EE570 Term Project

Gowtham Kaki

PURDUE UNIVERSITY

Overview

- Three papers on logic and applications
- Chu-Min Li, Zhu Z, Manya, F and Simon L, Minimum Satisfiability and Applications, IJCAI, 2011.
- Pulina L, and Tachella A, A Structural Approach to Reasoning with Quantified Boolean Formulas, IJCAI, 2009.
- Huang M, Shi X, Jin F, and Zhu X, Using First-Order Logic to Compress Sentences, AAAI, 2012

SAT Problem

- Given a formula in propositional logic, does there exist an assignment to literals such that the formula evaluates to true?

$$(x \vee z) \wedge (y \vee \neg z)$$

- SAT problem for formula with arbitrary number of literals is NP-complete.
- Nevertheless, very important problem as it finds applications in
  - Propositional theorem proving, and
  - Solving other NP-hard problems

SAT in Theorem Proving

- Given a propositional formula ϕ,

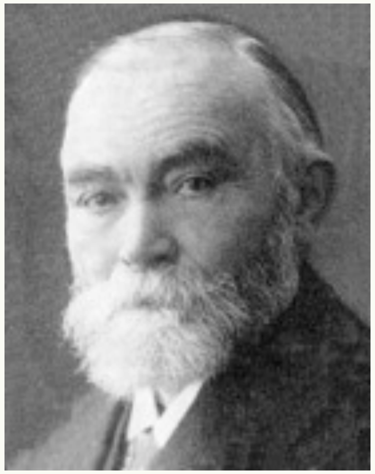
  ϕ is valid ≡ ¬ϕ is UNSAT

- Eg: How do you verify that Frege's theorem is valid?

Frege's theorem in propositional logic [edit]

In propositional logic, Frege's theorems refers to this tautology:

$$(P \to (Q \to R)) \to ((P \to Q) \to (P \to R))$$



SAT in Theorem Proving

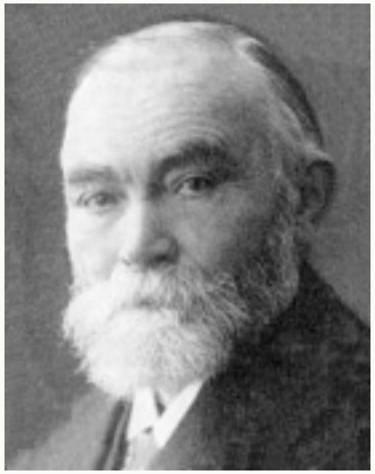
- Given a propositional formula ϕ,
  ϕ is valid ≡ ¬ϕ is UNSAT

- Eg: How do you verify that Frege’s theorem is valid?

Frege's theorem in propositional logic [edit]

In propositional logic, Frege's theorems refers to this tautology:

$$(P \to (Q \to R)) \to ((P \to Q) \to (P \to R))$$



- ¬ (Frege's theorem) → $\boxed{\text{SAT Solver}}$ → UNSAT

- SAT solver is a decision procedure for propositional logic

Encoding NP-hard problems in SAT

- Encode 4-Queens problem in SAT:
  - 16 boolean variables : x[i][j] $1 \leq i, j \leq 4$. x[i][j] is true if and only if board[i][j] has queen in the solution.
  - For every x[i][j], x[i][k], assert $\neg$(x[i][j] $\wedge$ x[i][k]). Similarly for columns and diagonals.
  - The result is a large conjunction of formulas like above.
- Conjunction → $\boxed{\text{SAT Solver}}$ → SAT with model M

F	F	T	F
T	F	F	F
F	F	F	T
F	T	F	F

MaxSAT and MinSAT problems

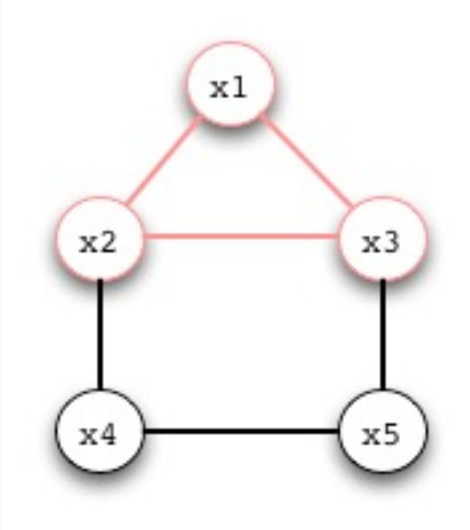
- Consider en extension of SAT problem where set of CNF clauses of the formula are divided into two sets:
  - Hard clauses - Inviolable constraints. Need to be satisfied at any cost
  - Soft clauses - Optional constraints. Some or all of them can remain unsatisfiable.
- MaxSAT problem - Maximize the number of SAT soft constraints (≡ Minimize UNSAT).
- MinSAT problem - Minimize the number of SAT soft constraints (≡ Maximize UNSAT)
- Observe that MaxSAT and MinSAT are dual problems.

MinSAT Application - MaxClique

- Clique of a graph $G = (V,E)$ is a sub-graph $G' = (V',E')$, such that $V' \subseteq V$, $E' \subseteq E$, and $E' = V' \times V'$.
- MaxClique problem - Given a graph $G=(V,E)$, find a clique of G with largest number of vertices.
- MinSAT encoding of MaxClique: Let $G'=(V',E')$ be the MaxClique sub-graph of G. We add:
  - A variable x_i for each vertex $v_i \in V$. x_i is true iff $v_i \in V'$.
  - A hard constraint $\neg x_i \vee \neg x_j$ for every pair of non-adjacent vertices v_i and v_j.
  - A soft constraint $\neg x_i$ for every i.
- Minimizing # of SAT soft constraints $\equiv$ Maximizing # of vertices in G'.

MinSAT Application - MaxClique

- Hard : (¬x1 ∨ ¬x4) ∧ (¬x1 ∨ ¬x5) ∧ (¬x2 ∨ ¬x5) ∧ (¬x3 ∨ ¬x4)
- Soft : (¬x1) ∧ (¬x2) ∧ (¬x3) ∧ (¬x4) ∧ (¬x5)




```
Welcome to DrRacket, version 5.3.6 [3m].
Language: racket; memory limit: 128 MB.
> (minsat-solve '(((not x1) (not x4)) ((not x1) (not x5))
                  ((not x2) (not x5)) ((not x3) (not x4)))
                '(((not x1)) ((not x2)) ((not x3)) ((not x4))
                                  ((not x5))));
SAT
Max number of unsat soft-constraints : 3
Model:
((x1 #t) (x4 #f) (x5 #f) (x3 #t) (x2 #t))
```

DPLL

- Davis-Putnam-Logemann-Loveland (DPLL) algorithm is an algorithm to decide satisfiability of propositional logic formulae in CNF form.
- Based on backtracking and propositional clause resolution (Robinson, 1965).

```
Data: φ : Set of CNF clauses
      Γ : Current assignment to variables
Result: A (possibly empty) model for the formula.
(φ,Γ) := unitPropagate (φ,Γ);
if φ contains empty clause then
|  return {}
end
if φ is empty then
|  return Γ
end
v := selectVariable (φ);
return (DPLL (φ ∪ {v},Γ) ∨ DPLL (φ ∪ {¬v},Γ))
```

Algorithm 1: DPLL (φ,Γ)

$$\frac{\Sigma \models \phi \qquad \Sigma \models \neg\phi \vee \psi 1 \vee .. \vee \psi n}{\Sigma \models \psi 1 \vee .. \vee \psi n}$$

$$\frac{\Sigma \models \phi \qquad \Sigma \models \phi \vee \psi 1 \vee .. \vee \psi n}{\Sigma \models \top}$$

MinSatz

- Algorithm proposed by the paper for MinSat problem
- Extends DPLL with the weighted soft constraints.
- Goal is to maximize the number of unsatisfied soft constraints.

```
Data: φ : Set of hard and soft CNF clauses
      LB : Lower bound (Lowest cost incurred so far)
Result: Minimum cost of satisfied soft constraints
(φ) := hardUnitPropagate (φ);
if φ contains empty clause then
    return -1
end
if φ is empty ∨ φ only contains empy soft clauses then
    return n-empty(φ)
end
UB := n-empty(φ) + overestimation(φ);
if UB ≤ LB then
    return LB
end
v := selectVariable (φ);
LB := MinSatz (φ ∪ {v},LB);
LB := MinSatz (φ ∪ {¬v},LB);
return LB
```

Algorithm 2: MinSatz (φ,LB)

- Similar to (α,β)-pruning, passes the maximum found so far.
- uses *overestimation* as static evaluator.

MinSatz - Overestimation

- *Overestimate(ϕ)* finds the maximum number of unsatisfiable soft constraints out of the remaining soft constraints when the current assignment is completed in some way.
- A naive overestimation just returns number of remaining soft constraints.
- Paper proposes an intelligent overestimation calculation based on following observations:
  - Consider the two soft clauses containing literals x and $\neg x$. Surely, both cannot be UNSAT simultaneously.
  - consider the case where there are two unit soft clauses with literals $x1$ and $x2$, respectively, and a hard clause $(x1 \vee x2)$. Again, both soft clauses cannot be simultaneously UNSAT.

MinSatz - Overestimation

- *Overestimate(ϕ)* makes use of aforementioned observations:
  - Construct a graph G” that contains a vertex for each soft constraint.
  - Add an edge between two vertices if corresponding soft constraints cannot be simultaneously satisfied (as per observations made).
  - Observe that for every clique in G”, utmost one soft constraint can be unsatisfiable. Therefore, number of cliques in the clique partition of G” gives upper bound on number of unsatisfiable soft constraints.

MinSatz - Critique

- Consider a ϕ with hard part = x1 ∧ (x2 ∨ x3) ∧ (¬x2 ∨ x3) ∧ (x2 ∨ ¬x3) ∧ (¬x2 ∨ ¬x3) and soft part = (¬x1)
- *hardUnitPropagate* propagates hard unit clause (x1), therefore makes lone soft constraint UNSAT (makes it empty).

```
Data: φ : Set of hard and soft CNF clauses
      LB : Lower bound (Lowest cost incurred so far)
Result: Minimum cost of satisfied soft constraints
1  (φ) := hardUnitPropagate (φ);
2  if φ contains empty clause then
3  |   return -1
4  end
5  if φ is empty  ∨  φ only contains empy soft clauses then
6  |   return n-empty(φ)
7  end
8  UB := n-empty(φ) + overestimation(φ);
9  if UB ≤ LB then
10 |   return LB
11 end
12 v := selectVariable (φ);
13 LB := MinSatz (φ ∪ {v},LB);
14 LB := MinSatz (φ ∪ {¬v},LB);
15 return LB
```

Algorithm 2: MinSatz (ϕ,LB)

- Predicate at Line 5 evaluates to true. MinSatz returns 1.
- But, hard part of ϕ is UNSAT!

MinSatz - Critique

- MinSatz returning values > 0 for UNSAT ϕ is clearly inappropriate.
- Interesting Qn: what is the weakest pre-condition under which the given algorithm can return correct answer?
- Ans: ϕ has to be a CNF formula with only Horn clauses. A CNF clause is a Horn clause if it contains atmost one +ve literal (a non-negated variable).
- Unit resolution is complete for Horn CNF formula [Boyer R S, 1971].
- Therefore, if formula is UNSAT, then it wouldn't even reach Line 5 of MinSatz.

MinSatz - Critique

- Consider the case when LB>0 (i.e., some soft constraints can be UNSAT when hard part is SAT)
- Assume that first recursive call (line 13) encounters unsatisfiability. Consequently, it returns -1.
- Now, -1 becomes LB for second recursive call

```
Data: φ : Set of hard and soft CNF clauses
      LB : Lower bound (Lowest cost incurred so far)
Result: Minimum cost of satisfied soft constraints
1  (φ) := hardUnitPropagate (φ);
2  if φ contains empty clause then
3  |  return -1
4  end
5  if φ is empty ∨ φ only contains empy soft clauses then
6  |  return n-empty(φ)
7  end
8  UB := n-empty(φ) + overestimation(φ);
9  if UB ≤ LB then
10 |  return LB
11 end
12 v := selectVariable (φ);
13 LB := MinSatz (φ ∪ {v},LB);
14 LB := MinSatz (φ ∪ {¬v},LB);
15 return LB
```

Algorithm 2: MinSatz (φ,LB)

- Unfortunately, we have lost the best solution found so far!
- More importantly, MinSatz might return non-optimal solution.

MinSatz - Critique

- Consider a ϕ with hard constraints: {(x1 ∨ x2), (¬x1 ∨ x3)} and soft part = {(¬x1), (¬x2), (¬x3)}
- Assume *selectVariable* selects x1. First recursive call sets x1 to *true*.
- hardUnitPropagate immediately returns SAT, assigning *true* to x3.

```
Data: ϕ : Set of hard and soft CNF clauses
      LB : Lower bound (Lowest cost incurred so far)
Result: Minimum cost of satisfied soft constraints
(ϕ) := hardUnitPropagate (ϕ);
if ϕ contains empty clause then
  | return -1
end
if ϕ is empty ∨ ϕ only contains empy soft clauses then
  | return n-empty(ϕ)
end
UB := n-empty(ϕ) + overestimation(ϕ);
if UB ≤ LB then
  | return LB
end
v := selectVariable (ϕ);
LB := MinSatz (ϕ ∪ {v},LB);
LB := MinSatz (ϕ ∪ {¬v},LB);
return LB
```

Algorithm 2: MinSatz (ϕ,LB)

- Now, ϕ is empty. So, MinSatz returns with number of empty soft constraints, which is 2. But, correct answer is 3!

MinSatzEE570

- MinSatzEE570 is modified MinSatz with corrections for inconsistencies.
- Reduces MinSAT to SAT problem and invokes DPLL when all soft constraints are UNSAT (orange box).
- Accounts for unsatisfiability result and consequent -ve return value from recursive calls (blue box).

```
Data: φ : Set of hard and soft CNF clauses
      LB : Lower bound (Lowest cost incurred so far)
Result: Minimum cost of satisfied soft constraints
(φ) := hardUnitPropagateEE570 (φ);
if φ contains empty clause then
    return -1
end
if φ is empty then
    return n-empty(φ)
end
if all soft clauses in φ are empty then
    if DPLL(hard(φ),{}) returns non-empty model then
        return n-empty(φ)
    else
        return -1
    end
end
UB := n-empty(φ) + overestimation(φ);
if UB ≤ LB then
    return LB
end
v := selectVariable (φ);
LB1 := MinSatz (φ ∪ {v},LB);
LB := (LB1≤0)?LB:LB1;
LB2 := MinSatz (φ ∪ {¬v},LB);
if LB2 = -1 ∧ LB1 = -1 then
    return-1
else
    return (LB2≤0)?LB:LB2
end
```

Algorithm 3: MinSatzEE570 (φ,LB)

MinSatzEE570

- Uses specialized hardUnitPropagateEE570.
- Adds tautological clauses of form (x ∨ ¬x) for each x that gets eliminated during unit propagation.
- Eg: usual unit propagation on (x1) ∧ (x1 ∨ x2 ∨ ... ∨ xn) returns ⊤(empty), after {x1↦ #t}.
- hardUnitPropagateEE570 returns (x2 ∨ ¬x2) ∧ ... ∧ (xn ∨ ¬xn) so that MinSatzEE570 can chose optimal assignments for x2, .. , xn

```
Data: φ : Set of hard and soft CNF clauses
      LB : Lower bound (Lowest cost incurred so far)
Result: Minimum cost of satisfied soft constraints
(φ) := hardUnitPropagateEE570 (φ);
if φ contains empty clause then
|  return -1
end
if φ is empty then
|  return n-empty(φ)
end
if all soft clauses in φ are empty then
|  if DPLL(hard(φ),{}) returns non-empty model then
|  |  return n-empty(φ)
|  else
|  |  return -1
|  end
end
UB := n-empty(φ) + overestimation(φ);
if UB ≤ LB then
|  return LB
end
v := selectVariable (φ);
LB1 := MinSatz (φ ∪ {v},LB);
LB := (LB1≤0)?LB:LB1;
LB2 := MinSatz (φ ∪ {¬v},LB);
if LB2 = -1 ∧ LB1 = -1 then
|  return-1
else
|  return (LB2≤0)?LB:LB2
end
```

Algorithm 3: MinSatzEE570 (φ,LB)

MinSatzEE570 - Implementation

- Both DPLL and MinSatzEE570 were implemented in racket, a dialect of scheme - https://github.com/gowthamk/ee570
- Racket provides exception handling, so failure case (eg: UNSAT in a recursive call) can be caught and handled at appropriate location (eg: reset LB value).
- Racket is still scheme, so rewrite system from HW-3 was reused to perform unit propagation (which is essentially *boolean-simplify* with appropriate rules). Also, HW-2 was reused to maintain consistent and non-redundant assignments to literals.
- Experiments performed on MaxClique calculation verified correctness of implementation.

Quantified Boolean Formulas

- Informally, quantified boolean formulas are propositional formulas extended with existential and universal quantifiers.
- Formally, A quantified boolean formula (QBF) is a formula of form $\varphi = Q_1 z_1 Q_2 z_2 \ldots Q_n z_n\ \phi$, where
  - $Q_i \in \{\forall, \exists\}$.
  - $z_1, z_2 \ldots z_n$ are distinct boolean variables. $Q_1 z_1 Q_2 z_2 \ldots Q_n z_n$ is called prefix of φ,
  - ϕ, called matrix of φ, is a propositional formula.
- Without loss of generality, we assume ϕ to be in CNF.

Semantics of QBFs

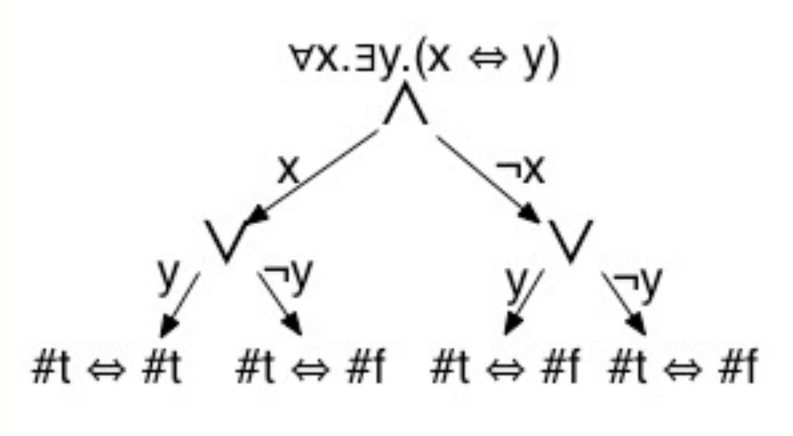
- Consider $(x \vee y)$. Is it SAT?
- Now, consider $\forall x.\forall y.(x \vee y)$. Is it SAT? -- Contradictory clause.
- Consider $(x \vee \neg x)$. SAT?
- Now, $\forall x. (x \vee \neg x)$. SAT?
- What do we observe?

> - If prefix is empty, then semantics of φ is same as semantics of propositional formula ϕ
> - If $\varphi = \exists x \psi$, φ is true iff φ_x is true or $\varphi_{\bar{x}}$ is true.
> - If $\varphi = \forall x \psi$, φ is true iff both φ_x and $\varphi_{\bar{x}}$ are true.

- A universally quantified formula is SAT, if its propositional matrix is valid, i.e., if it is true under all possible interpretations of universally quantified variables.
- Similarly, for existential quantification, it can be observed that QBF is SAT if there exists atleast one interpretation of existentially quantified variables for which propositional matrix is true.

Backtracking Search of QBF

- Semantics of QBF (the process of its evaluation) can be represented as an AND-OR tree.
- For eg, AND-OR tree for $\forall x.\exists y.(x \Leftrightarrow y)$ is shown here.



- The tree naturally leads to backtracking based search procedure.
- Optimizations possible. For eg: Backtrack as soon as search encounters a contradictory clause. No need to unroll fully.

Clause Resolution for QBFs

- General clause resolution rule for propositional logic is well known.

$$\frac{\Sigma \models \Phi_1 \ \vee \ \ldots \ \vee \ \Phi_n \ \vee \ \Psi \qquad \Sigma \models \neg\Psi \ \vee \ \Theta_1 \ldots \ \vee \ \Theta_m}{\Sigma \models \Phi_1 \ \vee \ \ldots \ \vee \ \Phi_n \ \vee \ \Theta_1 \ldots \ \vee \ \Theta_m}$$

- Clause resolution rule for QBFs is similar, except for
  - The literal that occurs with different polarity in both clauses (ψ in the fig.) should be an existential literal
  - $\Phi i \neq \Theta j$ where$0 \leq i \leq n$,and$0 \leq j \leq m$

STRUQS

- STRUQS is the solver for quantified boolean formulas described in the paper [Pulina L, and Tachella A, A Structural Approach to Reasoning with Quantified Boolean Formulas, IJCAI, 2009]
- Alternates between backtracking search with optimizations (backjumping) and clause resolution.
- But, DPLL already does that for propositional logic. What is the novelty, then?
- Unlike propositional logic, clause resolution for QBFs is complete![Kleine-Büning et al., 1995]
- Technically, one can construct a solver with only clause resolution. But, clause resolution may lead to exponential blowup of formulas. So, it is not better than backtracking search.

STRUQS

- STRUQS alternates between search and resolution based on a heuristic.
- The heuristic is a function of structure of the QBF (thence the name). Its optimal value was determined using experiments.
- Experiments performed on QBFEVAL'08 dataset. STRUQS solved 39% of dataset in allotted 600s of CPU time. Would have been placed 3rd in QBFEVAL'08.
- The conclusion is that it is profitable to employ search and resolution alternatively rather than relying on any one of them.

Compressing English Sentences

- Third paper [Huang M, Shi X, Jin F, and Zhu X, Using First-Order Logic to Compress Sentences, AAAI, 2012] uses a combination of following to compress English sentences:
  - NLP Parser aided Parts-of-speech (POS) tagging,
  - Inference rules in first-order logic (FOL),
  - Markovian Logic Network (MLN) to learn relative weights for rules and carry out compression.
- Main contributions: FOL rules for compressing a sentence by word/sentence deletion. Using MLNs to impose those rules on a dataset.

Rules

- Consider sentence "I am positively sure". Adjective "positive" can be removed.

  adjective(i) $\Rightarrow$ delete(i)

- Counterexample : "the relative velocity of object is 2 km/s".

  adjective(i) $\wedge$ ($\exists$ j.property(words[i, i + i], j)) $\Rightarrow$ $\neg$delete(i)

- Premises of 2nd rule subsume those of 1st rule, yet it arrives at a contradicting conclusion!
  - Dilemma : What rule to apply?
  - Solution : Add weights to rules to decide the *most-applicable* rule.

Markovian Logic Network

- The problem of deciding the *most-applicable* rule, when rules have relative weights, is an old one. Markovian Logic Network solves the problem for first-order logic.

- A Markovian Logic Network (MLN) is a probabilistic deductive system that makes uncertain inference based on first-order rules annotated with probabilities.

- Formally, a MLN is a set of pairs (φ, w), where φ is a formula in first-order logic and w is a weight for the formula, a real number

- For eg, when rules propositional logic are annotated with w=1, (assuming any other rule has w=0), an MLN would deduce tautology with w=1 and contradiction with w=0.

Experiments

- Many rules as described previously were constructed based on English grammar and experience. Rules makes use of POS tags generated by Stanford NLP Parser.
- Problem : How to annotate those rules with weights? Solution : Let MLN learn the weights utilizing a training set.
- Existing software (thebeast) was used to train MLN. About 1/3rd of the dataset is used to train MLN, which was used to compress the rest (2/3rds) of the dataset.
- One of their evaluations used humans to rank the quality of compressed sentences against those generated by others.
- Evaluators ranked new compression model as best for written corpus, and worse than others for spoken corpus.

Sample Results of Compression

Original	From the bottom of the list of nominees he climbed to the top.
Human	he climbed to the top.
Ours	he climbed to the top.
T3	the list of nominees climbed to the top.
SVTL	From the bottom of the list of nominees he climbed to the top.

Table 6: A good compression example from the written corpus.

Original	We 'll have Steve back next Monday morning to bring us a little bit more information.
Human	We 'll have Steve back Monday morning to bring us more.
Ours	We 'll have Steve back next Monday morning to bring us information.
T3	We 'll have Steve back next Monday morning to bring us a little bit more information.
SVTL	We 'll have Steve back to bring us bit more information.

Table 7: A good compression example from the spoken corpus.

Written	
Original	" Many of the things which bring joy to our hearts in the countryside have been destroyed , " said Sir David .
Human	" Many of the things which bring joy in the countryside have been destroyed , " said Sir David .
Ours	said Sir David .
T3	" Many In the things which , " said Sir David .
SVTL	" Many have been destroyed , " said Sir David .
Spoken	
Original	This is a breach that smells a lot–
Human	This breach smells
Ours	a –
T3	This is smells a lot –
SVTL	This is a breach that smells a lot –

Table 8: Two bad compression examples on the written and spoken corpora respectively.

Conclusion

- SAT problem for propositional logic and quantified boolean formulas (QBFs).
- DPLL algorithm for propositional SAT and STRUQS approach to QBF SAT.
- Propositional MinSAT problem. MinSatz algorithm, its inconsistencies and a corrected MinSatzEE570.
- Applications of SAT and MinSAT
- Compressing English sentences using first-order logic.

Thank you!