

Introduction to Artificial Intelligence

Propositional Logic

March 15, 2007

Henry Kautz

Outline

- Logic
- Efficient satisfiability testing by backtracking search
- Efficient satisfiability testing by local search
- Applications

Desiderata for Knowledge Representation

1. Declarative
 - separate knowledge from particular hard-coded application
2. Expressive
 - general rules as well as facts
 - incomplete information
3. Concise: can generate new conclusions
 - exponential / infinite number
4. Effectively computable
 - unambiguous, even without context

Basic Idea of Logic

- By starting with **true assumptions**, you can deduce **true conclusions**.

Blaise Pascal (1623-1662)

We know the truth, not only by the reason, but also by the heart.

Thomas Henry Huxley (1825-1895)

Irrationally held truths may be more harmful than reasoned errors.

John Keats (1795-1821)

Beauty is truth, truth beauty; that is all
Ye know on earth, and all ye need to know.

The Big Three

\models

\supset

\vdash

Entailment

- m – something that determines whether a sentence S is true or false – a “state of affairs”
- $m \models S$
 - S is true in m
 - m is a model of S
- $S \models T$
 - S entails T
 - Every model of S is a model of T
 - When S is true, then T must be true

Truth in the World vs Truth in a Model

- It is hard to formally talk about truth in the “real world”
- So m is generally a mathematical object that “mirrors” the world in some important way
- E.g.: m is a truth assignment – a function from sentences to $\{1, 0\}$



Satisfiability and Validity

Fix a language L and a set of possible models M .

S is **satisfiable** if it is true in some model:

$$m \models S$$

S is **unsatisfiable** if it is false in all models.

S is **valid** if it is true in all models:

$$\models S$$

Propositional Logic

Ingredients of a sentence:

- Propositions (variables)
 - literal = a variable or a negated variable
- Special symbols: *true, false*
- Logical Connectives $\neg, \wedge, \vee, \supset$

Ingredients of a model:

- Truth assignment: function that assigns true/false (1/0) to each variable
- Always assigns symbol *true* 1, symbol *false* 0
- Truth assignment of **sentences** computed by applying truth-tables for connectives

Some Basic Theorems of Model Theory

$$S \models T \quad \text{iff} \quad \models S \supset T$$

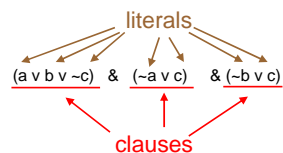
$$\models S \quad \text{iff} \quad \neg S \text{ is unsatisfiable}$$

$$S \models T \quad \text{iff} \quad (S \wedge \neg T) \models \text{false}$$

Formal Computational Complexity

- SAT = Prototypical NP-complete problem:
 - Given a Boolean formula, is there an assignment of truth values to the Boolean variables that makes it true?
 - As hard as any problem where an answer can be verified in polynomial time
 - Still NP-complete if formulas are restricted to Conjunctive Normal Form:

BUT:
HORN and 2SAT are solvable in linear time!



Inference

- **Mechanical** process for computing new sentences that are logically entailed
 - Does not depend on understanding intended meaning of sentences!

Examples:

Modus ponens

$$\{P, P \supset Q\} \vdash Q$$

Proof by refutation

$$(S \wedge \neg T) \vdash \text{false} \quad \text{then} \quad S \models T$$

Expert System for Automobile Diagnosis

Knowledge Base:

$\text{GasInTank} \wedge \text{FuelLineOK} \supset \text{GasInEngine}$
 $\text{GasInEngine} \wedge \text{GoodSpark} \supset \text{EngineRuns}$
 $\text{PowerToPlugs} \wedge \text{PlugsClean} \supset \text{GoodSpark}$
 $\text{BatteryCharged} \wedge \text{CablesOK} \supset \text{PowerToPlugs}$

Observed:

$\neg \text{EngineRuns}$,
 GasInTank , PlugsClean , BatteryCharged

Prove:

$\neg \text{FuelLineOK} \vee \neg \text{CablesOK}$

Solution by Forward Chaining

Knowledge Base and Observations:

~~$(\neg \text{GasInTank} \vee \neg \text{FuelLineOK} \vee \text{GasInEngine})$~~
 ~~$(\neg \text{GasInEngine} \vee \neg \text{GoodSpark} \vee \text{EngineRuns})$~~
 ~~$(\neg \text{PowerToPlugs} \vee \neg \text{PlugsClean} \vee \text{GoodSpark})$~~
 $(\neg \text{BatteryCharged} \vee \neg \text{CablesOK} \vee \text{PowerToPlugs})$
 $(\neg \text{EngineRuns})$
 (GasInTank)
 (PlugsClean)
 (BatteryCharged)

Negation of Conclusion:

(FuelLineOK)
 (CablesOK)

Resolution

$$\{ (p \vee \alpha), (\neg p \vee \beta) \} \vdash_R (\alpha \vee \beta)$$

Defined for clauses (CNF)
 α, β any disjunctions of literals
 Remove any repeated literals from conclusion

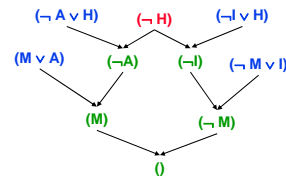
Provides a **complete** proof method for
 refutation style proofs in clausal logic
All true sentences can be derived!

Resolution Proof

DAG, where leaves are input clauses

Internal nodes are resolvents

Root is false (empty clause)



KB:

- If the unicorn is mythical, then it is immortal,
- if it is not mythical, it is an animal
- If the unicorn is either immortal or an animal, then it is horned.

Prove: the unicorn is horned.

Conversion to CNF

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$

1. Eliminate \Leftrightarrow , replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$$

2. Eliminate \Rightarrow , replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$$

3. Move \neg inwards using de Morgan's rules and double-negation:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$$

4. Apply distributivity law (\vee over \wedge) and flatten:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$$

Chapter 6, AI/AA/AS Chapter 7 10

New Variable Trick

Putting a formula in clausal form may increase its size exponentially

But can avoid this by introducing dummy variables

$$(a \wedge b \wedge c) \vee (d \wedge e \wedge f) \Rightarrow$$

$$(g \vee h)$$

$$(\neg a \vee \neg b \vee \neg c \vee g), (\neg g \vee a), (\neg g \vee b), (\neg g \vee c)$$

$$(\neg d \vee \neg e \vee \neg f \vee h), (\neg h \vee d), (\neg h \vee e), (\neg h \vee f)$$

Dummy variables don't change satisfiability!

Efficient Local Search for Satisfiability Testing

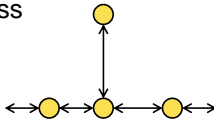
Greedy Local Search for SAT: GSAT

```
state = choose_start_state();
while ! GoalTest(state) do
  state := arg min { h(s) | s in Neighbors(state) }
end
return state;
```

- start = random truth assignment
- GoalTest = formula is satisfied
- h = number of false (unsatisfied) clauses
- neighbors = flip one variable (from true to false, or from false to true)

Smarter Noise Strategies

- For both random noise and simulated annealing, nearly all uphill moves are useless



- Can we find uphill moves that are more likely to be helpful?
- At least for SAT we can...

Random Walk for SAT

- Observation: if a clause is unsatisfied, at least one variable in the clause must be different in any global solution
($A \vee \sim B \vee C$)
- Suppose you randomly pick a variable from an unsatisfied clause to flip. What is the probability this was a good choice?

Random Walk for SAT

- Observation: if a clause is unsatisfied, at least one variable in the clause must be different in any global solution
($A \vee \sim B \vee C$)
- Suppose you randomly pick a variable from an unsatisfied clause to flip. What is the probability this was a good choice?

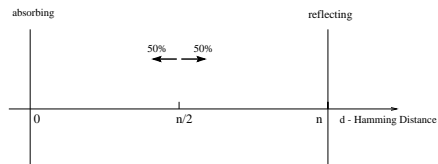
$$\Pr(\text{good choice}) \geq \frac{1}{\text{clause length}}$$

Random Walk Local Search

```
state = choose_start_state();
while ! GoalTest(state) do
  clause := random member { C | C is a clause of F and
                           C is false in state }
  var := random member { x | x is a variable in clause }
  state[var] := 1 - state[var];
end
return state;
```

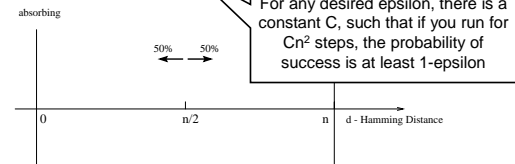
Properties of Random Walk

- If clause length = 2:
 - 50% chance of moving in the right direction
 - Converges to optimal with high probability in $O(n^2)$ time



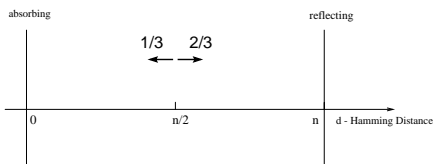
Properties of Random Walk

- If clause length = 2:
 - 50% chance of moving in the right direction
 - Converges to optimal with high probability in $O(n^2)$ time



Properties of Random Walk

- If clause length = 3:
 - 1/3 chance of moving in the right direction
 - Exponential convergence
 - Compare pure noise: $1/(n-\text{Hamming distance})$ chance of moving in the right direction
 - The closer you get to a solution, the more likely a noisy flip is bad



Greedy Random Walk

```

state = choose_start_state();
while ! GoalTest(state) do
  clause := random member { C | C is a clause of F and
                           C is false in state };
  with probability noise do
    var := random member { x | x is a variable in clause };
  else
    var := arg x min { #unsat(s) | x is a variable in clause,
                      s and state differ only on x };
  end
  state[var] := 1 - state[var];
end
return state;
    
```

Refining Greedy Random Walk

- Each flip
 - **makes** some false clauses become true
 - **breaks** some true clauses, that become false
- Suppose $s_1 \rightarrow s_2$ by flipping x . Then:
 - $\#unsat(s_2) = \#unsat(s_1) - \text{make}(s_1, x) + \text{break}(s_1, x)$
- **Idea 1:** if a choice breaks nothing, it is very likely to be a good move
- **Idea 2:** near the solution, only the break count matters
 - the make count is usually 1

Walksat

```

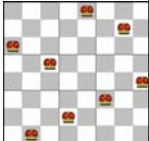
state = random truth assignment;
while ! GoalTest(state) do
  clause := random member { C | C is false in state };
  for each x in clause do compute break[x];
  if exists x with break[x]=0 then var := x;
  else
    with probability noise do
      var := random member { x | x is in clause };
    else
      var := arg x min { break[x] | x is in clause };
  endif
  state[var] := 1 - state[var];
end
return state;
    
```

Put everything inside of a restart loop.
Parameters: noise, max_flips, max_runs

SAT Translation of N-Queens

- At least one queen each row:
 $(Q_{11} \vee Q_{12} \vee Q_{13} \vee \dots \vee Q_{18})$
 $(Q_{21} \vee Q_{22} \vee Q_{23} \vee \dots \vee Q_{28})$
 ...
 $O(N^2)$ clauses

- No attacks:
 $(\neg Q_{11} \vee \neg Q_{12})$
 $(\neg Q_{11} \vee \neg Q_{22})$
 $(\neg Q_{11} \vee \neg Q_{21})$
 ...
 $O(N^3)$ clauses



Walksat Today

- Hard random 3-SAT: 100,000 vars, 15 minutes
 - Walksat (or slight variations) winner every year in "random formula" track of International SAT Solver Competition
 - Complete search methods: 700 variables
- Graph coloring, Latin squares, N-queens \approx 30,000 variables
 - But backtracking algorithms can be better for other interesting classes of problems
- Inspired huge body of research linking SAT testing to statistical physics (spin glasses)

Efficient Backtrack Search for Satisfiability Testing

Basic Backtrack Search for a Satisfying Model

Solve(F): return Search(F, { });

Search(F, assigned):

```

if all variables in F are in assigned then
  if evaluate(F, assigned) then return assigned;
  else return FALSE;
choose unassigned variable x;
return Search(F, assigned U {x=0}) ||
       Search(F, assigned U {x=1});
end;
    
```

State Space:

All partial or complete assignments of truth values to variables

Propagating Constraints

- Suppose formula contains
 $(A \vee B \vee \neg C)$
 and we set $A=0$.
- What is the resulting constraint on the remaining variables B and C?
 $(B \vee \neg C)$
- Suppose instead we set $A=1$. What is the resulting constraint on B and C?
No constraint

Empty Clauses and Formulas

- Suppose a clause in F is shortened until it become empty. What does this mean about F and the partial assignment?
F cannot be satisfied by any way of completing the assignment; must backtrack
- Suppose all the clauses in F disappear. What does this mean?
F is satisfied by any completion of the partial assignment

Unit Propagation

- Suppose a clause in F is shortened to contain a single literal, such as
(A)

What should you do?

*Immediately add the literal to assigned.
Repeat if another single-literal clause appears.*

- Applying resolution where one clause is a single literal is called **unit propagation**

DPLL

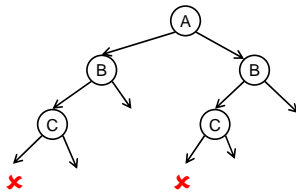
DPLL(F , assigned):

```

while F has a unit clause (c) do
  assigned = assigned U {c};
  shorten clauses containing -c;
  delete clauses containing c;
end
if F is empty then return assigned;
if F contains an empty clause then return FALSE;
choose an unassigned literal c; // variable and initial value
return Search(F U { (c) }, assigned) ||
       Search(F U { (-c) }, assigned);
end;
  
```

Improving Efficiency: Clause Learning

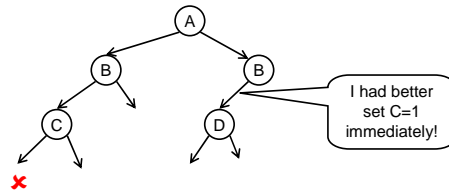
- Idea: backtrack search can repeatedly reach an empty clause (backtrack point) for the same reason



Example: Propagation from $B=0$ and $C=0$ leads to empty clause

Improving Efficiency: Clause Learning

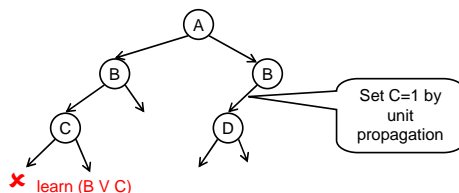
- If reason was remembered, then could avoid having to rediscover it



Example: Propagation from $B=0$ and $C=0$ leads to empty clause

Improving Efficiency: Clause Learning

- The reason can be remembered by adding a new **learned clause** to the formula



Example: Propagation from $B=0$ and $C=0$ leads to empty clause

Scaling Up

- Clause learning greatly enhances the power of unit propagation
- Tradeoff: memory needed for the learned clauses, time needed to check if they cause propagations
- Clever data structures enable modern SAT solvers to manage **millions** of learned clauses efficiently

Satplan in 15 Seconds

- Time = bounded sequence of integers
- Translate planning operators to propositional schemas that assert:
 - $action(i) \supset pre(i) \wedge effect(i+1)$
 - $\neg action_1(i) \vee \neg action_2(i)$ if interfering
 - $action_1$ negates a precondition of $action_2$
 - $\neg fact(i) \wedge fact(i+1) \supset action_1(i) \vee action_2 \vee \dots$
 - frame axioms
- initial_state₀, goal_state_n

Demo: SatPlan

Progress in SAT Solvers

- **Sato**: head/tail data structure: sub-linear time unit propagation
- **Satz**: select branch variable to maximize unit propagation
- **zChaff**: clause learning + head/tail
- **Jerusat**: use information from clause learning to help select branch variable
- **MiniSat**: open source SAT solver, easy to modify

wff	vars	clauses	sato 1997	satz 1997	zChaff 2001	jerusat 2003	siege 2003	MiniSat 2005
p05	3,656	31,089	13.23	0.61	0.01	0.01	0.01	0.02
p15	10,671	143,838	x	4.85	0.05	0.13	0.03	0.09
p18	34,325	750,269	x	x	13.92	6.59	4.85	2.55
p20	40,304	894,643	x	x	14.75	10.35	8.68	10.03
p28	249,738	13,849,105	x	x	846.72	79.59	12.74	27.80

Explaining the Success of DPLL: Backdoors to Tractability

Informally:

A backdoor to a given problem is a subset of the variables such that once they are assigned values, the polynomial propagation mechanism of the SAT solver solves the remaining formula.

Formal definition includes the notion of a “subsolver”:
a polynomial simplification procedure with certain general characteristics found in current DPLL SAT solvers.

Backdoors correspond to “clever reasoning shortcuts” in the search space.

Backdoors can be surprisingly small:

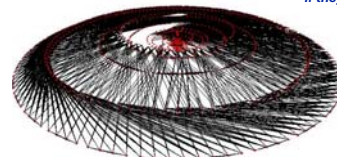
instance	# vars	# clauses	backdoor	fract.
logistics.d	6783	437431	12	0.0018
3bitadd_32	8704	32316	53	0.0061
pipe_01	7736	26087	23	0.0030
qg_30_1	1235	8523	14	0.0113
qg_35_1	1597	10658	15	0.0094

Most recent: Other combinatorial domains. E.g. graphplan planning, near constant size backdoors (2 or 3 variables) and $\log(n)$ size in certain domains. (Hoffmann, Gomes, Selman '04)

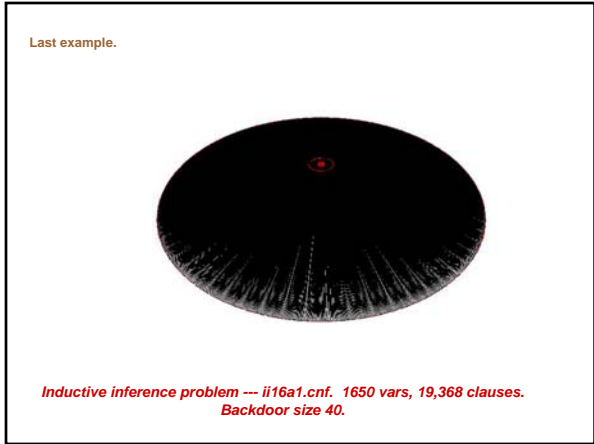
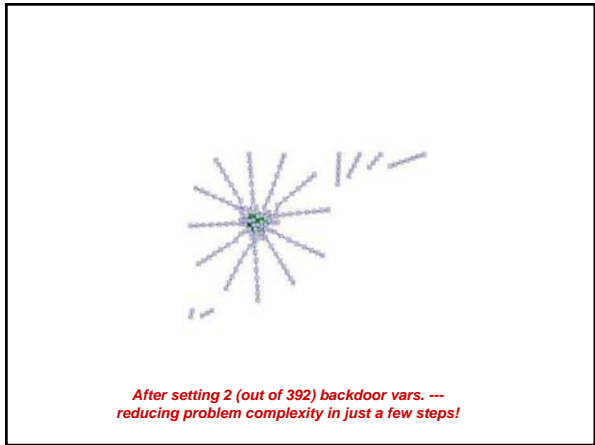
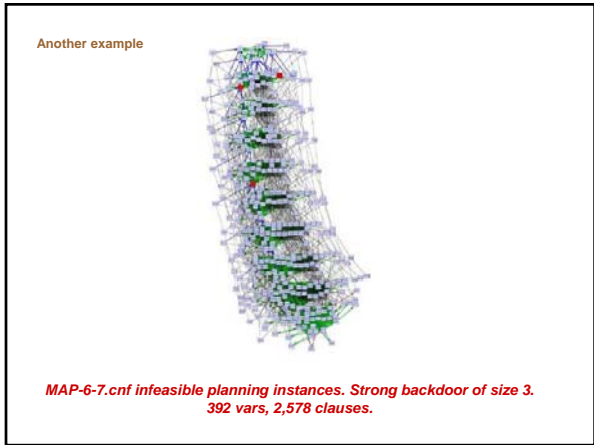
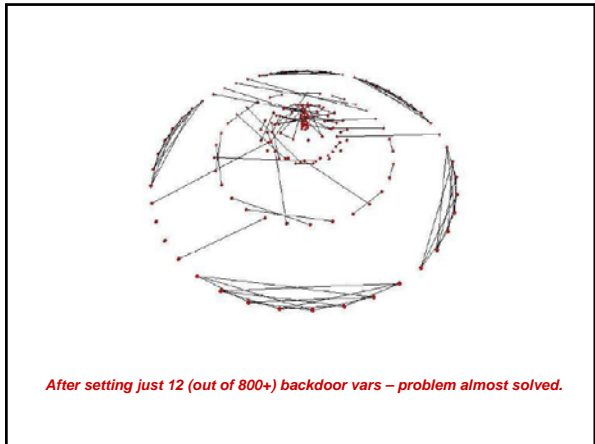
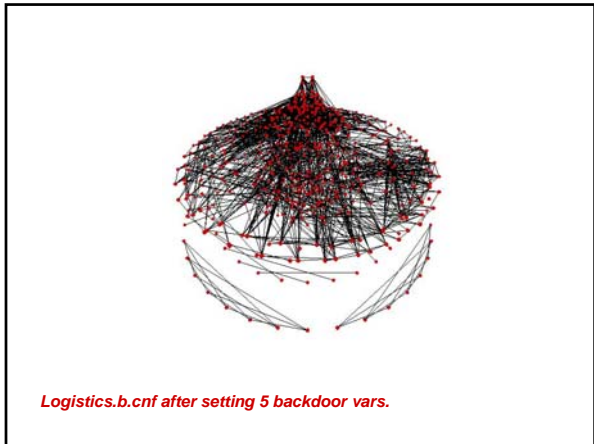
Backdoors capture critical problem resources (bottlenecks).

Backdoors --- “seeing is believing”

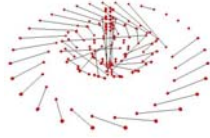
Constraint graph of reasoning problem.
One node per variable:
edge between two variables
if they share a constraint.



Logistics_b.cnf planning formula.
843 vars, 7,301 clauses, approx min backdoor 16
(backdoor set = reasoning shortcut)



Some other intermediate stages:



After setting 38 (out of 1600+) backdoor vars:

Good branching heuristics are ones that frequently identify backdoor variables!

