DM534
Introduction to Computer Science
Lecture on Satisfiability

Peter Schneider-Kamp

petersk@imada.sdu.dk

http://imada.sdu.dk/~petersk/

THE SAT PROBLEM

DM549: Propositional Variables

- Variable that can be either *false* or *true*
- Set P of **propositional variables**
- Example:

  $P = \{A, B, C, D, X, Y, Z, X_1, X_2, X_3, \ldots\}$

- A **variable assignment** is an assignment of the values *false* and *true* to all variables in P
- Example:

  X = *true*

  Y = *false*

  Z = *true*

DM549: Propositional Formulas

- **Propositional formulas**
  - If X in P, then X is a formula.
  - If F is a formula, then –F is a formula.
  - If F and G are formulas, then A $\wedge$ B is a formula.
  - If F and G are formulas, then A $\vee$ B is a formula.
  - If F and G are formulas, then A ➔ B is a formula.
- Example: (X ➔ (Y $\wedge$ –Z))

- Propositional variables or negated propositional variables are called **literals**
- Example: X, –X

Which formulas are satisfiable?

- X
- –X
- X $\wedge$ –X
- –X $\wedge$ –X
- X $\vee$ –X

- $X_1 \rightarrow X_2$
- $–X_1 \vee X_2$
- …

Satisfiability

- Variable assignment V **satisfies** formulas as follows:
  - V satisfies X in P iff V assigns X = *true*
  - V satisfies –A iff V does not satisfy A
  - V satisfies $A \wedge B$ iff V satisfies both A and B
  - V satisfies $A \vee B$ iff V satisfies at least one of A and B
  - V satisfies A ➔ B iff V does not satisfy A or V satisfies B

- A propositional formula A is **satisfiable** iff
  there is a variable assignment V such that V satisfies A.

- The Satisfiability Problem of Propositional Logic (**SAT**):
  - Given a formula A, decide whether it is satisfiable.

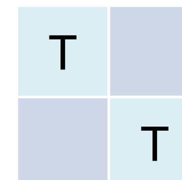
Modelling Problems by SAT

- propositional variables are basically bits
- model your problem by bits
- model the relation of the bits by a propositional formula
- solve the SAT problem to solve your problem

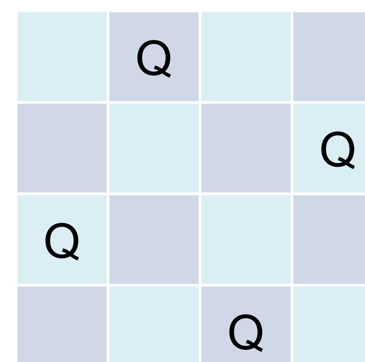
N-TOWERS & N-QUEENS

N-Towers & N-Queens

- N-Towers
  - How to place N towers on an NxN chessboard such that they do not attack each other?
  - (Towers attack horizontally and vertically.)



- N-Queens (restriction of N-Towers)
  - How to place N queens on an NxN chessboard such that they do not attack each other?
  - (Queens attack like towers + diagonally.)



Modeling by Propositional Variables

- Model NxN chessboard by NxN propositional variables $X_{i,j}$
- Semantics: $X_{i,j}$ is *true* iff there is a figure at row i, column j

- Example: 4x4 chessboard

$X_{1,1}$	$X_{1,2}$	$X_{1,3}$	$X_{1,4}$
$X_{2,1}$	$X_{2,2}$	$X_{2,3}$	$X_{2,4}$
$X_{3,1}$	$X_{3,2}$	$X_{3,3}$	$X_{3,4}$
$X_{4,1}$	$X_{4,2}$	$X_{4,3}$	$X_{4,4}$

- Example solution:
  - $X_{1,2} = X_{2,4} = X_{3,1} = X_{4,3} =$ *true*
  - $X_{i,j} =$ *false* for all other $X_{i,j}$

Reducing the Problem to SAT

- Encode the properties of N-Towers to propositional formulas
- Example: 2-Towers

$X_{1,1} \rightarrow -X_{1,2}$	"Tower at (1,1) attacks to the right"
$X_{1,1} \rightarrow -X_{2,1}$	"Tower at (1,1) attacks downwards"
$X_{1,2} \rightarrow -X_{1,1}$	"Tower at (1,2) attacks to the left"
$X_{1,2} \rightarrow -X_{2,2}$	"Tower at (1,2) attacks downwards"
$X_{2,1} \rightarrow -X_{2,2}$	"Tower at (2,1) attacks to the right"
$X_{2,1} \rightarrow -X_{1,1}$	"Tower at (2,1) attacks upwards"
$X_{2,2} \rightarrow -X_{1,2}$	"Tower at (2,2) attacks to the left"
$X_{2,2} \rightarrow -X_{2,1}$	"Tower at (2,2) attacks upwards"
$X_{1,1} \vee X_{1,2}$	"Tower in first row"
$X_{2,1} \vee X_{2,2}$	"Tower in second row"

$X_{1,1}$	$X_{1,2}$
$X_{2,1}$	$X_{2,2}$

- Form a conjunction of all encoded properties:

$$(X_{1,1} \rightarrow -X_{1,2}) \wedge (X_{1,1} \rightarrow -X_{2,1}) \wedge (X_{1,2} \rightarrow -X_{1,1}) \wedge (X_{1,2} \rightarrow -X_{2,2}) \wedge (X_{2,1} \rightarrow -X_{1,1}) \wedge$$
$$(X_{2,1} \rightarrow -X_{2,2}) \wedge (X_{2,2} \rightarrow -X_{1,2}) \wedge (X_{2,2} \rightarrow -X_{2,1}) \wedge (X_{1,1} \vee X_{1,2}) \wedge (X_{2,1} \vee X_{2,2})$$

Solving the Problem

- Determine satisfiability of

  $(X_{1,1} \Rightarrow -X_{1,2}) \wedge (X_{1,1} \Rightarrow -X_{2,1}) \wedge (X_{1,2} \Rightarrow -X_{1,1}) \wedge (X_{1,2} \Rightarrow -X_{2,2}) \wedge (X_{2,1} \Rightarrow -X_{1,1})$
  $\wedge (X_{2,1} \Rightarrow -X_{2,2}) \wedge (X_{2,2} \Rightarrow -X_{1,2}) \wedge (X_{2,2} \Rightarrow -X_{2,1}) \wedge (X_{1,1} \vee X_{1,2}) \wedge (X_{2,1} \vee X_{2,2})$

- Satisfying variable assignment (others are possible):
  - $X_{1,1} = X_{2,2} = \mathit{true}$
  - $X_{1,2} = X_{2,1} = \mathit{false}$

  (*true* ➔ −*false*) ∧ (*true* ➔ −*false*) ∧ (*false* ➔ −*true*) ∧ (*false* ➔ −*true*) ∧ (*false* ➔ −*true*)
  ∧ (*false* ➔ −*true*) ∧ (*true* ➔ −*false*) ∧ (*true* ➔ −*false*) ∧ (*true* ∨ *false*) ∧ (*false* ∨ *true*)

  (*true* ➔ *true*) ∧ (*true* ➔ *true*) ∧ (*false* ➔ −*true*) ∧ (*false* ➔ −*true*) ∧ (*false* ➔ −*true*) ∧
  (*false* ➔ −*true*) ∧ (*true* ➔ *true*) ∧ (*true* ➔ *true*) ∧ (*true* ∨ *false*) ∧ (*false* ∨ *true*)

  true ∧ *true* ∧ *true* ∧ *true* ∧ *true* ∧ *true* ∧ *true* ∧ *true* ∧ *true* ∧ *true*

  true

SAT Solving is Hard

- Given an assignment, it is easy to test whether it satisfies our formula
- BUT: there are many possible assignments!
- for m variables, there are 2^m possible assignments ☹
- SAT problem is a prototypical hard problem (NP-complete)

USING A SAT SOLVER

SAT Solvers

- SAT solver = program that determines satisfiability
- Plethora of SAT solvers available
  - For the best, visit http://www.satcompetition.org/
  - Different SAT solvers optimized for different problems
- In this course, we use the SAT solver **`lingeling`**
  - Very good overall performance at SAT Competition 2016
  - Parallelized versions available: `plingeling`, treengeling
  - Available from: http://fmv.jku.at/lingeling/

Conjunctive Normal Form (CNF)

- Nearly all SAT solvers require formulas in CNF
- CNF = **conjunction** of **disjunctions** of **literals**

- Example: 2-Towers

$(X_{1,1} \rightarrow -X_{1,2}) \wedge (X_{1,1} \rightarrow -X_{2,1}) \wedge (X_{1,2} \rightarrow -X_{1,1}) \wedge (X_{1,2} \rightarrow -X_{2,2}) \wedge (X_{2,1} \rightarrow -X_{1,1}) \wedge$
$(X_{2,1} \rightarrow -X_{2,2}) \wedge (X_{2,2} \rightarrow -X_{1,2}) \wedge (X_{2,2} \rightarrow -X_{2,1}) \wedge (X_{1,1} \vee X_{1,2}) \wedge (X_{2,1} \vee X_{2,2})$

- Conversion easy: $A \rightarrow B$ converted to $-A \vee B$

$(-X_{1,1} \vee -X_{1,2}) \wedge (-X_{1,1} \vee -X_{2,1}) \wedge (-X_{1,2} \vee -X_{1,1}) \wedge (-X_{1,2} \vee -X_{2,2}) \wedge (-X_{2,1} \vee -X_{1,1})$
$\wedge (-X_{2,1} \vee -X_{2,2}) \wedge (-X_{2,2} \vee -X_{1,2}) \wedge (-X_{2,2} \vee -X_{2,1}) \wedge (X_{1,1} \vee X_{1,2}) \wedge (X_{2,1} \vee X_{2,2})$

- Write formulas in CNF as a list of clauses (= lists of literals)
- Example:

$[[-X_{1,1}, -X_{1,2}],[-X_{1,1}, -X_{2,1}],[-X_{1,2},-X_{1,1}],[-X_{1,2},-X_{2,2}],[-X_{2,1},-X_{1,1}],[-X_{2,1},-X_{2,2}],[-X_{2,2},-X_{1,2}],$
$[-X_{2,2},-X_{2,1}],[X_{1,1},X_{1,2}],[X_{2,1},X_{2,2}]]$

Conversion to CNF

- Implications can be replaced by disjunction:
  - A ➔ B converted to $-A \vee B$
- DeMorgan's rules specify how to move negation “inwards”:
  - $–(A \wedge B) = –A \vee –B$
  - $–(A \vee B) = –A \wedge –B$
- Double negations can be eliminated:
  - $–(–A) = A$
- Conjunction can be distributed over disjunction:
  - $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$

Variable Enumeration

- SAT solvers expect variables to be identified with integers
- Starting from 1 and up to the number of variables used
- Necessary to map modeling variables to integer!
- Example: 4x4 chessboard
  - $X_{i,j}$ becomes 4*(i-1)+j

$X_{1,1}$	$X_{1,2}$	$X_{1,3}$	$X_{1,4}$
$X_{2,1}$	$X_{2,2}$	$X_{2,3}$	$X_{2,4}$
$X_{3,1}$	$X_{3,2}$	$X_{3,3}$	$X_{3,4}$
$X_{4,1}$	$X_{4,2}$	$X_{4,3}$	$X_{4,4}$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

(Simplified) DIMACS Format

- Description of DIMACS format for CNF (BB: dimacs.pdf)
- Simplified format (subset) implemented by most SAT solvers:
  - http://www.satcompetition.org/2016/format-benchmarks2016.html

- 2 types of lines for input
  - Starting with "c ": **comment**
  - Starting with "p ": **problem**
- 3 types of lines for output
  - Starting with "c ": **comment**
  - Starting with "s ": **solution**
  - Starting with "v ": **variable assignment**

Input Format 1/2

- **Comments**
  - Anything in a line starting with "c " is ignored
  - Example:

```
c This file contains a SAT encoding of the 4-queens problem!
c The board is represented by 4x4 variables:
c          1  2  3  4
c          5  6  7  8
c          9 10 11 12
c         13 14 15 16
c
```

Input Format 2/2

- **Problem**
  - Starts with "`p cnf #variables #clauses`"
  - Then one clause per line where
    - Variables are numbered from 1 to #variables
    - Clauses/lines are terminated by 0
    - Positive literals are just numbers
    - Negative literals are negated numbers
  - Example:

```
p cnf 16 80
 -1  -2 0
...
-15 -16 0
 1  2  3  4 0
...
13 14 15 16 0
```

Output Format 1/2

- **Comments**
  - just like for the input format
  - Example:

```
c reading input file examples/4-queens.cnf
```

- **Solution**
  - Starts with "`s `"
  - Then either "`SATISFIABLE`" or "`UNSATISFIABLE`"
  - Example:

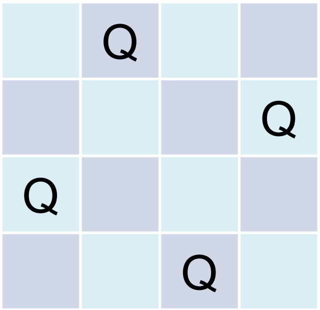
```
s SATISFIABLE
```

Output Format 2/2

- **Variable assignment**
  - Starts with "v "
  - Then list of literals that are assigned to true
    - "1" means variable 1 is assigned to true
    - "–2" means variable 2 is assigned to false
  - Terminated by "0"
  - Example:

```
v -1 2 -3 -4 -5 -6 -7 8 9 -10 -11 -12 -13 -14 15 -16 0
```

1	**2**	3	4	*false **true** false false*
5	6	7	**8**	*false false false **true***
9	10	11	12	***true** false false false*
13	14	**15**	16	*false false **true** false*



Running the SAT Solver

1. Save the comment and problem lines into .cnf file.
2. Invoke the SAT solver on this file.
3. Parse the standard output for the solution line.
4. If the solution is "SATISFIABLE", find variable assignment.

- Example:

```
lingeling 4-queens.cnf
```

WRITING A SAT SOLVER

Brute-Force Solver

- iterate through all possible variable assignments
- for each assignment
  - if the assignment satisfies the formula
    - output SAT and the assignment
- if no assignment is found, output UNSAT

Python Implementation

```
import itertools, sys

def parse_dimacs(lines):
  clauses = []
  while lines:
    line, lines = lines[0], lines[1:]
    if line[0] == "p":
      num_vars, num_clauses = [int(x) for x in line.split()[2:]]
      clauses = [[int(x) for x in line.split()[:-1]] for line in lines]
      return num_vars, [clause for clause in clauses if clause]

def output_dimacs(num_vars,d):
  if d:
    vars = [str(x) if d[x] else str(-x) for x in range(1,num_vars+1)]
    return "SATISFIABLE\ns "+" ".join(vars)
  return "UNSATISFIABLE"

...
```

Python Implementation

```
...

def reduce_clause(clause,d):
  new_clause = []
  for literal in clause:
    if not literal in d:
      new_clause.append(literal)
    elif d[literal]:
      return True
  return new_clause

def conflict(d,f):
  for clause in f:
    if not reduce_clause(clause,d):
      return True
  return False

...
```

Python Implementation

```
...

def solve(f,num_vars):
  for v in itertools.product([False,True],repeat=num_vars):
    d = {}
    for i in range(num_vars):
      d[i+1] = v[i]
      d[-i-1] = not v[i]
    if not conflict(d,f):
      return d
  return False

if __name__ == "__main__":
  num_vars, clauses = parse_dimacs(open(sys.argv[1]).readlines())
  result = solve(clauses,num_vars)
  print(output_dimacs(num_vars,result))
```

Empirical Evaluation

- For n variables, there are 2^n possible variable assignments

- **Example:**
- 2^{16} = 65,536 assignments for 4-queens (1 second)
- 2^{25} = 33,554,432 assignments for 5-queens (7 minutes)
- 2^{36} = 68,719,476,736 assignments for 6-queens (2 weeks)
- 2^{49} = 562949953421312 assignments for 7-queens (400 years)
- 2^{64} assignments for 8-queens (age of the universe)
- 2^{81} assignments for 9-queens (ahem … no!)

Fast Forwarding 60+ Years

- Incremental assignments
- Backtracking solver
- Pruning the search

Empirical Evaluation

- For n variables, there are 2^n possible variable assignments

- Example:
  - 2^{100} assignments for 10-queens (1.77 seconds)
  - 2^{121} assignments for 11-queens (1.29 seconds)
  - 2^{144} assignments for 12-queens (9.15 seconds)
  - 2^{169} assignments for 13-queens (5.21 seconds)
  - 2^{196} assignments for 14-queens (136.91 seconds)
  - ...

Fast Forwarding 60+ Years

- Incremental assignments
- Backtracking solver
- Pruning the search
- Backjumping
- Conflict-driven learning
- Restarts
- Forgetting

Empirical Evaluation

- For n variables, there are 2^n possible variable assignments

- Example:
  - 2^{256} assignments for 16-queens (0.02 seconds)
  - 2^{1024} assignments for 32-queens (0.10 seconds)
  - 2^{4096} assignments for 64-queens (1.08 seconds)
  - 2^{16384} assignments for 128-queens (17.92 seconds)
  - 2^{65536} assignments for 256-queens (366.05 seconds)
  - ...

Efficient SAT Solving

- in many cases, SAT problems can be solved efficiently
- state-of-the-art SAT solvers can be used as black Sboxes
- success of SAT solvers based on
  - relatively simple but highly-optimized algorithms
  - innovative and very pragmatic data structures
- used extensively for scheduling, hardware and software verification, mathematical proofs, ...

Take Home Slide

- SAT Problem = satisfiability of propositional logic formulas
- SAT used to successfully model hard (combinatorial) problems
- solving the SAT problem is hard in the general case
- advanced SAT solvers work fine (most of the time)