Propositional Logic

CSCE 420 – Spring 2023

read: Ch. 7

Knowledge-based programming

- As Feigenbaum said, one route to designing intelligent systems is to give them knowledge (expertise) to solve problems
- We need a general language for encoding knowledge.
- English is not sufficient (too ambiguous)
- The history of AI is tied to the search for higher-level languages that are “more expressive”.
  - AI drove advances in functional, object-oriented, and logic programming
  - LISP, Prolog, Smalltalk...

Logic

- Logic has become the standard for expressing knowledge in KBS
- Logic has advantages over procedural languages
  - context-independence: easier to judge correctness of a rule that 1 line buried in code (hence, easier to debug and maintain)
  - logic has a well-defined semantics (not subject to order, global variables, side-effects...)
  - rules can be used in many different ways (lots of different inferences)
- declarative vs. procedural programming: say “what”, not “how”
  - procedural languages require you to say HOW to do something
  - declarative languages let you describe the world, and the system can autonomously figure out the right thing to do as a consequence of the situation
- KBS: Knowledge-Based Systems
  - programming by writing “rule bases”

Example: Driving

- think about all the knowledge and inference you use while driving...
  - laws
  - mechanics of vehicle operation
  - right of away, turn signal, yellow lights...
  - safety (speed, following distance , changing lanes, pedestrians)
  - slippery roads, fire trucks, school buses...
  - other drivers: do they see you? can you infer their intentions? are they displaying erratic behavior?
- it is better to put this all in a giant KB, rather than trying to program an enormous if-then-else to handle all possible situations

Inference Algorithms

- Of course, we need a way to extract conclusions and make decisions from a rule base
- synonyms: "inference", "automated deduction", "theorem-proving"
- Inference algorithms are a foundation for Expert Systems
- expert systems “shells” are the architecture/environment in which you:
    1. load your rule base
    2. describe current situation
    3. ask questions...or what to do...or whether something is a consequence...
    4. get an explanation of the information used to get the answer (i.e. “proof”)

Defining a "logic"

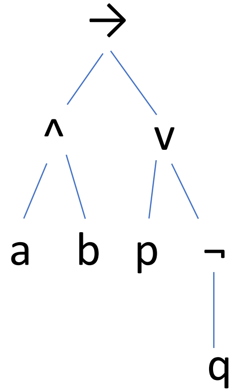
- There are actually many types of logics
  - propositional/Boolean logic
  - First-order logic (FOL), higher-order logics...
  - modal logics, epistemic logics (beliefs), temporal logics (used for program analysis)...
  - fuzzy logics, probabilistic logics...
  - non-monotonic logics (default reasoning, abduction)
- These logics differ in *expressiveness* and *computational complexity*
  - First-order logic (FOL) is the *lingua franca* for most KR in AI
- Each of these logic has its own:
  1. **Syntax** – the rules defining what sentences are legal expressions
  2. **Semantics** – defines "truth" of sentences, and relationship of meaning between sentences
  3. **Proof theory** – a method for answering queries

Syntax of Propositional Logic

- well-defined sentences
- atomic sentences = propositional symbols (A, P, battery_low, lights_on_room124)
- complex sentences: generated using operators
  - binary opers: and (^), or (v), xor (⊕), implication (→), biconditional (↔)
  - unary oper: negation (¬)
  - parentheses

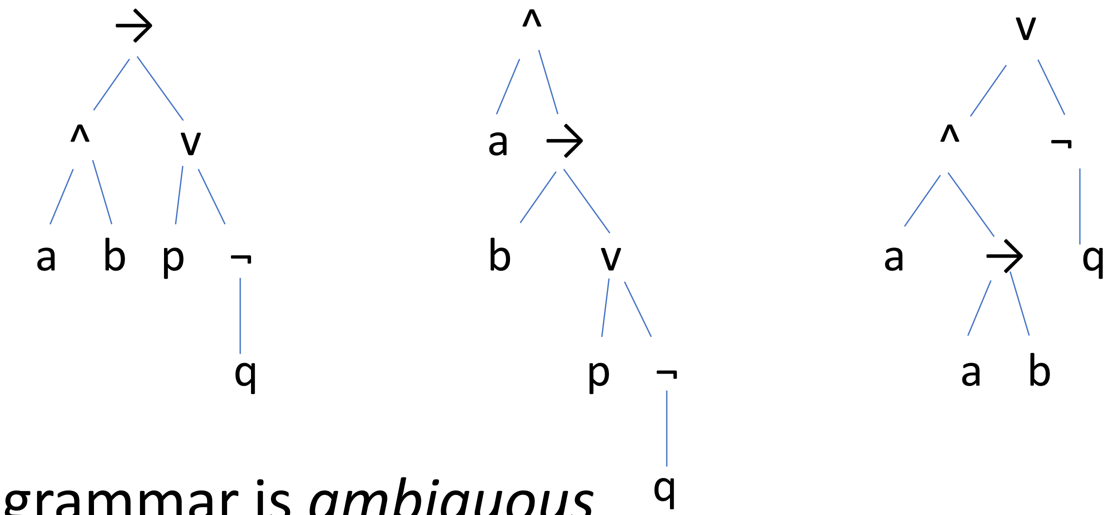
Syntax

- BNF grammar (Backus-Naur Form, production rules)
  - atomic ::= <prop> // tokens like “P”, “gas_tank_filled”...
  - binop ::= ^ | v | →|↔| ⊕
  - complex ::= <atomic> | ¬ complex | <complex> <binop> <complex> | (<complex>)
- examples:
  - legal: P, PvQ, ¬¬X^¬¬Y, ¬(¬X^(¬(¬Y)), Light ↔ Dark
  - not syntactically legal: "Win vv Lose", "(→Draw)"
- these can be used to derive the parse tree(s) for an expression
  - aka Abstract Syntax Trees (ASTs)
  - (I’m not going to give the algorithm for parsing this grammar here...)
  - example: a^b→pv¬q



Syntax

- of course, there are other possible parse trees...
  - a^b→pv¬q



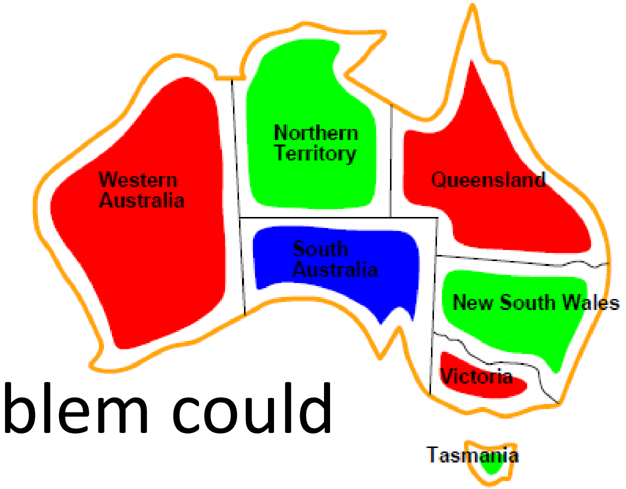
  - the grammar is *ambiguous*
  - one can always disambiguate an expression by adding parentheses
    - (a^b)→(pv¬q)   vs   a^(b→pv¬q)   vs   a^(b→p)v¬q

Syntax

- ...or one can rely on rules of precedence among operators
  - ¬ (highest)
  - ^
  - v, ⊕
  - →,↔ (lowest)
- There can still be parsing ambiguity: AvBvC = (AvB)vC or Av(BvC) ?
- each operator is left-associative: (AvB)vC
- these syntax rules are similar to mathematics:
  - all opers are left associative, except ^ (which is right associative)
  - 1+2*3/4/5+6^7^2 = (1+(((2*3)/4)/5))+(6^(7^2))
  - 1-2+3 = ? (2 or -4?)

- (unary minus) (highest)
^ (exponentiation)
*, /
+, -
=,<,> (lowest)

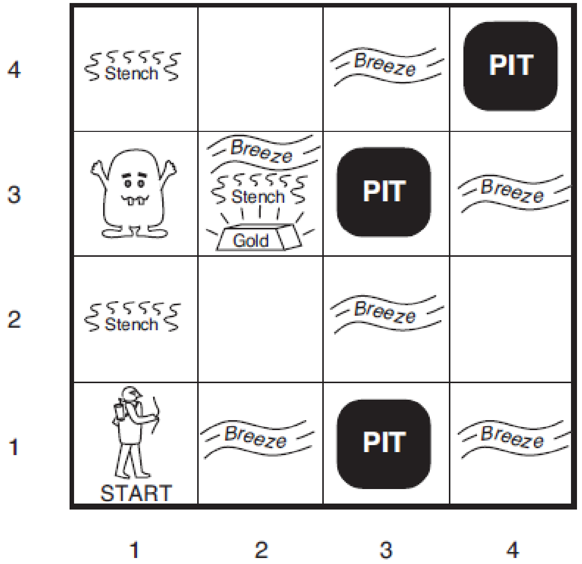
Example: Map-coloring

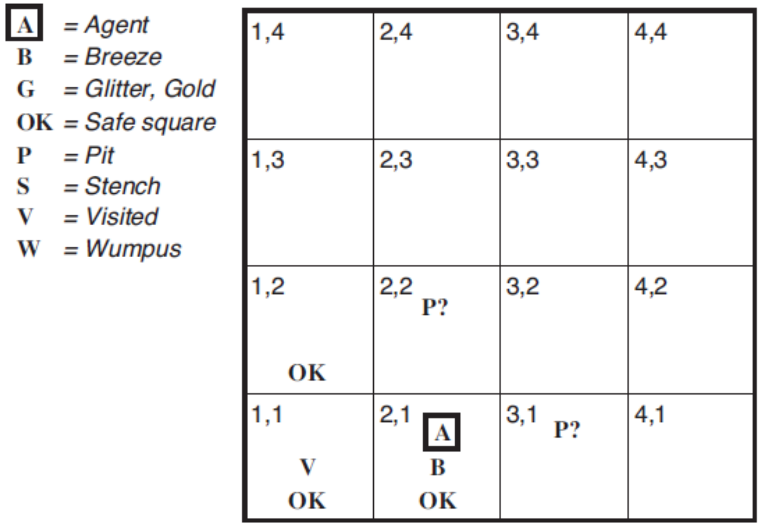


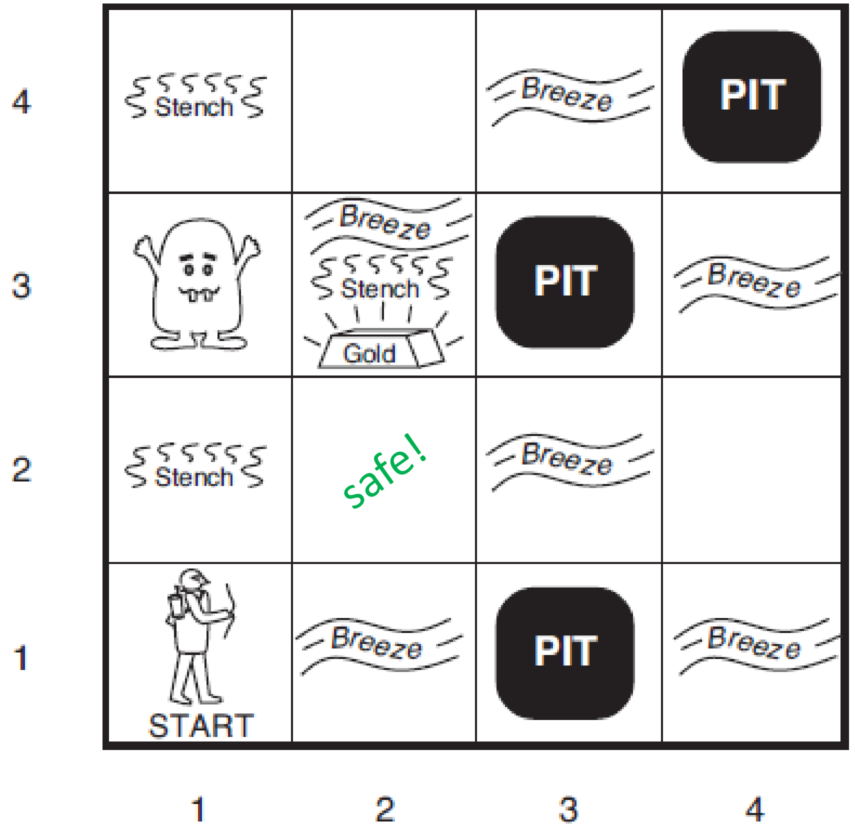
- A propositional encoding of the Australia map-color problem could look like this:
  - propositional symbols: WAR (Western Australia is Red), WAG, WAB, NTR, NTG, NTB (Northern Territories is Blue), QR, QG, QB...
  - KB:
    - WAR v WAG v WAB, NTR v NTG v NTB, QR v QG v QB... // each state is 1 of 3 colors
    - WAR→¬WAG^¬WAB, WAG→¬WAR^¬WAB, WAB→¬WAR^¬WAG,... // at most 1 color
    - // adjacent states must be different colors
    - WAR →¬NTR^¬SAR, WAG→¬NTG^¬SAG, WAB→¬NTB^¬SAB...
    - NTR →¬WAR^¬SAR^¬QR...
  - note: KB $\not\models$ WAR (does not entail)
  - however, KB $\models$ WAB→VB, and KB ∪ {WAB} $\models$ VB

Example: Wumpus World

- the goal of the agent is to find the gold without falling in a pit or getting eaten by the wumpus (there is only 1, and it can't move)
- the agent does not know a priori where the pits or wumpus are located
- the agent can only sense breezes, stenches, and glitter
- a breeze is felt in rooms adjacent to pits, and a stench can be sensed in rooms adjacent to the wumpus
- a room is *safe* to explore if it is known not have the pit wumpus







- after visiting rooms (1,1), (1,2), and (2,1), the agent should be able to infer that room (2,2) is safe
- KB∪{S12,B21} ⊨ safe22

We can use propositions like:

- W44="the wumpus is in room 4,4"
- B22 = "there is a breeze in 2,2"
- S13 = "there is a stench in 1,3"
- pit32 = "there is a pit in room 3,2"
- safe33 = "room 3,3 is safe"

KB = {

1. W11->S21^S12	17. P11->B21^B12	33. -W11^-P11->safe11
2. W12->S22^S11^S13	18. P12->B22^B11^B13	34. -W12^-P12->safe12
3. W13->S23^S12^S14	19. P13->B23^B12^B14	35. -W13^-P13->safe13
4. W14->S24^S13	20. P14->B24^B13	36. -W14^-P14->safe14
5. W21->S11^S31^S22	21. P21->B11^B31^B22	37. -W21^-P21->safe21
6. W22->S12^S32^S21^S23	22. P22->B12^B32^B21^B23	38. -W22^-P22->safe22
7. W23->S13^S33^S22^S24	23. P23->B13^B33^B22^B24	39. -W23^-P23->safe23
8. W24->S14^S34^S23	24. P24->B14^B34^B23	40. -W24^-P24->safe24
9. W31->S21^S41^S32	25. P31->B21^B41^B32	41. -W31^-P31->safe31
10. W32->S22^S42^S31^S33	26. P32->B22^B42^B31^B33	42. -W32^-P32->safe32
11. W33->S23^S43^S32^S34	27. P33->B23^B43^B32^B34	43. -W33^-P33->safe33
12. W34->S24^S44^S33	28. P34->B24^B44^B33	44. -W34^-P34->safe34
13. W41->S31^S42	29. P41->B31^B42	45. -W41^-P41->safe41
14. W42->S32^S41^S43	30. P42->B32^B41^B43	46. -W42^-P42->safe42
15. W43->S33^S42^S44	31. P43->B33^B42^B44	47. -W43^-P43->safe43
16. W44->S34^S43	32. P44->B34^B43	48. -W44^-P44->safe44

}

Semantics

- semantics refers to “meaning” of sentences, and relationships among them
  - this is defined using Model Theory
  - models describe states of the world, and are used to give “interpretations” of sentences
- Truth-functional semantics
  - each sentence is assumed to be either True or False in the world
  - (Law of the Excluded Middle) – there is no in-between
  - propositions correspond to “facts” about the state of the world, which can only be True or False
    - good example: plutoCold, mercuryCold (in our universe, the first is T, the second is F)
    - bad example: surface_temp_of_pluto (value can only be T or F)

also, uncertainty will be handled via multiple models...

- in Propositional Logic, *models* are *truth assignments* over all propositional symbols (that appear in the KB)
  - {A=F, B=F, C=T ... P=T, Q=F, R=F}
  - {mercuryCold=F, mercuryWarm=F, mercuryHot=T...earthCold=F, earthWarm=T, earthHot=F... plutoCold=T, plutoWarm=F, plutoHot=F}

Semantics

- Compositionality
  - a model defines the truth value for all atomic sentences
  - given a model, the truth value of ANY sentence can be computed by combining truth values of sub-sentences using *truth tables*
  - these are pretty straightforward in PropLog (except for →)

A	B	¬A	AvB	A^B	A⊕B	A→B	A↔B
T	T	F	T	T	F	T	T
T	F	F	T	F	T	**F**	F
F	T	T	T	F	T	T	F
F	F	T	F	F	F	T	T

P→Q: if the LHS (antecedents) are F, it doesn't matter what the RHS (consequent) is; only T->F is disallowed

in a model M={p=F , q=F , r=T}:

¬pv(q^r)	(¬F)v(F^T) = TvF = T
p→(q→p)	F→(F→F) = F→T = T
p^¬p	F^(¬F) = F^T = F

here's how you figure these out...

Semantics

- We say a model M *satisfies* a sentence s iff the interpretation (or truth value) of s in M is True
- a sentence s is *satisfiable* if there is at least 1 model that satisfies it
- a sentence s is *unsatisfiable* if no model that satisfies it
- a sentence s is a *tautology* (or *valid*) if it is satisfied by ALL models
- examples:
  - satisfiable: X, Xv¬YvZ, (X^¬Y)→Z (*what models make each of these True?*)
  - unsatisfiable: X^¬X, P⊕P, ¬Q↔Q (*convince yourself there are no models*)
  - tautologies: Av¬A, P→(Q→P) (*will be True in any model*)

Semantics

- semantic relationship between 2 sentences (or sets of sentences)
  - examples: {P} and {¬P} can't both be True (i.e. satisfied by same models)
  - if {PvQ, P→R, Q→S, ¬P} is True, then {S} must be True
- two sentences are *semantically equivalent* if they are satisfied by exactly the same models: $\alpha \equiv \beta$ iff $M(\alpha)=M(\beta)$
  - example: $A \rightarrow B \equiv \neg A \vee B$
- note: a set of sentences is True (or satisfied in a model) iff each sentence is True (equivalent to an implicit conjunction)

{PvQ ^ P→R ^ Q→S ^ ¬P}

- **entailment**
  - captures the notion of "logical consequence"
  - $\alpha \models \beta$ iff all models that satisfy α also satisfy β

Semantics

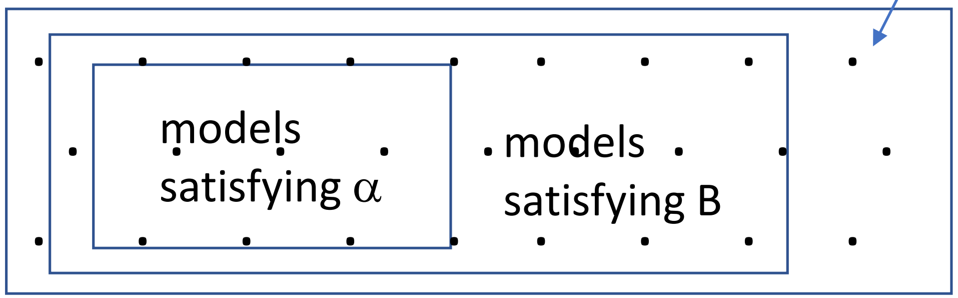
- show that {PvQ, Q→R, ¬P} ⊨ {R}
  - models that satisfy the premises (as a conjunction): {M3}
  - models the satisfy the consequents ({R}): {M1,M3,M5,M7}
  - {M3} ⊆ {M1,M3,M5,M7}

	P	Q	R	PvQ	Q→R	¬P	{PvQ, Q→R, ¬P}
M0	0	0	0	0	1	1	0
M1	0	0	1	0	1	1	0
M2	0	1	0	1	0	1	0
M3	0	1	1	1	1	1	1
M4	1	0	0	1	1	0	0
M5	1	0	1	1	1	0	0
M6	1	1	0	1	0	0	0
M7	1	1	1	1	1	0	0

Semantics

- Deduction Theorem
  - note that $P \rightarrow Q$ is valid looks like $P \models Q$, and they seem similar
  - but they are different:
    - $P \rightarrow Q$ is a sentence (defined by syntax)
    - $P \models Q$ means P entails Q (defined by semantics)
  - the Deduction Theorem shows that they are related:
    - $\alpha \models \beta$ iff $\alpha \rightarrow \beta$ is valid
    - (=>) see Venn diagram

a model M(i):
{A=F, B=T, C=F
D=T, E=F...}



in all models, either α is false or β is true, hence $\alpha \rightarrow \beta$

    - (<=) if $\alpha \rightarrow \beta$ is valid, then all models satisfy it, so all models either make α false or β true; hence those model that satisfy α also satisfy β; hence $\alpha \models \beta$

Inference

- is there a procedure to determine if $\alpha \models \beta$? (or KB $\models$ query)
- **model-checking**
  - of course, we can just enumerate all models and check if those satisfy α also satisfy β
  - how many models are there? 2^n (for *n* propositional symbols)
  - it is *finite*, so the procedure will halt and return yes (entailed) or no

- there are methods called ‘analytic tableau’ that allow you to do model-checking more efficiently by building a ‘semantic tree’ that represents different cases/possibilities, which can be used to construct a model of a set of sentences, or show that there are none (all branches closed->unsatisfiable); but these are still exponential in the worst case

Inference

- model-checking is inefficient
  - we need a more practical procedure to determine whether $\alpha \models \beta$
- Proof Procedures: methods to determine whether $\alpha \models \beta$ *purely by syntactic manipulation*
  - aka “Inference Methods”, “Theorem-Proving”, “Automated Deduction”...
- Propositional Rules of Inference (ROI)
  - rules for generating new sentences from old sentences
  - a *sound* ROI only generates new sentences that are entailed
  - in this context, ‘$\vdash$’ means ‘derives’ by a ROI, i.e. ‘$\alpha \vdash \beta$’ means ‘β is derived from α’
  - hence a rule R is sound iff for all sentences α,β, if $\alpha \vdash \beta$, then $\alpha \models \beta$
  - an ROI $\alpha \vdash \beta$ is *truth preserving* if the derived sentence β is semantically equivalent to α (satisfied by exactly the same models)

Rules of Inference

- example: Modus Ponens
  - from P and P→Q, we can derive Q
  - {P,P→Q} ⊢ Q
  - is MP sound?
  - all the models that satisfy the premises (conjunction of P and P→Q) also satisfy the derived sentence Q, so Q is entailed, so MP is sound

P	Q	P→Q	premises (conj)	derived (Q)
0	0	1	0	0
0	1	1	0	1
1	0	0	0	0
1	1	1	1	1

Rules of Inference

These are inference 'schemas'.
A, B, and C are patterns representing sub-sentences.

Can you prove that each of these is sound?
Make truth tables.
AE and MP are easy.
Resolution is worth doing...

	from this...	derive this...	comments
AndElimination (AE)	A^B	A	
AndIntroduction (AI)	A, B	A^B	
OrIntroduction	A, B	AvB	no such things as OrElimination!
Commutativity	A^B	B^A	truth-preserving
Distributivity	Av(B^C) A^(BvC)	(AvB)^(AvC) (A^B)v(A^C)	
DoubleNegationElim (DN)	¬¬A	A	
DeMorgan's Laws (DM)	¬(AvB) ¬(A^B)	¬A^¬B ¬Av¬B	flip the operator
ImplicationElimination (IE)	A→B	¬AvB	truth-preserving
Modus Ponens (MP)	A, A→B	B	pattern-matching, if LHS is matched, can derive RHS
Modus Tolens	A→B, ¬B	¬A	
contraposition	A→B	¬B→¬A	
Resolution	AvB, ¬AvC	BvC	requires 2 clauses with opposite literals

Natural Deduction

- Proof procedure to show that $\alpha \models \beta$
  - start by listing sentences in premise α
  - derive additional sentences using sound ROI
  - must be a *finite* sequence of steps ending in β
- number your sentences
- label each new sentence with ROI and sentences it was derived from

premises:

α_1
..
α_n

derived sentences:

γ_{n+1} (from 3 using AE)
γ_{n+2} (from 5 and 7 using MP)
...
β

Example of Nat Ded (1)

from this...

$$P \Rightarrow Q$$
$$L \wedge M \Rightarrow P$$
$$B \wedge L \Rightarrow M$$
$$A \wedge P \Rightarrow L$$
$$A \wedge B \Rightarrow L$$
$$A$$
$$B$$

...show that Q is entailed

premises

1. P→Q
2. L^M→P
3. B^L→M
4. A^P→L
5. A^B→L
6. A
7. B

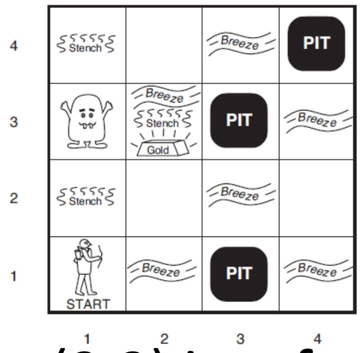
derivations

8.	A^B	[AndIntr, 6,7]
9.	L	[MP, 5,8]
10.	B^L	[AndIntr, 7,9]
11.	M	[MP, 10,3]
12.	L^M	[AndIntr, 9,11]
13.	P	[MP, 12,2]
14.	¬PvQ	[ImplElim, 1]
15.	Q	[Reso, 13,14]

Natural Deduction

- Why does Natural Deduction work?
  - we have to show that if Q is derived after a sequence of steps γ_1,... γ_n (and hence Q is entailed by KB$\cup${γ_1,... γ_n}), then KB $\models$ Q
  - whenever we derive a new sentence by a sound ROI and add it to the premises, the set of entailments stays the same (or possibly grows)
  - suppose KB $\models$ Q, and M(KB)$\subseteq$M(Q)
  - so if KB $\vdash \gamma_1$ (an intermediate sentence in proving Q) by a sound ROI, then KB $\models \gamma_1$, so M(KB)$\subseteq$M(γ_1), M(KB$\cup\gamma_1$)=M(KB)$\cap$M(γ_1)=M(KB), so KB$\cup${γ_1} $\models$ Q
  - this property is known as "*monotonicity*" (i.e. adding entailed intermediates doesn't affect what else is entailed)
  - similarly, if KB$\cup\gamma_1\vdash\gamma_2$, then KB$\cup${$\gamma_1$} $\models \gamma_2$ and M(KB$\cup${γ_1, γ_2})$\subseteq$M(Q), so KB$\cup${γ_1 , γ_2} $\models$ Q
  - if proof has n+1 steps KB, γ_1,... γ_{n1},Q, then KB$\cup${γ_1,... γ_n} $\models$ Q (by induction)

Example of Nat Ded (2): Wumpus World



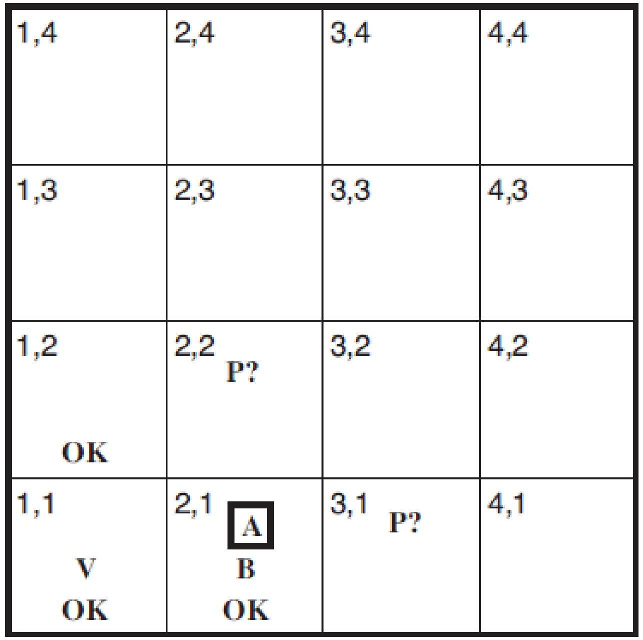
after visiting rooms (1,1), (1,2), and (2,1), the agent should be able to infer that room (2,2) is safe

KB = {

1. W11->S21^S12	17. P11->B21^B12	33. -W11^-P11->safe11
2. W12->S22^S11^S13	18. P12->B22^B11^B13	34. -W12^-P12->safe12
3. W13->S23^S12^S14	19. P13->B23^B12^B14	35. -W13^-P13->safe13
4. W14->S24^S13	20. P14->B24^B13	36. -W14^-P14->safe14
5. W21->S11^S31^S22	21. P21->B11^B31^B22	37. -W21^-P21->safe21
6. W22->S12^S32^S21^S23	22. P22->B12^B32^B21^B23	38. -W22^-P22->safe22
7. W23->S13^S33^S22^S24	23. P23->B13^B33^B22^B24	39. -W23^-P23->safe23
8. W24->S14^S34^S23	24. P24->B14^B34^B23	40. -W24^-P24->safe24
9. W31->S21^S41^S32	25. P31->B21^B41^B32	41. -W31^-P31->safe31
10. W32->S22^S42^S31^S33	26. P32->B22^B42^B31^B33	42. -W32^-P32->safe32
11. W33->S23^S43^S32^S34	27. P33->B23^B43^B32^B34	43. -W33^-P33->safe33
12. W34->S24^S44^S33	28. P34->B24^B44^B33	44. -W34^-P34->safe34
13. W41->S31^S42	29. P41->B31^B42	45. -W41^-P41->safe41
14. W42->S32^S41^S43	30. P42->B32^B41^B43	46. -W42^-P42->safe42
15. W43->S33^S42^S44	31. P43->B33^B42^B44	47. -W43^-P43->safe43
16. W44->S34^S43	32. P44->B34^B43	48. -W44^-P44->safe44

}

Natural Deduction Proof of KB^Facts |= safe22:



Facts = { 49. -B11, 50. -S11, 51. -B12, 52. S12, 53. B21, 54. -S21 }

55. -W22v(S12^S32^S21^S23) [Impl Elim, 6]
56. (-W22vS12)^ (-W22vS32))^ (-W22vS21)^ (-W22vS23) [Distrib, 55]
57. (-W22vS21) [And Elim, 56]
58. S21v-W22 [Commut, 57]
59. -S21->-W22 [Impl Intro, 58]
60. **-W22** [MP, 59, 54]
61. -P22v(B12^B32^B21^B23) [Impl Elim, 22]
62. (-P22vB12)^ (-P22vB32))^ (-P22vB21)^ (-P22vB23) [Distrib, 61]
63. -P22vB12 [And Elim, 62]
64. B12v-P22 [Commut, 63]
65. -B12->-P22 [Impl Intro, 64]
66. **-P22** [MP, 65, 51]
67. -W22^-P22 [And Intro, 60, 66]
68. **safe22** [MP, 38, 67]

Natural Deduction

- limitations
  - it can be difficult (but not impossible) to find the right sequence of derivations automatically
    - you could use a Search and try applying all ROI to all combinations of sentences till you generate the query (i.e. as the goal)
  - in theory, is Nat Ded a *complete* proof procedure???
    - can every query that is entailed by a KB be proved in a finite number of steps?
  - one can show that certain combinations of ROI are sufficient to guarantee that a proof always exists for entailed sentences (in Propositional Logic)

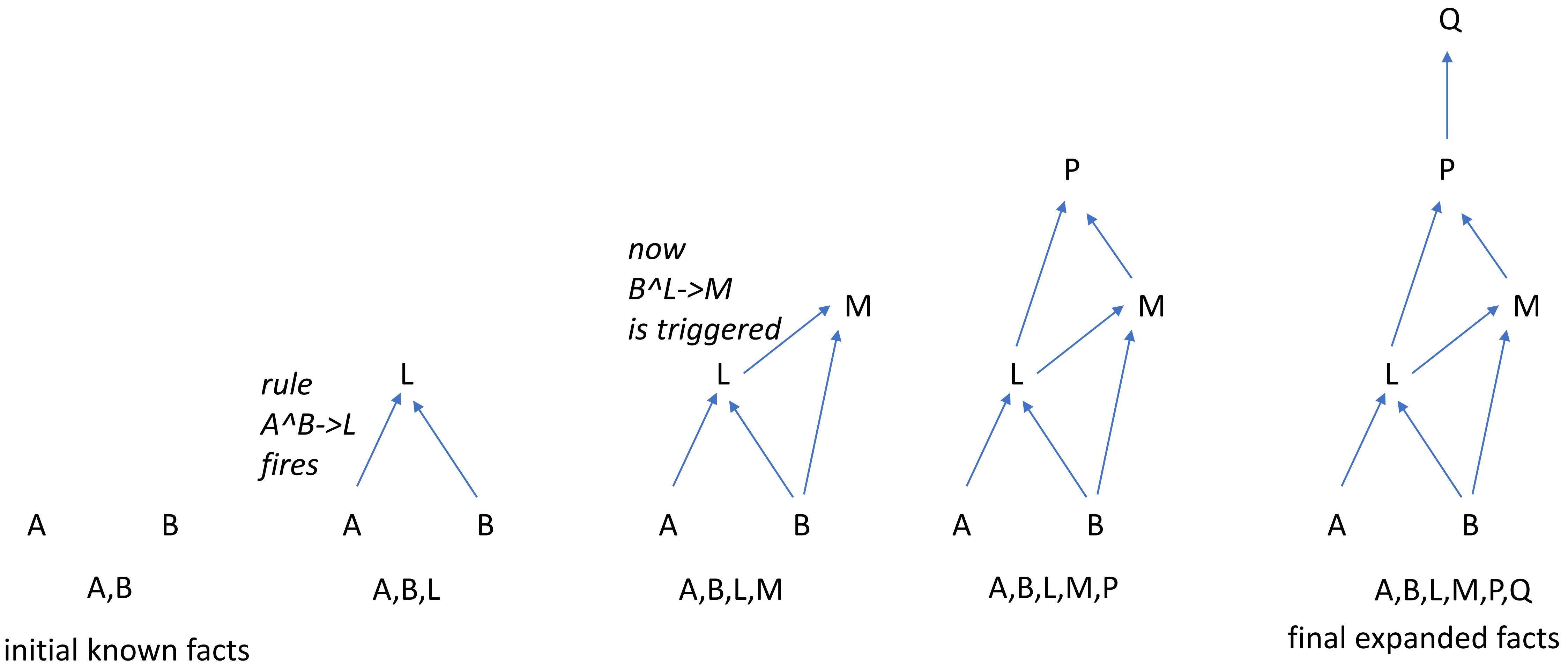
Forward Chaining

- let's explore more practical theorem-proving methods
- Forward-Chaining (FC) is super-easy: you only need Modus Ponens!
- however, FC only works on definite-clause KBs
  - a *clause* is a disjunction of literals, e.g. A v ¬B v C v ¬D
  - a *Horn clause* is a clause with at most one positive literal, e.g. ¬ A v B v ¬C
  - a *definite clause* is a clause with exactly one positive literal
- where do definite clauses come from? facts and conjunctive rules
  - facts: A, B (note – negations are not allowed!)
  - rules with conjunct. of pos. lits as antecedents and 1 consequent

    A ^ B ^ C → D (conjunctive rule)

    ¬A v ¬B v ¬C v D (definite clause, by Implication Elimination)

Intuitive idea of Forward-Chaining: expanding initial facts by triggering rules



```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
          q, the query, a proposition symbol
  count ← a table, where count[c] is initially the number of symbols in clause c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  queue ← a queue of symbols, initially symbols known to be true in KB

  while queue is not empty do
     p ← POP(queue)
     if p = q then return true
     if inferred[p] = false then
        inferred[p] ← true
        for each clause c in KB where p is in c.PREMISE do
           decrement count[c]
           if count[c] = 0 then add c.CONCLUSION to queue
  return false
```

in this context, 'clause' means 'rule' (i.e. definite clause)

initial facts

- the key idea in FC is to use a queue (sometimes called an 'agenda') to keep track of facts that have been inferred (initially, just the given facts)
- with each new fact inferred, we check which rules can be triggered (i.e. when all their antecedents have been satisfied), and then we put the consequents in the queue
- there are ways to make this efficient for large KBs by indexing on which rules have which propositions as antecedents (to quickly figure out which rules are triggered by new facts)

Example of Forward Chaining

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

A

B

- agenda:
  - 1. A, B
  - 2. ~~A, B~~, L // rule 5 is triggered
  - 3. ~~A, B, L~~, M // rule 4
  - 4. ~~A, B, L, M~~, P // rule 2
  - 5. ~~A, B, L, M, P~~, Q // rule 1 fires
  - stop since query was generated

note, in this illustration,
I am not popping things out
of the queue

Forward Chaining

- so using FC requires the KB to be formulated as <u>definite clauses</u> (i.e. a rule base)
- in theory, this is not always possible
  - for example, if {¬P} or {PvQ} or {P^Q→RvS} is in the KB, it can't be transformed into definite clauses
  - these examples represent *uncertainty*, which isn't permitted
  - hence this requirement <u>limits expressiveness</u> (not full Propositional Logic)
- in practice, it is often possible to express KBs for real problems in definite clause form, with judicious choice of propositions

To show that *safe22* in the Wumpus World, we have to re-write the KB as definite clauses, which can be achieved by using new propositions...

PropSyms:

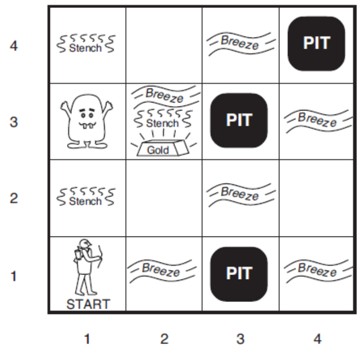
- WF = wumpus-free
- PF = pit-free
- C = calm
- US = unstenchy

note that we chose new propositional symbols representing information in a *negative* way

1. C11->PF21
2. C11->PF12
3. C12->PF22
4. C12->PF11
5. C12->PF13
6. C13->PF23
7. C13->PF12
8. C13->PF14
9. C14->PF24
10. C14->PF13
11. C21->PF11
12. C21->PF31
13. C21->PF22
14. C22->PF12
15. C22->PF32
16. C22->PF21
17. C22->PF23
18. C23->PF13
19. C23->PF33
20. C23->PF22
21. C23->PF24
22. C24->PF14
23. C24->PF34
24. C24->PF23
25. C31->PF21
26. C31->PF41
27. C31->PF32
28. C32->PF22
29. C32->PF42
30. C32->PF31
31. C32->PF33
32. C33->PF23
33. C33->PF43
34. C33->PF32
35. C33->PF34
36. C34->PF24
37. C34->PF44
38. C34->PF33
39. C41->PF31
40. C41->PF42
41. C42->PF32
42. C42->PF41
43. C42->PF43
44. C43->PF33
45. C43->PF42
46. C43->PF44
47. C44->PF34
48. C44->PF43
49. US11->WF21
50. US11->WF12
51. US12->WF22
52. US12->WF11
53. US12->WF13
54. US13->WF23
55. US13->WF12
56. US13->WF14
57. US14->WF24
58. US14->WF13
59. US21->WF11
60. US21->WF31
61. US21->WF22
62. US22->WF12
63. US22->WF32
64. US22->WF21
65. US22->WF23
66. US23->WF13
67. US23->WF33
68. US23->WF22
69. US23->WF24
70. US24->WF14
71. US24->WF34
72. US24->WF23
73. US31->WF21
74. US31->WF41
75. US31->WF32
76. US32->WF22
77. US32->WF42
78. US32->WF31
79. US32->WF33
80. US33->WF23
81. US33->WF43
82. US33->WF32
83. US33->WF34
84. US34->WF24
85. US34->WF44
86. US34->WF33
87. US41->WF31
88. US41->WF42
89. US42->WF32
90. US42->WF41
91. US42->WF43
92. US43->WF33
93. US43->WF42
94. US43->WF44
95. US44->WF34
96. US44->WF43
97. WF11^PF11->safe11
98. WF12^PF12->safe12
99. WF13^PF13->safe13
100. WF14^PF14->safe14
101. WF21^PF21->safe21
102. WF22^PF22->safe22
103. WF23^PF23->safe23
104. WF24^PF24->safe24
105. WF31^PF31->safe31
106. WF32^PF32->safe32
107. WF33^PF33->safe33
108. WF34^PF34->safe34
109. WF41^PF41->safe41
110. WF42^PF42->safe42
111. WF43^PF43->safe43
112. WF44^PF44->safe44

FC proof of *KB^Facts* ⊨ *safe22*

Facts:
113. C11 // room 1,1 is calm (no breeze)
114. US11 // room 1,1 is unstenchy
115. C12 // room 1,2 is calm
116. US21 // room 2,1 is unstenchy



inferred	agenda
	C11 US11 C12 US21 // initialize by pushing facts
C11	US11 C12 US21 PF12 PF21 // C11 causes 2 new facts to be pushed onto agenda from rules 1&2
US11	C12 US21 PF12 PF21 WF12 WF21
C12	US21 PF12 PF21 WF12 WF21 PF11 PF22 PF13 // C12 causes rules 3-5 to fire
US21	PF12 PF21 WF12 WF21 PF11 PF22 PF13 WF11 WF31 WF22 // rules 59-61
PF12 PF21 WF12 WF21 PF11 PF22 PF13	// these just pop off without pushing anything new
WF11	WF31 WF22 safe11 // since WF11 and PF11 have been inferred, rule 97 fires
WF31	WF22 safe11
WF22	safe11 safe22 // since WF22 and PF22 have been inferred, rule 102 fires
safe11	safe22
safe22	// found what we were looking for, showing the query is entailed; also, agenda becomes empty

Back-Chaining

- one of the problems with FC is that it can waste time generating a lot of unnecessary inferences that are irrelevant to the query
- back-chaining (BC) also works on definite-clause KBs, but it works *backwards* from the goal to find supporting facts
- hence BC is more efficient because it is goal-directed
- BC is in fact the basis of PROLOG (as we will see later when we cover FOL)

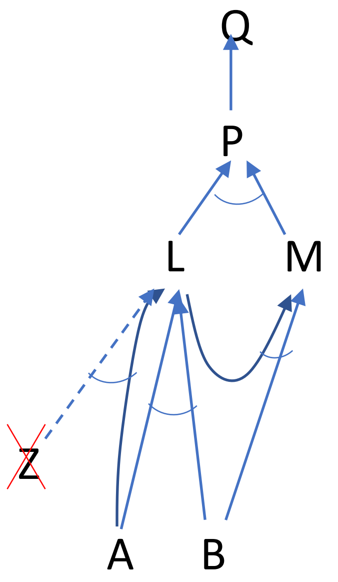
Back-Chaining

- BC uses a *goal stack* (initialized by pushing the query)
- with each iteration:
  - pop the goal on the top of the stack
  - check to see if it is a known fact
  - otherwise, find a rule that has the goal as consequent, and push the antecedents onto the stack as subgoals
  - the algorithm terminates when the stack becomes empty (success, showing the query is entailed, because it has been reduced to known facts)
- important: back-tracking
  - if some subgoals cannot be proved, BC must back-track and try another rule to prove goal

Example of Backward Chaining

1. P→Q
2. L^M→P
3. B^L→M
4. A^Z→L
5. A^B→L
6. A
7. B

- Q // initialize with query
- P // pop Q, replace with antecedent of rule 1
- L, M // replace P with ants. of rule 2
- A, Z, M // pop L, push A,P from rule 4
- Z, M // pop A (known fact)
- // since Z is not provable, back-track to other rule for L
- A, B, M
- B, M // pop A (fact)
- M // pop B (fact)
- B, L // pop M, try rule 3, push B,L
- L // pop B (fact)
- A,Z // pop L, try rule 4
- Z // pop A (fact)
- // since Z is not provable, back-track to other rule for L
- A, B
- B // pop A (fact)
- ∅ // pop B (fact); stack becomes empty; return success!



visualizing the proof tree as an "and-or" graph

(note: This example is modified from Fig 7.16 in the book to simplify for illustration purposes. The P in A^P→L was replaced with Z, to avoid the complication of checking for repeated subgoals, which would succeed implicitly, representing a loop. In this context, however, that technical detail is an unnecessary distraction.)

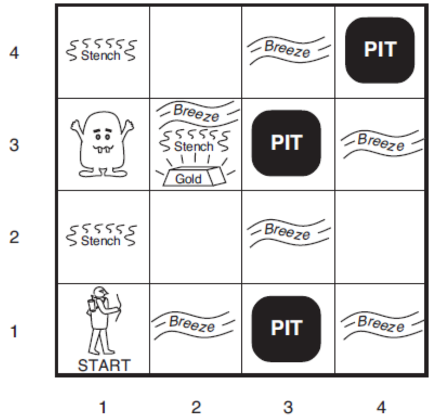
Back-chaining using Propositional Logic (Recursive stack-based version)

```
Backchain(KB,query)
  stack.push(query) // initialize
  return BC(KB,stack)

BC(KB,stack)
  if stack empty, return True
  subgoal ← stack.pop()
  if subgoal∈KB, return BC(KB,stack) // a known fact
  for each rule a1..an→subgoal in KB: // choice point for back-tracking
    stack.push(a1..an)
    result ← BC(KB,stack)
    if result=True, return True
    else remove a1..an from stack
  return False
```

BC proof of KB^Facts |= safe22
(using the definite-clause KB from slide 34)

Facts:
113. C11
114. US11
115. C12
116. US21



goal stack	
safe22	push query
WF22 PF22	rule102
US12 PF22	try rule 51 for WF22
US21 PF22	fail, back-track; try rule 61 for WF22
PF22	succeed (116); US21 pops off
C21	try rule 21 for PF22
C12	fail; back-track; try rule 61
∅	C12 is a known fact (115); pop off stack becomes empty; proof succeeds

Resolution Refutation

- FC and BC are effective proof procedures, but they are limited because the are not complete (not all KBs are in definite-clause form)
- Is there a complete proof procedure that is simpler than Nat. Ded.?
- Resolution Refutation proofs – you can prove any entailed sentence, and all you need is one ROI: *resolution*

$$\frac{A \lor B \lor \ldots,\ \neg A \lor C \lor \ldots}{B \lor \ldots \lor C \lor \ldots}$$

- prerequisite: you have to convert your KB into CNF (Conjunctive-Normal Form, i.e. clauses), which you can always do

simple example: A^B^¬C→D^E can be transformed
into 2 clauses (not necessarily Horn) that are equivalent:
(¬A v ¬B v C v D) , (¬A v ¬B v C v E)

Conversion to CNF

- procedure for converting any propositional sentence to CNF (p. 227)
  1. eliminate implications (and biconditionals)
  2. push negations inward (using DoubleNegElim and DeMorgan’s)
  3. distribute Or’s over And’s (till expression is 2-level Boolean CNF)
  4. break final conjunction into multiple clauses
- example: A^B^¬C→D^E

  1. ¬(A^B^¬C) v D^E // implication elimination

  2. (¬Av¬Bv¬¬C) v (D^E) // push negations inward

  3. (¬Av¬BvCvD) ^ (¬Av¬BvCvE) // distribution

  4a. (¬A v ¬B v C v D)

  4b. (¬A v ¬B v C v E)

Refutation Proofs

- negate the query and add it to the KB
- if the query was entailed, this creates an inconsistency (unsatisfiable), $M(KB \cup \{\neg q\}) = \emptyset$
- thus we should be able to derive the empty clause (which means "false" or "inconsistent")

simple example:

suppose KB={A,A→B} and q=B

negate query and append it: {A,A→B,¬B}

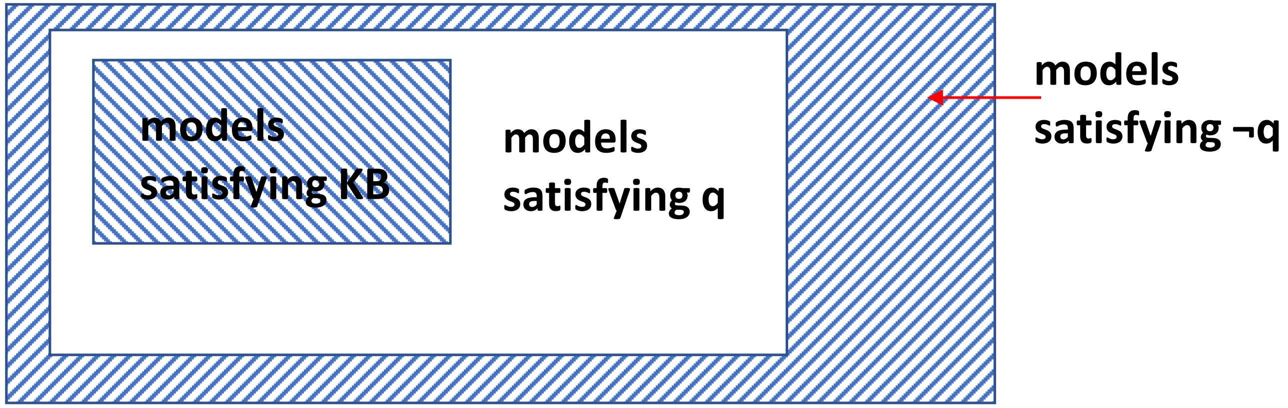
convert to CNF { A , ¬A v B, ¬B }

1. A
2. ¬A v B
3. ¬B
4. ¬A // resolve 2 and 3
5. ∅ // resolve 1 and 4, empty clause

this means we proved KB ⊨ B

Refutation Proofs

- Why do refutation proofs work?
- like “proof by contradiction”
- no models satisfy both KB and ¬q (empty intersection)



Example of Resolution Refutation Proof

KB:

1. P→Q
2. L^M→P
3. B^L→M
4. A^P→L
5. A^B→L
6. A
7. B

query: Q

CNF:

1. ¬P v Q
2. ¬L v ¬M v P
3. ¬ B v ¬L v M
4. ¬A v ¬P v L
5. ¬A v ¬B v L
6. A
7. B
8. ¬Q

// negated query

9. ¬P // reso on 1 and 8 (eliminate Q)
10. ¬L v ¬M // reso 2,9
11. ¬A v ¬B v ¬M // reso 5,10 (eliminate L)
12. ¬A v ¬B v ¬B v ¬L // reso 11,3 (eliminate M)
13. ¬A v ¬B v ¬L // (factoring, combine ¬Bs)
14. ¬A v ¬B v ¬A v ¬B // reso 13,5
15. ¬A v ¬B // factoring
16. ¬B // reso 15,7
17. ∅ // reso 16,8; empty clause!

Resolution Proof Procedure

function PL-RESOLUTION(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  new ← { }
  loop do
    for each pair of clauses C_i, C_j **in** *clauses* **do**
      resolvents ← PL-RESOLVE(C_i, C_j)
      if *resolvents* contains the empty clause **then return** *true*
      new ← *new* ∪ *resolvents*
    if *new* ⊆ *clauses* **then return** *false*
    clauses ← *clauses* ∪ *new*

you could stop when no new clauses are generated (disregarding repeated clauses); but query would not be entailed, since empty clause was not found

Wumpus World Clauses for Resolution

1. -W11vS21	25. -W31vS21	49. -P11vB21	73. -P31vB21	97. W11 v P11 v safe11
2. -W11vS12	26. -W31vS41	50. -P11vB12	74. -P31vB41	98. W12 v P12 v safe12
3. -W12vS22	27. -W31vS32	51. -P12vB22	75. -P31vB32	99. W13 v P13 v safe13
4. -W12vS11	28. -W32vS22	52. -P12vB11	76. -P32vB22	100. W14 v P14 v safe14
5. -W12vS13	29. -W32vS42	53. -P12vB13	77. -P32vB42	101. W21 v P21 v safe21
6. -W13vS23	30. -W32vS31	54. -P13vB23	78. -P32vB31	102. W22 v P22 v safe22
7. -W13vS12	31. -W32vS33	55. -P13vB12	79. -P32vB33	103. W23 v P23 v safe23
8. -W13vS14	32. -W33vS23	56. -P13vB14	80. -P33vB23	104. W24 v P24 v safe24
9. -W14vS24	33. -W33vS43	57. -P14vB24	81. -P33vB43	105. W31 v P31 v safe31
10. -W14vS13	34. -W33vS32	58. -P14vB13	82. -P33vB32	106. W32 v P32 v safe32
11. -W21vS11	35. -W33vS34	59. -P21vB11	83. -P33vB34	107. W33 v P33 v safe33
12. -W21vS31	36. -W34vS24	60. -P21vB31	84. -P34vB24	108. W34 v P34 v safe34
13. -W21vS22	37. -W34vS44	61. -P21vB22	85. -P34vB44	109. W41 v P41 v safe41
14. -W22vS12	38. -W34vS33	62. -P22vB12	86. -P34vB33	110. W42 v P42 v safe42
15. -W22vS32	39. -W41vS31	63. -P22vB32	87. -P41vB31	111. W43 v P43 v safe43
16. -W22vS21	40. -W41vS42	64. -P22vB21	88. -P41vB42	112. W44 v P44 v safe44
17. -W22vS23	41. -W42vS32	65. -P22vB23	89. -P42vB32	
18. -W23vS13	42. -W42vS41	66. -P23vB13	90. -P42vB41	
19. -W23vS33	43. -W42vS43	67. -P23vB33	91. -P42vB43	
20. -W23vS22	44. -W43vS33	68. -P23vB22	92. -P43vB33	
21. -W23vS24	45. -W43vS42	69. -P23vB24	93. -P43vB42	
22. -W24vS14	46. -W43vS44	70. -P24vB14	94. -P43vB44	
23. -W24vS34	47. -W44vS34	71. -P24vB34	95. -P44vB34	
24. -W24vS23	48. -W44vS43	72. -P24vB23	96. -P44vB43	

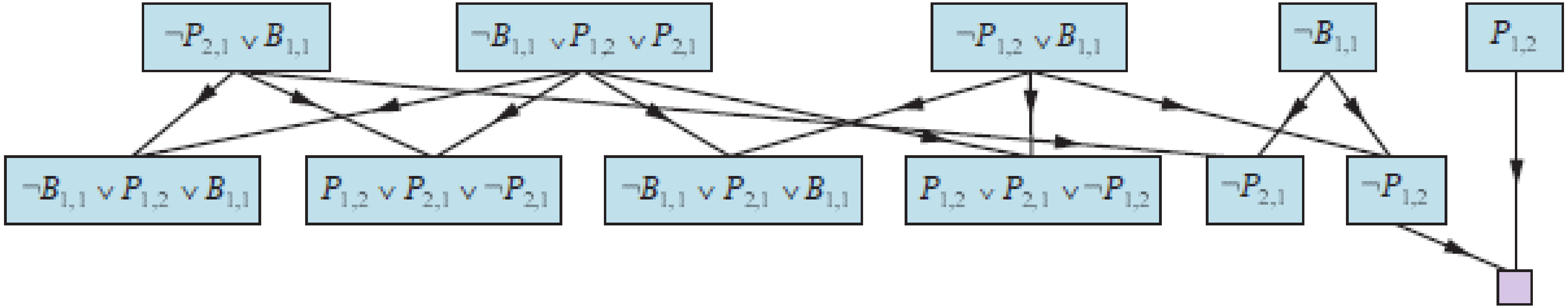
ResoRef proof of *KB^Facts* $\models$ *safe22*

Facts:
113. -B11
114. -S11
115. -B12
116. S12
117. B21
118. -S21
119. -safe22 // negation of query

new clauses	annotation
120. -P22	Reso on 115 & 62 (-P22vB12)
121. -W22	Reso on 118 & 16 (-W22vS21)
122. W22 v safe22	Reso on 120 & 102 (W22 v P22 v safe22)
123. safe22	Reso on 121 & 122
124. ∅ derived empty clause; proof succeeds	Reso on 119 & 123

Resolution Refutation

- Resolution as a <u>Search for the empty clause</u>
- nodes in Search Tree = clauses, but they have multiple parents
- there are often many clauses that can be resolved (most are irrelevant)



Resolution Refutation

- heuristics to make the resolution search more efficient (resolution strategies) :
  - unit-clause heuristic:
    - choose pairs of clauses that can resolve, where at least one clause is just a single literal
    - rationale: size of resolvent of clauses of size n and m is n+m-2, so if one is a unit clause, the resolvent shrinks in size to n-1 (closer to the goal of size 0 for the empty clause)
    - this is effective, but incomplete (there are some proofs you can't do if you always use the unit-clause heuristic)
  - input resolution: always choose one of the clauses from the input set (premise clauses)
  - linear resolution: always choose a previously resolved clause
- we will discuss these heuristics in more detail Ch. 9 (Inference in FOL)

Resolution Refutation

- Is it complete proof procedure? can it determine whether *any* sentence is entailed?
- Ground Resolution Theorem:
  - if a set of sentences S is unsatisfiable, then there exists a finite sequence of resolution steps that will generate the empty clause.
  - the textbook restates this as “the empty clause will be contained in the resolution closure”
  - the proof involves showing that: suppose S is unsat but RC(S) does not contain the empty clause; then we can construct a model out of the clauses in RC(S) - contradiction
- Theorem: Resolution refutation is a complete proof procedure.
  - if $\alpha \models \beta$, then there exists a finite sequence of resolution steps (starting from the CNF of $\{\alpha \wedge \neg\beta\}$) that will generate the empty clause.
- generally, we do not try to show the converse, i.e. that if b is not entailed, resolution should stop and say so, e.g. when it runs out of clauses that can be resolved
  - theoretically you could do it in Prop Log, but it depend on the RC(S) being finite (requires factoring)

Satisfiability

- another propositional theorem-proving strategy
- Sat methods can test if a set of sentences is unsatisfiable (like in a Refutation proof)
- more commonly, Sat methods are used on satisfiable KBs – the goal is to <u>generate a model</u> (where the truth values are the solution to a problem)
- this is a (slightly) more efficient form of model-checking

DPLL

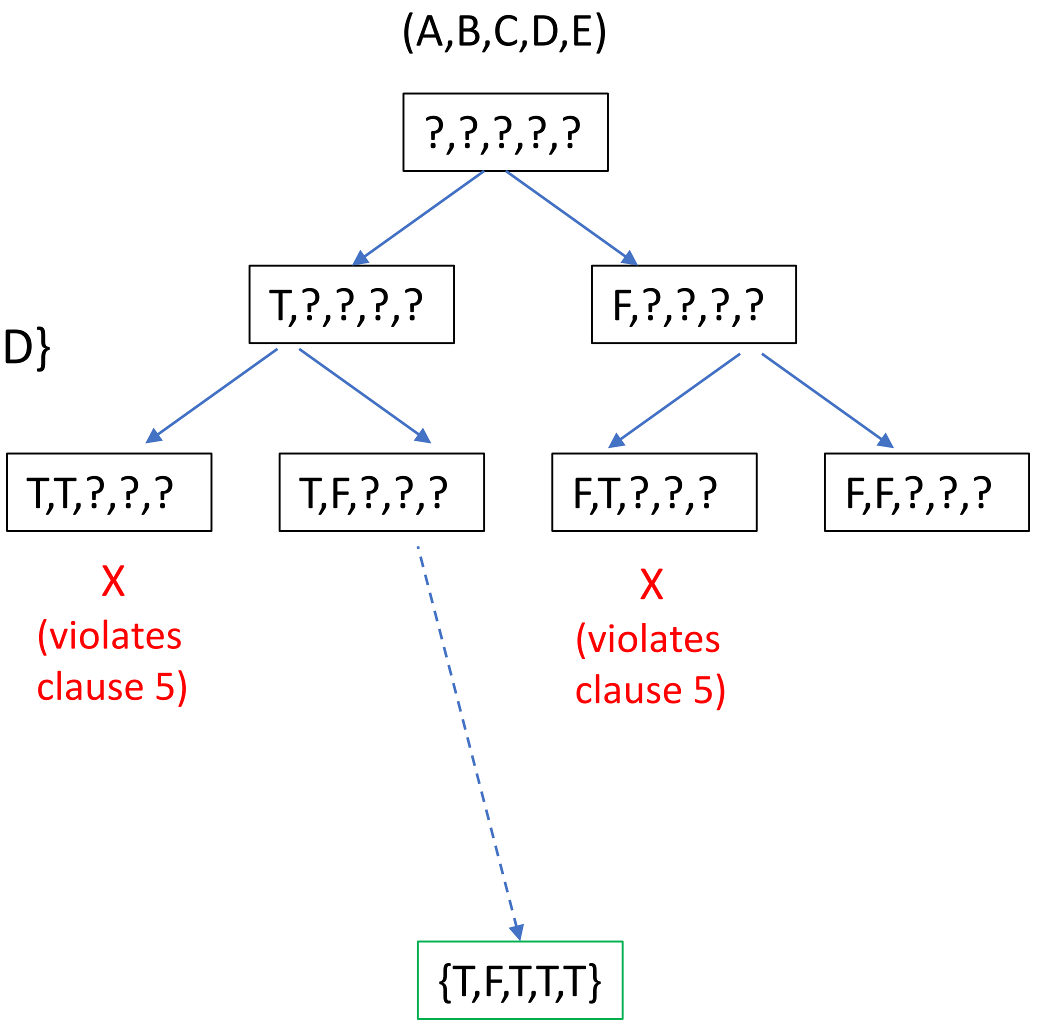
- a truth assignment (as a model) is a specification of truth values (T,F,?) for all propositional symbols in a KB
  - examples: {F,F,F,F,F}, {T,F,?,?,?}

- Search for complete truth assignment (like CSP)
  - KB = {¬DvC,A^B→E,¬D→¬E,C→E,¬B^D}, CNF={¬DvC,¬Av¬BvE,Dv¬E,¬CvE,¬B,D}
  - props are {A,B,C,D,E}
  - initial state={?,?,?,?,?}
  - goal states={F,F,T,T,T} and {T,F,T,T,T}

- "Davis-Putman-Logemann-Loveland" (DPLL) procedure
  - convert propositional KB into CNF
  - start with an empty truth assignment {?,?,?,...,?}
  - try binding one more variable at a time
  - back-track whenever a clause is violated
  - quit when a complete assignment is found that satisfies all clauses

DPLL

find a model for:
{¬DvC,¬Av¬BvE,Dv¬E,¬CvE,¬B,D}



(this tree assigns the props in alphabetical order by default, but DPLL could choose a different prop at each node using the unit-clause or pure symbol heuristics discussed on the next slide)

function DPLL-SATISFIABLE?(s) **returns** *true* or *false*
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(*clauses*, *symbols*, { })

---

function DPLL(*clauses*, *symbols*, *model*) **returns** *true* or *false*

  if every clause in *clauses* is true in *model* **then return** *true*
  if some clause in *clauses* is false in *model* **then return** *false*
  P, *value* ← FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)
  if P is non-null **then return** DPLL(*clauses*, *symbols* – P, *model* ∪ {P=*value*})
  P, *value* ← FIND-UNIT-CLAUSE(*clauses*, *model*)
  if P is non-null **then return** DPLL(*clauses*, *symbols* – P, *model* ∪ {P=*value*})
  P ← FIRST(*symbols*); *rest* ← REST(*symbols*)
  return DPLL(*clauses*, *rest*, *model* ∪ {P=*true*}) **or**
      DPLL(*clauses*, *rest*, *model* ∪ {P=*false*}))

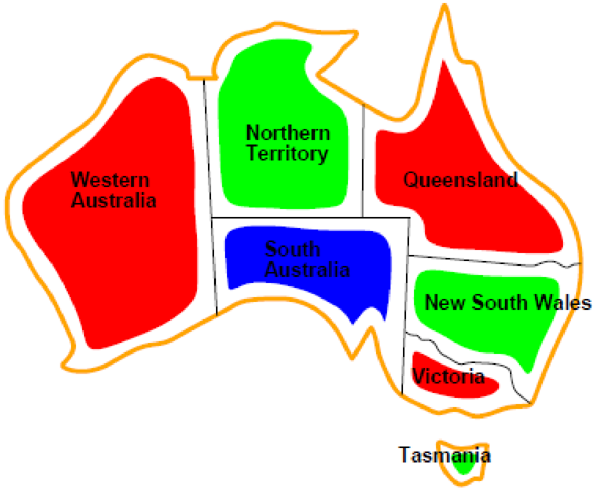
heuristics

the essence of DPLL is *guessing* a truth value for each proposition, and *backtracking* when a conflict is discovered

DPLL

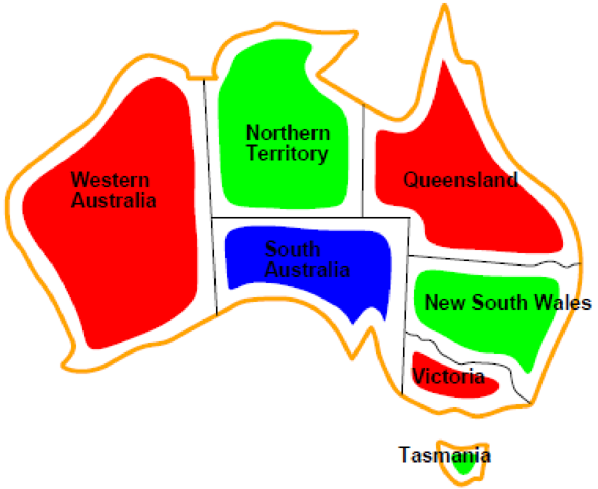
- DPLL systematically explores the space of models (which can be slow)
- heuristics to speed up DPLL
  - we can bias the choice of which proposition to assign next
  - unit clause heuristic – given a partial assignment, if there is a clause where all but one literal is False and the last is ?, then add the appropriate truth value to the model
    - example: CNF={¬B, D, ¬DvC , ¬Av¬BvE , Dv¬E , ¬CvE}
    - in model for A,B,C,D,E={T,?,?,T,?}; clause 1 is unit, so bind B=F: {T,F,?,T,?}
    - in model={T,F,?,T,?}; clause 3 is unit (because ¬D is false), so bind C=T: {T,F,T,T,?}
  - pure symbol heuristic - given a partial assignment, if prop X=? and X appears only as pos. lit. (X) in all unsatisfied clauses remaining, bind X=T
    - if it appears only as neg. lit. (¬X) in all *unsatisfied* clauses remaining, bind X=F
    - in the KB above, the proposition A only appears as "¬A", so we can just assume it is F
    - it doesn't mean X *has* to have that truth value, only it *can* (if there is a model of the KB, then there is a model in which X=T)

Solving Problems via Satisfiability



- example: map-coloring
  - convert KB (slide 11) to CNF
    - there are 21 propositions (7 states X 3 colors)
    - clauses={WAR v WAG v WAB, ¬WAR v ¬WAB, ¬WAR v ¬NTR, ...} (100-200 CNF sentences)
  - DPLL(<?,?,?,?.....?>,clauses) returns a complete truth assignment
    - <WAR=T, WAG=F, WAB=F, NTR=F, NTG=T, NTB=F, SAR=F...>
    - 7 T's and 14 F's
    - the DPLL algorithm can be modified to return additional models
    - how many times does back-tracking occur?
    - when does the unit-clause heuristic get invoked?
    - how much back-tracking would there be without the UC or PS heuristics?
    - size of search space?

Solving Problems via Satisfiability



- using DPLL to find other solutions
  - find a coloring of the map where Queensland is green
  - DPLL(<?,?,?,?.....?>,clauses∪{QG}) returns
    - <WAR=F,WAG=T,WAB=F,NTR=T,NTG=G,NTB=F,SAR=F,SAG=F,SAB=T,QR=F,QG=T,QB=F...>

- using DPLL to show something is entailed
  - show that if WA is red, then V has to be red: WAR→VR
  - negate the sentence and add to clauses: ¬(WAR→VR) = WAR ^ ¬VR (as CNF)
  - DPLL(<?,?,?,?.....?>,clauses∪{WAR, ¬VR}) returns *unsatisfiable*

DPLL

- many other problems can be solved by encoding them as Sat problems
  - **CSPs**
  - Sammy's sport shop, Wumpus world
  - planning (SatPlan), scheduling,
  - multi-agent coordination,...
  - vertex cover, knapsack,...

Complexity of Propositional Inference

- Cook’s Theorem: Boolean SAT is NP-complete.
  - proof involves showing that you can describe or "encode" a Turing machine that simulates any non-det. computation in the form of a Boolean expression with at size at most a polynomial in the number of states, tape symbols, etc
- Hence, *complete* proof procedures can’t be guaranteed to halt and return an answer in polynomial time (unless P=NP)
  - so we could wait a *long* time for a resolution proof to finish
  - however, restricted methods, like FC and BC can potentially run in poly time

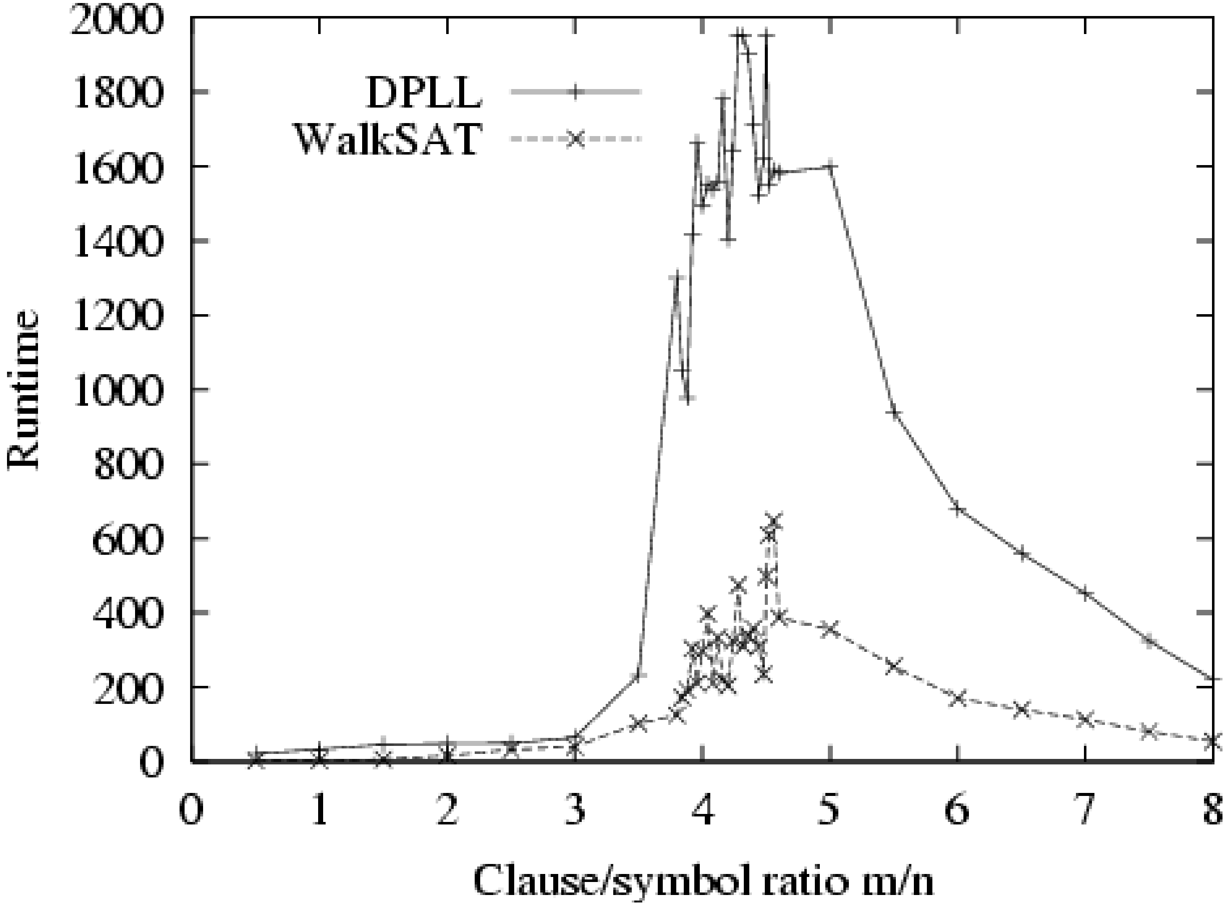
WalkSAT - a stochastic approach to satisfiability

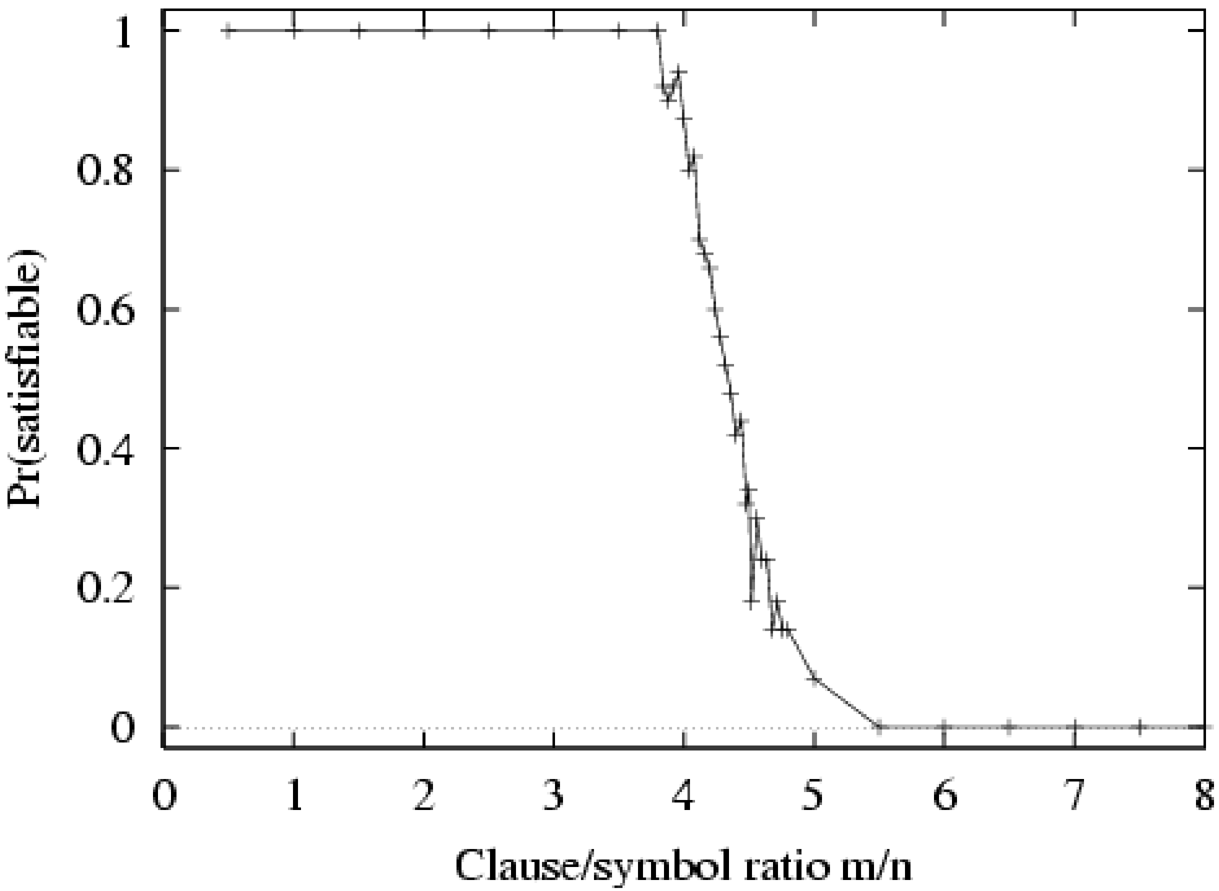
- not guaranteed to be complete, but it is fast and often effective at finding models of a set of clauses

function WALKSAT(*clauses*, *p*, *max_flips*) **returns** a satisfying model or *failure*
  inputs: *clauses*, a set of clauses in propositional logic
    p, the probability of choosing to do a "random walk" move, typically around 0.5
    max_flips, number of flips allowed before giving up

  model ← a random assignment of *true*/*false* to the symbols in *clauses*
  for $i = 1$ **to** *max_flips* **do**
    if *model* satisfies *clauses* **then return** *model*
    clause ← a randomly selected clause from *clauses* that is false in *model*
    with probability *p* flip the value in *model* of a randomly selected symbol from *clause*
    else flip whichever symbol in *clause* maximizes the number of satisfied clauses
  return *failure*

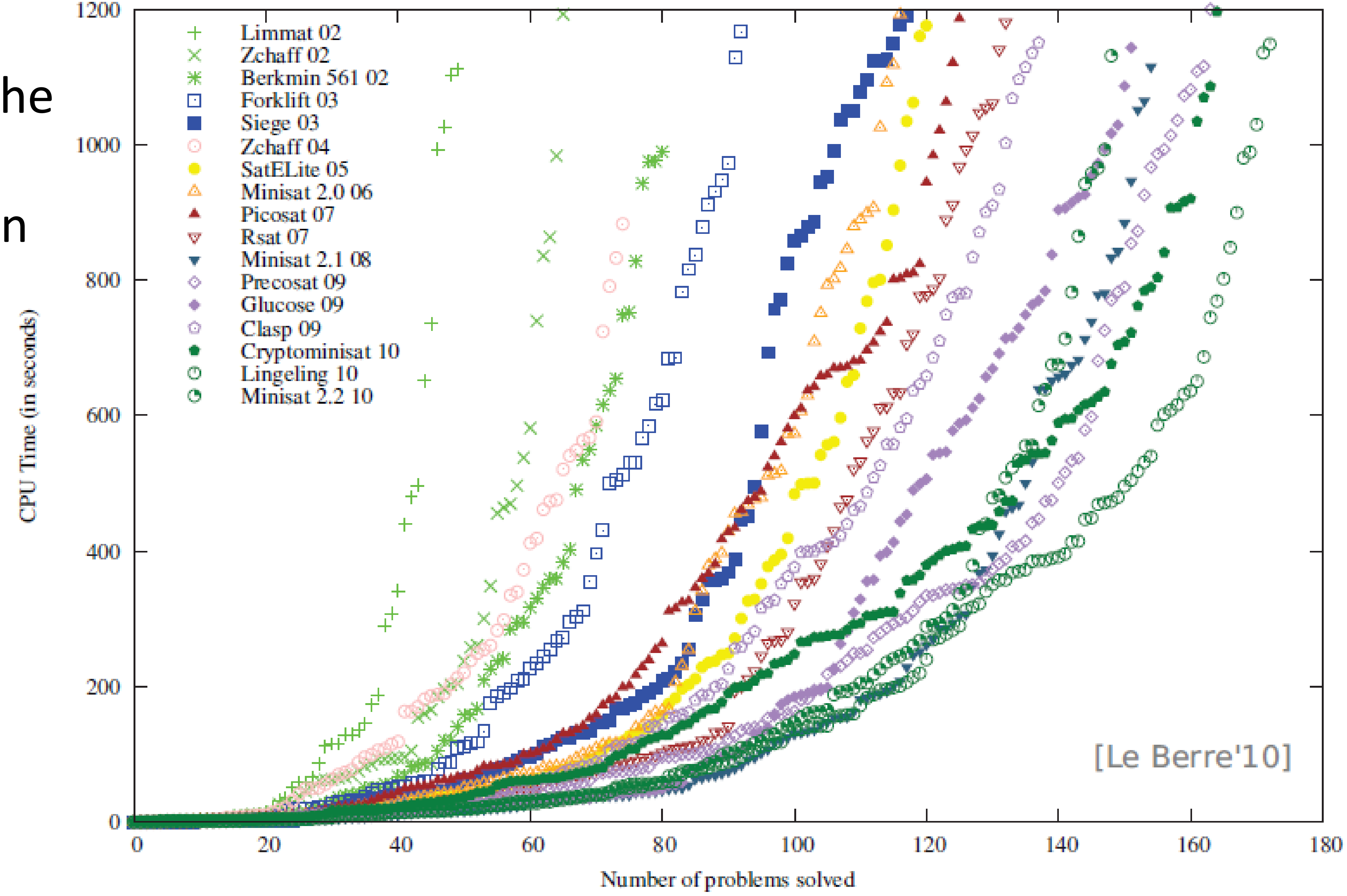
Hard Satisfiability Problems - The "Computational Cliff"





- experiments with *randomly generated* Sat problems (1000s of Boolean clauses)
- "computational cliff" at ~4.3 clauses per symbol

Results of the SAT 2009 Competition



CBMC - Concurrent Bounded Model Checker

- application of Boolean Sat to C program verification; uses MiniSat
- https://www.cprover.org/cbmc/
- SSA - single static assignment

```
void test(int x, int y)
  if(x > 5){
    x++;
    if (x < 3)
      x--;
    else
      y = x;
  }
  assert (x < 10 );
}
```

possible execution paths

path conditions

negation of invariant

$$\bigvee_{i=0}^{M} (pc_i \wedge \neg \mathcal{P}|_{\sigma_i})$$

$$((x_0 > 5) \wedge \neg(x_0 + 1 < 3) \wedge \neg(x_0 + 1 < 10)) \vee$$
$$((x_0 > 5) \wedge (x_0 + 1 < 3) \wedge \neg(x_0 < 10)) \vee$$
$$(\neg(x_0 > 5) \wedge \neg(x_0 < 10))$$