Planning and search

Search and SAT

Outline

◇ Propositional (Boolean) logic

◇ Equivalence, validity, satisfiability

◇ How to encode problems in propositional logic

◇ SAT

◇ (Search) algorithms for solving SAT

Propositional logic: Syntax

The proposition symbols P_1, P_2 etc are sentences

If S is a sentence, $\neg S$ is a sentence (negation)

If S_1 and S_2 are sentences, $S_1 \land S_2$ is a sentence (conjunction)

If S_1 and S_2 are sentences, $S_1 \lor S_2$ is a sentence (disjunction)

If S_1 and S_2 are sentences, $S_1 \;\Rightarrow\; S_2$ is a sentence (implication)

If S_1 and S_2 are sentences, $S_1 \;\Leftrightarrow\; S_2$ is a sentence (biconditional)

Propositional logic: Semantics

Each model specifies true/false for each proposition symbol

E.g. $\begin{array}{cccc} P_{1,1} & P_{1,2} & P_{2,1} & P_{2,2} \\ true & false & false & true \end{array}$

2^4 possible models (assignments), can be enumerated automatically

Rules for evaluating truth with respect to a model m:

$\neg S$	is true iff	S	is false		
$S_1 \wedge S_2$	is true iff	S_1	is true **and**	S_2	is true
$S_1 \vee S_2$	is true iff	S_1	is true **or**	S_2	is true
$S_1 \Rightarrow S_2$	is true iff	S_1	is false **or**	S_2	is true
i.e.,	is false iff	S_1	is true **and**	S_2	is false
$S_1 \Leftrightarrow S_2$	is true iff	$S_1 \Rightarrow S_2$	is true **and**	$S_2 \Rightarrow S_1$	is true

Simple recursive process evaluates an arbitrary sentence, e.g.,
$(P_{1,1} \vee P_{1,2}) \wedge (P_{2,1} \vee P_{2,2}) = (true \vee false) \wedge (false \vee true) = true \wedge true = true$

Truth tables

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

Logical equivalence

Two sentences are logically equivalent iff true in same models:

$\alpha \equiv \beta$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$

$$
\begin{array}{rcll}
(\alpha \land \beta) & \equiv & (\beta \land \alpha) & \text{commutativity of } \land \\
(\alpha \lor \beta) & \equiv & (\beta \lor \alpha) & \text{commutativity of } \lor \\
((\alpha \land \beta) \land \gamma) & \equiv & (\alpha \land (\beta \land \gamma)) & \text{associativity of } \land \\
((\alpha \lor \beta) \lor \gamma) & \equiv & (\alpha \lor (\beta \lor \gamma)) & \text{associativity of } \lor \\
\neg(\neg\alpha) & \equiv & \alpha & \text{double-negation elimination} \\
(\alpha \;\Rightarrow\; \beta) & \equiv & (\neg\beta \;\Rightarrow\; \neg\alpha) & \text{contraposition} \\
(\alpha \;\Rightarrow\; \beta) & \equiv & (\neg\alpha \lor \beta) & \text{implication elimination} \\
(\alpha \;\Leftrightarrow\; \beta) & \equiv & ((\alpha \;\Rightarrow\; \beta) \land (\beta \;\Rightarrow\; \alpha)) & \text{biconditional elimination} \\
\neg(\alpha \land \beta) & \equiv & (\neg\alpha \lor \neg\beta) & \text{De Morgan} \\
\neg(\alpha \lor \beta) & \equiv & (\neg\alpha \land \neg\beta) & \text{De Morgan} \\
(\alpha \land (\beta \lor \gamma)) & \equiv & ((\alpha \land \beta) \lor (\alpha \land \gamma)) & \text{distributivity of } \land \text{ over } \lor \\
(\alpha \lor (\beta \land \gamma)) & \equiv & ((\alpha \lor \beta) \land (\alpha \lor \gamma)) & \text{distributivity of } \lor \text{ over } \land
\end{array}
$$

Satisfiability

A sentence is satisfiable if it is true in **some** model

e.g., $A \lor B$, C

A sentence is unsatisfiable if it is true in **no** models

e.g., $A \land \neg A$

Entailment: a conjunction of sentences KB entails a sentence A if and only if A is true in all models which make KB true (written $KB \models A$

Satisfiability is connected to entailment via the following:

$KB \models A$ if and only if $(KB \land \neg A)$ is unsatisfiable

SAT problem: is a given propositional sentence satisfiable?

n-SAT: sentence with n propositional variables

Encoding n-queens problem

Let $P_{i,j}$ be true if there is a queen in column i, row j

For 2-queen problem (sorry!), there is exactly one queen in every column:

$$C \;=\; (P_{1,1} \vee P_{1,2}) \wedge (P_{2,1} \vee P_{2,2}) \wedge \neg(P_{1,1} \wedge P_{1,2}) \wedge \neg(P_{2,1} \wedge P_{2,2})$$

Queens should not attack each other, so they should not be

in the same row: $R \;=\; \neg((P_{1,1} \wedge P_{2,1}) \vee (P_{1,2} \wedge P_{2,2}))$

or on the same diagonal: $D \;=\; \neg((P_{1,1} \wedge P_{2,2}) \vee (P_{1,2} \wedge P_{2,1}))$

Solution is a model where $C \wedge R \wedge D$ is true

Truth table for 2-queens problem

$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	C	R	D
true	*true*	*true*	*true*	*false*	*false*	*false*
true	*true*	*true*	*false*	*false*	*false*	*false*
true	*true*	*false*	*true*	*false*	*false*	*false*
true	*true*	*false*	*false*	*false*	*true*	*true*
true	*false*	*true*	*true*	*false*	*false*	*false*
true	*false*	*true*	*false*	*true*	*false*	*true*
true	*false*	*false*	*true*	*true*	*true*	*false*
true	*false*	*false*	*false*	*false*	*true*	*true*
false	*true*	*true*	*true*	*false*	*false*	*false*
false	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*true*	*true*	*false*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*
false	*false*	*true*	*true*	*false*	*true*	*true*
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*false*	*false*	*true*	*false*	*true*	*true*
false	*false*	*false*	*false*	*false*	*true*	*true*

Checking satisfiability

can check if there is a model by exhaustive enumeration

for a formula with n variables, $O(2^n)$ (exponential)

famous NP-complete problem

NP means non-deterministic polynomial time

A class of problems where we can **guess** the answer and verify that it is indeed the answer in polynomial time

Complete means: any NP problem can be polynomially reduced (reformulated) to satisfiability problem

Algorithms for solving SAT

also use a kind of search

complete depth-first search: Davis–Putnam–Logemann–Loveland (DPLL) algorithm

local search (sound but incomplete) WalkSAT

operate on clauses - need to learn to rewrite propositional sentences to clausal form

CNF

Conjunction normal form: conjunction of disjunctions where each disjunct is a literal (a variable or its negation)

Clause: a disjunction of literals (can be represented as a set of literals)

For example: $(A \vee \neg B) \wedge (\neg A \vee C)$ is in CNF

Clauses: $A \vee \neg B$, $\neg A \vee C$ or simply $\{A, \neg B\}$, $\{\neg A, C\}$

Logical equivalence

Two sentences are logically equivalent iff true in same models:
$\alpha \equiv \beta$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$

$$
\begin{array}{rcll}
(\alpha \wedge \beta) & \equiv & (\beta \wedge \alpha) & \text{commutativity of } \wedge \\
(\alpha \vee \beta) & \equiv & (\beta \vee \alpha) & \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) & \equiv & (\alpha \wedge (\beta \wedge \gamma)) & \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) & \equiv & (\alpha \vee (\beta \vee \gamma)) & \text{associativity of } \vee \\
\neg(\neg\alpha) & \equiv & \alpha & \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) & \equiv & (\neg\beta \Rightarrow \neg\alpha) & \text{contraposition} \\
(\alpha \Rightarrow \beta) & \equiv & (\neg\alpha \vee \beta) & \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) & \equiv & ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) & \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) & \equiv & (\neg\alpha \vee \neg\beta) & \text{De Morgan} \\
\neg(\alpha \vee \beta) & \equiv & (\neg\alpha \wedge \neg\beta) & \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) & \equiv & ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) & \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) & \equiv & ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) & \text{distributivity of } \vee \text{ over } \wedge
\end{array}
$$

Conversion to CNF

$A \Leftrightarrow (B \vee C)$

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

   $(A \;\Rightarrow\; (B \vee C)) \wedge ((B \vee C) \;\Rightarrow\; A)$

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

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

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

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

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

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

Basic-DPLL (my terminology)

function BASIC-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 B-DPLL(*clauses*, *symbols*, [ ])

---

function B-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*
$P \leftarrow$ FIRST(*symbols*); *rest* ← REST(*symbols*)
return B-DPLL(*clauses*, *rest*, [$P =$ *true*|*model*]) **or** B-DPLL(*clauses*, *rest*, [$P =$ *false*|*model*])

Proper DPLL

Real DPLL uses a number of heuristics

The version in the next slide uses pure symbol heuristic and unit clause heuristic

Pure symbol heuristic: pure symbol is a symbol which occurs in all clauses with the same sign (for example only as $\neg A$). If a sentence has a model, then it has a model where pure symbols are assigned so as to make their literals true (for example A assigned false). This is the value which this heuristic assigns to the symbol. Purity is recalculated as some clauses become true and can be ignored (so purity is defined relative to the set of remaining clauses)

Unit clause heuristic: unit clause is a clause with only one literal. In DPLL, it is a clause where only one symbol is yet unassigned. Unit clauses also have obvious assignment (for example for $\{\neg A\}$ to be true we have to assign false to A). This is the value which this heuristic assigns to the symbol.

DPLL

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, [P = *value*|*model*])
P, *value* ← FIND-UNIT-CLAUSE(*clauses*, *model*)
if P is non-null **then return** DPLL(*clauses*, *symbols*–P, [P = *value*|*model*])
P ← FIRST(*symbols*); *rest* ← REST(*symbols*)
return DPLL(*clauses*, *rest*, [P = *true*|*model*]) **or** DPLL(*clauses*, *rest*, [P = *false*|*model*])

Other DPLL heuristics

variable ordering

random restarts

intelligent backtracking

component analysis

WalkSat

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*

Which SAT problems are hard?

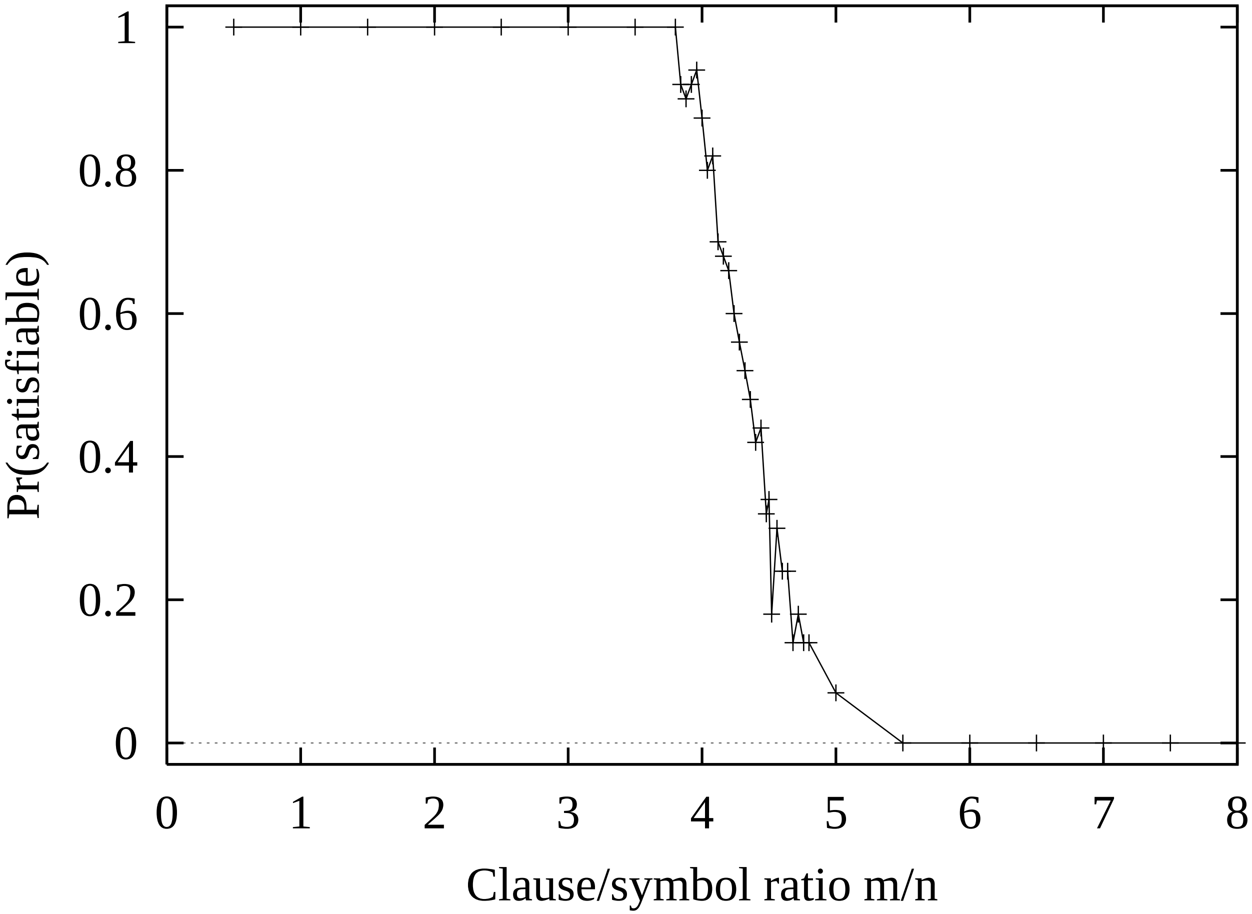
$CNF_k(m, n)$: k symbols per clause, m clauses and n symbols

as m/n ratio increases, probability of satisfiability decreases

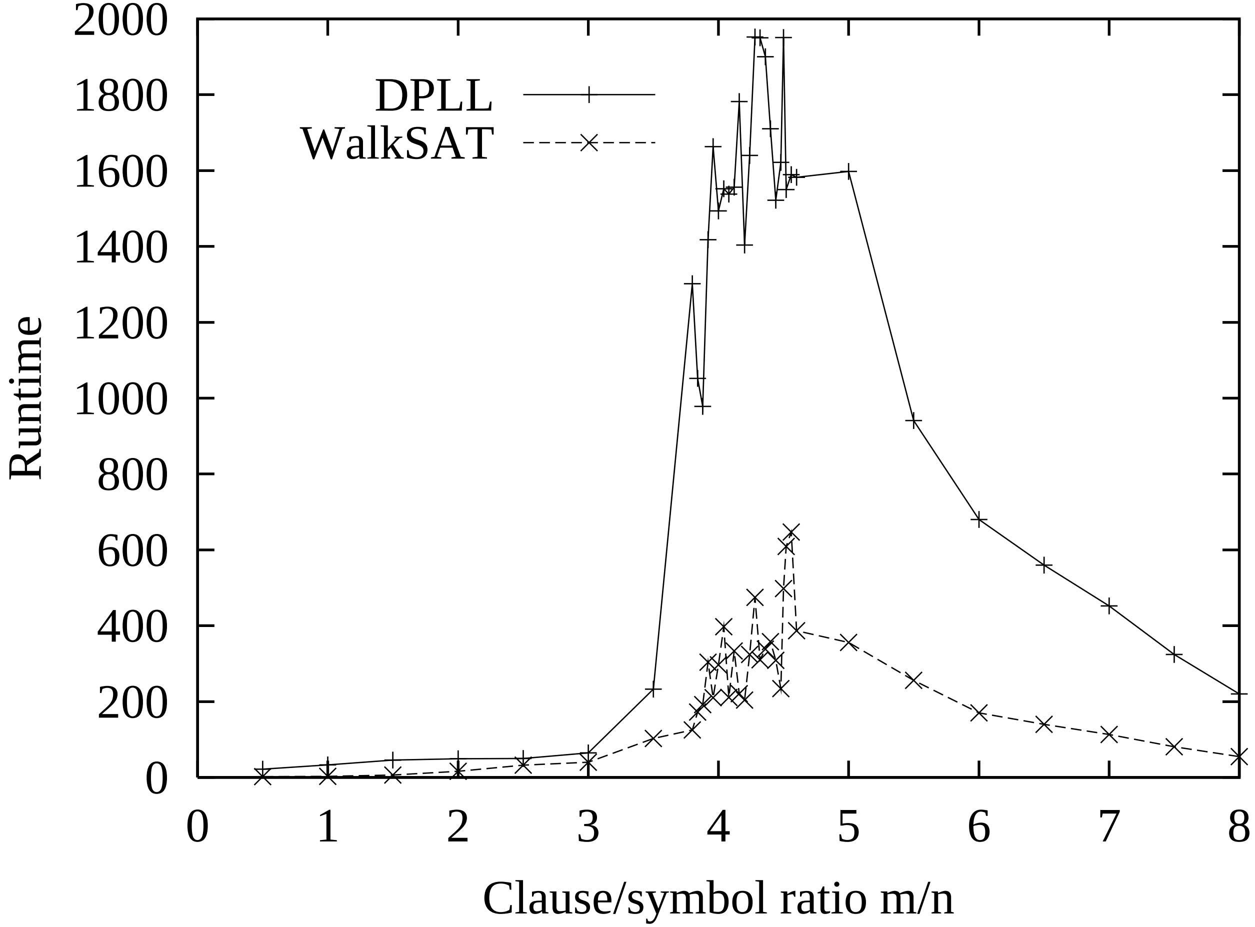
for $CNF_3(m, 50)$, (randomly generated sentences) threshold for a sharp drop in probability is around 4.3

hard problems: around the threshold value

$CNF_3(m, 50)$


Pr(satisfiable)
1
0.8
0.6
0.4
0.2
0
0
1
2
3
4
5
6
7
8
Clause/symbol ratio m/n

Performance of SAT solving algorithms



Summary

Logic can be used to encode search (and as we shall see later, planning) problems

SAT problems are solved with search algorithms (complete or local-search hill-climbing type)

Next lecture and reading

Search with non-deterministic actions and incomplete information

End of chapter 4 in 3rd edition