Synthesis of Propositional Satisfiability Solvers

Douglas R. Smith,

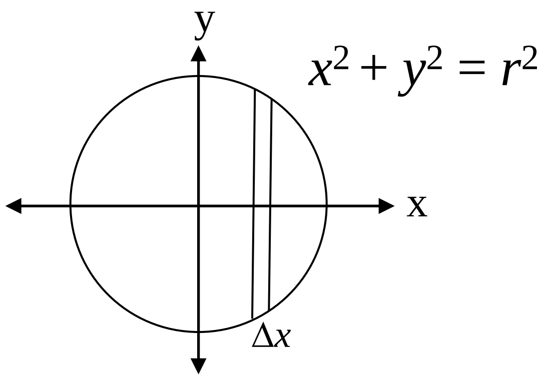
Stephen Westfold

Kestrel Institute

Palo Alto, California

Calculus

Problem: find the volume Vol(r) of a sphere of radius r



$$\text{Vol}(r) = \lim_{\Delta x \to 0} \sum_{-r}^{r} \pi\, y^2\, \Delta x$$

$$= \lim_{\Delta x \to 0} \sum_{-r}^{r} \pi\, (r^2 - x^2)\, \Delta x$$

$$= \int_{-r}^{r} \pi\, (r^2 - x^2)\, dx$$

$$= \int_{-r}^{r} \pi\, r^2\, dx \;-\; \int_{-r}^{r} x^2\, dx$$

= using $\int u^n\, du = u^{n+1}/(n+1) + C$

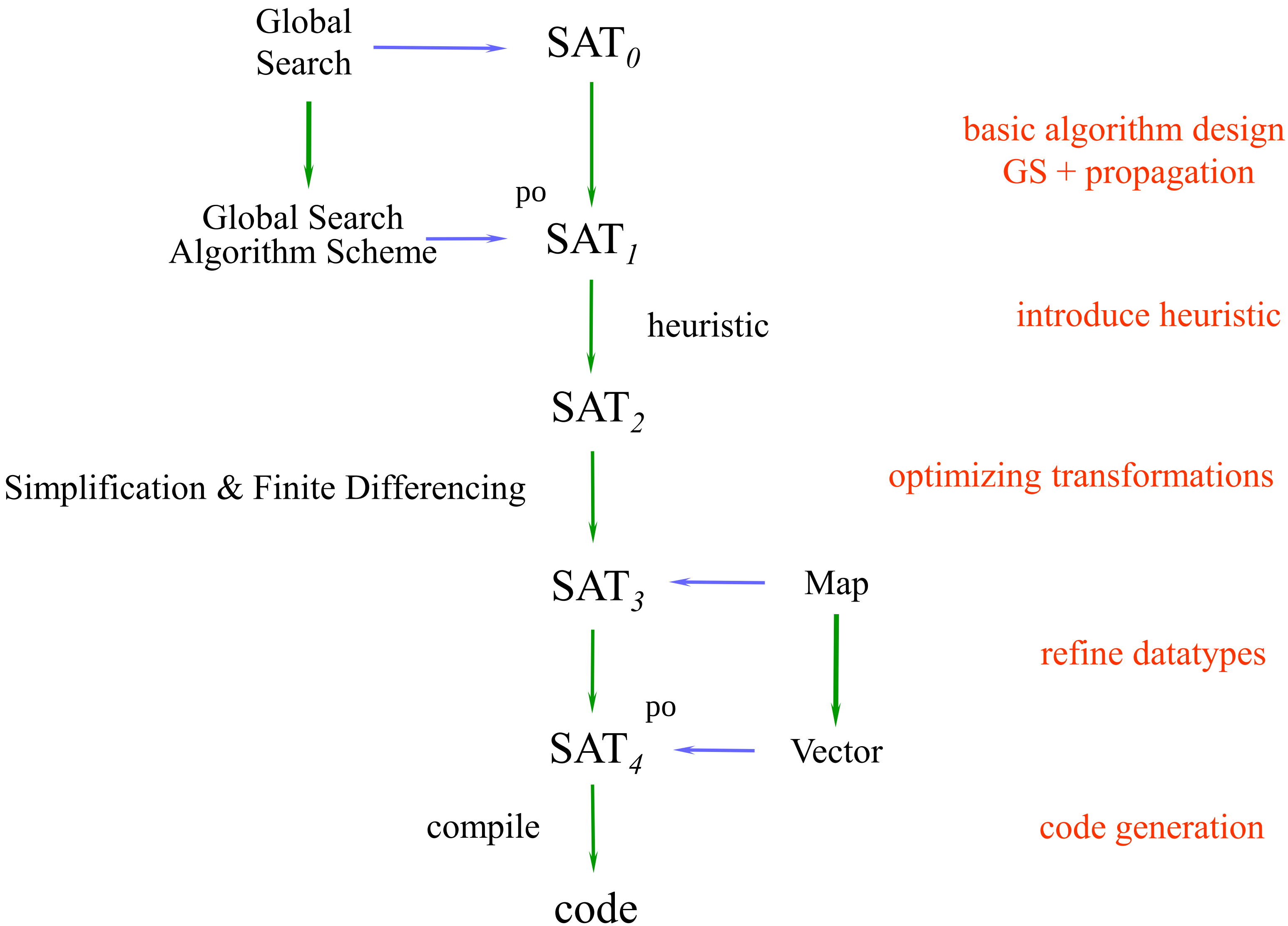
$$\pi\, r^2 x \Big]_{-r}^{r} - \; x^3 dx/3 \Big]_{-r}^{r}$$

$$= (4/3)\pi\, r^3$$

SAT Problem Specification

```
type Option A = | None | Some A

op eval : CNF * Valuation → Boolean
op SAT : CNF → Option Valuation

axiom SAT_spec is
  fa(p:CNF, v:Valuation)
     case SAT(p) of
       | Some v → eval(p, v)=true
       | None →    ~satisfiable(p)
```

Derivation Structure



Demo

Specifying the SAT Problem

Propositional Satisfiability (SAT)

Given a propositional formula,
prove that it is satisfiable/consistent
usually by constructing a model.

$$(A \lor \neg B \lor C) \land (B \lor \neg C) \land C$$

has several models: $\{A \mapsto \text{true},\ B \mapsto \text{true},\ C \mapsto \text{true}\}$
$\{A \mapsto \text{false},\ B \mapsto \text{true},\ C \mapsto \text{true}\}$

Note: The formula is a conjunction of disjuncts of literals.
This is called conjunctive normal form (CNF).

(A, -B, C), (B, -C), (C) 3 clauses

SAT Domain Theory

1. type Logic3 = | true | false | unk

   3-valued Kleene semilattice: unk $\sqsubseteq$ true, unk $\sqsubseteq$ false

2. type Valuation = map(Var, Logic3)

   operators:

   $m \sqsubseteq n \;=\; \forall(v)(v \in dom(m) \Rightarrow m(v) \sqsubseteq n(v))$

   domain(m) domain

   $m \oplus n$ composition (disjoint domains)

   laws:

   $m \sqsubseteq n \Rightarrow p \oplus m \sqsubseteq p \oplus n$

   $m \sqsubseteq n \Rightarrow \quad m \sqsubseteq p \oplus n$

   $m \sqsubseteq p \wedge n \sqsubseteq p = m \oplus n \sqsubseteq p$ if disjoint m,n

   $\bigwedge_i (m_i \sqsubseteq n) = (\bigoplus_i m_i) \sqsubseteq n$ if mutually disjoint m_i

SAT Domain Theory

```
type Variable      = Nat
type Valuation     = Map(Variable, Logic3)
type Literal       = | Pos Variable | Neg Variable
type Clause        = Set Literal
type CNF           = Set Clause
```

```
eval(p:CNF, vm:Valuation) : Logic3

   = if      ∀(cl)(cl∈p ⇒ evalC(cl,vm)=true)   then true
     else if ∃(cl)(cl∈p  ∧ evalC(cl,vm)=false)  then false
                                                else unk

evalC(cl:Clause, vm:Valuation) : Logic3 = …
evalL(lit:Literal, vm:Valuation) : Logic3 = …
```

SAT Domain Theory

simplify (p: CNF, vm: Valuation) : CNF

satisfiable(p:CNF) = $\exists$(vm) eval(p,vm)=true

satisfiable(p:CNF, pm:Valuation)

= $\exists$(vm)(pm $\sqsubseteq$ vm $\wedge$ eval(p,vm)=true)

Laws:

eval is monotone in its 2nd argument:

$m \sqsubseteq n \Rightarrow (\text{eval}(p,m) \sqsubseteq \text{eval}(p,n))$

satisfiable is antimonotone in its 2nd argument:

$m \sqsubseteq n \Rightarrow (\text{satisfiable}(p,m) \Leftarrow \text{satisfiable}(p,n))$

SAT Problem Specification

```
type Option A = | None | Some A

op eval : CNF * Valuation → Boolean
op SAT : CNF → Option Valuation

axiom SAT_spec is
  fa(p:CNF)
     case SAT(p) of
       | Some v → eval(p, v)=true
       | None →    ~satisfiable(p)
```

} Output Condition

Lattice-Based Quantifier Elimination Laws

for monotone $F: \langle A, \preccurlyeq \rangle \rightarrow \langle C, \sqcup, \sqcap \leq \rangle$

preorder lattice

$$\left(\bigsqcup_{a \preccurlyeq \hat{a}} F(a) \right) = F(\hat{a})$$

for monotone $F: \langle A, \preccurlyeq \rangle \rightarrow \langle \text{Boolean}, \vee, \wedge, \Rightarrow \rangle$

$$\exists(a)(a \preccurlyeq \hat{a} \wedge F(a)) = F(\hat{a})$$

for monotone $F: \langle \text{Boolean}, \Rightarrow \rangle \rightarrow \langle \text{Boolean}, \vee, \wedge, \Rightarrow \rangle$

$$\exists(a)\, F(a) = F(\text{true})$$

Quantifier Elimination Laws

$$F\colon \langle A, \preccurlyeq \rangle \to \langle C, \sqcup, \sqcap\ \leq \rangle$$

preorder — lattice

monotone F

$\bigsqcup_{a \preccurlyeq \hat{a}} F(a) = F(\hat{a})$	$\bigsqcap_{\check{a} \preccurlyeq a} F(a) = F(\check{a})$

antimonotone F

$\bigsqcup_{\check{a} \preccurlyeq a} F(a) = F(\check{a})$	$\bigsqcap_{a \preccurlyeq \hat{a}} F(a) = F(\hat{a})$

Quantifier Elimination Laws

specialization to predicates

$$F \colon \langle A, \preccurlyeq \rangle \to \langle \text{Boolean}, \vee, \wedge, \Rightarrow \rangle$$

preorder

monotone F

$\exists(a)(a \preccurlyeq \hat{a} \wedge F(a)) = F(\hat{a})$	$\forall(a)(\check{a} \preccurlyeq a \Rightarrow F(a)) = F(\check{a})$

antimonotone F

$\exists(a)(\check{a} \preccurlyeq a \wedge F(a)) = F(\check{a})$	$\forall(a)(a \preccurlyeq \hat{a} \Rightarrow F(a)) = F(\hat{a})$

Quantifier Elimination Laws

specialization to propositions

$F: \langle \text{Boolean}, \Rightarrow \rangle \to \langle \text{Boolean}, \vee, \wedge, \Rightarrow \rangle$

monotone F

$\exists(a)F(a) = F(true)$	$\forall(a)F(a) = F(false)$

antimonotone F

$\exists(a)F(a) = F(false)$	$\forall(a)F(a) = F(true)$

Lattice-Based Quantifier Change Laws

Lattice $\langle L, \sqcap, \sqcup, \leq \rangle$

$$\frac{g: T \rightarrow L, \quad S \subseteq T}{\bigsqcup_{x \in S} g(x) \;\leq\; \bigsqcup_{x \in T} g(x)}$$

$$\frac{g: T \rightarrow L, \quad S \subseteq T}{\bigsqcap_{x \in S} g(x) \;\geq\; \bigsqcap_{x \in T} g(x)}$$

e.g. Lattice $\langle \text{Boolean}, \wedge, \vee, \Rightarrow \rangle$

$$\frac{g: T \rightarrow L, \; S \subseteq T}{\exists(x{:}S)\; g(x) \Rightarrow \exists(x{:}T)\; g(x)}$$

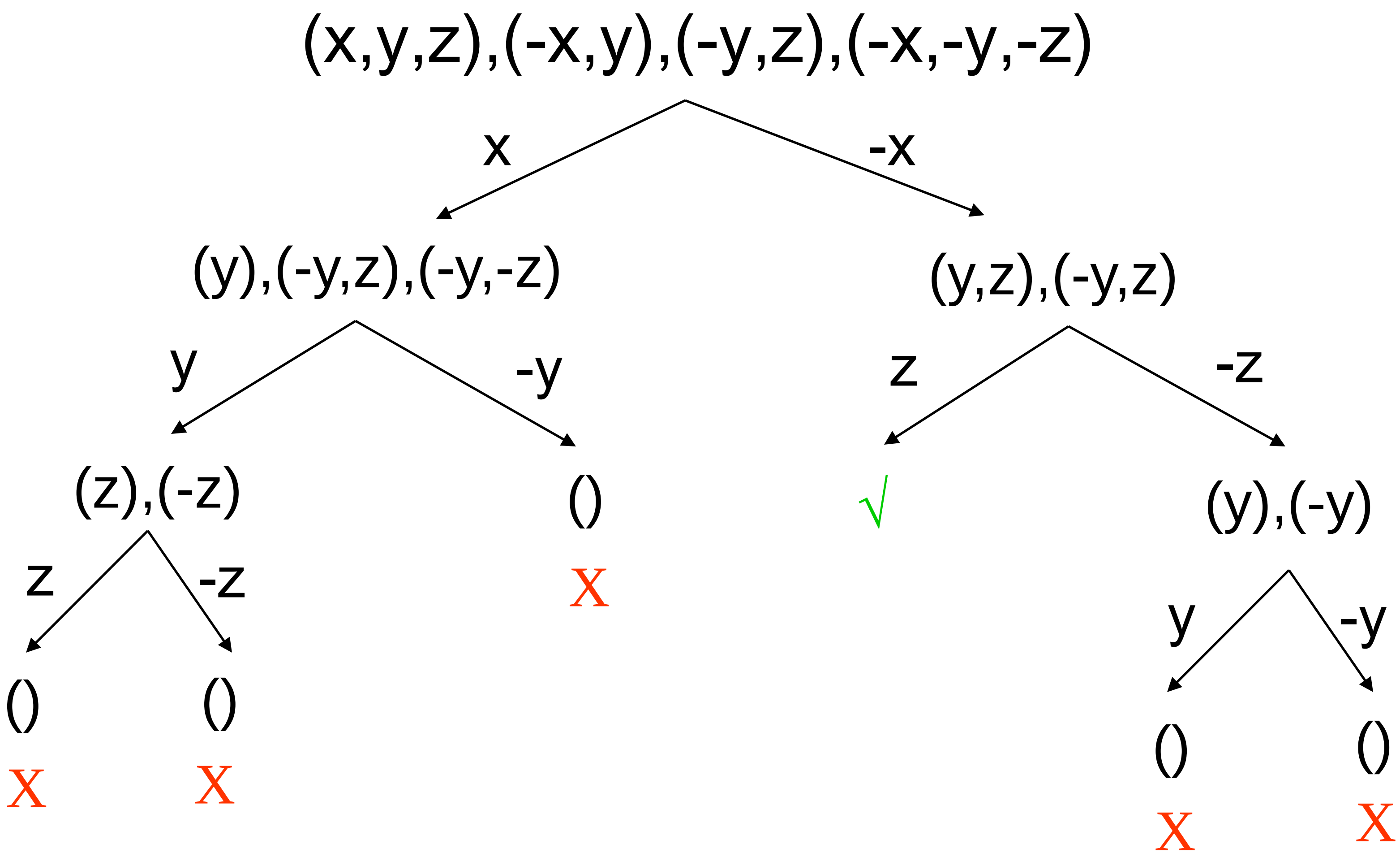
$$\frac{g: T \rightarrow L, \; S \subseteq T}{\forall(x{:}S)\; g(x) \Leftarrow \forall(x{:}T)\; g(x)}$$

Propositional Satisfiability (SAT)

```
type Option A = | None | Some A

op evalCNF : CNF * Valuation → Boolean
op SAT : CNF → Option Valuation

axiom SAT_spec is
  fa(p:CNF, v:Valuation)
     case SAT(p) of
       | Some v → eval(p, v)=true
       | None →    ~satisfiable(p)
```

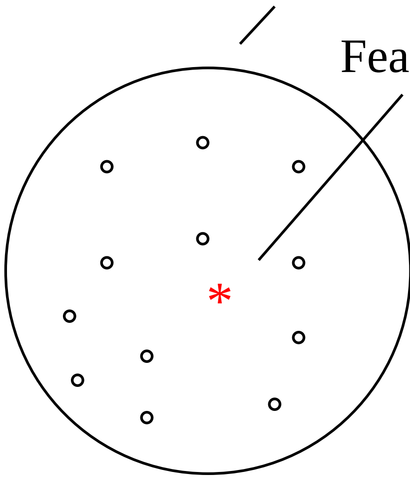
Deriving a SAT Algorithm

Basic SAT algorithm
(Davis-Putnam-Loggeman-Loveland)

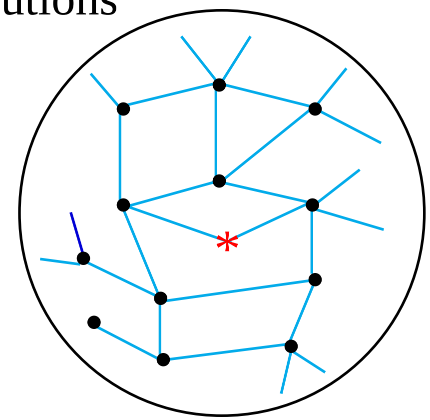


model: {x ↦ false, z ↦ true}

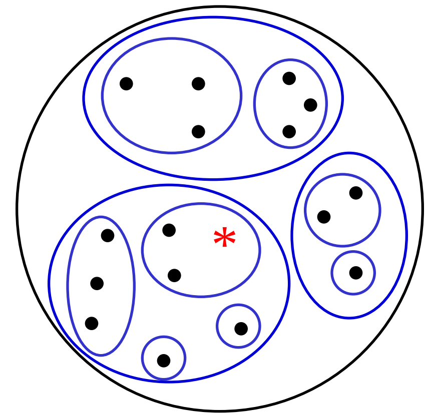
Problem Solving Structure



Solution space



Local structure
=
solution space
+
binary relation



Global structure
=
solution space
+
recursive partition

Global Search Theory

```
GlobalSearchTheory =  spec
  type D                                          input type
  type R                                          output type
  op O              : D * R → Boolean             output condition
  op mkInitial      : D → R
  op ⊑              : R * R → Boolean
  op Split          : D * R * R → Boolean
  op Subspaces      : D * R → List R
  op Extract        : D * R → Option R

   axiom ⟨R, ⊔, ⊑⟩ is a semilattice

   axiom  fa(x:D, z:R) mkInitial(x) ⊑ z

   axiom  fa(x:D, r:R, z:R)  r ⊑ z = ( Extract(x,r)=z  ∨  ex (s:R) (Split(x,r,s)  & s ⊑ z))

   axiom fa(x:D, r:R, s:R)   Split(x,r,s) = member(s, Subspaces(x,r))

end-spec
```

GS Scheme with Pruning + Propagation

```
F(x:D)  =  case propagate(x, mkInitial(x)) of
                  | none → none
                  | some r → GS(x,r)

GS(x:D, r:Rhat | Phi(x,r)) : option R
 =  case extract(x,r) of
        | some z → some z
        | none → GSAux(x, Subspaces(x,r))

GSAux(x:D, rs:List Rhat | fa(r:R)r∈rs⇒Phi(x,r)) : Option R
 = case rs of
      | nil → none
      | hd::tl → case propagate(x, hd) of
                       | none → GSAux(x,tl)
                       | some r → case GS(x,r) of
                                        | none → GSAux(x,tl)
                                        | some z → some z
```

theorem: fa(x:D) O(x, F(x))   ***provable from GS axioms***

Global Search Concepts → SAT concepts

Global Search Concept	SAT Concept
Output Condition O	Given proposition is satisfied by a valuation
Basic GS branching	Extend a partial model by alternate values of a variable
Pruning	Prune partial models that falsify a clause
Constraint Propagation: Necessary Propagation Consistent Refinement	Boolean Constraint Propagation Unit Rule (BCP) Pure Literal Rule
Conflict-Directed Backjumping	Conflict-Directed Backjumping
Learning	Learning

Propagation Mechanisms

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

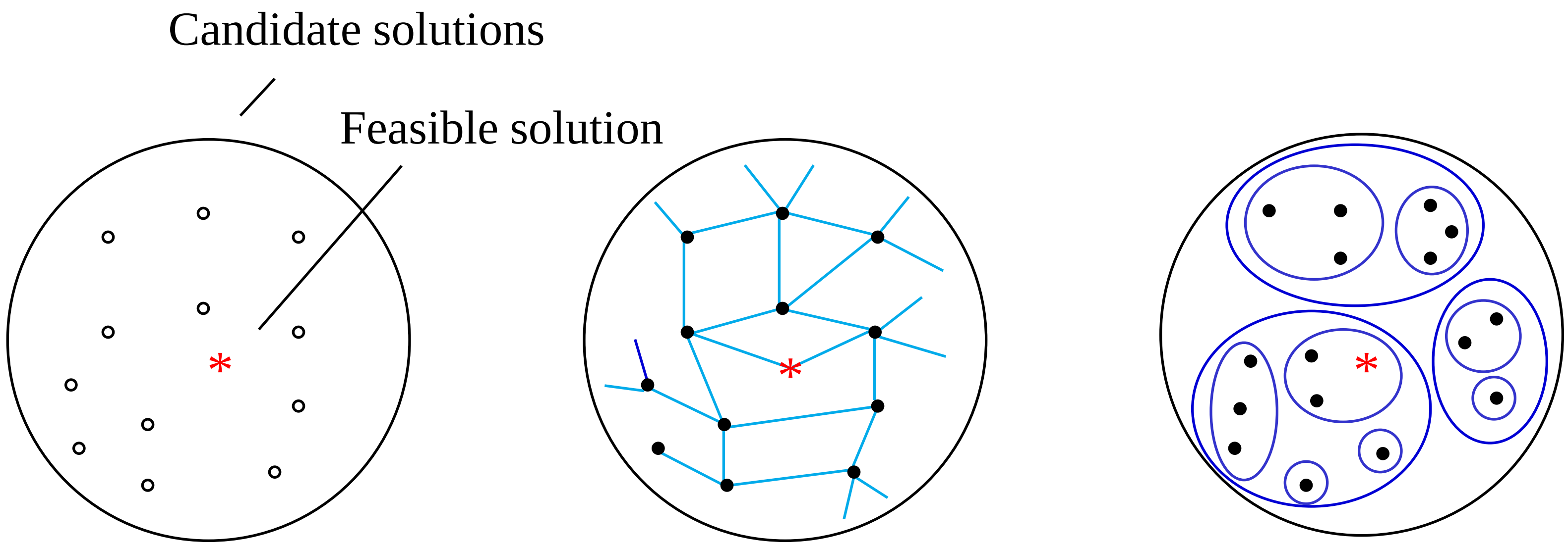
Unit-rule (Boolean Constraint Propagation):

if a clause only has one open (unassigned) literal
then its value is forced.

Pure Literal Rule:

if a literal has all-positive (all-negative) occurrences
then its value may be set to true (false).

Problem Solving Structure

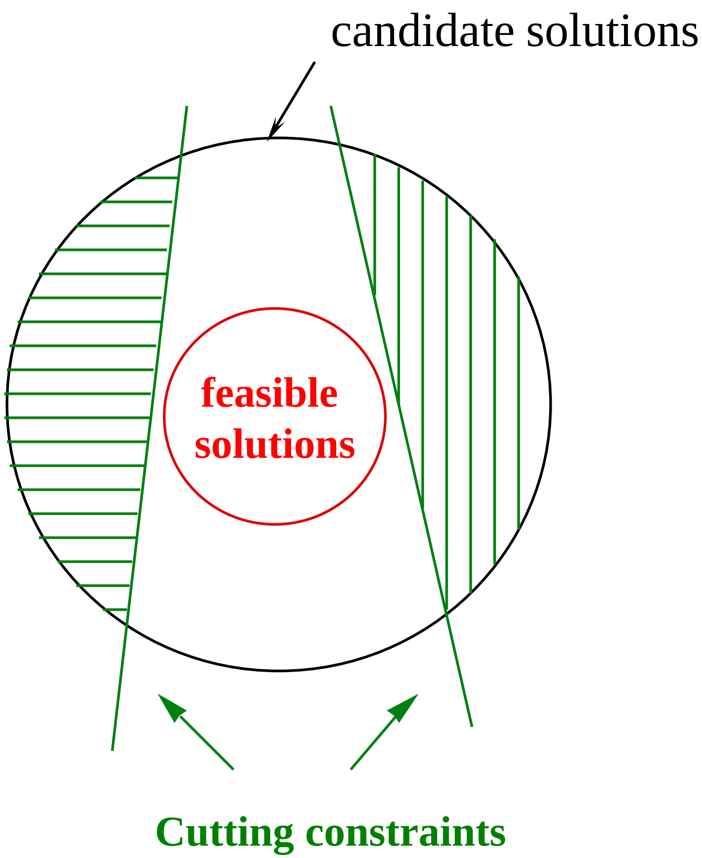


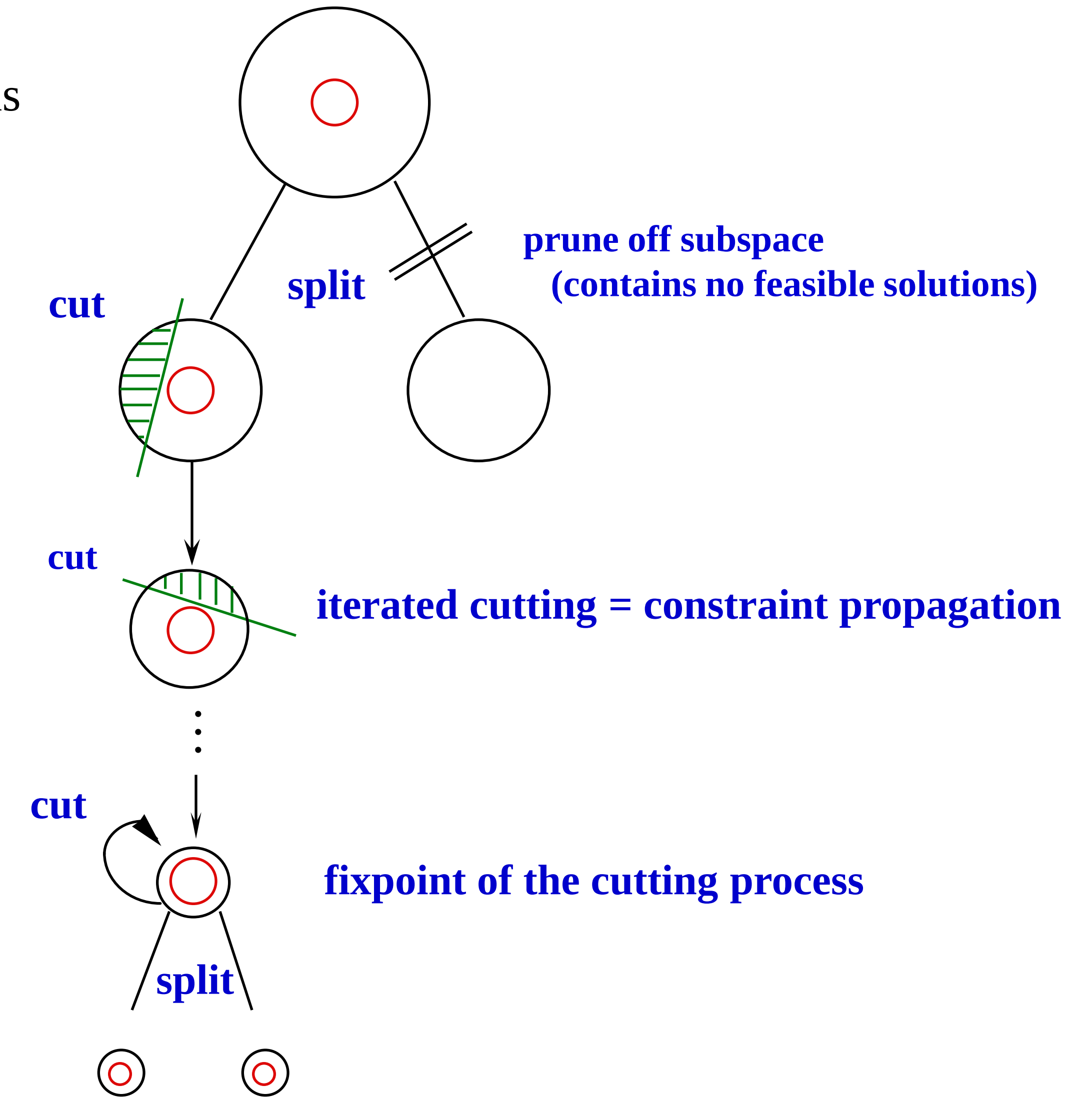
Solution space

Local structure
=
solution space
+
binary relation

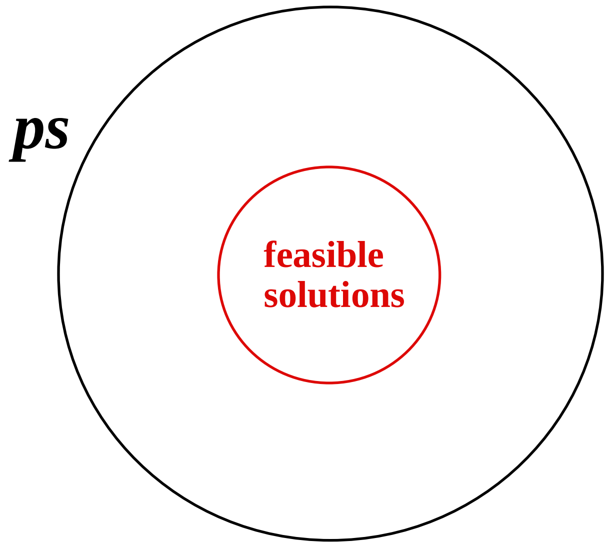
Global structure
=
solution space
+
recursive partition

Global Search Problem Solving





Deriving Pruning Constraints



Let *ps* be a partial solution that represents a set of candidate solutions

$$\exists(s)\ (ps \sqsubseteq s \wedge O(x,s)) \Rightarrow \Phi(x,ps)$$

ideal pruning test – decides if *ps* contains feasible solutions

derived pruning test – if false then *ps* contains no feasible solutions

want the *strongest* necessary test rooted in both conjuncts

Deriving a Pruning Test

$$\exists(s)\ (ps \sqsubseteq s\ \wedge O(x,s))\ \Rightarrow \Phi(x,ps)$$

$\exists(vm)$ ($pm \sqsubseteq vm \wedge eval(p,vm) = true$)

$\Rightarrow$ weakening = to $\sqsubseteq$

$\exists(vm)$ ($pm \sqsubseteq vm \wedge eval(p,vm) \sqsubseteq true$)

= Quantifier Elimination: $\exists(vm)$ ($pm \sqsubseteq vm \wedge F(vm)$) $= F(pm)$

if antimonotone $F(vm)$

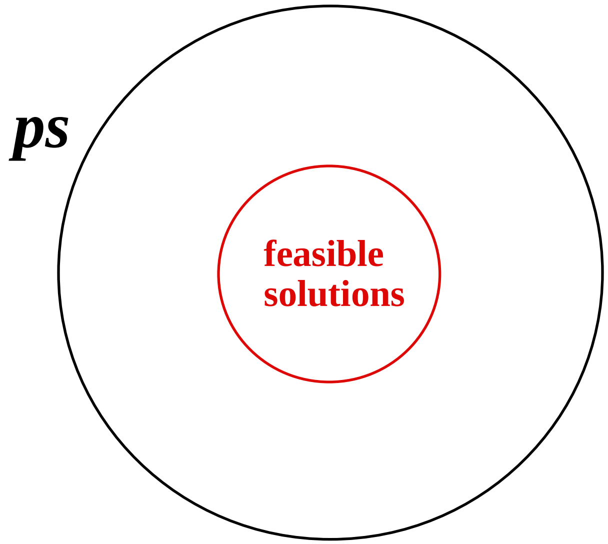
$eval(p, pm) \sqsubseteq true$

= unfolding def of eval (to get a CNF-specific pruning test)

$\forall(c)(c \in p \Rightarrow \exists(lit)(lit \in c \wedge evalLit(lit,pm) \sqsubseteq true))$.

$\neg\Phi(p,pm) = \exists(c)(c \in p \wedge \forall(lit)(lit \in c \Rightarrow evalLit(lit,pm) = false))$.

Deriving Pruning Constraints



Let *ps* be a partial solution that represents a set of candidate solutions

$$\exists(s)\ (ps \sqsubseteq s \wedge O(x,s)) \Rightarrow \Phi(x,ps)$$

ideal pruning test – decides if *ps* contains feasible solutions

derived pruning test – if false then *ps* contains no feasible solutions

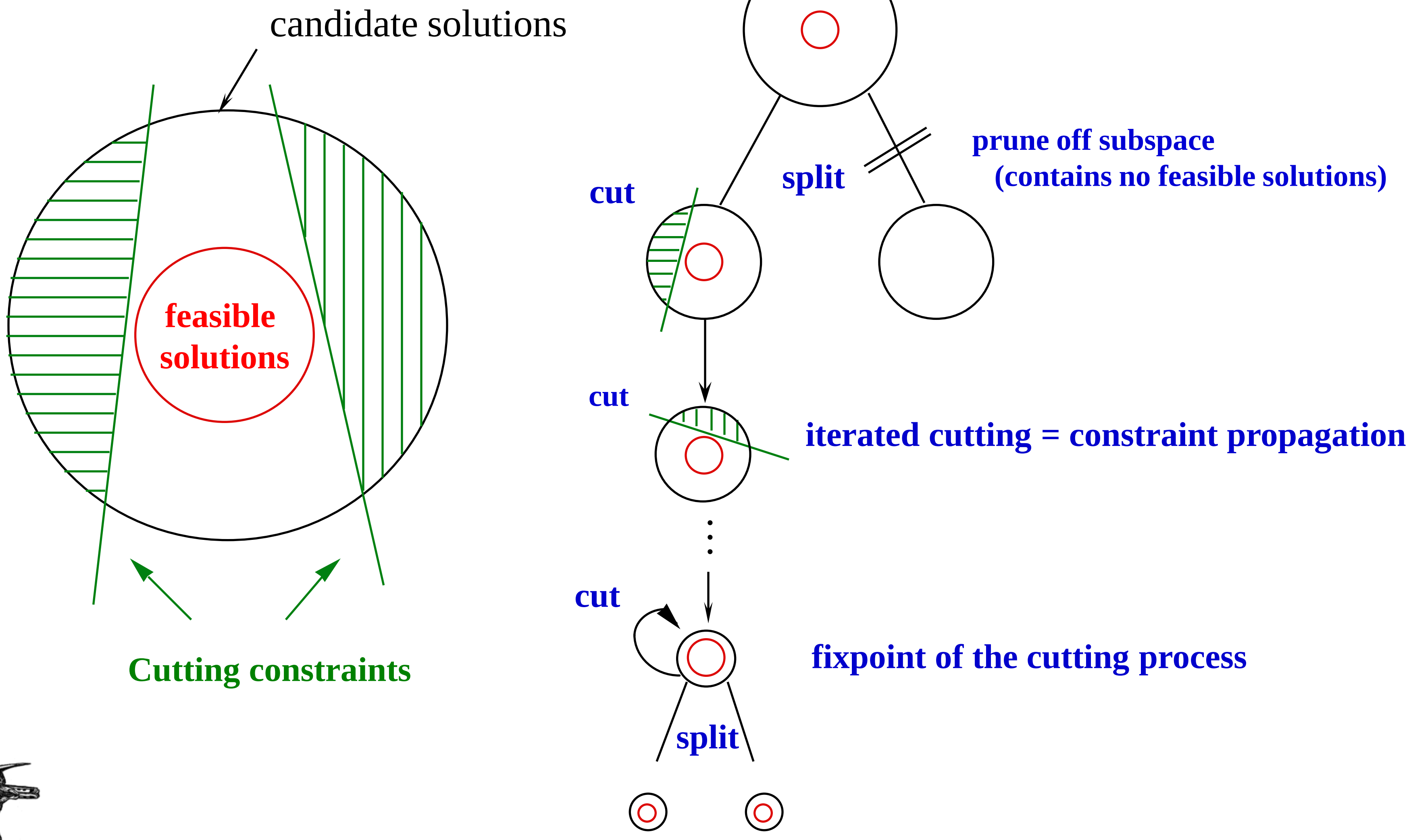
want the *strongest* necessary test rooted in both conjuncts

Algorithm Design Principles

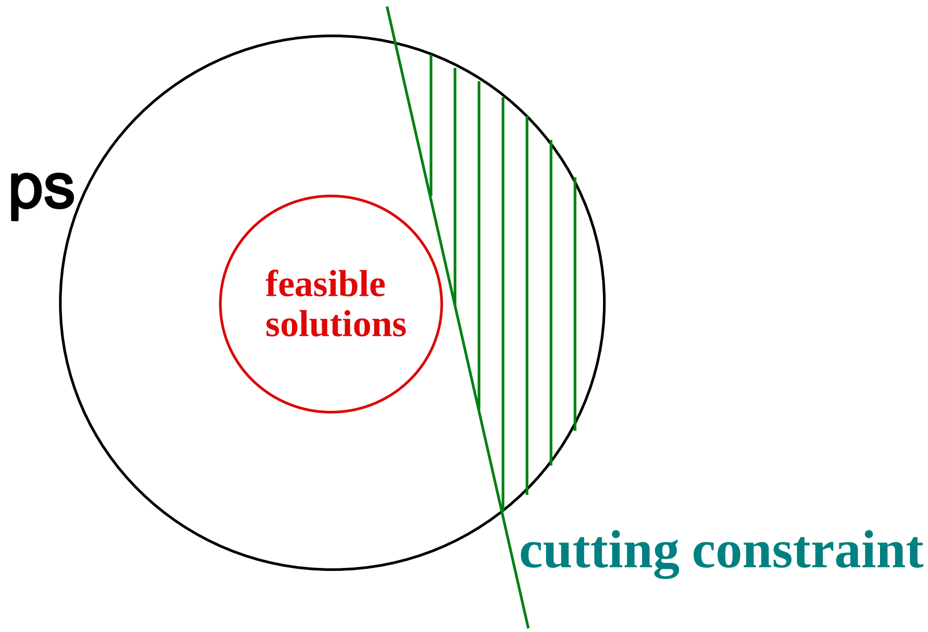
Prinicple 1. Characterize the ideal information needed
then derive an approximation to it

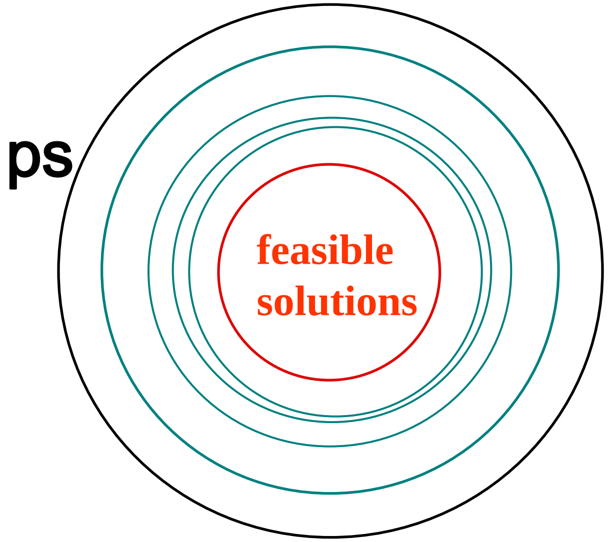
Principle 2. The closer semantically the approximation is to the ideal,
the better the performance of the resulting algorithm.

Global Search Problem Solving



Deriving Cutting Constraints





Let **ps** be a partial solution that represents a set of candidate solutions

$$ps \sqsubseteq f(x,ps) \;\wedge\; \forall(s)\,(ps \sqsubseteq s \wedge O(x,s) \Rightarrow f(x,ps) \sqsubseteq s)$$

Deriving (Boolean) Constraint Propagation: from cutting constraint to fixpoint iteration

Theorem:

If $ps \sqsubseteq f(x,ps) \wedge \forall(s)(ps \sqsubseteq s \wedge O(x,s) \Rightarrow f(x,ps) \sqsubseteq s)$ characterization of the propagation function f

then $\text{least } qs.\ ps \sqsubseteq qs \wedge f(x,qs) \sqsubseteq qs$ least fixpoint of f

$\sqsubseteq \bigsqcup qs \text{ s.t. } ps \sqsubseteq qs \wedge fa(s)(ps \sqsubseteq s \Rightarrow (qs \sqsubseteq s = O(x,s)))$

specification of the perfect cut

Deriving (Boolean) Constraint Propagation: from cutting constraint to fixpoint iteration

$\bigsqcup qs \text{ s.t. } ps \sqsubseteq qs \wedge fa(s)(ps \sqsubseteq s \Rightarrow (qs \sqsubseteq s = O(x,s)))$

specification of the tightest representation

$\sqsupseteq$ weakening formula, Quantifier change

$\bigsqcup qs \text{ s.t. } ps \sqsubseteq qs \wedge fa(s)(ps \sqsubseteq s \Rightarrow (qs \sqsubseteq s \Rightarrow O(x,s)))$

$\sqsupseteq$ weakening using $qs \sqsubseteq s \wedge O(x,s) \Rightarrow f(x,qs) \sqsubseteq s$, Quantifier change

$\bigsqcup qs \text{ s.t. } ps \sqsubseteq qs \wedge fa(s)(ps \sqsubseteq s \Rightarrow (qs \sqsubseteq s \Rightarrow f(x,qs) \sqsubseteq s))$

$=$ simplifying, using $qs \sqsubseteq s \Rightarrow ps \sqsubseteq s$

$\bigsqcup qs \text{ s.t. } ps \sqsubseteq qs \wedge fa(s)(qs \sqsubseteq s \Rightarrow f(x,qs) \sqsubseteq s)$

$=$ Quantifier elimination: $f(qs) \sqsubseteq s$ monotone in s

$\bigsqcup qs \text{ s.t. } ps \sqsubseteq qs \wedge f(x,qs) \sqsubseteq qs$

$\sqsupseteq$ $\text{least } qs.\ ps \sqsubseteq qs \wedge f(x,qs) \sqsubseteq qs$

Implement by iterating f to a fixpoint starting at ps

Deriving Boolean Constraint Propagation

$$ps \sqsubseteq f(x,ps) \land \forall(s)\ (ps \sqsubseteq s \land O(x,s) \Rightarrow f(x,ps) \sqsubseteq s\ )$$

$pm \sqsubseteq vm \land eval(p,vm)$

= def of map refinement

$pm \oplus qm = vm \land eval(p,vm)$

= project out vm

$eval(p, pm \oplus qm)$

= distribute eval over $\oplus$

$eval(\ simplify(p,pm),\ qm)$

= case analysis on Boolean variables;
let p' = simplify(p, pm), b:boolean

$$\bigwedge_{v \in domain(qm)} (\{v \mapsto b\} \sqsubseteq qm \lor \{v \mapsto \neg b\} \sqsubseteq qm) \land eval(p', qm)$$

Deriving Boolean Constraint Propagation

$$ps \sqsubseteq f(x,ps) \wedge \forall(s)\ (ps \sqsubseteq s \wedge O(x,s) \Rightarrow f(x,ps) \sqsubseteq s\ )$$

$$\bigwedge_{v \in domain(qm)} (\{v \mapsto b\} \sqsubseteq qm \vee \{v \mapsto \neg b\} \sqsubseteq qm) \wedge eval(p', qm)$$

= replace disjunction by implication

$$\bigwedge_{v \in domain(qm)} (\neg\{v \mapsto b\} \sqsubseteq qm \Rightarrow \{v \mapsto \neg b\} \sqsubseteq qm) \wedge eval(p', qm)$$

$\Rightarrow$ antimonotonicity of satisfiable:

$$m \sqsubseteq n \Rightarrow (satisfiable(p,m) \Leftarrow satisfiable(p,n))$$

$$\bigwedge_{v \in domain(qm)} (\neg(satisfiable(p', \{v \mapsto b\}\ ) \Leftarrow satisfiable(p',qm)) \Rightarrow \{v \mapsto \neg b\} \sqsubseteq qm) \wedge eval(p', qm)$$

Deriving Boolean Constraint Propagation

$$ps \sqsubseteq f(x,ps) \wedge \forall(s)\ (ps \sqsubseteq s \wedge O(x,s) \Rightarrow f(x,ps) \sqsubseteq s\ )$$

$$\bigwedge_{v \in domain(qm)} (\neg(satisfiable(p',\{v \mapsto b\}) \Leftarrow satisfiable(p',qm)) \Rightarrow \{v \mapsto \neg b\} \sqsubseteq qm) \wedge eval(p', qm)$$

$= \quad eval(p',qm) \Rightarrow satisfiable(p',qm)$

$$\bigwedge_{v \in domain(qm)} (\neg satisfiable(p',\{v \mapsto b\}) \Rightarrow \{v \mapsto \neg b\} \sqsubseteq qm) \wedge eval(p', qm)$$

$\Rightarrow \quad domain(qm)=Vars \backslash domain(pm)$, unfold p'

$$\bigwedge_{\substack{v \in Vars \backslash domain(pm) \\ \neg satisfiable(p,\, pm \oplus \{v \mapsto b\})}} \{v \mapsto \neg b\} \sqsubseteq qm$$

Deriving Boolean Constraint Propagation

$$ps \sqsubseteq f(x,ps) \wedge \forall(s)\ (ps \sqsubseteq s \wedge O(x,s) \Rightarrow\ f(x,ps) \sqsubseteq s\ )$$

$$\bigwedge_{\substack{v \in Vars \setminus domain(pm) \\ \neg satisfiable(p,\ pm \oplus \{v \mapsto b\})}} \{v \mapsto \neg b\} \sqsubseteq qm$$

$=$ using the law: $\bigwedge (m_i \sqsubseteq n) = (\bigoplus m_i) \sqsubseteq n$

$$\left(\bigoplus_{\substack{v \in Vars \setminus domain(pm) \\ \neg satisfiable(p,\ pm \oplus \{v \mapsto b\})}} \{v \mapsto \neg b\} \right) \sqsubseteq qm$$

$\Rightarrow$ precomposing with pm

$$pm \oplus \left(\bigoplus_{\substack{v \in Vars \setminus domain(pm) \\ \neg satisfiable(p,\ pm \oplus \{v \mapsto b\})}} \{v \mapsto \neg b\} \right) \sqsubseteq pm \oplus qm = vm$$

$f(p,pm)$

Unit Rule and other forms of Propagation

$$pm \oplus \bigoplus_{\substack{v \in Vars \setminus domain(pm) \\ \neg satisfiable(p,\ pm \oplus \{v \mapsto b\})}} \{v \mapsto \neg b\}$$

Calculate a sufficient condition of $\neg satisfiable(p,\ pm \oplus \{v \mapsto b\})$:

$\neg satisfiable(p,\ pm \oplus \{v \mapsto b\})$

$\Leftarrow$ $\neg(eval(p,\ pm \oplus \{v \mapsto b\}) = true)$

= using CNF def of eval, evalC

$\exists\ (c{:}Clause)(c \in p \wedge \forall(lit)(lit \in c \Rightarrow evalL(lit,\ pm \oplus \{v \mapsto b\}) = false))$

= distributing evalL over $\oplus$

$\exists\ (c{:}Clause)(c \in p \wedge \forall(lit)(lit \in c \Rightarrow$ if $var(lit)=v$

then $evalL(lit,\ \{v \mapsto b\}) = false$

else $evalL(lit,\ pm) = false))$

other sufficient conditions lead to BCP2 – speculative BCP

Boolean Constraint Propagation Code

$$f(p,pm) \;=\; pm \oplus \bigoplus_{\substack{v \in \text{domain}(pm) \\ \neg \text{satisfiable}(p,\, pm \oplus \{v \mapsto b\})}} \{v \mapsto \neg b\}$$

code: propagate(p,pm)

```
propagate(p:CNF, m: Valuation): Valuation =
   if m = f(p,m)
   then m
   else propagate(p, f(p,m)).
```

use Finite Differencing to reduce the cost of this expensive expression

Conflict Analysis and Learning

A GS path fails when $\neg\Phi(x,ps)$

The decisions leading up to the failure include

- split decisions: sds
- propagation refinements: prs

Conflict Analysis: a sufficient condition on the failure:

$$\theta(x,ps,sds,prs) \Rightarrow \neg\Phi(x,ps)$$

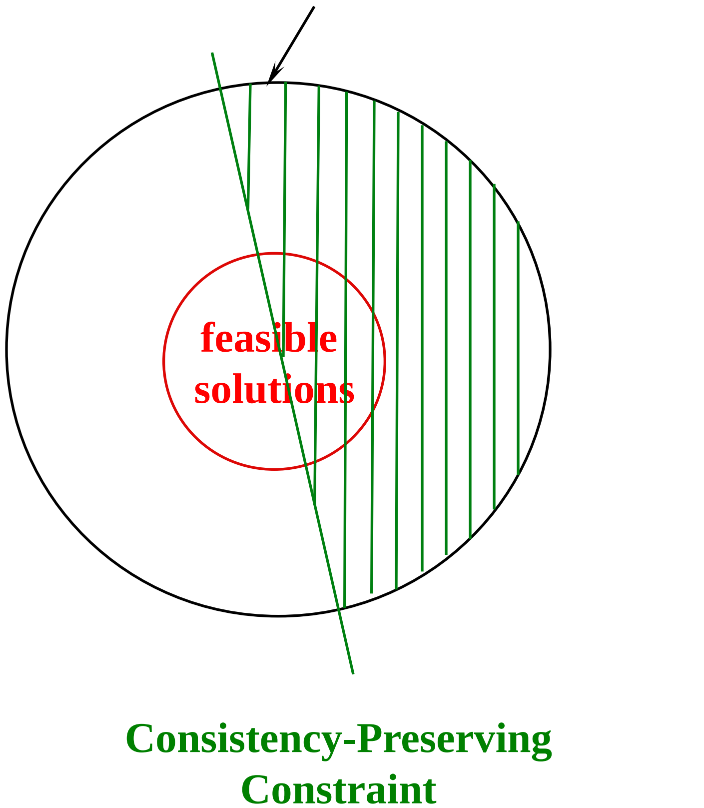
or

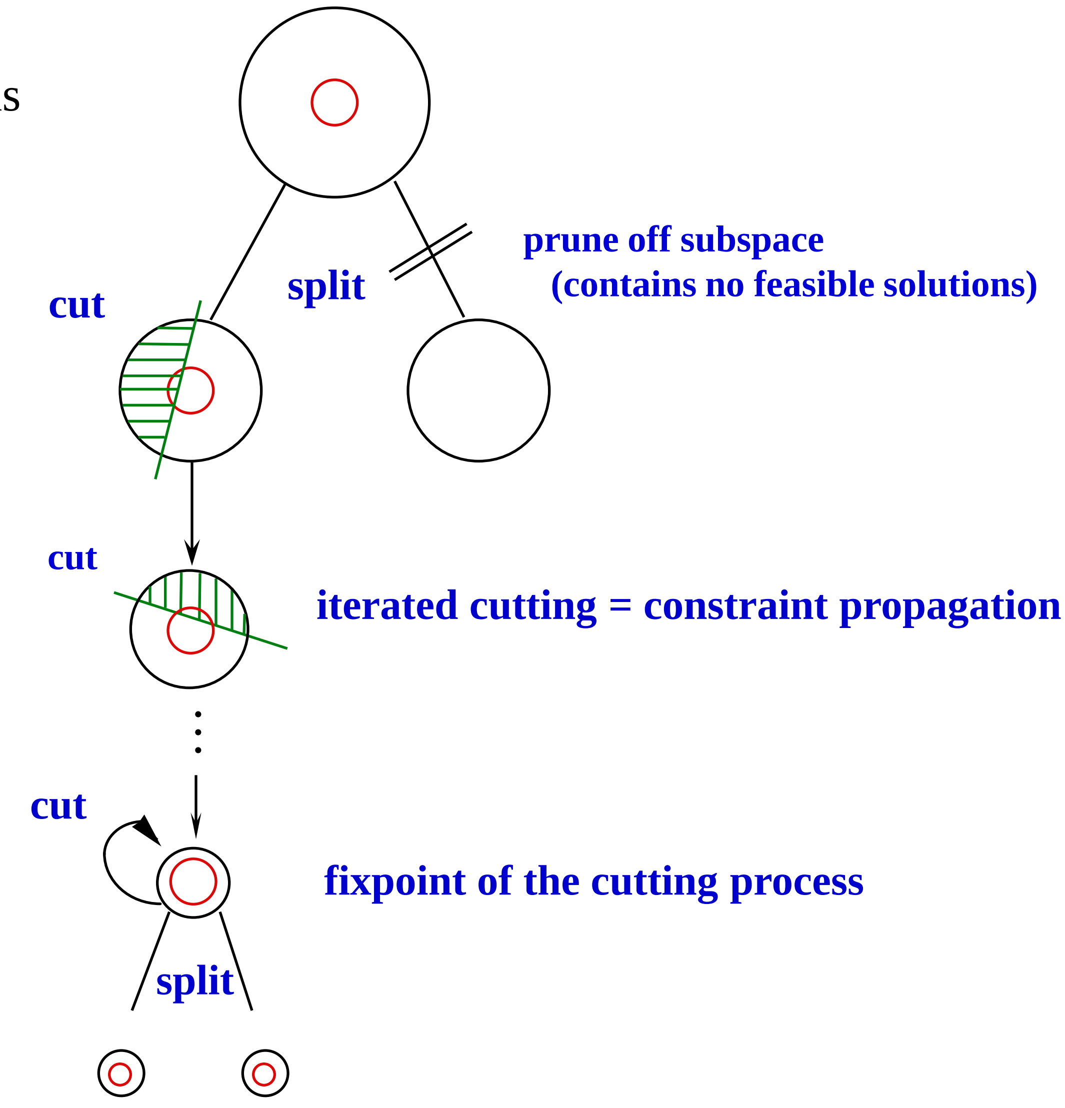
$$\exists(s)(ps \sqsubseteq s \;\wedge O(x,s)) \;\Rightarrow \Phi(x,ps) \Rightarrow \neg\theta(x,ps,sds,prs)$$

Learning: Incorporating $\neg\theta$ as a propagation constraint:

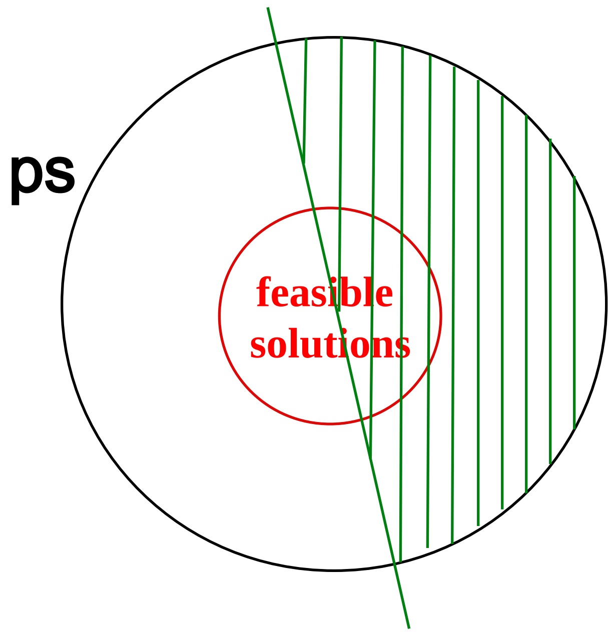
- easy in SAT since $\neg\theta$ is a clause
- in general GS?

Global Search Problem Solving





Consistency-Preserving Refinement



Let ps be a partial solution that represents a set of candidate solutions

$$\exists(s)\ (ps \sqsubseteq s \wedge O(x,s)) \;=\; \exists(s)\ (f(x,ps) \sqsubseteq s \wedge O(x,s)) \wedge ps \sqsubseteq f(x,ps)$$

ps has a feasible solution iff **f(x,ps)** has a feasible solution

Pure Literal Rule: A Consistency-Preserving Refinement

$$\exists(s)\ (ps \sqsubseteq s \wedge O(x,s)) \ = \ \exists(s)\ (f(x,ps) \sqsubseteq s \wedge O(x,s)) \wedge ps \sqsubseteq f(x,ps)$$

$\exists(vm)(\ pm \sqsubseteq vm \wedge eval(p,vm)=true)$

= by definition

satisfiable(p, pm)

= a digression is in order

Pure Literal Rule: A Consistency-Preserving Refinement

Want to apply the following Quantifier Elimination law about functions:

$$\exists(a)F(a) = F(\text{true}) \quad \text{for monotone } F$$

but we need to apply it to the *CNF representation* of a function:

$$\forall(v)(\text{monotone}(p,v) \Rightarrow \text{satisfiable}(p) = \text{satisfiable}(p,\{v \mapsto \text{true}\}))$$

$$\text{satisfiable}(p,m) = \text{satisfiable}(p,\ m \oplus \bigoplus_{\text{monotone}(p,v)} \{v \mapsto \text{true}\})$$

$$\text{satisfiable}(p,m) = \text{satisfiable}(p,\ m \oplus \bigoplus_{\text{antimonotone}(p,v)} \{v \mapsto \text{false}\})$$

Pure Literal Rule: A Consistency-Preserving Refinement

$$\exists(s)\ (ps \sqsubseteq s \wedge O(x,s)) \ = \ \exists(s)\ (f(x,ps) \sqsubseteq s \wedge O(x,s)) \wedge ps \sqsubseteq f(x,ps)$$

$\exists(vm)(\ pm \sqsubseteq vm \wedge eval(p,vm))$

= by definition

satisfiable(p, pm)

= CNF version of Quantifier Elimination laws

$$satisfiable(p,\ pm \oplus \bigoplus_{monotone(p,v)} \{v \mapsto true\} \oplus \bigoplus_{antimonotone(p,v)} \{v \mapsto false\})$$

= unfolding the def of satisfiable

$\exists(vm)(\ f(p,pm) \sqsubseteq vm \wedge eval(p,vm))$

$$\text{where } f(p,pm) = pm \oplus \bigoplus_{monotone(p,v)} \{v \mapsto true\} \oplus \bigoplus_{antimonotone(p,v)} \{v \mapsto false\})$$

Summary: Propagation Code

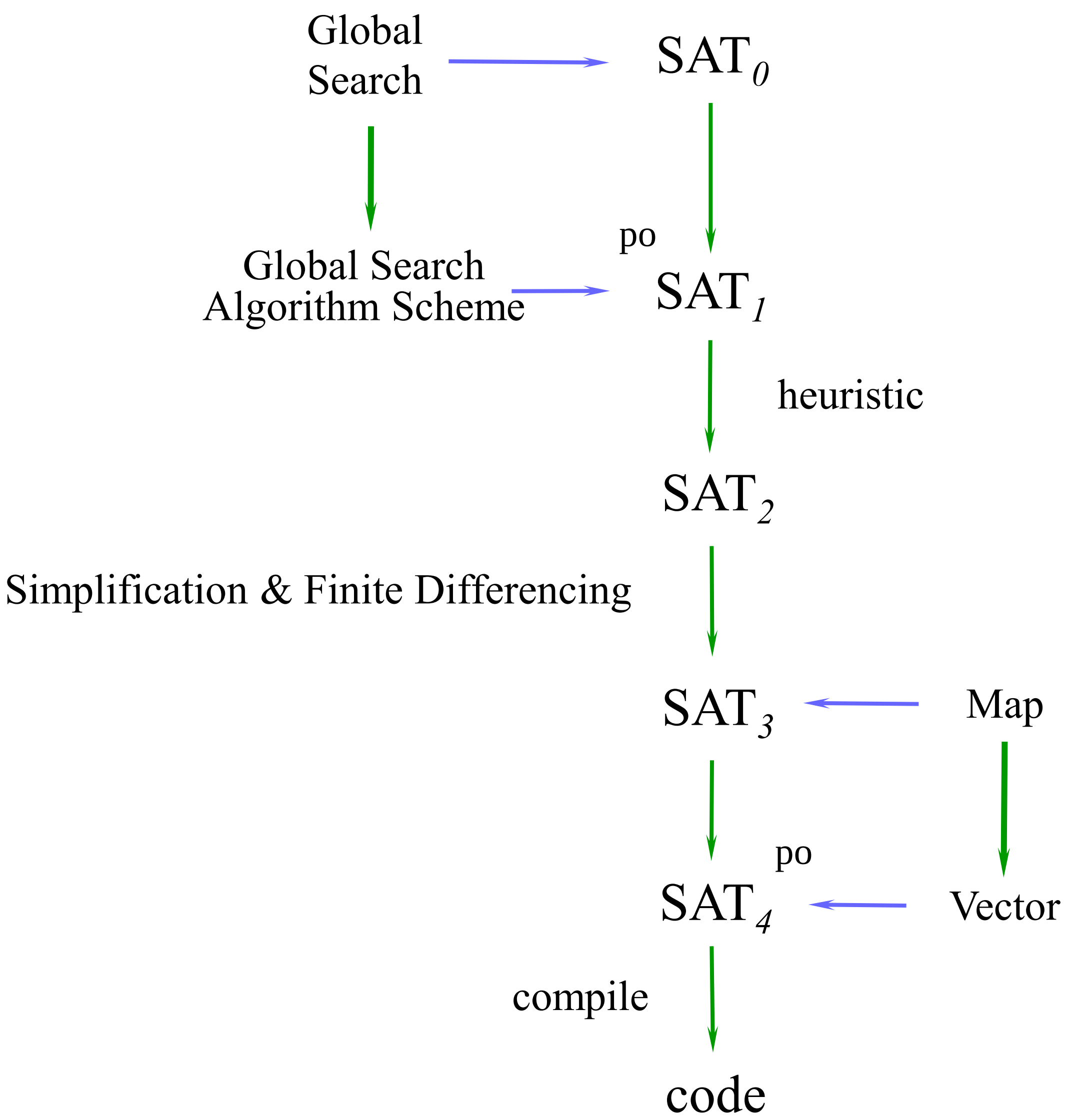
$$ur(p,pm) \;=\; pm \oplus \bigoplus_{\substack{v \in Vars \backslash domain(pm) \\ \neg satisfiable(p,\, pm \oplus \{v \mapsto b\})}} \{v \mapsto \neg b\}$$

$$plr(p,pm) = pm \oplus \bigoplus_{monotone(p,v)} \{v \mapsto true\} \oplus \bigoplus_{antimonotone(p,v)} \{v \mapsto false\})$$

```
propagate(p:CNF, m: Valuation): Valuation =
   if m = plr(ur(p,m))
   then m
   else propagate(p, plr(ur(m))).
```

use Finite Differencing to reduce the cost of this expensive expression

Derivation Structure



basic algorithm design: GS + propagation

introduce heuristic

optimizing transformations

refine datatypes

code generation

Heuristics: Variable Choice and Value Ordering

Maximum Occurrences of Minimum Size (MOMs):

Let F(v) = number of occurrences of v in the shortest open clauses; branch on v such that F(v) and F($\neg v$) are maximal

Dynamic Largest Individual Sum:

For a given variable v:

- $C_{v,p}$ = # unresolved clauses in which v appears positively
- $C_{v,n}$ = # unresolved clauses in which v appears negatively
- Let v be the literal for which $C_{v,p}$ is maximal
- Let w be the literal for which $C_{w,n}$ is maximal
- If $C_{v,p} > C_{w,n}$ choose v and assign it TRUE
- Otherwise choose w and assign it FALSE

Derivation of Heuristics?

idea: formally specify the ideal situation,
and then derive a tractably computable approximation

Goal: choose v to minimize the cost of deciding satisfiability of p

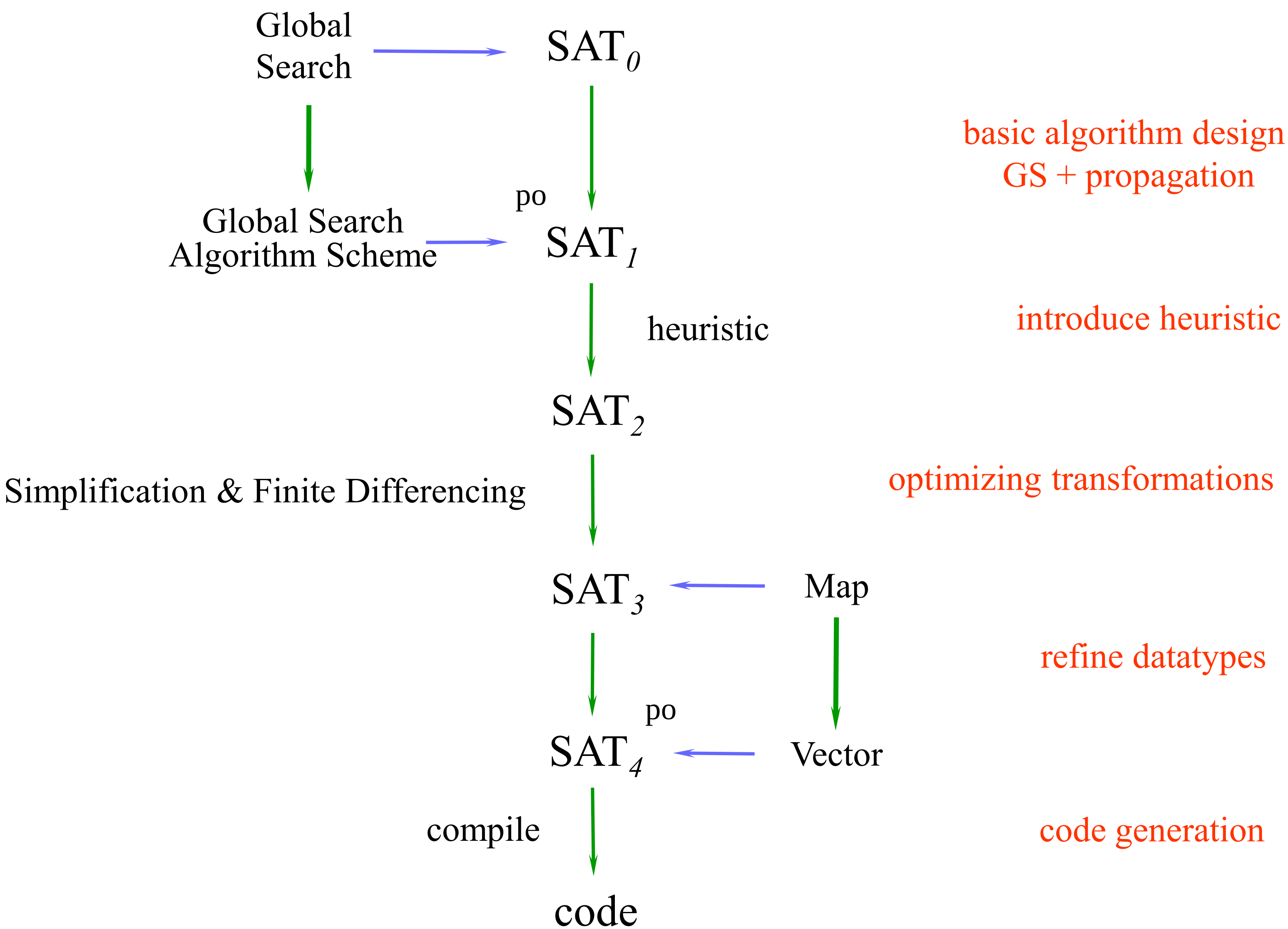
≈ let |p| = number of unassigned vars in p, assume cost of subtree ≈ $C^{|p|}$

$$\min_v \; C^{|propagate(simplify(p,\{v \mapsto true\}))|} + C^{|propagate(simplify(p,\{v \mapsto false\}))|}$$

≈ minimize each term by maximizing the impact of propagation;
minimize the sum by attempting to make the exponents roughly equal

$$\max_v \; occurrences(v, p) \; * \; occurrences(\neg v, p)$$

Derivation Structure



Finite Differencing

. . .
F(*nil*)
. . .

def F(*x*:*List*)=
. . .
length(*x*)
. . .
F(cons(a,x))
. . .

abstract on *length*(*x*) → then simplify

. . .
F´(*nil*, *length*(*nil*))   [*length*(*nil*) → 0]
. . .

def F´(*x*, *c* | *c*=*length*(*x*)) =
. . .
length(*x*) → *c*
. . .
F´(*cons*(*a*,*x*), *length*(*cons*(*a*,*x*)))
. . .

distribute length over cons

length(*cons*(*a*,*x*)) → *1*+ *length*(*x*) → *1* + *c*

Summary: Propagation Code

$$ur(p,pm) = pm \oplus \bigoplus_{\substack{v \in Vars \backslash domain(pm) \\ \neg satisfiable(p,\, pm \oplus \{v \mapsto b\})}} \{v \mapsto \neg b\}$$

$$plr(p,pm) = pm \oplus \bigoplus_{monotone(p,v)} \{v \mapsto true\} \oplus \bigoplus_{antimonotone(p,v)} \{v \mapsto false\})$$

```
propagate(p:CNF, m: Valuation): Valuation =
    if m = plr(ur(p,m))
    then m
    else propagate(p, plr(ur(m))).
```

Finite Differencing for Open Variables

Maintain current set of open variables: openVars = Vars\domain(pm)

0. Initialization

Context: st = mkInitialState(p)
Simplify: openVars = Vars\domain(st.varVals)

= substituting st

openVars = Vars\domain(mkInitialState(p).varVals)

= unfold

openVars = Vars\domain({})

= simplify

openVars = Vars.

Finite Differencing for Open Variables

1. UpdateState

Context: st' = updateState(st,var,val)
& openVars = Vars\domain(st.varVals)
Simplify: openVars' = Vars\domain(st'.varVals)

= substituting st'

Vars\domain(updateState(st,var,val).varVals)

= unfold

Vars\domain(st.varVals $\oplus$ {var $\mapsto$ val})

= simplify

Vars\domain(st.varVals)\{var}

= openVars\{var}.

Finite Differencing for the Unit Rule

What are the current units clauses?

let open?(st,lit) decide if literal lit has a value

Maintain: OLC (Open Literal Count per clause)

$$OLC(c) = size\ \{\ lit \mid lit \in c \wedge open?(st, lit)\ \}$$ for clauses c

Finite Differencing for the Unit Rule

0. Initialization

Context: st = mkInitialState(p)
Simplify: OLC(c) = size { lit | lit∈c ∧ open?(st,lit) }

= substituting st

OLC(c) = size { lit | lit∈c ∧ open?(mkInitialState(p),lit) }

= unfold open?

size { lit | lit∈c ∧ Apply(mkInitialState(p).varVal,lit)=unk }

= size { lit | lit∈c ∧ true }

= size(c).

Finite Differencing for the Unit Rule

1. UpdateState

Context: st' = updateState(st,v,b)
& OLC(c) = size { lit | lit$\in$c $\wedge$ open?(st,lit) }
Simplify: OLC'(c) = size { lit | lit$\in$c $\wedge$ open?(st’,lit) }

= substituting st'

size { lit | lit$\in$c $\wedge$ open?(updateState(st,v,b), lit) }

= unfold open?

size { lit | lit$\in$c $\wedge$ Apply(updateState(st,v,b).varVal=unk, lit) }

= …

if v $\in$ map(varofLit,c)
then OLC(c) - 1
else OLC(c).

Finite Differencing for the Unit Rule

Maintain the set of clauses that have one open literal

currentUnitClauses = filter(fn(cl)→ OLC(cl)=1), domain(st.prop))

Context: st' = updateState(st,v,b)

currentUnitClauses' = currentUnitClauses
$\setminus$ {c | v $\in$ c $\wedge$ c $\in$ currentUnitClauses}
$\uplus$ {c | v $\in$ c $\wedge$ OLC(c)=2}

Data Type Refinement

Simple specification for finite maps:

```
spec
  import Sets
  type Map(a,b)

  op [a,b] apply : Map(a,b) -> a -> Option b
  op [a,b] empty_map :  Map(a,b)
  op [a,b] update : Map(a,b) -> a -> b -> Map(a,b)
  op [a,b] singletonMap : a -> b -> Map(a,b)
  op [a,b] domain: Map(a,b) -> Set a

  op [a,b] TMApply(m:Map(a,b),x:a | x in? domain(m)): b =
                  the(z:b)( apply m x = some z)
   …
endspec
```

Refinement by Spec Morphism

Refine Maps to Vector structures equipped for fast backtracking

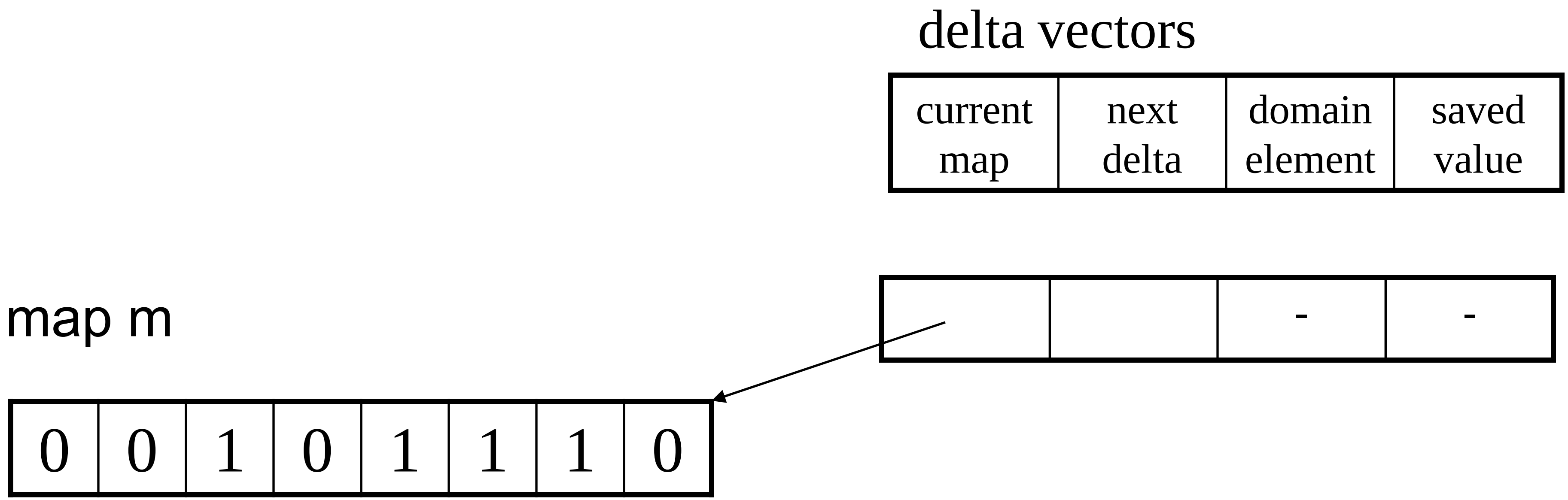
```
Map = spec
  import Sets
  type Map(a,b)

  op  apply
  op  empty_map
  op  update
  op  singletonMap
  op  domain

  op  TMApply
  …
endspec
```

```
BTVector = spec
  import Sets
  type Map(a,b)

  op BTV_apply
  op BTV_empty_map
  op BTV_update
  op singletonMap =
     λ (x, y) BTV_update(BTV_empty_map,x,y)

  op BTV_domainToList
  op domain(m: Map(a,b)) =
        foldl (λ(x,s) set_insert(x,s))
              empty_set (BTV_domainToList m)

  op BTV_eval
  …

endspec
```

Proof obligation: all map axioms remain provable under translation

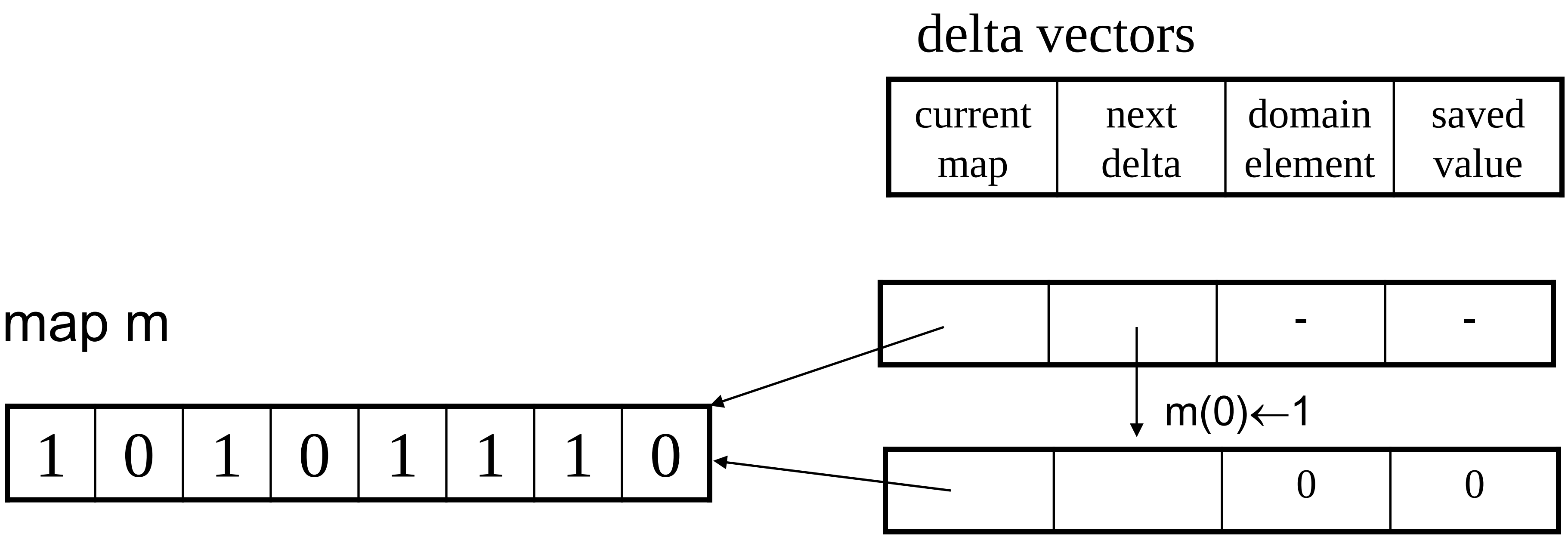
BTVectors

Datatype to represent maps with backtrack info



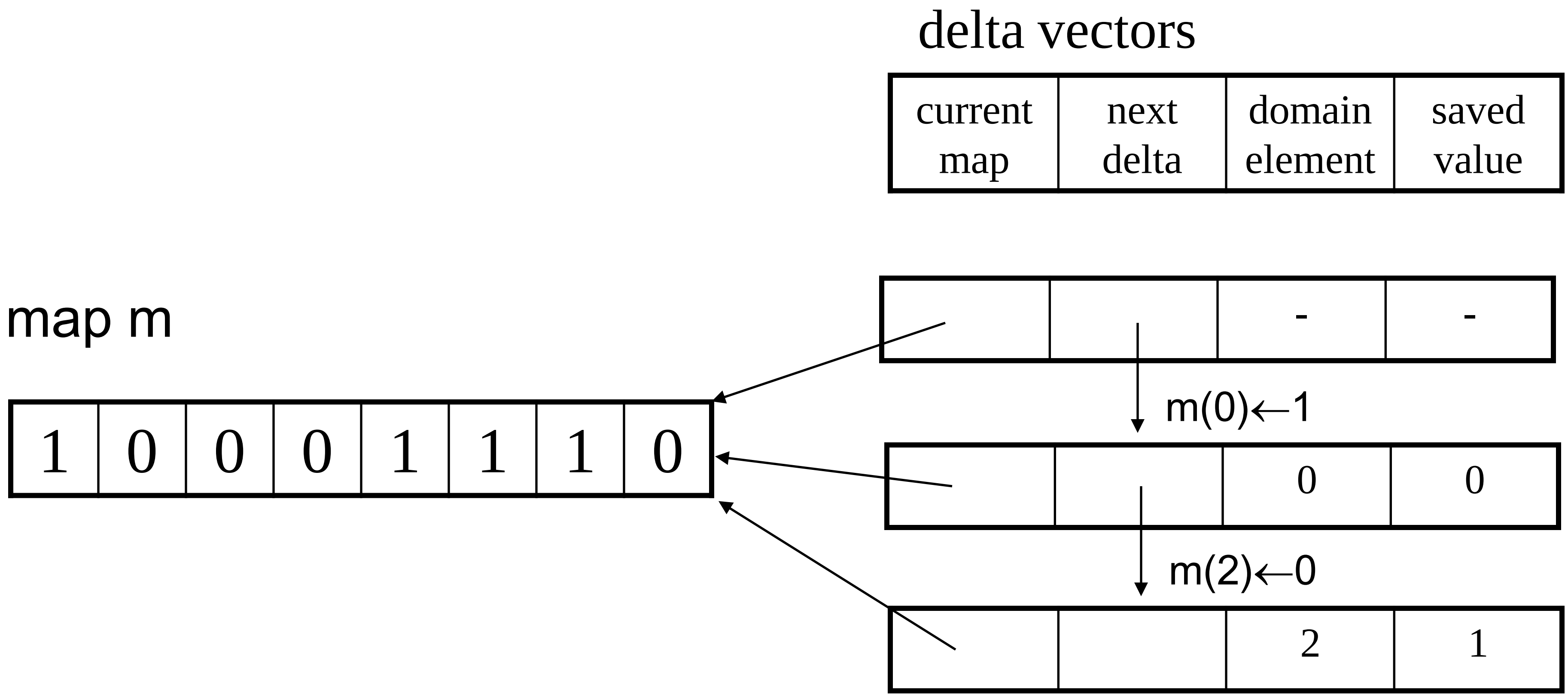
BTVectors

Datatype to represent maps with backtrack info

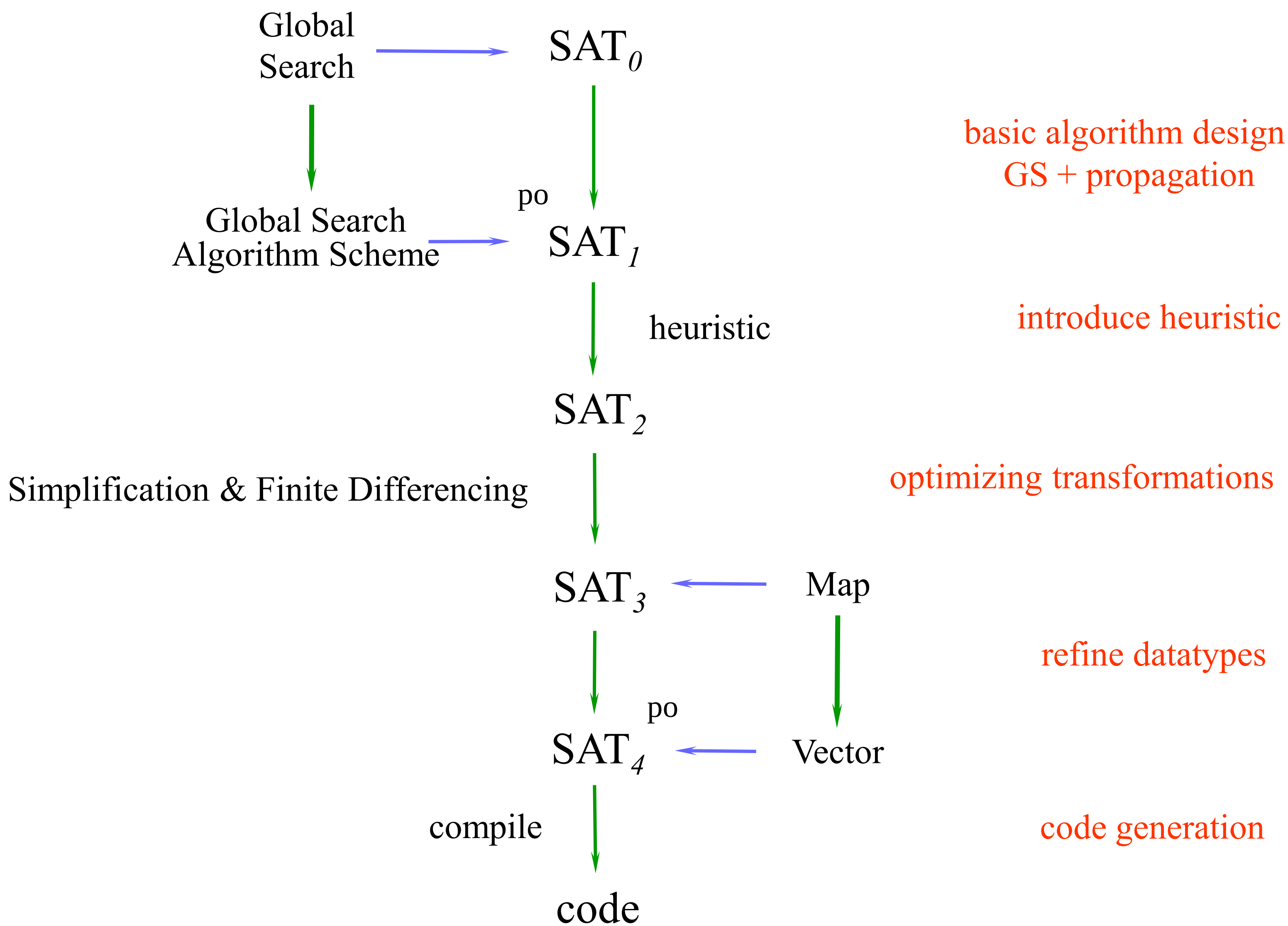


BTVectors

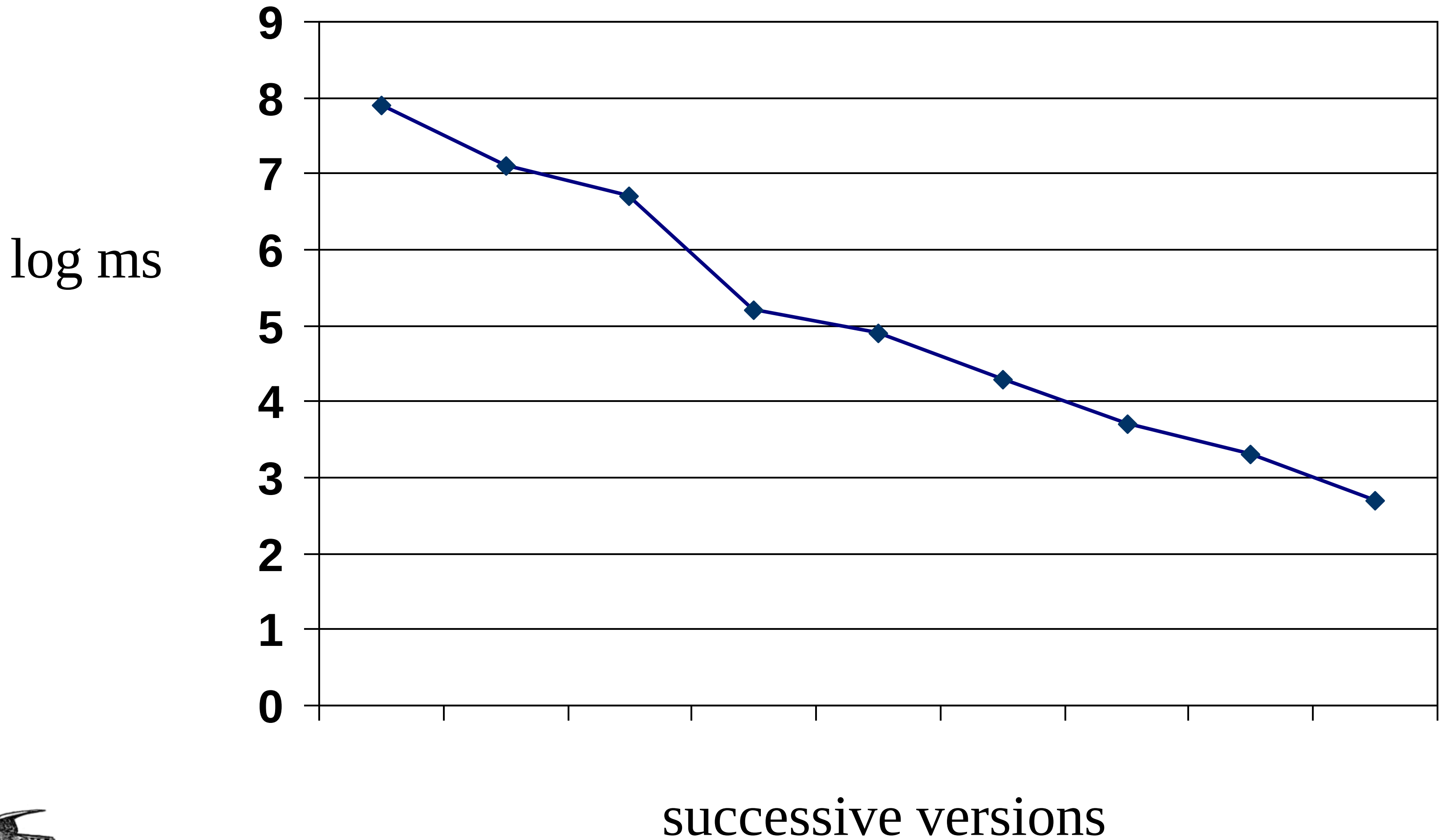
Datatype to represent maps with backtrack info



Derivation Structure



Log Plot of Runtimes for Consecutive Versions



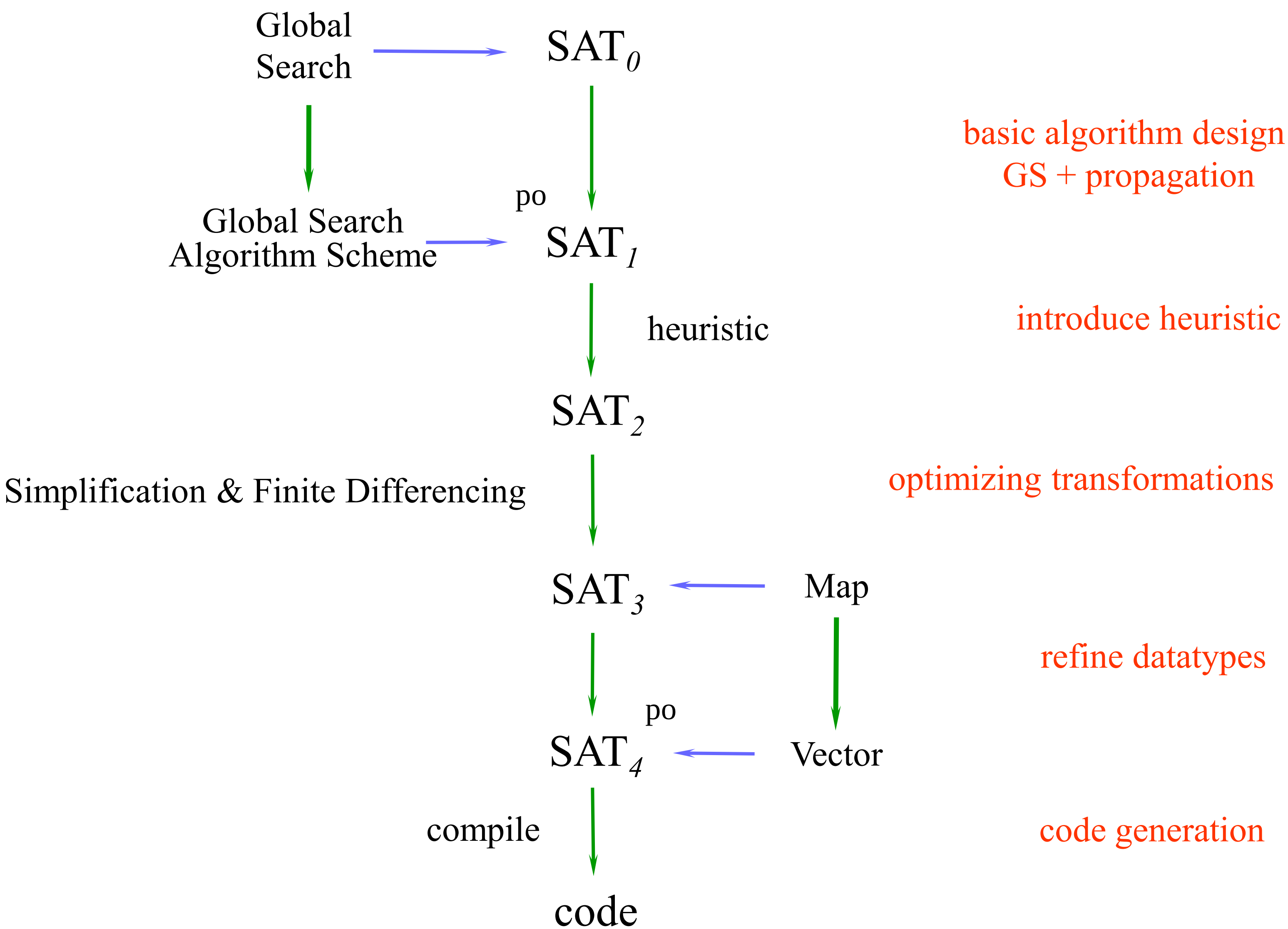
What's Next

- better conflict analysis
- learning
- restarts
- preprocessing
- watched literals
- locality tuning
- better data structures

General Goals:

1. Capture best-practice abstract design theories
2. Apply theories to generate native solvers for other problems

Derivation Structure



Extras

SAT-like Problems

- k-SAT (all clauses have size k)
- max-SAT (find an assignment that maximizes the satisfied clauses)
- QBF (Quantified Boolean Formula)
- 2QBF (QBF restricted to 2 quantifiers)
- Pseudo Boolean SAT (counting constraints + objectives: 0,1-ILP)
- Horn-SAT (clauses have at most one positive literal)
- game-SAT
- SMT (Satisfiability Modulo Theories)

non-SAT constraint problems, e.g.

Discrete CSP
Knapsack
Integer Linear Programming
Scheduling
Set Covers