Outline for 4/16

- **Recap**
  - **Logistics**
  - **Search**
  - **Constraint Satisfaction**
  - **Knowledge Representation 1**
- **Model-Based Autonomous Systems**
  - Motivation & Overview
  - Model-based configuration management
  - Viewed as Optimizing CSP Search
  - LTMS: Incremental Propositional Solver

Brian Williams &
Pandu Nayak
NASA Ames

Logistics

- PS 1 past due
- PS 2 due in 1.5 weeks
- Project on Internet Integration Systems
- PS 3 at the end of the quarter

Course Topics by Week

- Search & Constraint Satisfaction
- KR 1: Propositional Logic
- Autonomous Spacecraft 1: Configuration Mgmt
- Autonomous Spacecraft 2: Reactive Planning
- KR2: Modeling Internet Information Sources
- Information Integration 2: Planning & Execution
- Supervised Learning & Datamining
- Reinforcement Learning
- Bayes Nets: Inference & Learning
- Review & Future Forecast

Unifying View of AI

- Knowledge Representation
  - Expressiveness
  - Reasoning (Tractability)
- Search
  - Space being searched
  - Algorithms & performance

Specifying a search problem?

- What are states (nodes in graph)?
- What are the operators (arcs between nodes)?
- Initial state?
- Goal test?
- [Cost?, Heuristics?, Constraints?]

E.g., Eight Puzzle

1	2	3
4	5	6
7	8	

Search Summary

		Time	Space	Complete?	Opt?
Brute force	DFS	b^d	d	N	N
	BFS	b^d	b^d	Y	Y
	Iterative deepening	b^d	bd	Y	Y
	Iterative broadening	b^d			
Heuristic	Best first	b^d	b^d	N	N
	Beam	b^d	b+L	N	N
	Hill climbing	**b^d**	b	N	N
	Simulated annealing	**b^d**	b	N	N
	Limited discrepancy	b^d	bd	Y/N	Y/N
Optimizing	A*	b^d	b^d	Y	Y
	IDA*	b^d	b	Y	Y
	SMA*	b^d	[b-max]	Y	Y

Binary Constraint Network

- Set of n variables: $x_1 \ldots x_n$
- Value domains for each variable: $D_1 \ldots D_n$
- Set of binary ***constraints*** (also known as relations)
  - $R_{ij} \subseteq D_i \times D_j$
  - Specifies which values of x_i are consistent w/ those of x_j
- Partial ***assignment*** of values with a tuple of pairs
  - {...(x,a)…} means variable x gets value a...
  - ***Consistent*** if all constraints satisfied on all vars in tuple
  - Tuple = full solution if consistent & all vars included
- Tuple $\{(x_i, a_i) \ldots (x_j, a_j)\}$ ***consistent w/ a set of vars*** $\{x_m \ldots x_n\}$ iff $\exists\ a_m \ldots a_n$ such that this tuple is consistent: $\{(x_i, a_i) \ldots (x_j, a_j), (x_m, a_m) \ldots (x_n, a_n)\}$ }

Constraint Satisfaction Summary

- Preprocessing Strategies
- Search Algorithms
  - Chronological Backtracking (BT)
  - Backjumping (BJ)
  - Conflict-Directed Backjumping (CBJ)
  - Forward checking (FC)
- Dynamic variable ordering heuristics

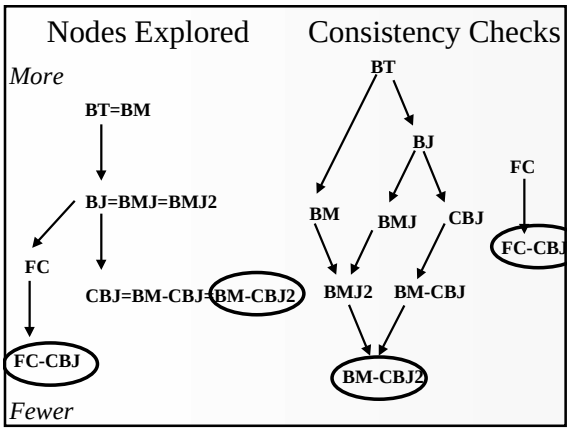
Backjumping (BJ)

- Similar to BT, but more efficient when no consistent instantiation can be found for the current var
- Instead of backtracking to most recent var…
- BJ reverts to deepest var which was checked against the current var

BJ Discovers
(2, 5, 3, 6) inconsistent with x_6
No sense trying other values of x_5

Other Strategies

- CBJ
  - More sophisticated backjumping behavior
  - Each variable has ***conflict set CS***
    - Set of vars that failed consistency checks w/ current val
  - Discovers (2, 5, 3) inconsistent with $\{x_5, x_6\}$
- FC
  - Perform Consistency Check Forward
  - Whenever assign var a value
    - Prune inconsistent values from
    - As-yet unvisited variables
    - Backtrack if domain of any var ever collapses



Knowledge Repr. Summary

- All KR systems $\subset$ logic or probability theory
- Propositional Logic
  - Syntac
  - Semantics
  - Inference
    - DPLL
    - GSAT
- First Order Predicate Calculus
  - Terms, $\forall$, $\exists$, ...
- Bayesian Belief Networks

Resolution

$$\frac{A \vee B \vee C, \quad \neg C \vee D \vee \neg E}{A \vee B \vee D \vee \neg E}$$

- Refutation Complete
  - Given an unsatisfiable KB in CNF,
  - Resolution will eventually deduce the empty clause
- Proof by Contradiction
  - To show $\Sigma \models Q$
  - Convert $\Sigma \cup \{\neg Q\}$ to CNF
    - Conjunction of disjunctions (clauses)
  - Show result is unsatisfiable!

Davis Putnam (DPLL)

[1962]

Procedure DPLL (CNF formula: Φ)
  If Φ is empty, return yes.
  If there is an empty clause in Φ return no.
  If there is a pure literal u in Φ return DPLL(Φ(u)).
  If there is a unit clause {u} in Φ return DPLL(Φ(u)).
  Else
    Select a variable v mentioned in Φ.
    If DPLL(Φ(v))=yes,
      then return yes.
    Else
      return DPLL(Φ(¬v)).

Recall: Φ(u) means set u := true in Φ, then simplify

GSAT

[1992]

Procedure GSAT (CNF formula: Φ, max-restarts, max-climbs)
  For I := I o max-restarts do
    A := randomly generated truth assignment
    for j := 1 to max-climbs do
      if A satisfies Φ then return yes
      A := random choice of one of best successors to A
        ;; successor means only 1 var val changes from A
        ;; best means making the most clauses true

Today's Outline

- **Recap**
  - **Search**
  - **Constraint Satisfaction**
  - **Knowledge Representation 1**
- **Model-Based Autonomous Systems**
  - Motivation & Overview
    - Model-Based Programming
    - Model-based Deductive Executive
    - RISC-like Deductive kernel
  - Model-based configuration management
  - Viewed as Optimizing CSP Search
  - LTMS: Incremental Propositional Solver

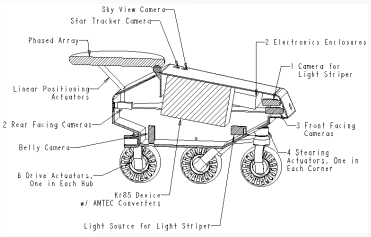
Emergence of Complex Autonomous Systems

- Availability of cheap networked control processors
- Emergence of realtime graphical programming environments
  - control shell, labview, simulink,
  - rule-based control G*
- More sophisticated realtime operating systems
  - VXWORKS
- Platform independent embedded language kernels
  - Java.

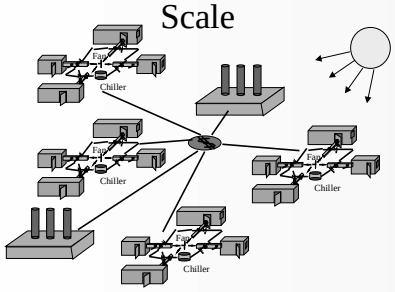
Mobile Robots

Indoor (Xavier)

Field (Lunar Rover)

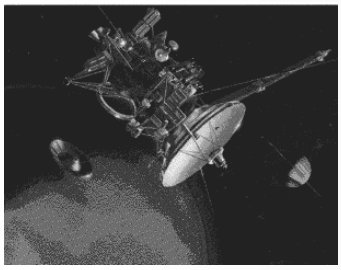


Immobile Robots Example 1: Autonomy on a Global Scale



Intelligent buildings, power grids, factories and highways.

Immobile Robots Example 2: Cassini Saturn Mission



- ~ 1 billion $
- 7 years to build
- 7 year cruise
- ~ 150 - 300 ground operators

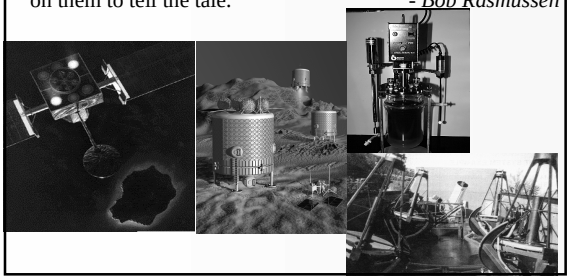
- *150 million $*
- *2 year build*
- *0 ground ops*

Properties of Complex Autonomous Systems

Mobile Robots	*Immobile Robots*
Structure:	
• 3D vision, articulated arms/legs	• 1D and binary sensors & actuators
• mobile	• highly reconfigurable
• few	• thousands
• mostly alike	• heterogenous
Function:	
• Navigation & path planning	• Control of internal systems
• Map and terrain learning	– conscious + autonomic control
	– immune, regulatory & nervous

NASA's Autonomous Systems

"With autonomy we declare that no sphere is off limits. We will send our spacecraft to search beyond the horizon, accepting that we cannot directly control them, and relying on them to tell the tale." *- Bob Rasmussen*

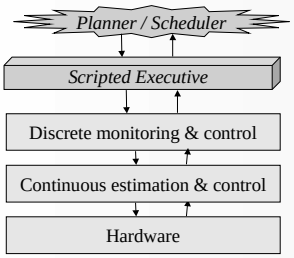


Additional Examples of Complex Autonomous Systems

- Berkeley Path Project and BatMobile
- Xerox PARC and CMU Smart Building projects
- Honeywell onboard control of Boeing Jet liner
- Rockwell Automation Assembly-line contrl
- DARPA Survivable Systems Initiative
- European smart car initiative
- NASA New Millennium and Outer Planets programs
- NASA Human Exploration and Development of Mars
- NASA Reusable Launch Vehicle program

Programming Complex Autonomous Systems

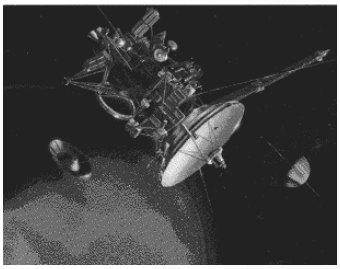
Are Traditional Robot Control Architectures adequate?



typically eliminated

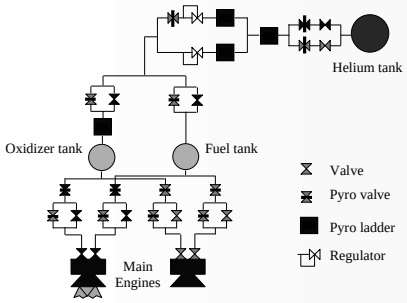
Challenge: Autonomous Saturn Orbital Insertion of Cassini

[Williams & Nayak 96a; Pell *et. al.* 97

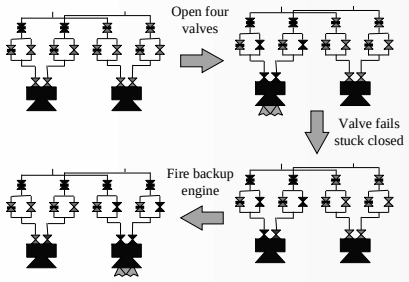


- *no Earth Comm*
- *~ 1 hr insertion window*
- *engines idle for several years*
- *moves through ring plane*

Cassini Propulsion System Schematic



Reconfiguring for a failed engine



Autonomous System Coding Challenge

Programmers must reason through system-wide interactions to generate codes for:

- monitoring
- hardware mode confirmation
- goal tracking
- detecting anomalies
- isolating faults
- diagnosing causes
- parameter estimation
- hardware reconfiguration
- fault recovery
- standby
- safing
- fault avoidance
- adaptive control
- control policy coordination

⇨ poor reuse, poor coverage, error prone

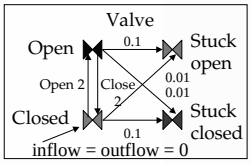
Solution: Part 1 Model-based Programming

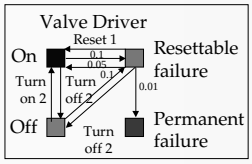
Programmers and operators generate breadth of functions from commonsense hardware models in light of mission-level goals.

Have engineers program in models, automate synthesis of code:

- models are compositional & highly reusable.
- generative approach covers broad set of behaviors.
- commonsense models are easy to articulate at concept stage and insensitive to design variations.

Single Core Model Describes Software, Hardware Dynamics



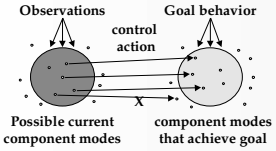


- **Finite Constraint, Concurrent Probabilistic Transition Systems**
  - Models qualitative behavior of component modes
  - Models failure, intermittency and optimal control

AI Code Identifies + Reconfigures Component Modes

Identifying Modes:
- mode confirmation
- goal tracking
- detecting anomalies
- isolating faults
- diagnosing causes

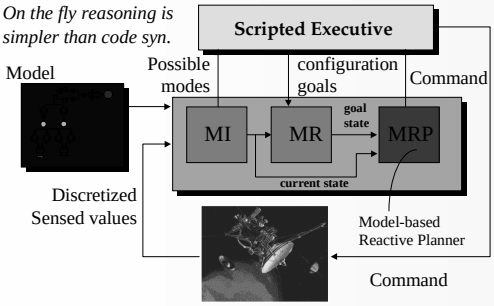
Reconfiguring Modes:
- hardware reconfiguration
- fault recovery
- standby, safing & shutdown
- fault avoidance
- control policy coordination



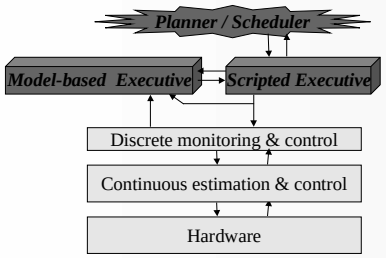
~~Traditional Control System~~ Agent Architecture

Solution: Part 2 Model-based Deductive Executive

On the fly reasoning is simpler than code syn.



Autonomy architecture with hybrid model-based & scripted executives



Combining Reactivity and Deduction: Is it practical?

- **Planning is NP-Hard:** Can we do real-time search?
  - " Hoping for the best amounts to arguing that for the particular cases that come up in practice, extensions to current planning techniques will happen to be efficient. My intuition is that this is not the case." -- [Chapman 87]
- **Proposed Solution:** Compile inference into reactive networks, guarantee response time ...

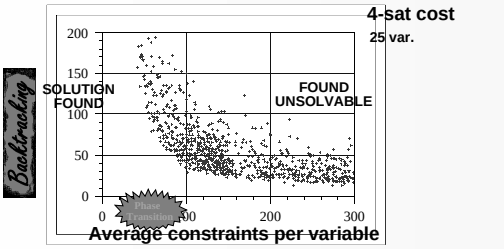
But is this solution sufficient?
But is this intuition correct?

Can autonomous systems be built w/ low cost, robust reactive networks?

- Most (semi) autonomous systems operate for long periods of time, often in harsh environments.
  - Saturn probes, automobiles, chemical plants, Antarctic habitats, assembly lines, power networks, computer networks, jetliners, submarines, ...
- Anomalies build over time, combinatorial blowup
  - 1 failure per year over a 7 year mission = n^7 likely anomalies.
- Reactive networks that precompile contingencies to likely anomalies become enormous!

However, most NP hard problems aren't hard.

- Satisfiability problems exhibit a phase transition from easily solvable to unsolvable. [Cheeseman et. al. 91]



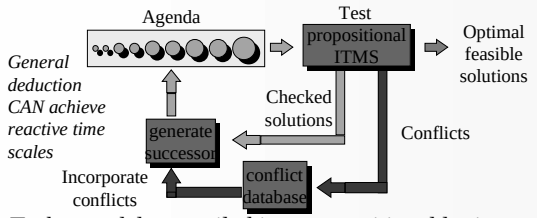
SAT, CSP problems for physical systems are typically easy

Physical systems ...
- are causal
- can be modeled with few feedback loops

- Hence, local propagation determines assignments to most state variables -- little search is required to test satisfiability.

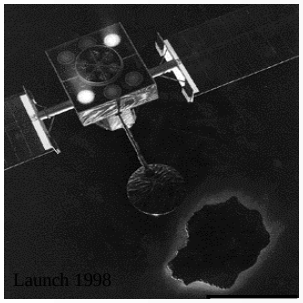
Approach: Build RISC-like onboard deductive kernel
- Propositional SAT or CSP algorithm at its core.
- Minimize SAT queries (e.g., < 10) using highly focused, best first search methods.

Solution: Part 3 Risc-like Best-first, Deductive Kernel



- Tasks, models compiled into propositional logic
- Conflicts dramatically focus search
- Careful enumeration grows agenda linearly
- ITMS efficiently tracks changes in truth assignments

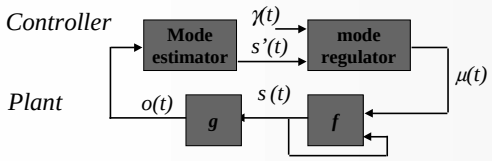
Deep Space One



Today's Outline

- **Recap**
- **Model-Based Autonomous Systems**
  - Motivation & Overview
    - Model-Based Programming
    - Model-based Deductive Executive
    - RISC-like Deductive kernel
  - **Model-based configuration management**
    - Modeling a spacecraft using propositional logic
    - Mode Identification
    - Mode Reconfiguration
  - Viewed as Optimizing CSP Search
  - LTMS: Incremental Propositional Solver

Consider a sub family of model-based optimal controllers where...



$\mu = argmin\ C(s', \gamma, \mu')\ s.t.\ \mu' \in M$

- *s(t), s'(t), o(t), γ(t), μ(t)* have discrete, finite domains and time *t* is discrete.
- *f* and *g* are specified declaratively.
- the estimator and regulator are implemented as queries to a fast, best first, propositional inference kernel.

A family of increasingly powerful deductive model-based optimal controllers

- Step 1: Model-based configuration management with a partially observable state-free plant.
- Step 2: Model-based configuration management with a dynamic, concurrent plant.
- Step 3: Model-based executive with a reactive planner, and an indirectly controllable dynamic, concurrent plant.

Modeling the Plant

- System *S* is a tuple (Π, Σ) [Williams & Nayak 96b]
- Π : set of *variables* ranging over finite domains
  - *state* variables (Π_s)
    - values are partitioned into *nominal* and *failure* values
  - *control* variables (Π_c)
  - *dependent* variables (Π_d)
  - *observable* variables (Π_o)
- A set of *feasible assignments*, Σ
- Specification ρ_Σ of Σ is in *propositional logic*.
- Propositions are of the form $y_k = e_k$

  y_k is a variable in Π, and e_k is in y_k's domain

  Σ is the set of assignments that satisfy ρ_Σ

Specifying a valve

- Variables Π = {*mode*, f_{in}, f_{out}, p_{in}, p_{out} }
  - *mode* $\in$ {*open, closed, stuck-open, stuck-closed*}
  - f_{in}, and f_{out} range over {*positive, negative, zero*}
  - p_{in}, and p_{out} range over {*high, low, nominal*}
- Specifying Σ with ρ_Σ

  $mode = open \Rightarrow (p_{in} = p_{out}) \wedge (f_{in} = f_{out})$

  $mode = closed \Rightarrow (f_{in} = zero) \wedge (f_{out} = zero)$

  $mode = stuck\text{-}open \Rightarrow (p_{in} = p_{out}) \wedge (f_{in} = f_{out})$

  $mode = stuck\text{-}closed \Rightarrow (f_{in} = zero) \wedge (f_{out} = zero)$

Configuration System (S, γ)

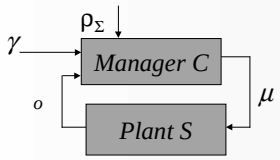
- S is a system (Π, Σ)
- γ: g_0, g_1 … , called goal configurations, is a sequence of propositional formulae on Π.

(S, γ) generates a *configuration trajectory*

$$\sigma : s_0, s_1 \ldots$$

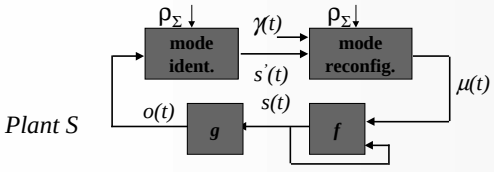
- $s_i \in \Sigma$
- s_{i+1} satisfies g_i or contains a failure value not in s_i

Model-based Configuration Manager



- C incrementally generates a *control sequence* $\mu : \mu_0, \mu_1 \ldots$, such that C and S together form a configuration system
- o are the observables (values for Π_o)
- And μ_i are control values (values for Π_c)
  - Assume (for now) in the absence of failures, there exist procedures to control nominal values of all state variables

Mode identification + reconfiguration



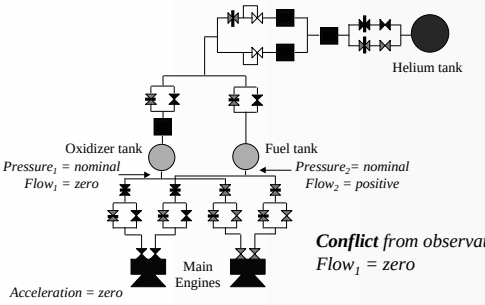
Configuration management achieved by

- Mode identification
  - identifies the system state based only on observables
- Mode reconfiguration
  - reconfigures the system state to achieve goals

Mode identification via Consistency-based diagnosis

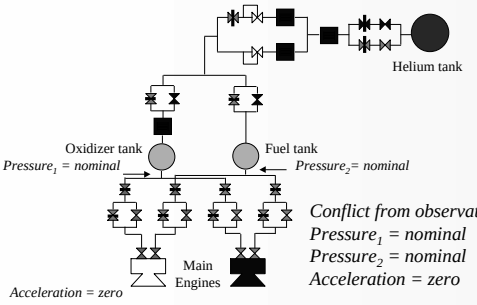
- Each component has an associated state variable
  - values partitioned into *nominal* values and *failure* values
- Mode identification involves finding state variable assignments consistent with the observations
  - each state variable can be assigned either the most recently commanded nominal value or one of the failure values

Example: Cassini propulsion system



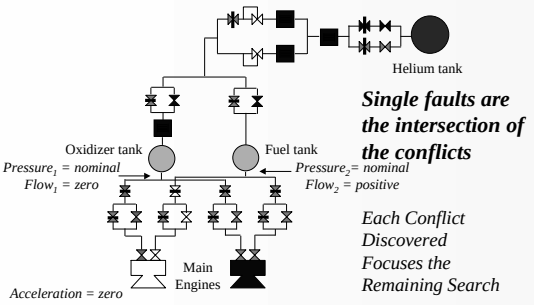
Conflict *from observation*
$Flow_1$ *= zero*

More ***conflicts***



Conflict from observations
$Pressure_1$ *= nominal*
$Pressure_2$ *= nominal*
Acceleration = zero

A ***candidate*** covers each conflict



Single faults are the intersection of the conflicts

Each Conflict Discovered Focuses the Remaining Search

Mode Identification as CSP

- Set of n variables: $x_1 \ldots x_n$
  - Modes for each component
- Value domains for each variable: $D_1 \ldots D_n$
  - Valve37 $\in$ {*open, closed, stuck-open, stuck-closed*}
- Set of ***constraints*** (also known as relations)
  - $Valve37 = open \Rightarrow (p_{in} = p_{out}) \wedge (f_{in} = f_{out})$
- Search
  - Goal a consistent set of modes, maximizing probability
  - Start State
    - NOT the null state where no variables have been assigned
    - Instead: hysteresis - previous values

Mode ***Reconfiguration***

- Each component has an associated control variable
  - in the absence of failures, each control variable directly controls the nominal value of the corresponding state variable
- Mode reconfiguration involves finding control variable assignments such that the goal is entailed

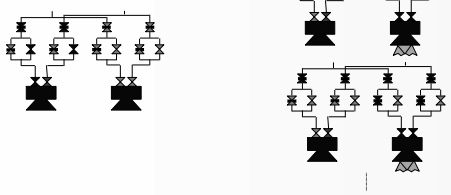
$\Rightarrow$ Equivalently, mode reconfiguration involves finding a set of component modes that entail the goal

$$\rho_\Sigma \cup \text{modes} \models g_0$$

Reconfiguring to restore thrust

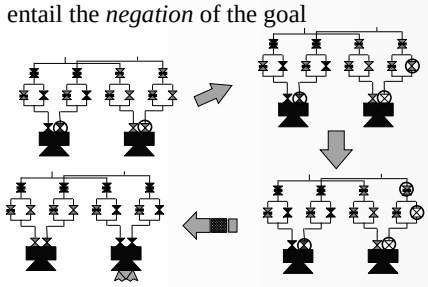
Possible reconfigurations

Current state



Conflicts focus search

- Conflicts are generated when a subset of the modes entail the *negation* of the goal



Statistically Optimal Configuration Management

Statistical Mode Identification [deKleer & Williams 86]

- $p(s_i \mid o_i) = p(o_i \mid s_i)\, p(s_i) / p(o_i)$ Bayes Rule
  $\propto p(o_i \mid s_i)\, p(s_i)$
- $p(o_i \mid s_i)$ is approximated from the model
  - $p(o_i \mid s_i) = 1$ if s_i entails o_i
  - $p(o_i \mid s_i) = 0$ if s_i and o_i are inconsistent
  - $p(o_i \mid s_i) = ?$ Otherwise
- Optimal Mode Reconfiguration

  Control $\mu_{i+1} = argmin\ C(s_i, \mu_{i+1}')$ s.t. $\mu_{i+1}' \models$ goal

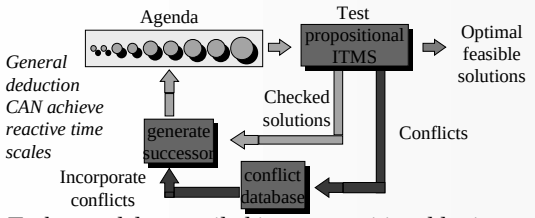
MI and MR performance

	MI time	MR time

Number of components: 80
Number of clauses: 11101

Diagnosis of Combinatorial Circuits

Solution: Part 3
Risc-like Best-first, Deductive Kernel



- Tasks, models compiled into propositional logic
- Conflicts dramatically focus search
- Careful enumeration grows agenda linearly
- ITMS efficiently tracks changes in truth assignments

Today's Outline

- **Recap**
- **Model-Based Autonomous Systems**
  - Motivation & Overview
  - Model-based configuration management
    - Modeling a spacecraft using propositional logic
    - Mode Identification
    - Mode Reconfiguration
  - **Viewed as Optimizing CSP Search**
    - **MI/MR as combinatorial optimization**
    - **Cost Model**
    - **Best-First Search**
  - LTMS: Incremental Propositional Solver

Combinatorial optimization problem

- Combinatorial opt. problem is a tuple (V, f, c)
- V is a set of *discrete* variables with *finite* domains
- *Assignment* maps each $v \in V$ to value domain
- f is a function that decides *feasibility* of assignments
  - *f(a)* returns *true* if and only if assignment *a* is feasible
- c is a function that returns the *cost* of an assignment
  - *c(a)* is the cost of assignment *a*
  - a_1 is *preferred* over assignment a_2 if $c(a_1) < c(a_2)$
- Problem:

  $min\ c(V)$ st $f(V)$

MI/MR as combinatorial optimization

- MI
  - variables: components with domains the possible modes
    - an assignment corresponds to a candidate diagnosis
  - feasibility: consistency with observations
  - cost: probability of a candidate diagnosis
- MR
  - variables: components with domains the possible modes
    - an assignment corresponds to a candidate repair
  - feasibility: entailment of goal
  - cost: cost of repair

Simple cost model

- Each variable has cost associated with each value
  - $c(v_i = l_i)$ is the cost of assigning value l_i to variable v_i
- Cost of a complete assignment is the *sum* of the costs of the individual variable assignments
  - if assignment a is $v_1=l_1,\ldots,v_n=l_n$ then $c(a) = \Sigma_i\, c(v_i=l_i)$
- Costs of all variable values are non-negative
  - $c(v_i = l_i) \quad 0$
- Each variable has a minimum cost value with cost *0*
- Generating a least cost assignment is straightforward
  - each variable is assigned a value with cost *0*

Using the simple cost model for MI

- Most probable diagnosis with *independent* component failures [de Kleer & Williams 89; de Kleer 91; Williams & Nayak 96]
  - $p(v_1=l_1,\ldots,v_n=l_n) = p(v_1=l_1) \times \ldots \times p(v_n=l_n)$
  - let m_i be the most probable mode for component v_i
  - $c(v_i=l_i) = -log(p(v_i=l_i) / p(v_i=m_i))$
    - ⇒all costs are non-negative with $c(v_i=m_i) = 0$
    - ⇒for any assignments a_1 and a_2, $c(a_1) > c(a_2)$ iff $p(a_1) < p(a_2)$

Limitations of the simple cost model

- *Dependent* faults [Srinivas & Nayak 96]
  - probabilistic dependence between component failures captured using a Bayesian network
  - need to use a special enumeration algorithm

Best first search

• Used in [de Kleer & Williams 89; ...; Williams & Nayak 96]

function *BFS(V, f, c)*
  Initialize *Agenda* to a least cost assignment
  Initialize *Solutions* to the empty set
  while *Agenda* is non-empty **do**
    Let *A* be one of the least cost assignments in *Agenda*
    Remove *A* from *Agenda*
    if *f(A)* is *true* **then** Add *A* to *Solutions* **endif**
    Add *immediate successor* assignments of *A* to *Agenda*
    if enough solutions **then return** *Solutions* **endif**
  endwhile
  return *Solutions*
end *BFS*

Required subroutines for *BFS*

- Generating a least cost assignment
- Generating the immediate successors of an assignment
  - *completeness*: every feasible assignment must be the (eventual) successor of the least cost assignment
  - *monotonicity*: if *b* is an immediate successor of *a*, then $c(a) \geq c(b)$
- Deciding that enough solutions have been generated
  - maximum number of solutions
  - minimum difference between cost of best feasible solution and the cost of the best assignment on the *Agenda*
  - minimum difference between costs of the last two assignments
- *Agenda* management as a priority queue

Representing assignments

- Each assignment is represented by the set of variable values that *differ* from the least cost assignment

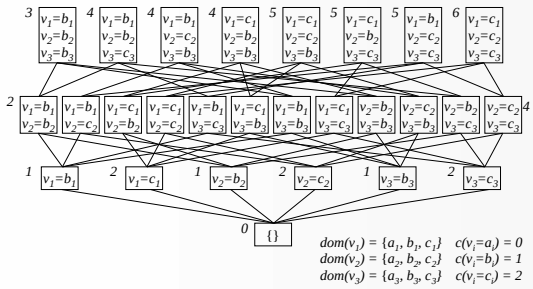
$dom(v_1) = \{a_1, b_1, c_1\}$ $\quad c(v_i=a_i) = 0$
$dom(v_2) = \{a_2, b_2, c_2\}$ $\quad c(v_i=b_i) = 1$
$dom(v_3) = \{a_3, b_3, c_3\}$ $\quad c(v_i=c_i) = 2$

- Least cost assignment $\{v_1=a_1, v_2=a_2, v_3=a_3\}$
- Assignment $\{v_1=a_1, v_2=a_2, v_3=b_3\}$ represented as just $\{v_3=b_3\}$

Basic successor function

- Assignment A_2 is an *immediate* successor of A_1 if
  - the representation of A_1 is a *subset* of the representation of A_2; and
  - the representations of A_1 and A_2 differ by exactly 1 variable value
  - *e.g.*, $\{v_3=b_3\}$ is an immediate successor of {}
  - *e.g.*, $\{v_3=b_3, v_2=b_2\}$ is an *eventual* successor, but *not* an immediate successor, of {}
- Definition of immediate successors is
  - *complete*: all assignments are eventual successors of the least cost assignment
  - *monotonic*: if A_2 is an immediate successor of A_1, then $c(A_1) \geq c(A_2)$

Successor lattice



Conflicts

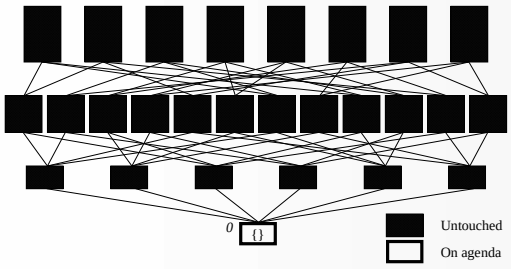
- A *conflict* is a *partial* assignment that is guaranteed to be infeasible
  - any assignment that *contains* (or is *subsumed* by) a conflict is infeasible
  - [Davis 84; Genesereth 84; de Kleer & Williams 87]
  - *e.g.*, if the partial assignment $\{v_3=a_3, v_2=a_2\}$ is a conflict, then the assignment $\{v_3=a_3, v_2=a_2, v_1=b_1\}$ is infeasible
- *Requirement*: whenever *f* determines that an assignment is infeasible, it returns a conflict
  - if assignment *A* is infeasible, then *A* itself is trivially a conflict
  - ideally, *f* returns a *minimal* infeasible subset of *A* as a conflict
  - conflicts can be generated using dependency tracking in a truth maintenance system

Focusing with conflicts

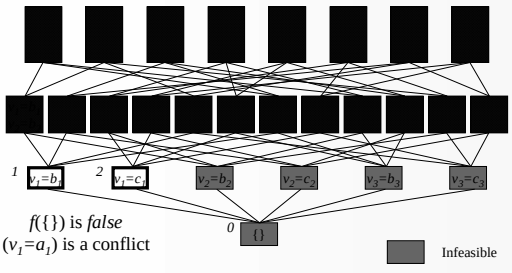
- *Lemma*: Let A_2 be an (eventual) successor of A_1 such that A contains a conflict N, but A_2 does not. Then there exists an immediate successor A_3 of A_1 that doesn't contain N such that A_2 is an (eventual) successor of A_3.

⇒If an assignment A_1 is infeasible and contains a conflict N, then we need only generate those immediate successors of A_1 that ***don't contain*** N
  - the lemma ensures that completeness is preserved
  - the **smaller the conflict, the fewer** the immediate successors

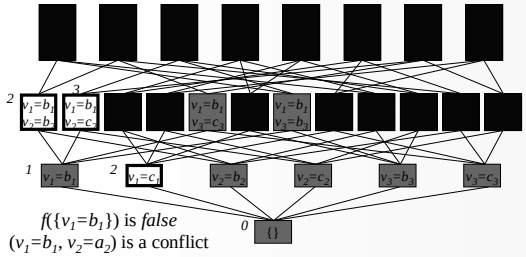
Initializing the agenda



Assignment {} is infeasible

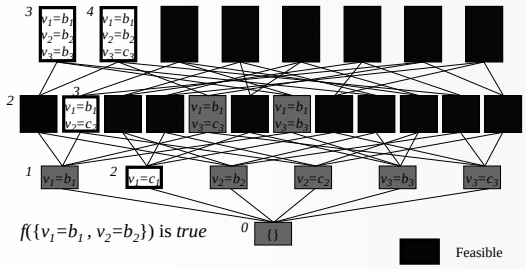


Assignment $\{v_1=b_1\}$ is infeasible



$f(\{v_1=b_1\})$ is *false*
$(v_1=b_1, v_2=a_2)$ is a conflict

Least cost feasible assignment found



$f(\{v_1=b_1, v_2=b_2\})$ is *true*

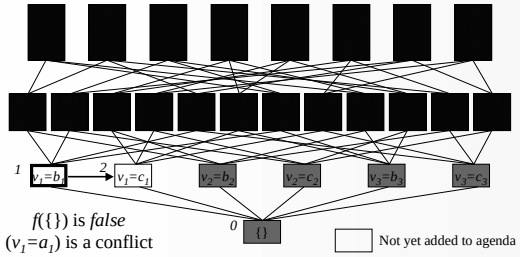
Decreasing agenda size

- Agenda size can be problematic in a best first search
  - for a branching factor *b*, agenda grows to size *O(bk)* after *k* checks
  - inserting *b* elements into the agenda after *k* checks is *O(b logb+b logk)*
- Immediate successors of an assignment are totally ordered
  - non-least cost successors only checked *after* least cost successor

⇒Insert only least cost successor onto agenda
Sort remaining successors
Each assignment has exactly two successors

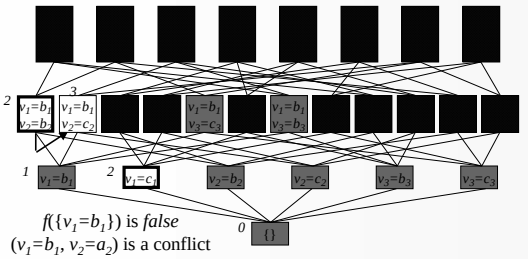
  - least cost immediate successor
  - next more expensive sibling
- Size of the agenda is *bounded by* the number of checks
  - inserting *b* successors after *k* checks is *O(b logb + 2logk)*

Only $\{v_1=b_1\}$ added to agenda



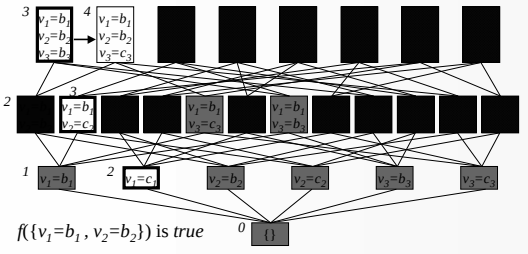
$f(\{\})$ is *false*
$(v_1=a_1)$ is a conflict

Immediate successor and sibling of $\{v_1=b_1\}$ added to agenda



$f(\{v_1=b_1\})$ is *false*
$(v_1=b_1, v_2=a_2)$ is a conflict

Least cost feasible assignment found



$f(\{v_1=b_1, v_2=b_2\})$ is *true*

Today's Outline

- **Recap**
- **Model-Based Autonomous Systems**
  - Motivation & Overview
  - Model-based configuration management
  - Viewed as Optimizing CSP Search
    - MI/MR as combinatorial optimization
    - Cost Model
    - Best-First Search
  - **LTMS: Incremental Propositional Solver**
    - **Truth Maintenance System**
    - **Incremental SAT**

LTMS: Truth Maintenance System

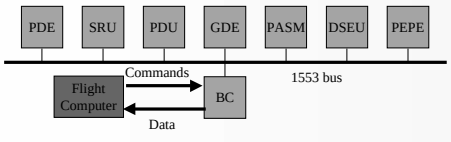
- Incrementally maintains consequences of a propositional theory Σ [Doyle 79; McAllester 80]
  - incrementally manages addition and deletions from Σ
  - ideal for *real-time* propositional reasoning
- Set Σ contains propositional clauses
  - a *clause* is disjunction of propositional literals
  - a *unit clause* is a clause with exactly one disjunct
  - a *literal* is a proposition or the negation of a proposition

    $\neg\, rain \vee \neg\, umbrella \vee dry$
  - a clause can be read as an implication in different ways

    $rain \wedge\ umbrella \Rightarrow dry$

    $rain \wedge \neg\, dry \Rightarrow \neg umbrella$

Example: DS-1 bus communication



- Some of the clauses describing bus communication

C_1: $\neg\, nci \vee \neg\, a \vee nco$ C_4: $\neg\, rf \vee ia$ C_7: $\neg\, ok \vee \neg\, uf$

C_2: $\neg\, ia \vee nco$ C_5: $\neg\, uf \vee ia$ C_8: $\neg\, rf \vee \neg\, uf$

C_3: $\neg\, ok \vee a$ C_6: $\neg\, ok \vee \neg\, rf$ C_9: $\neg\, a \vee \neg\, ia$

BC health: *ok, rf, uf* BC activity: *a, ia*

No input cmd: *nci* No output cmd: *nco*

Generic LTMS interface

- Updating the clauses in Σ
  - *add-clause* (*clause,* Σ)
  - *delete-clause* (*clause,* Σ)
- Propositional inference
  - *consistent?* (Σ)
  - *follows-from?* (*literal,* Σ)
- Justification structure
  - *supporting-clause* (*literal,* Σ)
  - *supporting-literals* (*literal,* Σ)
    - the *supporting-clause* together with the *supporting-literals* entail *literal*
    - each literal in *supporting-literals* follows from Σ

Using the LTMS in MI and MR

- LTMS database Σ contains clauses describing component behavior in each mode
- MI searches for component mode assignments that are consistent with the observations
- MR searches for component mode assignments that entail the goal
- MI and MR add and delete clauses corresponding to assumptions that a component is in a particular mode
- Justification structure is used to generate conflicts from an inconsistent Σ

ITMS = Incremental LTMS

- Optimizes context switching
- Propagates newly added clauses before deleting unsupported clauses
- Frequently 80% improvement over an LTMS
- Worst case: 5% slower than LTMS in 4/387 cases

LTMS labels

- The LTMS *labels* each proposition *true*, *false*, or *unknown*
  - if *p* is labeled *true* (*false*), then Σ logically entails *p* (¬*p*)
  - labeling algorithm is *sound*, but not necessarily *complete*
  - a positive (negative) literal is *true* if and only if the corresponding proposition is *true* (*false*)

Σ	
$\neg p \vee \neg q \vee r$	$\neg s$
$s \vee p$	q
$\neg u \vee v$	$u \vee v$

Labels	
p: true	*u: unknown*
q: true	*v:unknown*
r: true	
s: false	

Conflicting clauses

- A *conflicting clause* is one in which all literals are labeled *false*
  - $\neg p \vee \neg q \vee r$ is a conflicting clause if the labels are *p: true, q: true, r: false*
- Existence of a conflicting clause means that Σ is *inconsistent*
  - since LTMS labels are sound, a conflicting clause is unsatisfiable
- If Σ is inconsistent, *supporting-clause*(⊥, Σ) returns a conflicting clause and *supporting-literals*(⊥, Σ) returns the set of literals in that clause

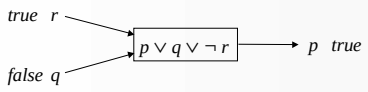
Unit propagation: the basic idea

- Let *C* be the clause $l_1 \vee l_2 \vee \ldots \vee l_n$
  - *C* can be viewed as the implication
    $\neg l_1 \wedge \neg l_2 \wedge \ldots \wedge \neg l_{n-1} \Rightarrow l_n$
- Suppose literals $l_1, l_2, \ldots, l_{n-1}$ are labeled *false* and l_n is labeled *unknown*
  - antecedent of the above implication version of clause *C* is *true*
  - hence, clause *C* implies that the consequent (l_n) must be *true*

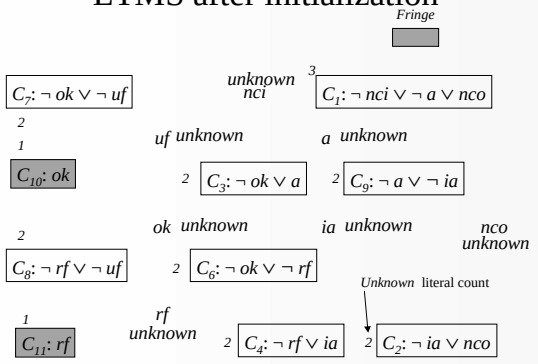
  $\Rightarrow l_n$'s label must be changed from *unknown* to *true*
- Example

Label inference + proposition support

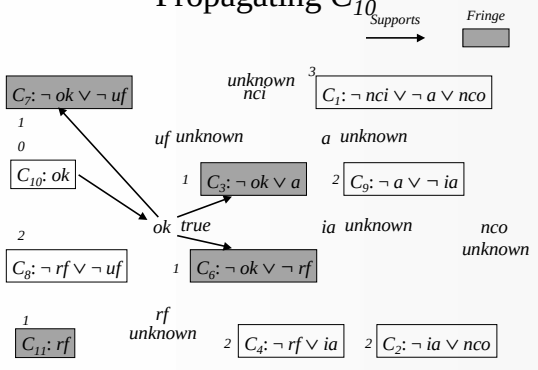
- Let *p* be the proposition labeled *unknown* in *clause*
  - *clause* has just been removed from *fringe*, and hence *p* is unique
- If *p* occurs as a positive (negative) literal in *clause*, its label is changed to *true* (*false*)
- *clause* and other propositions in *clause* are said to *support p*



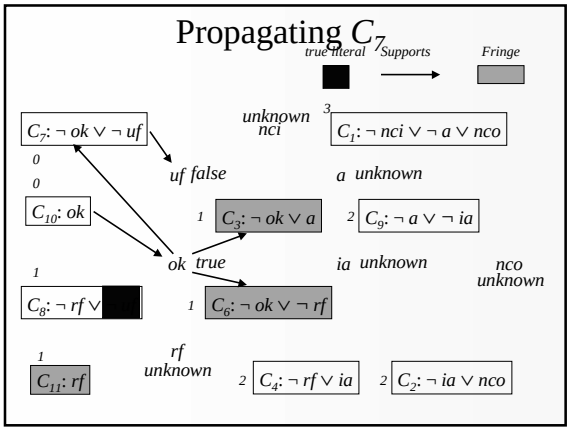
LTMS after initialization



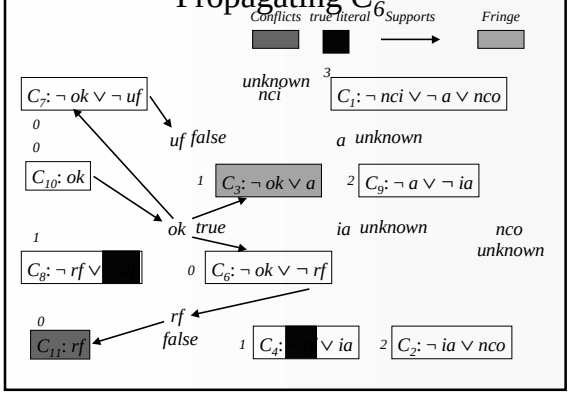
Propagating C_{10}



Propagating C_7



Propagating C_6



Unit propagation at the *fringe*

- Unit propagation takes place at the *fringe*, which consists of all clauses that have
  - exactly one literal labeled *unknown*
  - all other literals labeled *false*
- Basic unit propagation algorithm
  - select a clause from the *fringe* and propagate until the *fringe* is empty or a conflicting clause is detected

Unit propagation algorithm

function *compute- labels* (Σ)
  Initialize all labels to *unknown*
  Set Σ's *fringe* to Σ's unit clauses, and Σ's *conflicts* to the empty set
  propagate (Σ)
end *compute-labels*

function *propagate* (Σ)
  while Σ's *fringe* is non-empty and Σ's *conflicts* is empty **do**
    Remove a *clause* from Σ's *fringe*
    Infer a new label using *clause*
    Update Σ's *fringe* and *conflicts* with the newly propagated label
  endwhile
end *propagate*

Updating *fringe* and *conflicts*

- *fringe* and *conflicts* updated when a proposition's label changes
  - only clauses in which the proposition occurs can update *fringe* or *conflicts*
- Membership in *fringe* and *conflicts* determined *incrementally*
  - track the count of literals in the clause labeled *unknown*
    - decrement (increment) the count when an *unknown* (*true* or false) literal becomes *true* or *false* (*unknown*)
  - track whether the clause is satisfied (*i.e.*, contains a literal labeled *true*)

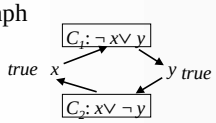
⇒A clause is added to (removed from) the *fringe* if the *unknown* literal count becomes (changes from) 1 and it is not (or it is) satisfied

Well-founded support

- Proposition supports generated by unit propagation form a *directed acyclic graph*

⇒Unit propagation produces *well-founded support*

- *Non-well-founded support* contains cycles in the support graph



Implementing the generic interface

- *consistent?* (Σ)
  - returns *true* iff Σ has no conflicts after unit propagation terminates
- *follows-from?* (*literal*, Σ)
  - returns *literal*'s label after unit propagation terminates
- *supporting-clause* (*literal*, Σ)
  supporting-literals (*literal*, Σ)
  - returns the clause and literals, respectively, that support *literal* after unit propagation terminates

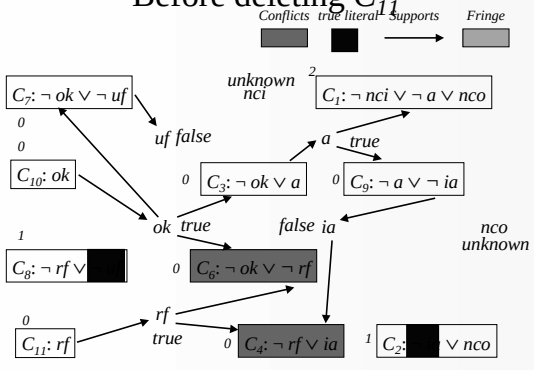
Incrementally modifying Σ

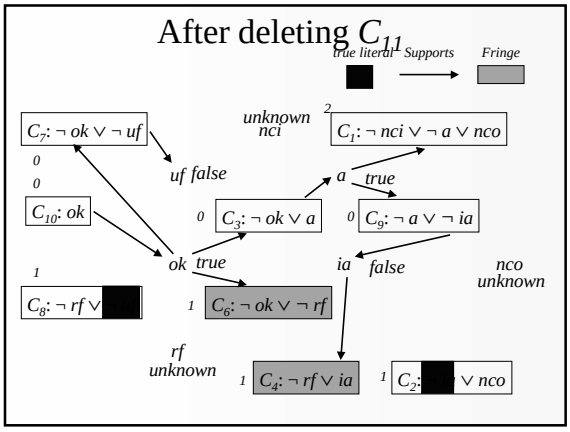
- *add-clause* (*clause*, Σ)
  - update *clause*'s *unknown* literal count and whether it is satisfied
  - update Σ's *fringe* and *conflicts* appropriately
  - call *propagate* (Σ)

  ⇒need only do propagations (directly or indirectly) dependent on *clause*
- *delete-clause* (*clause*, Σ)
  - follow the support structure to set the label of all propositions (directly or indirectly) supported by *clause* to *unknown*
    - update Σ's *fringe* and *conflicts* as labels are changed

  ⇒only propagations (directly or indirectly) dependent on *clause* are undone
  - call *propagate* (Σ)

Before deleting C_{11}




After deleting C_{11}
true literal
Supports
Fringe
C_7: ¬ ok ∨ ¬ uf
unknown
nci
2
C_1: ¬ nci ∨ ¬ a ∨ nco
0
uf false
a
true
0
C_{10}: ok
0
C_3: ¬ ok ∨ a
0
C_9: ¬ a ∨ ¬ ia
ok
true
ia
false
nco
unknown
1
C_8: ¬ rf ∨
1
C_6: ¬ ok ∨ ¬ rf
rf
unknown
1
C_4: ¬ rf ∨ ia
1
C_2: ∨ nco