodage de M au niveau octet (ici : 3 octets)

let tmpA = load(M,addr(A)) @ load(M,addr(A)+1) @ load(M,addr(A)+2)
and tmpB = load(M,addr(B)) @ load(M,addr(B)+1) @ load(M,addr(B)+2)
in
let nX = tmpA+tmpB
in

$$store(store(store(M, addr(X), nX[0]), addr(X)+1, nX[1]), addr(X)+2, nX[2])$$

ok pour du C ... mais la formule est complexe

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

PB ouvert : affiner automatiquement le niveau d'abstraction de la modélisation

Plan

- Automatisation de la génération de tests
  - Prédicat de chemins
  - Exécution symbolique
  - Limites
  - Exécution dynamique symbolique
  - Optimisations
  - En pratique
  - Discussion
  - Complément : Aspects logiques
  - Complément : Optimisations

Optimisations de la DSE

Diminuer le nombre d'appels au solveur

- méthodes correctes d'élagage de chemins
- heuristiques : parcours de chemin plus malins que la DFS

Diminuer le coût moyen d'un appel au solveur

- on se place dans le cas où le solveur est utilisé en boite noire
- on peut quand même améliorer ses performances

Optimisation de chemins : élagage

Élimination de chemins redondants : certains chemins sont redondants pour le critère de couverture choisi, on peut les éviter

- techniques complètes

Gestion des appels de fonction : cause principale de l'explosion du nombre de chemins

- pas encore satisfaisant

Chemins redondants

Couper les chemins qui ne peuvent atteindre de nouvelles instructions

- pour chaque chemin actif, calcul des instructions accessibles à partir de l'état actif
- on stop le chemin si le préfixe ne peut atteindre de cible non couverte
- calcul des accessibles peut être fait très efficacement
- technique complète vis à vis de la couverture d'instructions

Technique complète vis à vis de la couverture d'instructions

- on ne perd rien

Technique puissante : (DFS + optim) meilleure que bcp de parcours avancés

Chemins redondants (2)

Couper les chemins amenant à un état symbolique déjà couvert : si on a deux préfixes π et σ tq $\phi_\pi \Rightarrow \phi_\sigma$, alors on garde seulement le préfixe σ

- technique complète vis à vis de la couverture d'instructions
- potentiellement très coûteuse, demande de vérifer $\Rightarrow$
- on peut utiliser un calcul approché de $\Rightarrow$ via $\preceq$

Technique complète vis à vis de la couverture d'instructions

Gestion des appels de fonction

Reste un problème ouvert

Quelques solutions partielles

- couper l'exploration à une certaine profondeur [incomplet !] (Osmose, Java PathFinder)
- gestion paresseuse des fonctions (Sage)
- construction itérative de résumés de fonctions (Dart)
- spécifications de fonctions (PathCrawler)

Gestion des appels de fonction (2)

Couper l'exploration à une certaine profondeur

- simple à mettre en oeuvre ! !
- pas complet

Variantes

- concrétiser : prédicat de chemin correct et simple, mais très (trop ?) contraint
- remplacer l'appel de fonction par $\top$: formule simple mais incorrecte (!)
- rentrer dans la fonction mais empêcher l'énumération de chemin : correct, mais formule compliquée

Gestion des appels de fonction (3)

Résumés de fonction de type $Pre(\vec{In}) \wedge Post(\vec{In}, \vec{Out})$

C'est le cas idéal

- pas besoin d'entrer dans la fonction appelée pour énumérer les chemins
- pas besoin d'entrer dans la fonction appelée pour générer des contraintes

Attention

- le solveur doit pouvoir gérer la spéc
- la spéc doit être fonctionnelle, et suffisamment précise pour permettre de déduire des valeurs (pas de surapproximation)
- qui donne la spéc ?

Gestion des appels de fonction (4)

Alternatives pour les résumés de fonction

- résumés en sous-approximation
- utilisation incrémentale de résumés en surapproximation

Résumés en sous-approximation de type $\bigvee \phi(\vec{in}) \Rightarrow \psi(\vec{out})$

- correct, et peut être construit pendant l'exploration
- rappelle la logique des "stubs" en test usuel
- ajoute des $\vee$, formules plus lourdes
- peut être construit incrémentallement

Rappel : Optimisations de la DSE

Diminuer le nombre d'appels au solveur

- méthodes correctes d'élagage de chemins
- heuristiques : parcours de chemin plus malins que la DFS

Diminuer le coût moyen d'un appel au solveur

Heuristique : Couvrir plus vite

Technique usuelle : parcours en profondeur (DFS)

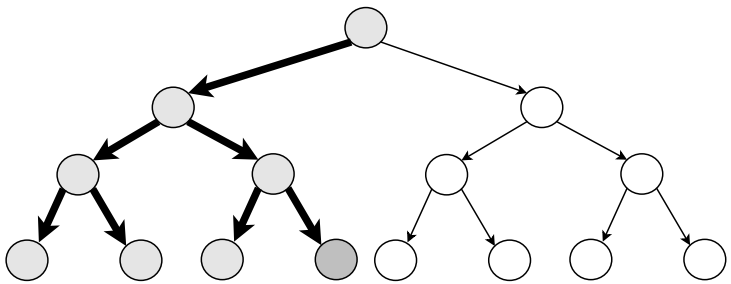
Avantage classique de DFS

- un seul contexte ouvert à la fois (mémoire)
- simple à implanter en récursif

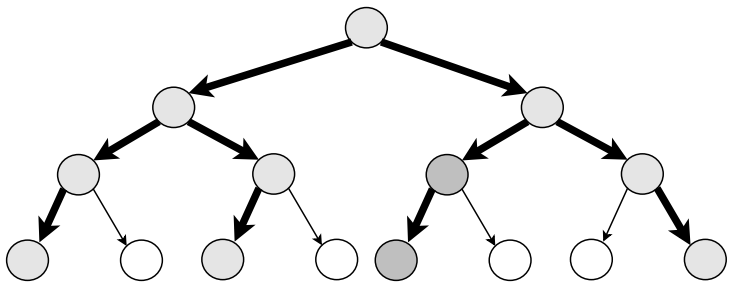
Problème de la DFS pour la génération de tests

- si #DT limité, la DFS se concentre sur une portion très restreinte du code

Couvrir plus vite (2)



(a) DFS



(b) BFS

Couverture pour 4 tests générés

Couvrir plus vite (3)

Couverture des chemins par DSE : tous les parcours de chemins se valent

Couverture des branches (budget limité ou non) : tous les parcours de chemins ne se valent pas

- DFS est souvent très mauvais

Quelques solutions

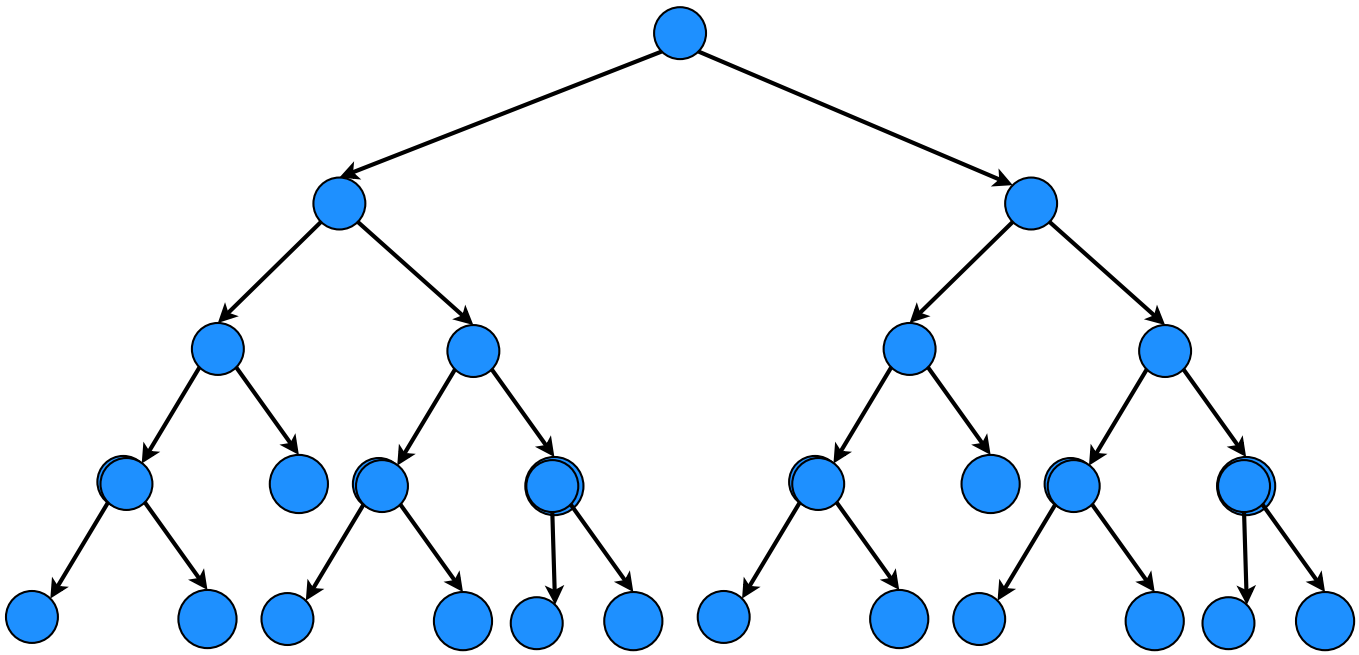
- parcours hybride (CUTE) : DFS + aléatoire
  [simple, meilleur que dfs]
- *fitness guided* [EXE, SAGE, PEX] : les préfixes actifs sont évaluées, celui de plus haut score est étendu
  [mécanisme très versatile]

Parcours fitnex-guided

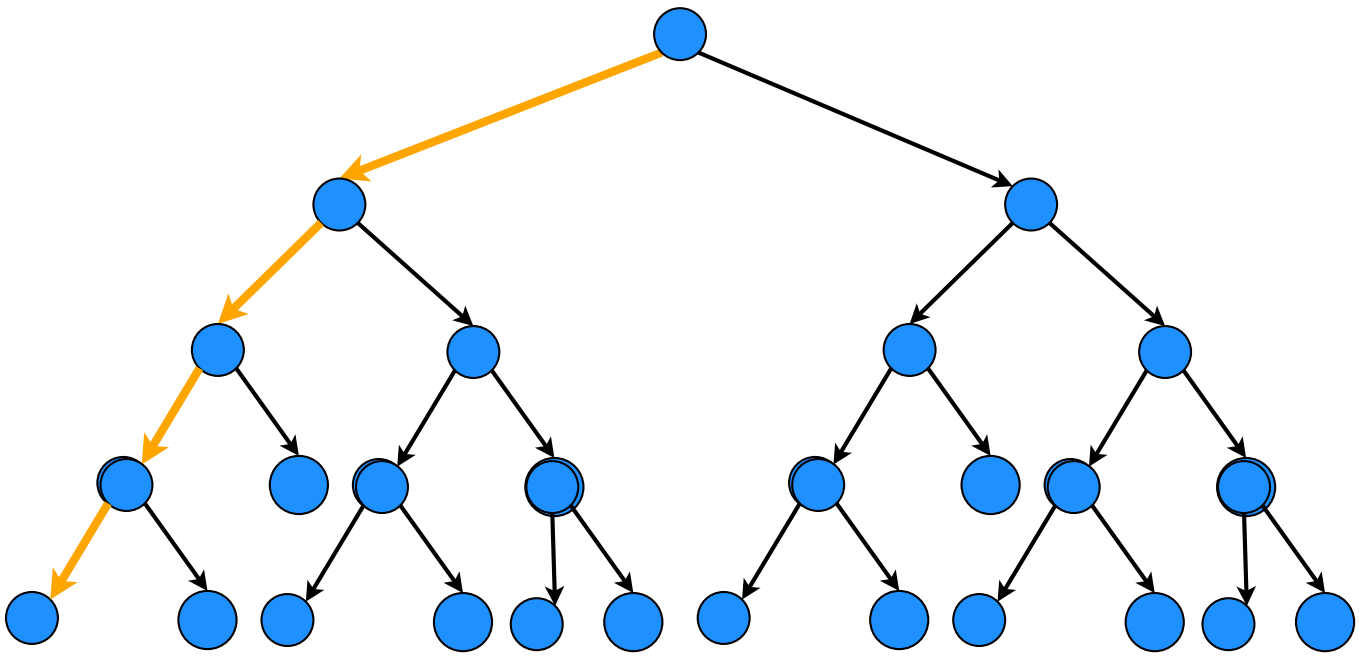
Ingrédients de l'exécution symbolique "Fitness-guided"

- **chemin actif** : chemin non couvert, dont le plus long (strict) préfix est couvert [on dit aussi préfix actif]
- notion de **score** d'un chemin actif
- à chaque étape : **sélection** + **extension** +
  - choisir le chemin actif ayant le meilleur score
  - "étendre" ce chemin : résolution + exécution
  - ajouter les nouveaux chemins actifs créés par cette exécution

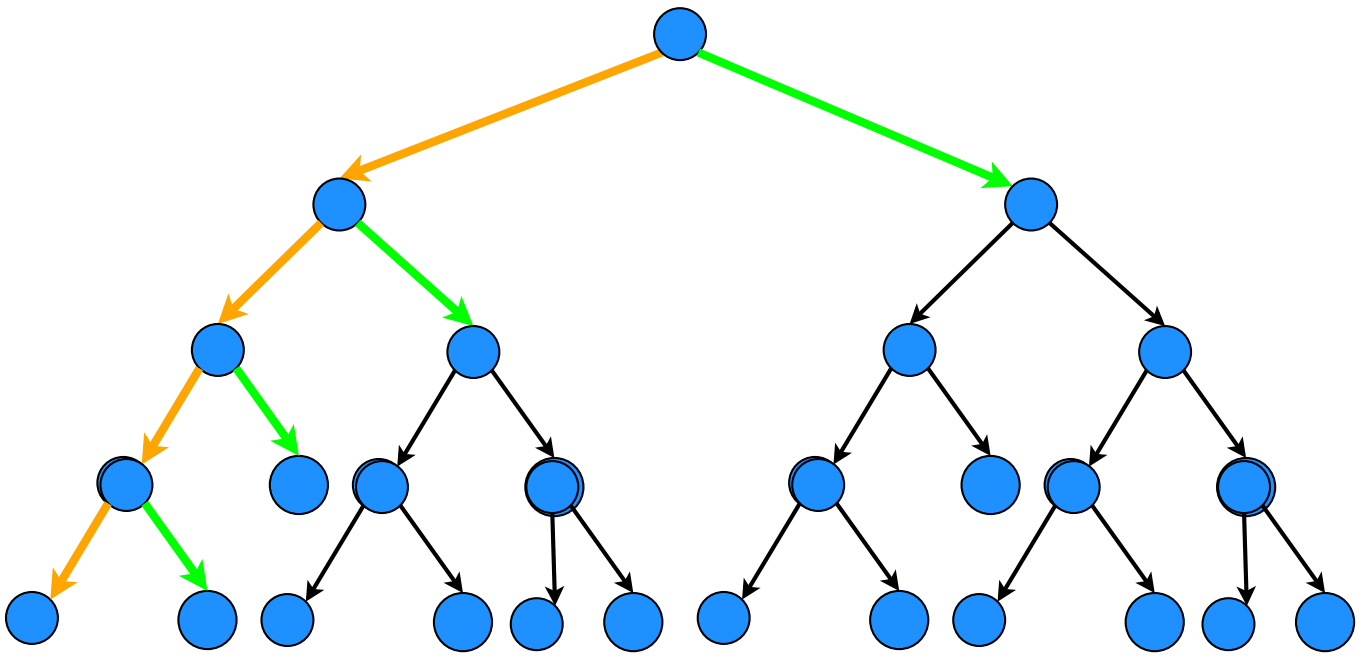
Parcours fitnex-guided (2)



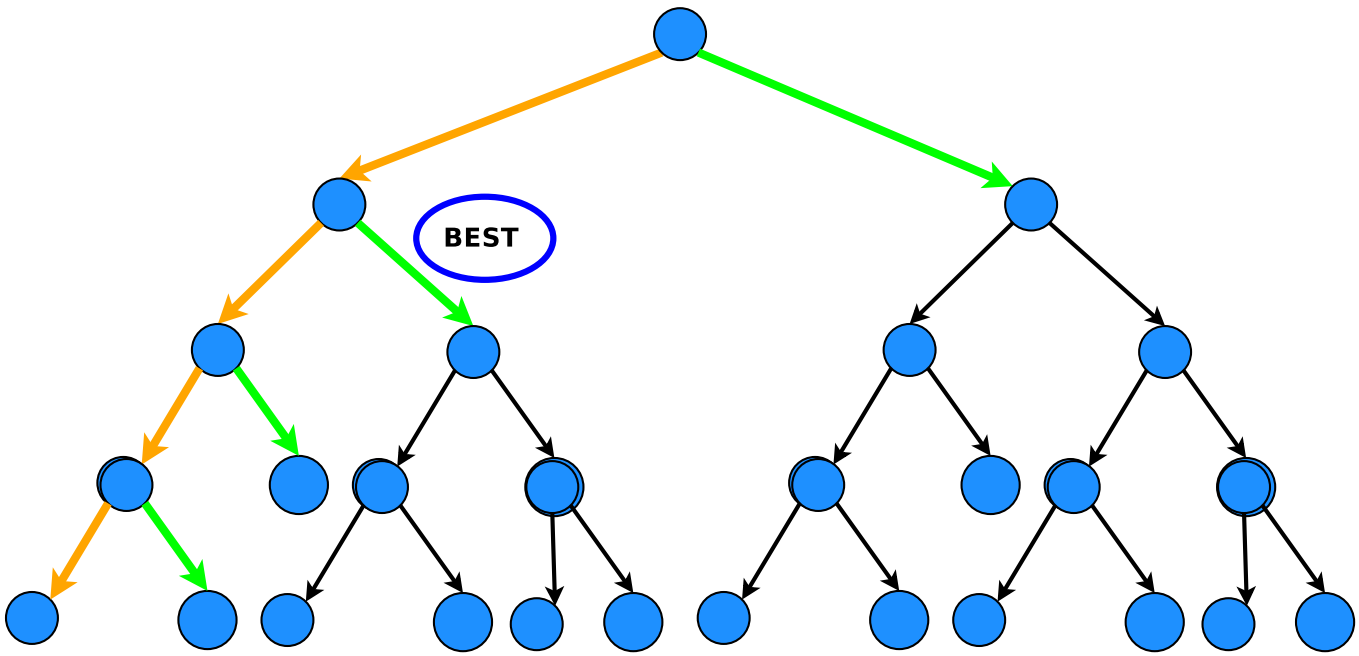
Parcours fitnex-guided (2)



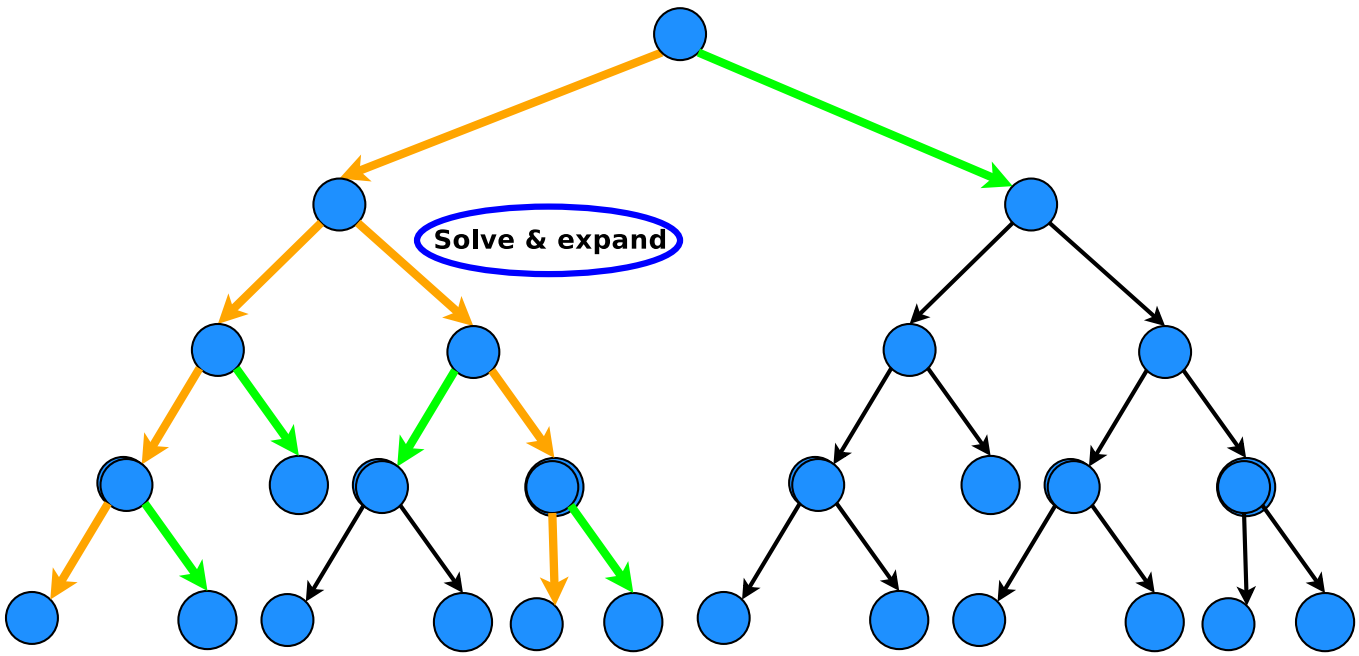
Parcours fitnex-guided (2)



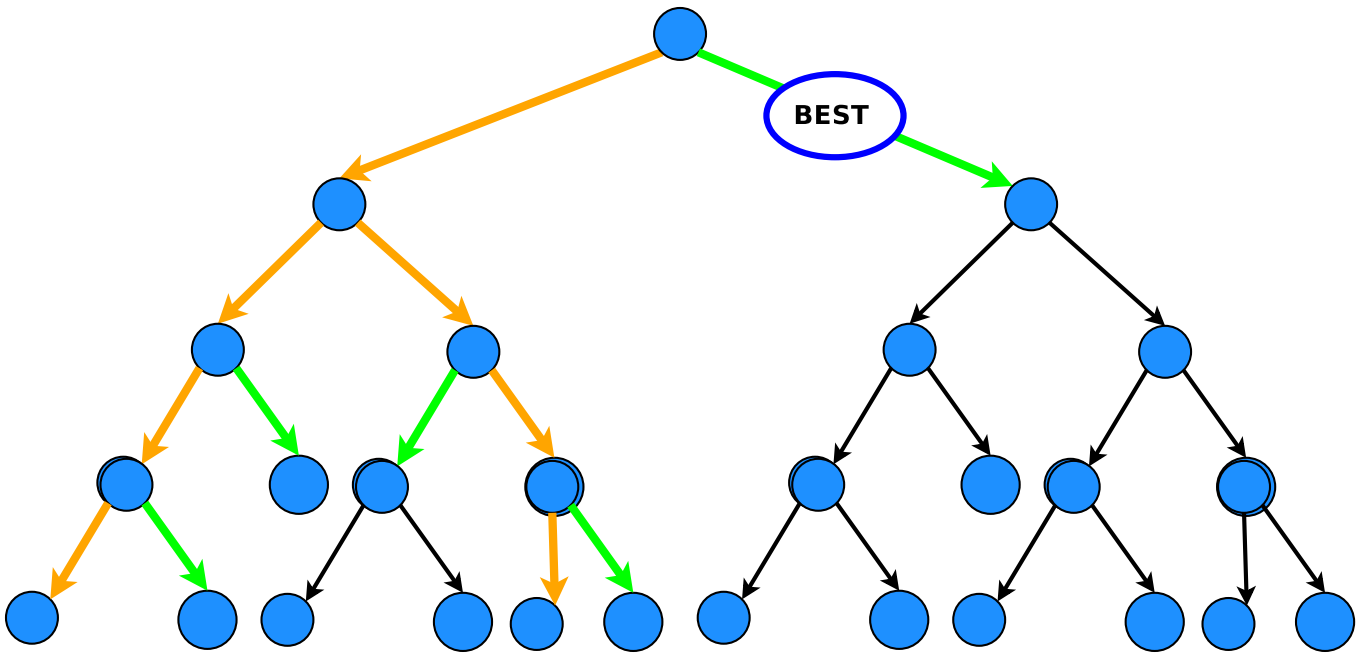
Parcours fitnex-guided (2)



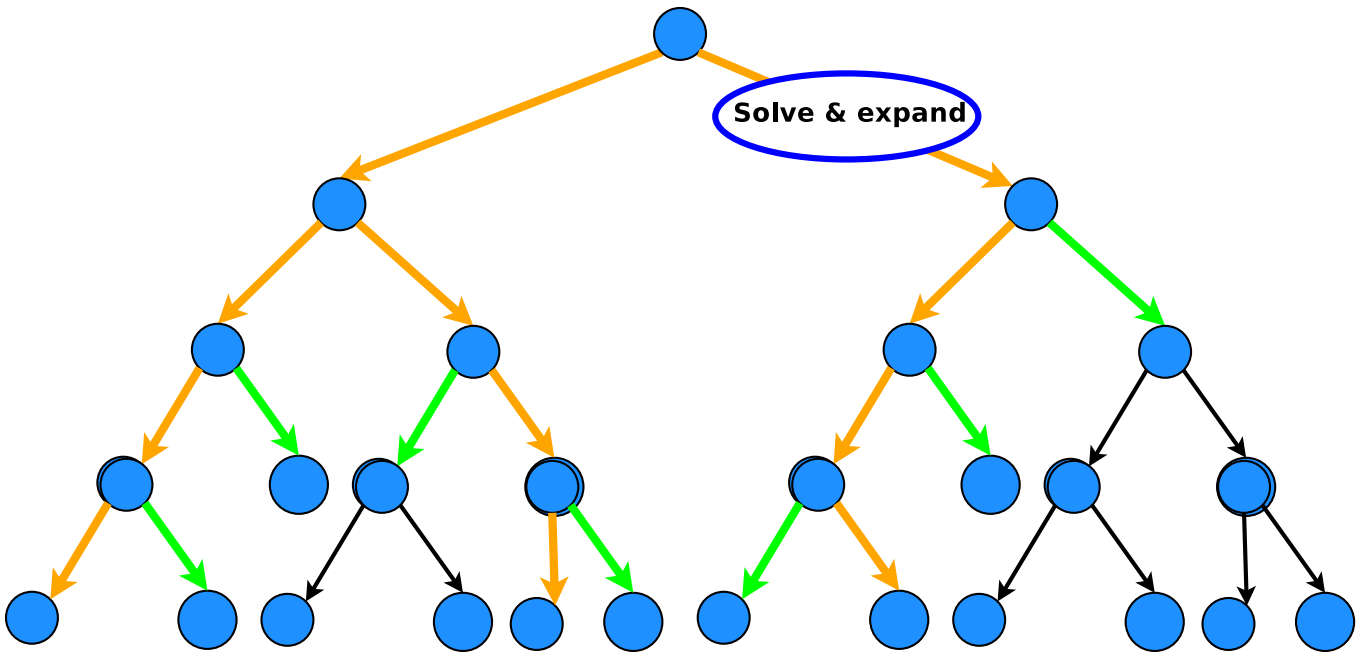
Parcours fitnex-guided (2)



Parcours fitnex-guided (2)



Parcours fitnex-guided (2)



Parcours fitnex-guided (3)

Nous supposons disponible les deux fonctions de base suivantes :

- `get_initial_value` : $\varepsilon \mapsto$ `inputData` : *fournit une donnée d'entrée concrète initiale (constante ou aléatoire).*
- `get_new_paths` : `inputData` $\mapsto$ `set<path>` : *lance une exécution concrète à partir de valeurs d'entrées, observe le chemin π suivi à l'exécution et retourne les préfixes actifs de π qui n'ont pas encore été collectés. Une implémentation réelle demande un type plus complexe, prenant en compte l'historique des préfixes actifs collectés jusque là.*

L'algorithme générique nécessite un type abstrait `VAL` de score, et les deux fonctions abstraites suivantes :

- `score` : `path` $\mapsto$ `VAL` : *évaluation d'un préfixe*
- `compare` : `VAL` $\times$ `VAL` $\mapsto \{<, =, >\}$: *comparaison à partir du score*

La fonction suivante est déduite facilement :

- `get_best` : `set<path>` $\mapsto$ `path` : *utilise* `compare`

Parcours fitnex-guided (4)

input : un programme P
output : RES : ensemble de couples (π - dt), tq pour chaque couple, P(dt) couvre π et l'ensemble des dt couvre toutes les branches faisables de P

```
 1: RES := ∅
 2: v0 := get_initial_value /* arbitrary initial concrete value */
 3: H := get_new_paths(v0) /* get all active prefixes from an execution */
 4: While still uncovered branches or paths do
 5:       π := get_best(H)
 6:       H := H\{π}
 7:       case SOLVE(SYMB(π)) of
 8:            | UNSAT → nop
 9:            | SAT(v) →
10:                 RES := RES ∪ {(π, v)} ;
11:                 H := H ∪ get_new_paths(v)
12:       end case
13: end while
14: return RES
```

Quelle fonction de Score ?

Par exemple, on peut baser le score sur :

- longueur du chemin, profondeur d'appel de la dernière instruction
- nb de fois où la dernière instruction a été couverte
- ...

Intérêts : permet d'intégrer facilement de nombreuses heuristiques de parcours de chemin

- chemin choisi aléatoirement
- dfs, bfs, dfs avec seuil
- dfs modulée par la profondeur d'appel et priorité aux branches non couvertes
- ...

Quelles fonctions de Score (2) ?

Quelques exemples d'heuristiques

minCallDepth-dfs

- les chemins dont le noeud final a la plus petite profondeur d'appel sont prioritaires
- puis dfs sur ces préfixes

hybrid dfs

- alterner k_1 test aléatoire puis k_2 étapes de dfs à partir du dernier chemin aléatoire

Best first

- le prochain chemin est celui ayant le gain le plus élevé
- gain, ex1 : nb de nouvelles instructions couvertes à coup sûr
- gain, ex2 : nb d'instructions successeurs non encore couvertes

Rappel : Optimisations de la DSE

Diminuer le nombre d'appels au solveur

- méthodes correctes d'élagage de chemins
- heuristiques : parcours de chemin plus malins que la DFS

Diminuer le coût moyen d'un appel au solveur

Optimisations avant l'appel au solveur

- simplification de formule (dont slicing)
- séparation des sous-formules indépendantes
- solveur "léger"
- système de cache
- réutilisation des solutions précédentes

Préprocessing : slicing

Enlever toutes les contraintes qui n'affectent pas le contrôle du chemin courant

- ex : expressions de calcul du résultat final
- à faire sur la formule, ou sur l'expression de chemin (plus simple)

Exemple de chemin :

```
y := y+1; x := a+b ; assume(y < 10) ; return x
```

- prédicat de chemin : $Y_1 = Y_0 + 1 \wedge X_1 = A_0 + B_0 \wedge Y_1 < 10$
- avec slicing : $Y_1 = Y_0 + 1 \wedge Y_1 < 10$

Préprocessing : slicing (2)

Remarque : si le langage de formules permet la définition de termes, alors le slicing se fait aussi bien niveau chemin que niveau formule

chemin `y := y+1; x := a+b ; assume(y < 10) ; return x`

- prédicat en avant :
  $\text{let } Y_1 := Y_0 + 1 \text{ in } \text{let } X_1 := A_0 + B_0 \text{ in } Y_1 < 10$
- le terme X_1 n'est jamais utilisé, on l'enlève
- formule simplifiée : $\text{let } Y_1 := Y_0 + 1 \text{ in } Y_1 < 10$

Préprocessing : simplification de formule

propagation de constantes : règles de la forme

- (élim de var) $X = 5$: remplacer X par 5 dans toute la formule, se souvenir de $X = 5$
- (élim de déf) let $X :=5$: remplacer X par 5 dans toute la formule, éliminer X
- (calcul de terme) $5 + 3$: faire le calcul, remplacer par 8
- (calcul partiel de terme)

  $X + 0$: remplacer par X

  $X \times 0$: remplacer par 0

  $X \times 1$: remplacer par X

  ...

propagation d'égalités

- si $X = Y$, garder seulement X ou seulement Y dans la formule
- on peut étendre aux opérateurs : $X = A + B \wedge Y = A + B$ alors $X = Y$

Préprocessing : simplification de formule (2)

unification des sous-termes identiques (introduction de défs)

- transformer $X = A + B + 1 \wedge Y = A + B + 2 \wedge B \leq Z$ en
  let $T := A + B$ in $X = T + 1 \wedge Y = T + 2 \wedge B \leq Z$

On peut normaliser les opérateurs AC pour trouver plus d'égalités

- AC : associatif - commutatif
- fonctionne pour propagation des égalités et unification de sous-termes identiques
- ex : réécrire les additions en ordonnant les opérandes selon ordre lexicographique
- ex : $X = A + B \wedge Y = B + A$, on normalise en $X = A + B \wedge Y = A + B$, on déduit que $X = Y$

Préprocessing : simplification de formule (3)

reconnaissance et utilisation des variables “proxy”

- sur une formule de type $X = A + B \wedge X + 3 \leq 100$
- X est un proxy pour A et B : A et B ne sont pas utilisés ailleurs, et $X = A + B$ est satisfaisable pour toute valeur de X
- dans ce cas : on enlève A et B de la formule (on résoud avec X), et on résoud à part $X = A + B$
- ici on peut même imposer directement par exemple $A = X$ et $B = 0$

Remarques

- ne marche pas si A ou B ont d'autres contraintes
- ce n'est pas un cas particulier de formules indépendantes

Séparation des sous-formules indépendantes

Séparation de formules : Soit $\vec{V}, \vec{V_1}, \vec{V_2}$ des ensembles de variables. Si $\varphi(\vec{V})$ peut se décomposer en $\varphi_1(\vec{V_1}) \wedge \varphi_2(\vec{V_2})$ et $\vec{V_1} \cap \vec{V_2} = \emptyset$, alors :

- si $\textsc{Solve}(\varphi_1)$ retourne UNSAT alors UNSAT
- sinon si $\textsc{Solve}(\varphi_2)$ retourne UNSAT alors UNSAT
- sinon SAT, et une solution = solution$(\vec{V_1}) \cup$ solution$(\vec{V_2})$

Rmq 1 : dès que $\textsc{Solve}$ a une complexité supérieure à linéaire, on gagne à séparer la formule

Rmq 2 : (pratique, cas UNSAT) dans quel ordre résoudre les φ_i ?

Rmq 3 : plus les formules sont petites, mieux le cache fonctionne

Utilisation d'un solveur léger, résolution en 2 étapes

On utilise déjà un solveur peu coûteux mais incomplet

- répond UNSAT ou MAYBE

Le solveur léger est lancé avant le solveur complet

- on gagne du temps si la formule est UNSAT

Comment obtenir le solveur léger ?

- le faire soi-même
- utiliser un solveur existant avec juste des théories simples
- un preprocessing élaboré peut servir de solveur léger

Cache sur les formules

Cache de formules : pour certaines φ' déjà résolues, on garde dans un cache $\mathcal{C} : \varphi' \mapsto$ SAT (et une solution) ou $\varphi' \mapsto$ UNSAT.

Soit φ à résoudre, à chaque appel du solveur on fait :

- si $\exists \varphi' \in \mathcal{C}$ tq $\varphi \Rightarrow \varphi'$ et φ' UNSAT, alors φ UNSAT
- sinon si $\exists \varphi' \in \mathcal{C}$ tq $\varphi' \Rightarrow \varphi$ et φ' SAT, alors φ SAT (et même solution)
- sinon $\textsc{Solve}(\varphi)$

Remarque : calculer $\Rightarrow$ est coûteux, on approxime $A \Rightarrow B$ par $B \preceq A$ (sous-terme syntaxique)

Réutilisation de solutions précédentes

- on résoud les préfixes de chemin de manière incrémentale
- donc on résoud une formule du type $\phi(...) \wedge \mathit{pred}(\vec{X})$, en connaissant déjà une solution de $\phi(...)$
- on peut réutiliser l'ancienne solution comme suit : toute la sous-formule de ϕ n'affectant pas $\vec{X}$ est enlevée, les variables concernées prennent leurs valeurs anciennement trouvées, et on résoud ce qui reste

Réutilisation de solutions précédentes (2)

* Supposons que l'on a déjà résolu
  $X = Y + 3 \wedge X \leq 5 \wedge B \geq 0$

solution trouvée : $X = 6, Y = 8, B = 0$

* pour résoudre
  $X = Y + 3 \wedge X \leq 5 \wedge B \geq 0 \wedge B + 12 \leq Z$

on réutilise les anciennes valeurs de X, Y (6 et 8), et on résoud seulement $B \geq 0 \wedge B + 12 \leq Z$

Réutilisation de solutions précédentes (3)

Remarque : peut être simulé en utilisant séparation des sous-formules indépendantes (cf après) et utilisation du cache

Solution tout de même intéressante

- plus facile à mettre en oeuvre qu'un cache de calcul (mais moins général)
- même si le cache est dispo, économise la recherche dans le cache

Remarque

L'efficacité de ces optims dépend du type de logiciel

- pour le code de type "parseur simple", les systèmes de cache et de séparation de sous-formules fonctionnent très bien

Les optimisations se combinent bien

- plus on simplifie, plus on peut séparer les formules
- plus les formules sont simples / petites, plus le cache fonctionne
- ...