Design and Implementation of Efficient Techniques for ALLSAT

- Tejas Nikumbh (10D070019)
- Guide : Prof. Virendra Singh
- Co-Guide : Prof. Shankar Balachandran

Outline

- Introduction
  - Problem Definition
  - Overview & Focus Area
- Background Literature
  - All Clause All SAT
  - All SAT Using Minimal Blocking Clauses
- Implementation & Observation
- Using Prediction Technique
- Proposed Idea – Majority Rule
- Conclusions
- Future Work

Problem Definition

- Problem for All SAT: Convert CNF form to equivalent DNF form
- DNF form is representative of all satisfying solutions of the corresponding CNF form
- CNF form:

$$(x_1 \ V \ x_2 \ V \ x_3) \ \Lambda \ (x_2 \ V \ \bar{x}_1 \ V \ x_3) \ \Lambda \ (\overline{x_4})$$

- DNF form:

$$(x_1 \ \Lambda \ x_2 \ \Lambda \ x_3) \ V \ (x_2 \ \Lambda \ \overline{x}_1 \ \Lambda \ x_3) \ V \ (\overline{x_4})$$

Overview & Focus Area

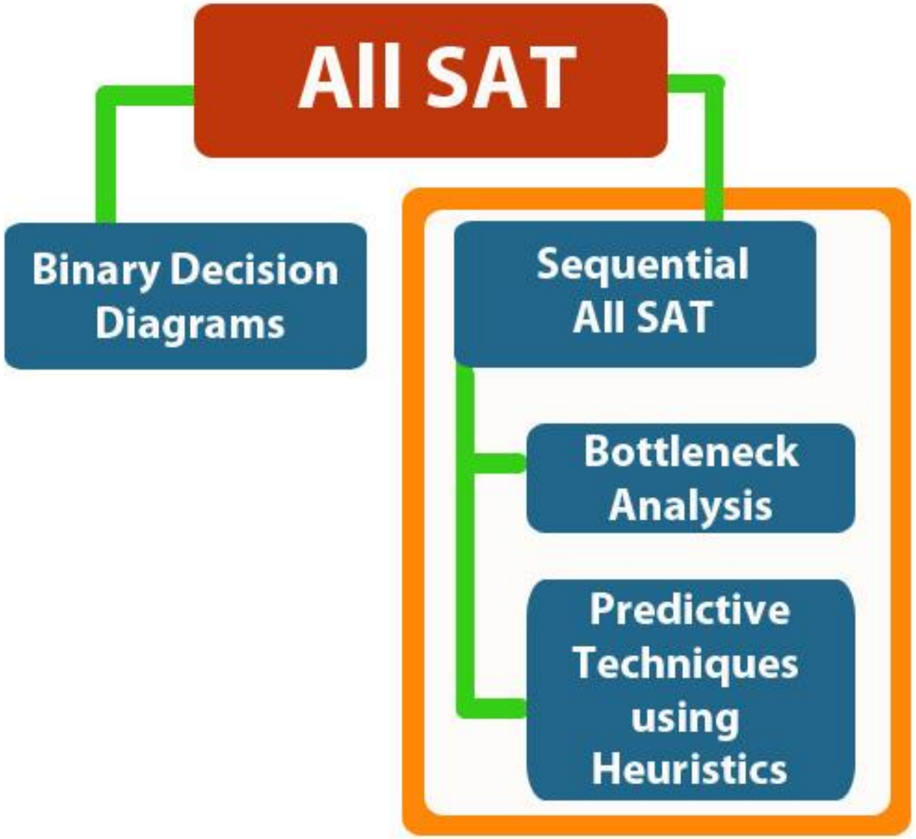

Fig 1: Project Organization Overview

All Clause All SAT Technique

- Improvement upon Naïve Algorithm
  - Step 1: Produce a solution
  - Step 2: Produce a cube cover for this solution
      - » Add this cover to the DNF form
  - Step 3: Generate blocking clause for cover
  - Step 4: Update problem using blocking clause
  - Do 1 to 4 until problem becomes UNSAT

Cube Cover

- Might be possible that only a subset of variable assignments from solution are required to satisfy the formula
- Other variable assignment values are don’t care
- Such a subset is called a cube cover

Intuition [1]

- What if we let cubes overlap ?

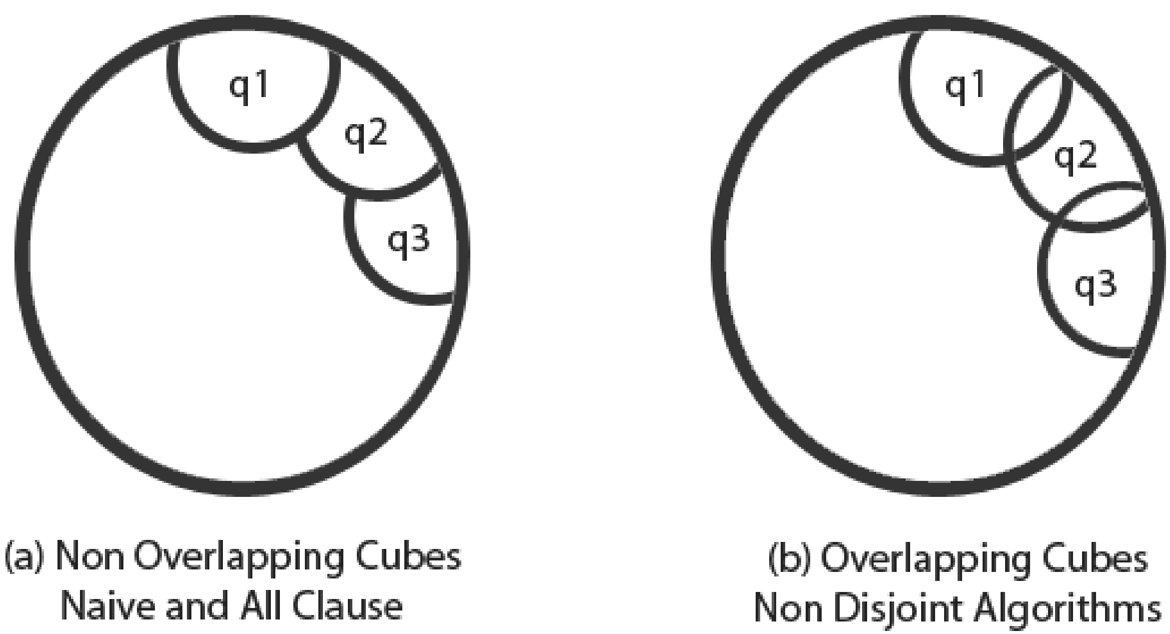

Fig 2: Comparison of Disjoint and Non Disjoint All SAT Algorithms

Intuition [1]

- Model Counting (Computing number of solutions of a SAT Instance), is Sharp –P complete ($P^{\#P}$)- As in case of All Clause & Naïve
- However, Minimal DNF Cover computation in most cases is Sigma P2 (P^{NP}) complex (less than Sharp - P)
- Since our problem is about computing DNF Cover, we might be doing more work than necessary

Implementation

- Implementation was done in python
- SAT solver: MiniSAT python bindings
- Additional Data Structure : ‘Membership Matrix’ for computation of cube covers
- Based on two algorithm templates
  - Algorithm Template 1
  - Algorithm Template 2 [called from Template 1]

Implementation

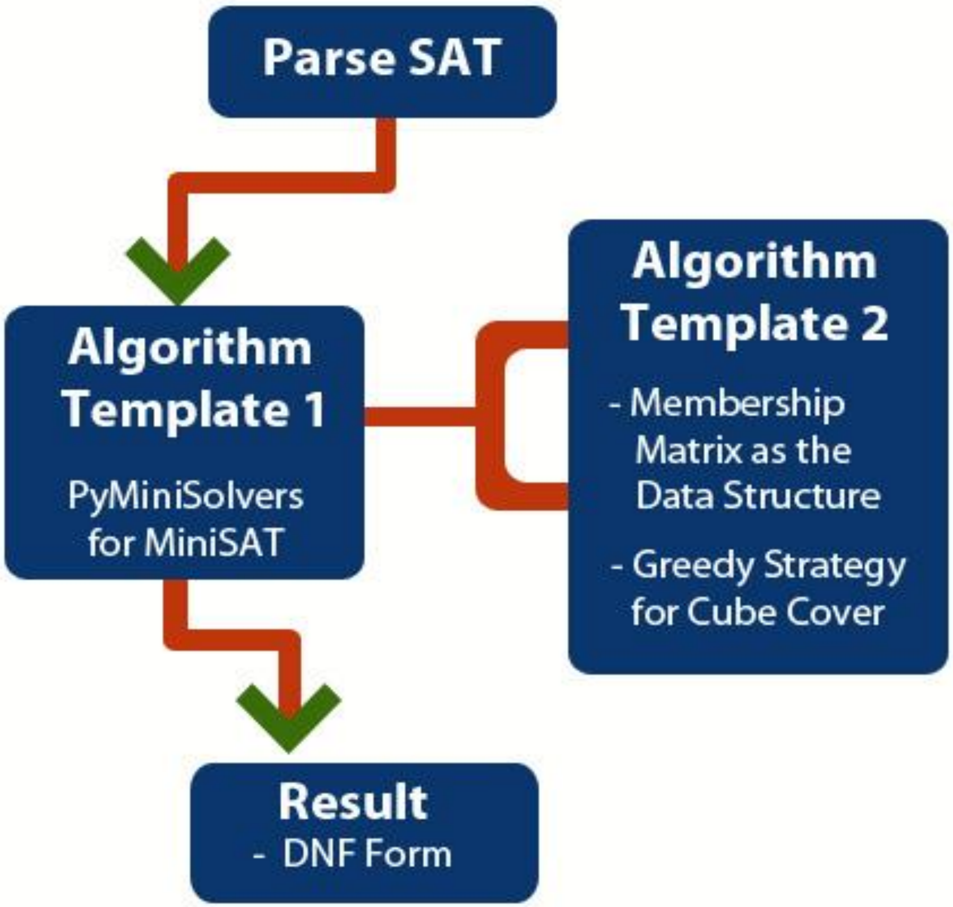

Fig 3: Implementation Flow

Algorithm Template 1

```
Input: CNF formula
Output:  Equivalent DNF formula
function cover_allSAT (CNF)
      DNF := null
      CNF_Init := CNF
      while(unsat(CNF))
            solution := solve(CNF)
            cover := get_cover(solution,CNF_Init)
            blocking_clause = negate(cover)
            DNF := DNF | cover
            CNF := CNF ^ blocking_clause
      end
return DNF
```

Algorithm Template 2

- Involves computing cube cover
- Prime functions in this template are: -
  - Computing additional data structure called Membership Matrix or UCP Matrix
  - Selecting essential literals for single solution
  - Selecting and pruning clauses covered by essential literals selected above

Benchmarks

- Subset of standard benchmarks from satlib.org and SAT competitions was used
- These benchmarks are split across three categories
  - Random
  - Crafted
  - Industrial

Results

- We identified two categories of problems in our results
- Category I: Problems solved by solver without memory insufficiency in limited time
- Category II: Problems that either timed out after running for long time or ran out of memory

Results – Category I

	Membership Matrix	Essential Literals	Clauses Covered
Random	50.41%	9.12%	36.02%
Crafted	47.60%	7.87%	32.71%
Industrial	48.68%	7.80%	41.78%

	Greedy Cover	Parsing Time	Other Functions
Random	0.7%	0.0001%	3.74%
Crafted	0.7%	0.0001%	11.12%
Industrial	0.0002%	0.0001%	1.73%

Table 1: Timing Analysis for Category I problems

Results – Category I

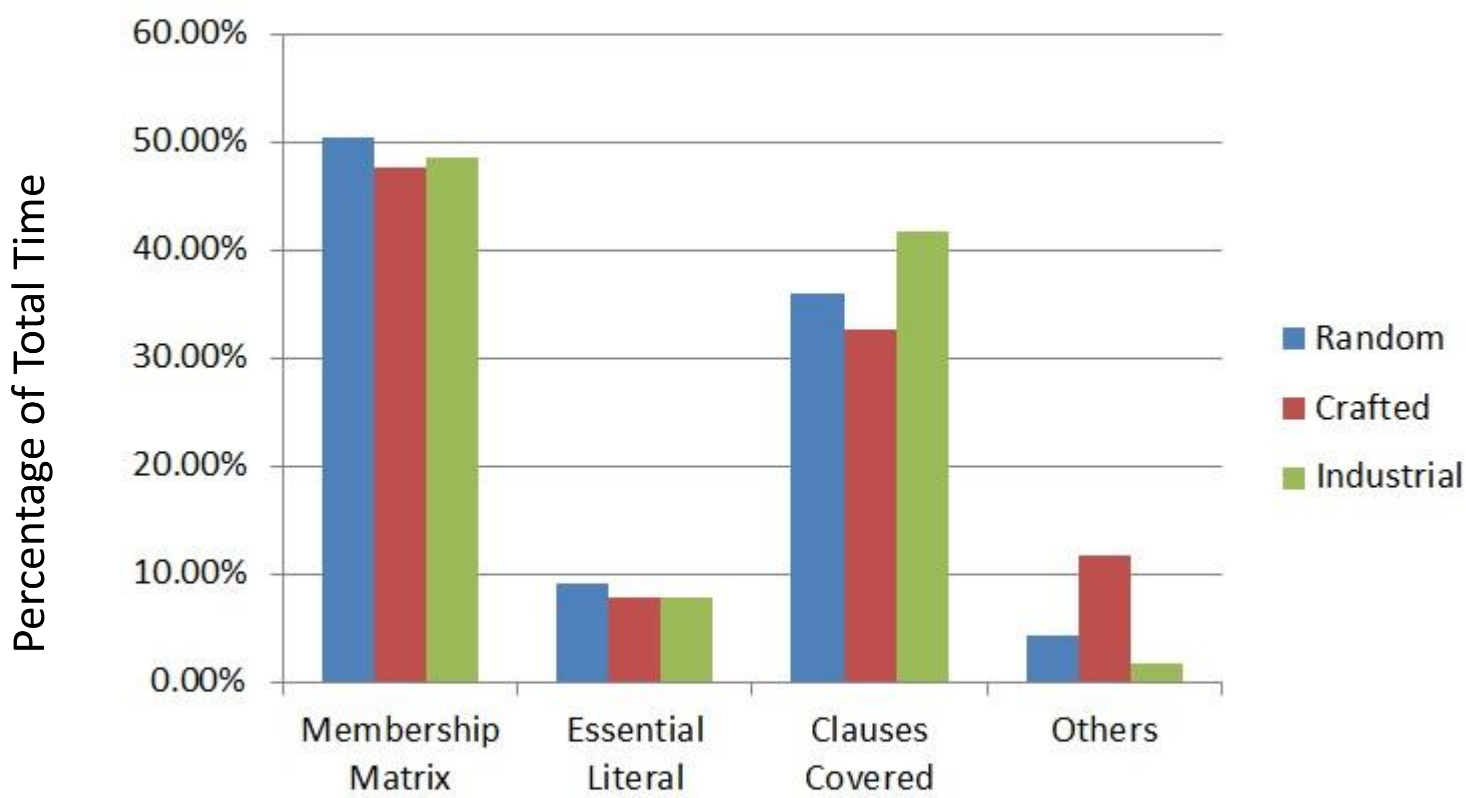

Fig 4: Timing Analysis for Category I problems – Bar Graph

Observations

- Most time is taken by **Membership Matrix, Essential Literals** and **Clause Cover Computation**
- These three can be safely specified as the bottlenecks for Category I problems in our implementation

Observations

- Some SAT problems are inherently tough
- A large part of the exponential search space needs to be searched for such problems in order to determine the outcome
- Most solvers, including MiniSAT, fail on such instances without even concluding whether SAT or UNSAT

Using Prediction Technique

- Instead of getting no results, better to **predict** results based on heuristics or features
- Solution: Use predictive techniques to estimate if a problem is satisfiable or not
- Use in our application:
  - If predictor says SAT:
    - continue to solve and product all solutions
  - else:
    - discard

Desired Characteristics

- Need very low error margins for SAT problems classified as UNSAT
- A margin of error for UNSAT classified as SAT is relatively permissible because
  - In this case, solver would still try to solve the problem, and eventually conclude the problem being UNSAT after search space exploration

Features for Classification

- We need certain features to predict if a problem is satisfiable or not
- We used the features as that of SATZilla [3]
- These features estimate the hardness of the problem by taking into account various factors like size of the problem, clustering coefficient of graphical representation, local search contradiction depth, etc.

Benchmarks

- Same as benchmarks used for implementation
- This time additional category of mixed problems was introduced
  - Random
  - Crafted
  - Industrial
  - Mixed (All three combined)

Prediction Process

- Only two major steps are involved
  – Step 1 : Feature extraction from problem instance
  – Step 2 : Prediction based on these features , using a classifier

Results

- Step 1: Following are the times for feature extraction on our benchmarks [SAT ’07 and before]

	Average	Min	Max
Random	7.99s	0.44s	14.34s
Crafted	9.86s	0.68s	140.09s
Industrial	145.94s	0.25s	6217.97s

Table 2: Feature Computation Times for Benchmarks

Classifiers

- We decided to train multiple classifiers to obtain accuracies across benchmarks
- Classifiers used were
  - Neural Network
  - Discriminant Analysis Classifier
  - Classification Tree
  - 1 Nearest Neighbor
  - Naïve Bayes

Classifiers – Neural Net

- We used a 3 layered Neural Network for classification. We decided to sweep through number of neurons in hidden layer to obtain optimum architecture

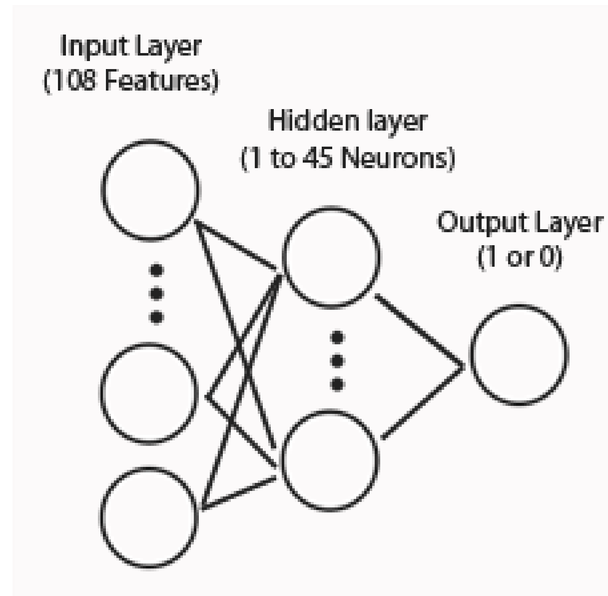

Figure 5: Neural Network Architecture

Classifiers – Neural Net

- Results for Random, Crafted and Industrial

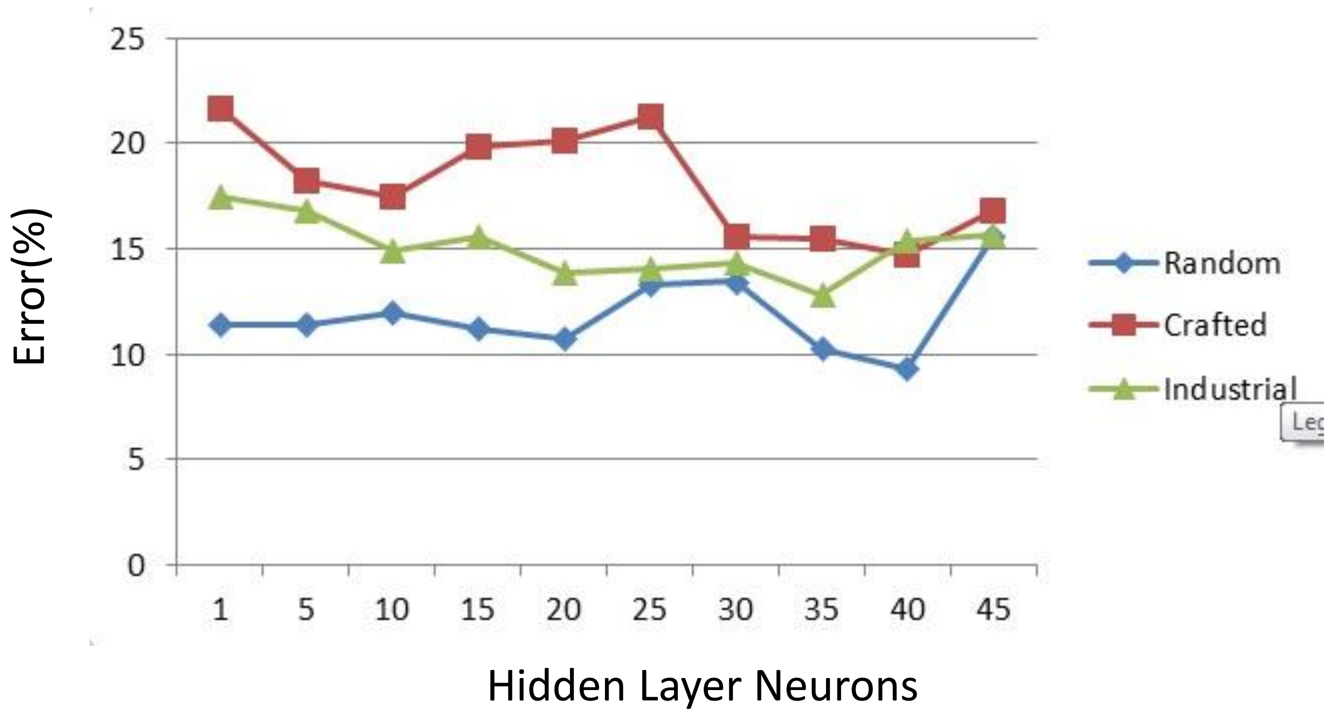

Figure 6: Error percentage versus Hidden Layer Neurons

Other Classifiers

- We decided to use 35 neurons in the hidden layer as we can observe an approximate dip in error rate in all three categories around that number
- We also tried simulations using other classifiers to obtain how well they perform

Classification Results

- Accuracies across 5 different classifiers

	Neural Network	Discriminant Analysis	1 Nearest Neighbor	Classification Tree	Naïve Bayes
Random	89.80%	89.77%	87.82%	**89.81%**	69.88%
Crafted	84.55%	80.15%	81.77%	**87.13%**	64.22%
Industrial	87.21%	87.91%	88.41%	**88.84%**	74.21%
Mixed	85.40%	78.47%	85.85%	**87.48%**	55.05%

Table 3: Overall accuracies with different classifiers

Classification Results

- SAT as SAT Accuracies

	Neural Network	Discriminant Analysis	1 Nearest Neighbor	Classification Tree	Naïve Bayes
Random	**95.8%**	95.26%	91.27%	93.03%	63.36%
Crafted	86.9%	72.65%	72.65%	83.37%	**95.63%**
Industrial	87.2%	85.66%	87.32%	**87.87%**	56.25%
Mixed	87.9%	85.71%	87.11%	**88.52%**	83.79%

Table 4: SAT problem correct classification accuracies

Classification Results

- UNSAT as UNSAT Accuracies

	Neural Network	Discriminant Analysis	1 Nearest Neighbor	Classification Tree	Naïve Bayes
Random	74.7%	74.26%	79.08%	81.74%	**88.38%**
Crafted	**89.6%**	84.94%	77.59%	89.54%	45.22%
Industrial	**93.4%**	89.87%	89.36%	89.68%	89.52%
Mixed	82.2%	69.49%	84.12%	**86.06%**	45.95%

Table 5: UNSAT problem correct classification accuracies

Observations

- Choosing appropriate classifiers, we can obtain accuracy in excess of 85% for any application
- From our application point of view (SAT correctly as SAT), Classification Tree proved out to be the best classifier as it gave 88.52% accuracy

Proposed Idea : Majority Rule

- We wanted to improve accuracy for application specific data (SAT as SAT)
- For this, we are ready to compromise on UNSAT as UNSAT since the error produced by this will later be caught be the solver
- We use a Majority rule

Majority Rule

- Take 4 classifiers & produce outputs
- 2^4 possible outputs from 0000 to 1111
- Predict 1 only in case of 1111
- Decreases probability of classification of a SAT instance classified as UNSAT since all 4 classifiers going wrong together is less probable

Results

- SAT specific Majority Rule
- Harder to wrongly predict SAT
- 4 classifiers

	Overall	SAT as SAT	UNSAT as UNSAT
Random	88.04%	**98.95%**	57.15%
Crafted	78.45%	**94.53%**	69.21%
Industrial	87.74%	**96.88%**	80.03%
Mixed	81.44%	**98.23%**	59.23%

Table 6: Majority Rule Results

Majority Rule - Extended

- If application is UNSAT specific
- Make it harder to predict UNSAT wrongly
- Predict 0 only in case of output 0000
- Tried the simulations for this version too

Results

- UNSAT specific Majority Rule
- Harder to wrongly predict UNSAT
- 4 classifiers

Problem	Overall	SAT as SAT	UNSAT as UNSAT
Random	87.64%	85.64%	**93.19%**
Crafted	78.53%	49.36%	**97.77%**
Industrial	84.95%	71.60%	**96.88%**
Mixed	82.44%	74.33%	**95.05%**

Table 7: Majority Rule – Extended Results

Overall Conclusions

- Two Categories of problems were identified
- For Category I bottlenecks were,
  - Membership Matrix
  - Essential Literals
  - Clauses Covered by Essential Literals Two Categories of problems were identified

Overall Conclusions

- For Category II, prediction technique was suggested
- Without using Majority Rule, we obtained accuracies as high as 88.52% for our application using Classification Tree Algorithm
- With Majority Rule, accuracies as high as 98.23% can be obtained

Future Work

- Try parallelizing bottlenecks of our implementation for Category I
- Try and apply predictors developed to other applications such as SAT Solvers
  - Taking decision of which branch to take while assigning variables

References

[1] Yu, Yinlei, et al. "All-SAT Using Minimal Blocking Clauses." *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*. IEEE, 2014.

[2] Devlin, David, and Barry O'Sullivan. "Satisfiability as a classification problem."*Proc. of the 19th Irish Conf. on Artificial Intelligence and Cognitive Science*. 2008.

[3] Xu, Lin, et al. "SATzilla: portfolio-based algorithm selection for SAT." *Journal of Artificial Intelligence Research* (2008): 565-606.

References

[4] Wong, M. Anthony, and Tom Lane. "A kth nearest neighbour clustering procedure." *Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface*. Springer US, 1981.

[5] Hansen, Matthew, R. Dubayah, and R. DeFries. "Classification trees: an alternative to traditional land cover classifiers." *International journal of remote sensing* 17.5 (1996): 1075-1081.

[6] Fraley, Chris, and Adrian E. Raftery. "Model-based clustering, discriminant analysis, and density estimation." *Journal of the American statistical Association* 97.458 (2002): 611-631.

Thank You

Problem Definition

- Given a problem in CNF form, find all satisfying solutions of the problem
- CNF form
- Product of Sums

$$(x_1 \ V \ x_2 \ V \ x_3) \ \Lambda \ (x_2 \ V \ \bar{x}_1 \ V \ x_3) \ \Lambda \ (\overline{x_4})$$

Problem Definition

- SAT Problem: Find a variable assignment such that the CNF form evaluates to TRUE
- All SAT Problem: Find all the variable assignments that satisfy the formula
- Example:

$$(x_1 \ V \ x_2 \, V \ x_3 \,) \, \Lambda \ (x_2 \ V \ \bar{x}_1 \, V \ x_3 \,) \, \Lambda \, (\overline{x_4})$$

Solution: x_1 = 1, $x_2 = 1$, $x_3 = 1$, $x_4 = 0$

Naive All SAT

- Function on the following principle
  - Step 1: Produce a solution
  - Step 2: Produce a blocking clause
  - Step 3: Update the problem using blocking clause
  - Step 4: Go-to Step 1
- Do the above until problem becomes UNSAT

Naïve Algorithm for All SAT

```
Input: CNF formula
Output:  Set of all solutions of CNF
function naïve_allSAT (CNF)
     solution_set := empty_set
     while(unsat(CNF))
          solution := solve(CNF)
          solution_set.add(solution)
          blocking_clause = negate(solution)
          CNF := CNF ^ blocking_clause
     end
return solution_set
```

All Clause Algorithm

```
Input: CNF formula
Output:  Equivalent DNF formula
function cover_allSAT (CNF)
      DNF := null
      while(unsat(CNF))
            solution := solve(CNF)
            cover := get_cover(solution,CNF)
            blocking_clause = negate(cover)
            DNF := DNF | cover
            CNF := CNF ^ blocking_clause
      end
return DNF
```

Algorithm for Cube Cover

- ‘s’ is one particular solution to CNF problem
- ‘ess_lit’ is set of essential literals

```
function get_cover(s,CNF)
  mat := get_membership_matrix(s,CNF)
  ess_lit :=  get_essentials(s,mat)
  clauses := get_clauses_cov(ess_lit,mat)
  CNF := prune_essential_cover(clauses,mat)
  literals := greedy_cover(mat)
 return ess_lit ^ literals
```

All SAT Using Minimal Blocking Clauses

```
Input: CNF formula
Output:  Equivalent DNF formula
function cover_allSAT (CNF)
      DNF := null
      CNF_Init := CNF
      while(unsat(CNF))
            solution := solve(CNF)
            cover := get_cover(solution,CNF_Init)
            blocking_clause = negate(cover)
            DNF := DNF | cover
            CNF := CNF ^ blocking_clause
      end
return DNF
```

Algorithm Template 1

```
Input: CNF formula
Output:  Equivalent DNF formula
function cover_allSAT (CNF)
      DNF := null
      CNF_Init := CNF
      while(unsat(CNF))
            solution := solve(CNF)
            cover := get_cover(solution,CNF_Init)
            blocking_clause = negate(cover)
            DNF := DNF | cover
            CNF := CNF ^ blocking_clause
      end
return DNF
```

Algorithm Template 2

- ‘s’ is one particular solution to CNF problem
- ‘ess_lit’ is set of essential literals

```
function get_cover(s,CNF)
  mat := get_membership_matrix(s,CNF)
  ess_lit :=  get_essentials(s,mat)
  clauses := get_clauses_cov(ess_lit,mat)
  CNF := prune_essential_cover(clauses,mat)
  literals := greedy_cover(mat)
 return ess_lit ^ literals
```

Classifiers – Neural Net

- Results for All

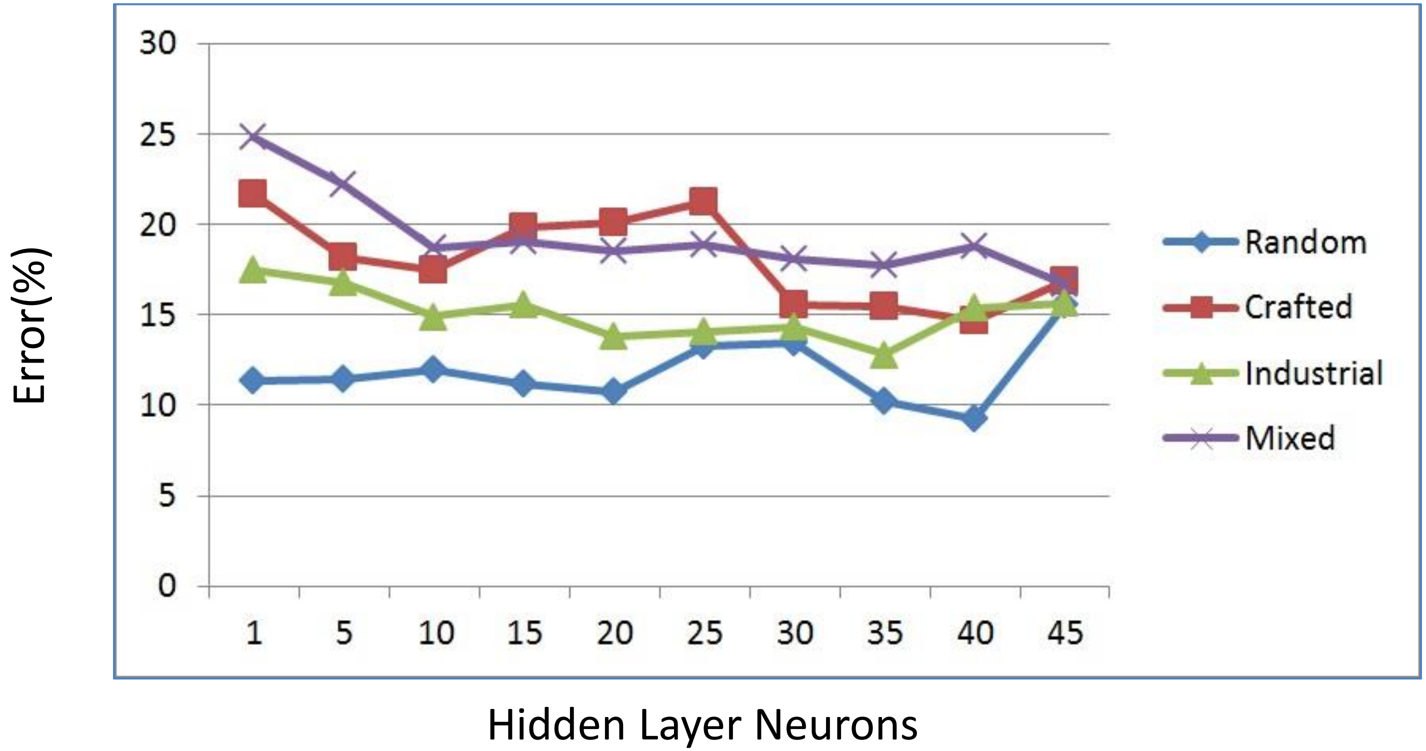

Figure 6: Error(%) versus Hidden Layer Neurons