Planning and Scheduling: Propositional Satisfiability Techniques

Prof. Dr.-Ing. Gerhard K. Kraetzschmar

Hochschule
Bonn-Rhein-Sieg

b-it
Bonn-Aachen
International Center for
Information Technology

Acknowledgements

These slides are based on those slides by Dana Nau, Stuart Russell, Sheila McIlrath and Alen Fern

Motivation

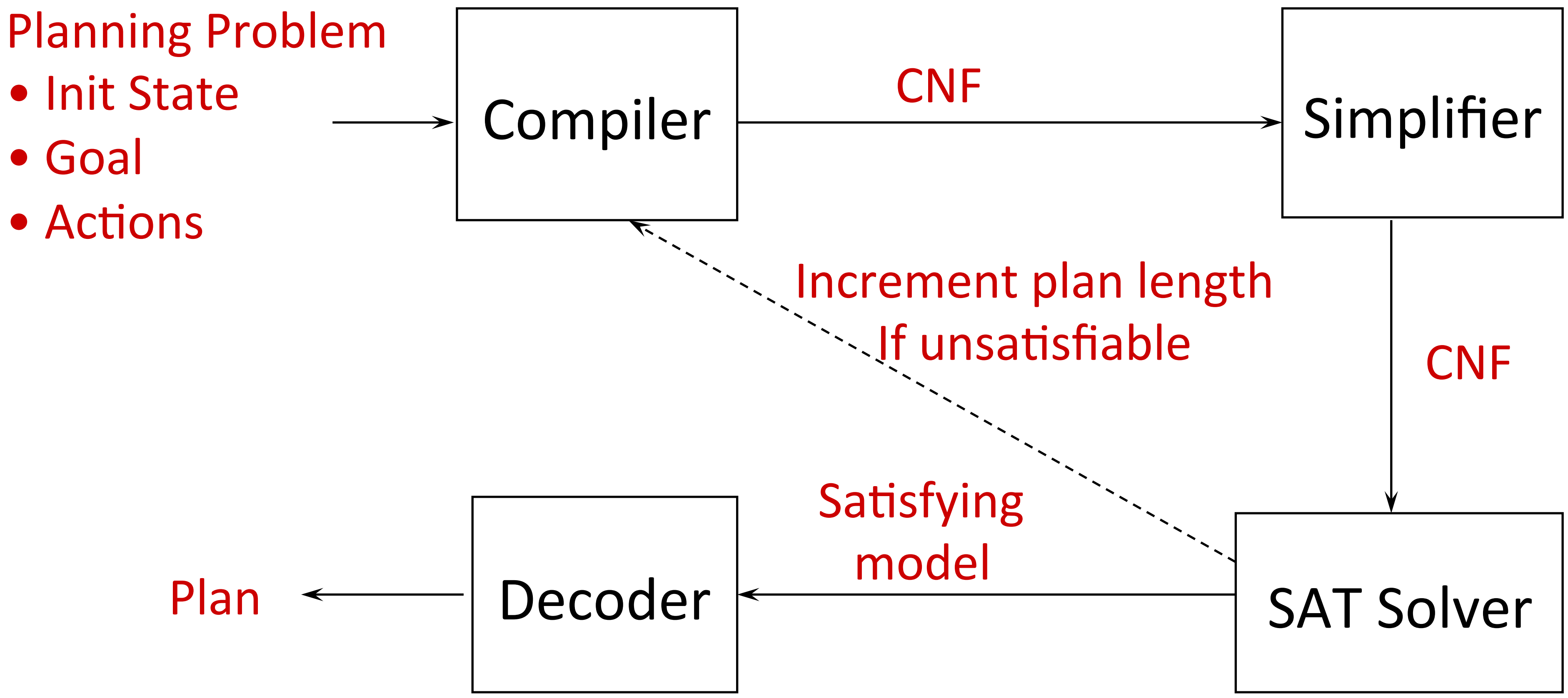
- Propositional satisfiability:
  - Given a Boolean formula, e.g. $(P \lor Q) \land (\neg Q \lor R \lor S) \land (\neg R \lor \neg P)$ does there exist a **model**, i.e. an assignment of truth values to the propositional variables that makes the formula true?
- This was the very first problem shown to be NP-complete
- Lots of research on algorithms for solving it
  - Algorithms are known for solving all but a small subset in average-case polynomial time
- Therefore:
  - Try translating classical planning problems into satisfiability problems, and solving them that way

Outline

- **Brief** review of propositional logic & satisfiability
- Planning as propositional satisfiability:
  - Encoding planning problems as satisfiability problems
  - Checking for satisfiability with a SAT solver
  - Extracting plans from truth values
- Satisfiability techniques
  - Davis-Putnam (actually DPLL, sometimes referred to as DP)
  - Local search
    - GSAT and WalkSAT
- Combining satisfiability techniques with planning graphs
  - BlackBox & SATPlan

* Terminology: “SATPLAN approach” (circa 1992) vs. the SATPLAN planner of 2004, 2006 etc., the successor of Blackbox.

Architecture of a SAT-based Planner



History of SAT-based approach

- 1969 Plan synthesis as theorem proving (Green IJCAI-69)
- 1971 STRIPS (Fikes & Nilsson AIJ-71)
- …
- 1992 Satplan Approach (Kautz & Selman ECAI-92)
- 1996 (Kautz & Selman AAAI-96) (Kautz, McAllester & Selman KR-96)
- 1997 MEDIC (Ernst *et al.* IJCAI-97)
- 1998 Blackbox (Kautz & Selman AIPS98 workshop)
- 1998 IPC-1 Blackbox performance comparable to the best
- 2000 IPC-2 Blackbox performs terribly (Graphplan-style planners dominated)
- 2002 IPC-3 No SAT-based planners entered
- 2004 IPC-4 Satplan04 was clear winner of Optimal propositional planners
- 2006 IPC-5 Satplan06 & Maxplan* (Chen Xing & Zhang IJCAI-07) dominated

BG - Propositional Logic: Syntax

We are given a set of primitive propositions $\{P_1, \ldots, P_n\}$

- These are the basic statements we can make about the “world”

- From basic propositions we can construct compound sentences (also called *formulas*)
  - If S is a sentence, $\neg S$ is a sentence (negation)
  - If S_1 and S_2 are sentences, $S_1 \wedge S_2$ is a sentence (conjunction)
  - If S_1 and S_2 are sentences, $S_1 \vee S_2$ is a sentence (disjunction)
  - If S_1 and S_2 are sentences, $S_1 \Rightarrow S_2$ is a sentence (implication)
  - If S_1 and S_2 are stences, $S_1 \Leftrightarrow S_2$ is a sentence (biconditional)

BG - Propositional Logic: CNF

- A *literal* is either a proposition or the negation of a proposition

- A *clause* is a disjunction of literals

- A formula is in *conjunctive normal form (CNF)* if it is the conjunction of clauses:

  E.g.: $(\neg R \vee P \vee Q) \wedge (\neg P \vee Q) \wedge (\neg P \vee R)$

- Any formula can be represented in conjunctive normal form (CNF)

BG - Propositional Logic: Semantics

- A *model* is a truth assignment to the propositions:
  e.g. p1 = false, p2 = true, p3 = false
  that makes the formula true!
- A formula is either true or false with respect to a model
- The truth of a formula is evaluated recursively as given below: (P and Q are formulas)

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
$false$	$false$	$true$	$false$	$false$	$true$	$true$
$false$	$true$	$true$	$false$	$true$	$true$	$false$
$true$	$false$	$false$	$false$	$true$	$false$	$false$
$true$	$true$	$false$	$true$	$true$	$true$	$true$

BG - Propositional Satisfiability

- A formula is *satisfiable* iff there exists some model
  - e.g. $A \vee B$, C
- A formula is *unsatisfiable* iff there does not exist any model
  - e.g. $A \wedge \neg A$

Encoding Planning as Satisfiability: Basic Idea

- A *bounded planning problem* is a pair (*P,n*):
  - *P* is a planning problem; *n* is a positive integer
  - Find a solution for *P* of length *n*

- Create a propositional formula that represents:
  - Initial state
  - Goal
  - Action dynamics

  for *n* time steps

- We will define the formula for (*P*,*n*) such that:
  1) ***any*** model (i.e. satisfying truth assignment) of the formula represents a solution to (*P*,*n*)
  2) if (*P*,*n*) has a solution then the formula is satisfiable

Example of Complete Formula for (P,1)

$at(r1,l1,0) \wedge \neg at(r1,l2,0) \wedge$
$at(r1,l2,1) \wedge$
$move(r1,l1,l2,0) \Rightarrow at(r1,l1,0) \wedge at(r1,l2,1) \wedge \neg at(r1,l1,1) \wedge$

$move(r1,l2,l1,0) \Rightarrow at(r1,l2,0)] \wedge at(r1,l1,1) \wedge \neg at(r1,l2,1) \wedge$

$\neg move(r1,l1,l2,0) \vee \neg move(r1,l2,l1,0) \wedge$

$\neg at(r1,l1,0) \wedge at(r1,l1,1) \Rightarrow move(r1,l2,l1,0) \wedge$

$\neg at(r1,l2,0) \wedge at(r1,l2,1) \Rightarrow move(r1,l1,l2,0) \wedge$

$at(r1,l1,0) \wedge \neg at(r1,l1,1) \Rightarrow move(r1,l1,l2,0) \wedge$

$at(r1,l2,0) \wedge \neg at(r1,l2,1) \Rightarrow move(r1,l2,l1,0)$

Overall Approach

- Do iterative deepening (like we did with Graphplan):
  - for $n = 0, 1, 2, \ldots,$
    - encode (P,n) as a satisfiability problem Φ
    - if Φ is satisfiable, then
      - From the set of truth values that satisfies Φ,
        a solution plan can be constructed,
        so return it and exit

- With a complete satisfiability tester, this approach will produce optimal layered plans for solvable problems

- We can use a GraphPlan analysis to determine an upper bound on n, providing us with a way to detect unsolvability

Notation

- For satisfiability problems we need to use propositional logic
- Need to encode ground atoms into propositions
  - For set-theoretic planning we encoded atoms into propositions by rewriting them as shown here:
    - Atom: at(r1,loc1)
    - Proposition: at-r1-loc1
- For planning as satisfiability we'll do the same thing
  - But we won't bother to do a syntactic rewrite
  - Just use at(r1,loc1) itself as the proposition
- Also, we'll write plans starting at a_0 rather than a_1
  - $\pi = \langle a_0, a_1, \ldots, a_{n-1} \rangle$

Fluents

- If $\pi = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ is a solution for (P,n), it generates these states:

  $s_0, \quad s_1 = \gamma(s_0,a_0), \quad s_2 = \gamma(s_1,a_1), \quad \ldots, \quad s_n = \gamma(s_{n-1}, a_{n-1})$

- A *fluent* is a proposition used to describe what's true in each s_i

  - at(r1,loc1,*i*) is a fluent that's true iff at(r1,loc1) is in s_i
  - We'll use l_i to denote the fluent for a literal l in state s_i
    - e.g., if l = at(r1,loc1)
      then l_i = at(r1,loc1,*i*)
  - a_i is a fluent saying that a is the i'th step of π
    - e.g., if a = move(r1,loc2,loc1)
      then a_i = move(r1,loc2,loc1,*i*)

Encoding Planning Problems

- Encode (P,n) as a formula Φ such that

  $\pi = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ is a solution for (P,n)
  if and only if

  Φ can be satisfied in a way that makes the fluents
  $a_0, \ldots, a_{n-1}$ true

- Let
  - A = {all actions in the planning domain}
  - S = {all states in the planning domain}
  - L = {all literals in the language}

- Φ is the conjunction of many other formulas …

Formulas in Φ

- Formula describing the *initial state*:

  $$\bigwedge\{l_0 \mid l \in s_0\} \wedge \bigwedge\{\neg l_0 \mid l \in L - s_0\}$$

  Describes the complete initial state (both positive and negative facts)
  E.g. on(A,B,0) ∧ ¬on(B,A,0)

- Formula describing the *goal*:

  $$\bigwedge\{l_n \mid l \in g^+\} \wedge \bigwedge\{\neg l_n \mid l \in g^-\}$$

  Says that goal facts must be true in the final state (i.e. at time-step *n)*
  E.g. on(B,A,n)

- Is this enough?

Formulas in Φ

- For every action *a* and timestep *i,* formula describing what fluents must be true if *a* were in the *i*'th step of the plan:
  - $a_i \Rightarrow \bigwedge\{p_i \mid p \in \text{Precond}(a)\} \wedge \bigwedge\{e_{i+1} \mid e \in \text{Effects}(a)\}$

    or

  - $a_i \Rightarrow \bigwedge\{l_i \mid l \in \text{Precond}(a)\}$, a's preconditions must be true
  - $a_i \Rightarrow \bigwedge\{l_{i+1} \mid l \in \text{ADD(a)}\}$, *a*'s ADD effects must be true in *i+1*
  - $a_i \Rightarrow \bigwedge\{\neg l_{i+1} \mid l \in \text{DEL}(a)\}$, *a*'s DEL effects must be false in *i+1*
- *Complete exclusion axiom*:
  - For all actions *a* and *b* and time-steps *i*, formulas saying *a* and *b* can't occur at the same time (e.g. $\neg a_i \vee \neg b_i$)
- Is this enough?
  - The formulas say nothing about what happens to facts if they are **not** effected by an action.

Frame Axioms

- *Frame axioms*:
  - Formulas describing what *doesn't* change between steps i and i+1

Several ways to write these…

- One way: *explanatory frame axioms*
  - One axiom for every possible literal l at every timestep i
  - Says that if l changes between s_i and s_{i+1}, then the action at step i must be responsible:

$$(\neg l_i \wedge l_{i+1} \Rightarrow \bigvee_{a \text{ in } A}\{a_i \mid l \in \text{effects}^+(a)\})$$
$$\wedge\ (l_i \wedge \neg l_{i+1} \Rightarrow \bigvee_{a \text{ in } A}\{a_i \mid l \in \text{effects}^-(a)\})$$

Example

- Planning domain:
  - one robot r1
  - two adjacent locations l1, l2
  - one operator (move the robot)

- Encode (P,n) where $n = 1$

  - Initial state: {at(r1,l1)}
    Encoding: $at(r1,l1,0) \wedge \neg at(r1,l2,0)$

  - Goal: {at(r1,l2)}
    Encoding: $at(r1,l2,1) \wedge \neg at(r1,l1,1)$

  - Operator: …

Example (continued)

- Operator: move(r,l,l')

  precond: at(r,l)

  effects: at(r,l'), ¬at(r,l)

  Encoding:

  $move(r1,l1,l2,0) \Rightarrow at(r1,l1,0) \wedge at(r1,l2,1) \wedge \neg at(r1,l1,1)$

  $move(r1,l2,l1,0) \Rightarrow at(r1,l2,0) \wedge at(r1,l1,1) \wedge \neg at(r1,l2,1)$

  $move(r1,l1,l1,0) \Rightarrow at(r1,l1,0) \wedge at(r1,l1,1) \wedge \neg at(r1,l1,1)$

  $move(r1,l2,l2,0) \Rightarrow at(r1,l2,0) \wedge at(r1,l2,1) \wedge \neg at(r1,l2,1)$

  contradictions (easy to detect)

  $move(l1,r1,l2,0) \Rightarrow \ldots$

  $move(l2,l1,r1,0) \Rightarrow \ldots$

  $move(l1,l2,r1,0) \Rightarrow \ldots$

  $move(l2,l1,r1,0) \Rightarrow \ldots$

  nonsensical

- How to avoid generating the last four actions?
  - Assign data types to the constant symbols like we did for state-variable representation

Example (continued)

- Locations: l1, l2
- Robots: r1

- Operator: move(r : robot, l : location, l' : location)

  precond: at(r,l)

  effects: at(r,l'), ¬at(r,l)

Encoding:

$$move(r1,l1,l2,0) \Rightarrow at(r1,l1,0) \wedge at(r1,l2,1) \wedge \neg at(r1,l1,1)$$

$$move(r1,l2,l1,0) \Rightarrow at(r1,l2,0) \wedge at(r1,l1,1) \wedge \neg at(r1,l2,1)$$

Example (continued)

- Complete-exclusion axiom:

$$\neg move(r1,l1,l2,0) \vee \neg move(r1,l2,l1,0)$$

- Explanatory frame axioms:

$$\neg at(r1,l1,0) \wedge at(r1,l1,1) \Rightarrow move(r1,l2,l1,0)$$
$$\neg at(r1,l2,0) \wedge at(r1,l2,1) \Rightarrow move(r1,l1,l2,0)$$
$$at(r1,l1,0) \wedge \neg at(r1,l1,1) \Rightarrow move(r1,l1,l2,0)$$
$$at(r1,l2,0) \wedge \neg at(r1,l2,1) \Rightarrow move(r1,l2,l1,0)$$

Example of Complete Formula for (P,1)

$at(r1,l1,0) \wedge \neg at(r1,l2,0) \wedge$

Initial State

$at(r1,l2,1) \wedge$

Goal

$move(r1,l1,l2,0) \Rightarrow at(r1,l1,0) \wedge at(r1,l2,1) \wedge \neg at(r1,l1,1) \wedge$

$move(r1,l2,l1,0) \Rightarrow at(r1,l2,0)] \wedge at(r1,l1,1) \wedge \neg at(r1,l2,1) \wedge$

Operator

$\neg move(r1,l1,l2,0) \vee \neg move(r1,l2,l1,0) \wedge$

Complete exclusion axiom

$\neg at(r1,l1,0) \wedge at(r1,l1,1) \Rightarrow move(r1,l2,l1,0) \wedge$

$\neg at(r1,l2,0) \wedge at(r1,l2,1) \Rightarrow move(r1,l1,l2,0) \wedge$

$at(r1,l1,0) \wedge \neg at(r1,l1,1) \Rightarrow move(r1,l1,l2,0) \wedge$

$at(r1,l2,0) \wedge \neg at(r1,l2,1) \Rightarrow move(r1,l2,l1,0)$

Explanatory frame axioms

Convert to CNF and give it to a SAT solver

Extracting a Plan

- Suppose we find an assignment of truth values that satisfies Φ.
  - This means *P* has a solution of length *n*
- For i=1,…,n, there will be exactly one action *a* such that a_i = *true*
  - This is the *i*'th action of the plan.
- Example (from the previous slides):
  - Φ can be satisfied with move(r1,l1,l2,0) = *true*
  - Thus, ⟨move(r1,l1,l2,0)⟩ is a solution for (*P*,1)
  - It's the only solution - no other way to satisfy Φ

Supporting Layered Plans

- Complete exclusion axiom:
  - For ***all*** actions a and b and time steps i include the formula
    $\neg a_i \lor \neg b_i$
  - This guaranteed that there could be only one action at a time
- Partial exclusion axiom:
  - For ***any pair of incompatible*** actions a and b and each time step i include the formula $\neg a_i \lor \neg b_i$
  - This encoding allows more than one action to be taken at a time step
    - resulting in layered plans

SAT Algorithms

- Systematic Search
  - DPLL (Davis Putnam Logemann Loveland)
    - backtrack search and unit propagation
    - Extend partial assignment into complete assignment
    - Sound and complete
- Local Search
  - GSAT
  - Walksat
    - Greedy local search and noise to escape minima
    - Modify randomly chosen total assignment
    - Sound but not complete (but very fast!)

Planning

- How to find an assignment of truth values that satisfies Φ?
  - Use a satisfiability algorithm

- DPLL is a complete SAT-solver:
  - First need to put Φ into conjunctive normal form
    e.g., $\Phi = D \wedge (\neg D \vee A \vee \neg B) \wedge (\neg D \vee \neg A \vee \neg B) \wedge (\neg D \vee \neg A \vee B) \wedge A$
  - Write Φ as a set of *clauses* (disjunctions of literals)
    $\Phi = \{\{D\}, \ \{\neg D, A, \neg B\}, \ \{\neg D, \neg A, \neg B\}, \ \{\neg D, \neg A, B\}, \ \{A\}\}$
  - Two special cases:
    - If $\Phi = \varnothing$ (I.e. no clauses) is a formula that is always *true*
    - If $\Phi = \{\ldots, \varnothing, \ldots\}$ (i.e. an empty clause) is a formula that's always *false* and hence unsatisfiable
  - DPLL simply searches the space of truth assignments, assigning one proposition a value at each step of the search tree

The DPLL Algorithm

1. Early termination

   A clause is true if any literal is true.

   A sentence is false if any clause is false. E.g. **$(\neg A \lor \neg B)$** $\land (A \lor C)$

2. Pure symbol heuristic

   Pure symbol: always appears with the same "sign" in all clauses.

   e.g., In the three clauses $(A \lor \neg B)$, $(\neg B \lor \neg C)$, $(C \lor A)$, A and B are pure, C is impure.

   Make a pure symbol literal true.

3. Unit clause heuristic

   Unit clause: only one literal in the clause

   The only literal in a unit clause must be true.

Basic Observations

- If literal L_1 is true,
  then clause $(L_1 \vee L_2 \vee \ldots)$ is true
- If clause C_1 is true,
  then $C_1 \wedge C_2 \wedge C_3 \ldots$ has the same value as $C_2 \wedge C_3$
  - *Therefore*: Its ok to delete clauses containing true literals!
- If literal L_1 is false,
  then clause $(L_1 \vee L_2 \vee L_3 \vee \ldots)$ has the same value as $(L_2 \vee L_3 \vee \ldots)$
  - *Therefore*: Its ok to shorten clauses containing false literals!
- If literal L_1 is false,
  then clause (L_1) is false
  - *Therefore*: the empty clause means false!

The Davis-Putnam Procedure

μ = {literals to which we have assigned the value *true*}; initially empty

DPLL:

if Φ contains an empty clause
  then return false; // backtrack
if Φ is a consistent set of literals
  then return true; // solution
Unit-Propagate // see below
Choose a literal P from Φ;
DPLL(Φ ∧ P,μ);
DPLL(Φ ∧ ¬P,μ);

Unit-propagate: // simplify - B.C. P.
  For every unit clause l in Φ
   Add l to the set of true literals
   Delete all clauses containing l
   Delete all occurrences of ¬l

```
Davis-Putnam(Φ,μ)
    if ∅ ∈ Φ then return
    if Φ = ∅ then exit with μ
    Unit-Propagate(Φ,μ)
    select a variable P such that P or ¬P occurs in ϕ
    Davis-Putnam(Φ ∪ {P},μ)
    Davis-Putnam(Φ ∪ {¬P},μ)
end

Unit-Propagate(Φ,μ)
    while there is a unit clause {l} in Φ do
        μ ← μ ∪ {l}
        for every clause C ∈ Φ
            if l ∈ C then Φ ← Φ − {C}
            else if ¬l ∈ C then Φ ← Φ − {C} ∪ {C − {¬l}}
end
```

Backtracking search through alternative assignments of truth values to literals

DPLL Example

- Given: formula Φ, with list of symbols {A,B,C,D,E}
  - DPLL({ (¬D ∨ ¬B ∨ C), (B ∨ ¬A ∨ ¬C), (¬C ∨ ¬B ∨ E), (¬E ∨ D ∨ B), (B ∨ E ∨ ¬C) }, {A, B, C, D, E}, [])
- Step 1: not every clause true, none false, pure symbol ¬A
  - DPLL({ (¬D ∨ ¬B ∨ C), (¬C ∨ ¬B ∨ E), (¬E ∨ D ∨ B), (B ∨ E ∨ ¬C) }, {B,C,D,E}, [A=false])
- Step 2: not every clause true, none false, no pure symbols, no unit clause
  - DPLL({ (¬E ∨ D), (E ∨ ¬C) }, {C,D,E}, [A=false, B=false])
  - DPLL({ (¬D ∨ C), (¬C ∨ E) }, {C,D,E}, [A=false, B=true])
- Step 3b: not every clause true, none false, pure D, ¬C, no unit clause
  - DPLL({ (}, {E}, [A=false, B=false, C=false, D=true]) DONE!
- Step 3a: not every clause true, none false, pure ¬D, E, no unit clause
  - DPLL({}, {C}, [A=false, B=true, D=false, E=true]) DONE!
- Done! (We found even two models.)

DPLL: Example 1

$$(\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b)$$

a=T

a=F

$$(\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b) \wedge a$$

$$(\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b) \wedge \neg a$$

Simplify

Simplify

$$b \wedge \neg b$$

...Satisfiable! $\neg b$

Simplify

{} ...Unsatisfiable!

Assignment: $a \rightarrow 0$, $b \rightarrow 0$

Remove: all clauses which become true

all literals that become false from the remaining clauses

DPLL: Example 2

$$(P \lor R) \land (\neg P \lor \neg Q \lor R) \land (P \lor \neg R)$$

P=T

P=F

$$(P \lor R) \land (\neg P \lor \neg Q \lor R) \land (P \lor \neg R) \land P \qquad (P \lor R) \land (\neg P \lor \neg Q \lor R) \land (P \lor \neg R) \land \neg P$$

Simplify

Simplify

$\neg Q \lor R$ ...Satisfiable!

$R \land \neg R$

Simplify

$\{\}$

Assignment: $P \rightarrow 1$, $Q \rightarrow 0$, $R \rightarrow 1$

...Unsatisfiable!

Remove: all clauses which become true

all literals that become false from the remaining clauses

Local Search - Basic Idea

- Keep only a single (complete) state in memory
- Generate only the neighbours of that state
- Keep one of the neighbours and discard others

- Key features:
  - no search paths
  - neither systematic nor incremental

- Key advantages:
  - use very little memory (constant amount)
  - find solutions in search spaces too large for systematic algorithms

Local Search

- Local search over space of ***complete*** truth assignments
- Let *u* be an assignment of truth values to all of the variables
  - cost(u,Φ) = number of clauses in Φ that aren't satisfied by *u*
  - flip(P,u) = u except that *P*'s truth value is reversed
- Local search:
  - Select a random assignment *u*
  - while cost(u,Φ) $\neq$ 0
    - if there is a P such that cost(flip(P,u),Φ) < cost(u,Φ) then
      - randomly choose any such *P*
      - u $\leftarrow$ flip(P,u)
    - else return failure
- Local search is sound
- If it finds a solution it will find it very quickly
- Local search is not complete: can get trapped in local minima

GSAT

- Basic-GSAT:
  - Select a random assignment u
  - while cost(u,Φ) ≠ 0
    - choose the P that **minimizes** cost(flip(P,u),Φ), and flip it
- Not guaranteed to terminate
- GSAT:
  - restart after a max number of flips
  - return failure after a max number of restarts
- WalkSAT is like GSAT but differs in the method used to pick which variable to flip
  - Both algorithms may restart with a new random assignment if trapped in local minima.
  - Many versions of GSAT/WalkSAT. WalkSAT superior for planning.

WalkSat

- With probability P:
  - flip **any** variable in any unsatisfied clause
- With probability (1-P):
  - flip **best** variable in any unsatisfied clause
  - The best variable is the one that when flipped causes the most clauses to be satisfied
- P controls the randomness of search
- Randomness can help avoid local minima
- Best DPLL-based solvers (e.g., Siege) are currently best!

What SAT-based planning shows

- General propositional reasoning can compete with state of the art specialized planning systems
  - Radically new stochastic approaches to SAT can provide very low exponential scaling
  - Best solvers for SAT-based planning are currently DPLL-based solvers such as Satzilla, PrecoSAT (and previously RelSAT and before that Siege and before that ZChaff) that have the option of using random restarts and some other local-search tricks.
- Why does it work?
  - More flexible than forward or backward chaining
  - Randomized algorithms less likely to get trapped along bad paths

Discussion:

- Recall the overall approach:
  - for $n = 0, 1, 2, \ldots,$
    - encode (P,n) as a satisfiability problem Φ
    - if Φ is satisfiable, then
      - From the set of truth values that satisfies Φ, extract a solution plan and return it

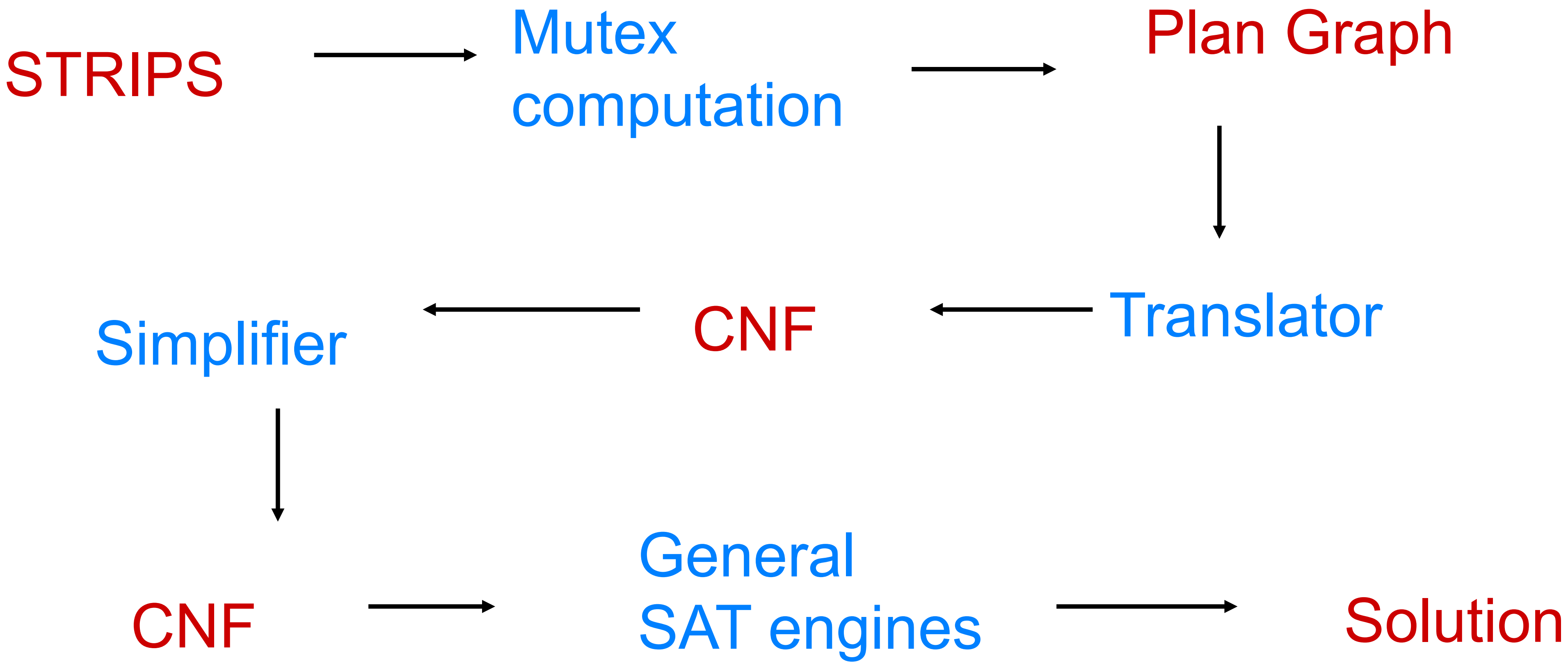
- How well does this work?

Discussion:

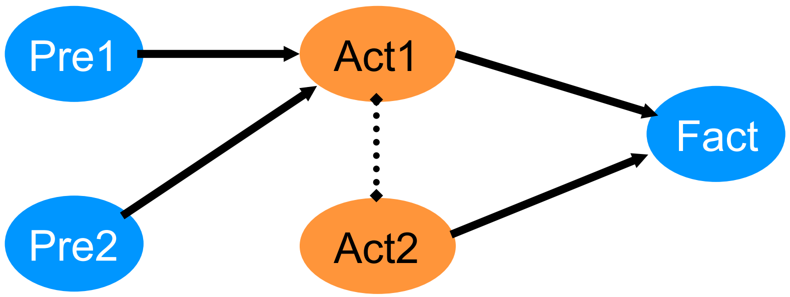
- Recall the overall approach:
  - for $n = 0, 1, 2, \ldots,$
    - encode (P,n) as a satisfiability problem Φ
    - if Φ is satisfiable, then
      - From the set of truth values that satisfies Φ, extract a solution plan and return it

- How well does this work?
  - By itself, not very practical (takes too much memory and time)
  - But it can be combined with other techniques
    - e.g., planning graphs

BlackBox


STRIPS
Mutex computation
Plan Graph
Translator
CNF
Simplifier
CNF
General SAT engines
Solution

Translation of Plan Graph



- Fact $\Rightarrow$ Act1 $\vee$ Act2
- Act1 $\Rightarrow$ Pre1 $\wedge$ Pre2
- $\neg$Act1 $\vee$ $\neg$Act2

Can create such constraints for every node in the planning graph

BlackBox

- The BlackBox procedure combines planning-graph expansion and satisfiability checking
- Roughly as follows:
  - for *n = 0, 1, 2, …*
  - Graph expansion: create a “planning graph” that contains *n* “levels”
  - Check whether the planning graph satisfies a necessary (but insufficient) condition for plan existence
  - If it does, then
    - Encode (P,n) as a satisfiability problem Φ but include **only** the actions in the planning graph
    - If Φ is satisfiable then return the solution

BlackBox

- Memory requirement still is combinatorially large,
  but less than satisfiability alone
- It was one of the two fastest planners in the 1998 Planning Competition
- When is BlackBox not a good idea?

1. When the domain is
   too large for a propositional planning approach
2. When long sequential plans are needed
3. When the solution time
   is dominated by graph-expansion and not plan extraction.

SatPlan: BlackBox's successor

- SatPlan combines planning-graph expansion and satisfiability checking, roughly as follows:
  - for *n* = 0, 1, 2, …
    - Create a planning graph that contains *n* levels
    - Encode the planning graph as a satisfiability problem
    - Try to solve it using a SAT solver
      - If the SAT solver finds a solution within some time limit,
        - Remove some unnecessary actions
        - Return the solution
- Memory requirement still is combinatorially large
  - but less than what's needed by a direct translation into satisfiability
- SatPlan was one of the best planners in the 2004 and 2006 planning competitions

* Terminology: "SATPLAN approach" (circa 1992) vs. the SATPLAN planner of 2004, 2006 etc., the successor of Blackbox.