XOR Local Search for Boolean Brent Equations

Wojciech Nawrocki Zhenjun Liu Andreas Fröhlich
Marijn Heule Armin Biere

Carnegie Mellon University

JKU
JOHANNES KEPLER UNIVERSITY LINZ

SAT 2021

2021-06-26

XOR Local Search for Boolean Brent Equations

XOR Local Search for Boolean Brent Equations

Wojciech Nawrocki Zhenjun Liu Andreas Fröhlich
Marijn Heule Armin Biere

Carnegie Mellon University

JKU
JOHANNES KEPLER UNIVERSITY LINZ

SAT 2021

When SLS outperforms CDCL

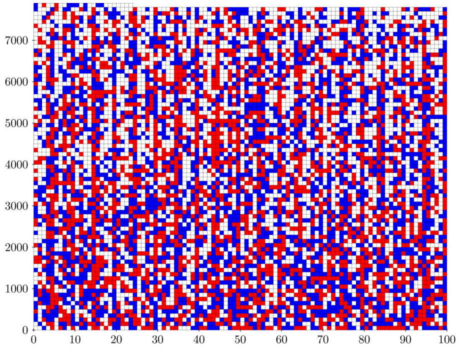
- Random k-SAT and satisfiable, hard-combinatorial problems.

When SLS outperforms CDCL
- Random k-SAT and satisfiable, hard-combinatorial problems.

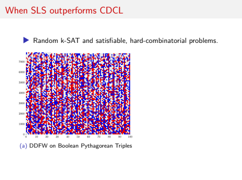
- While CDCL is the dominating SAT solving paradigm, there are problems on which Stochastic Local Search performs significantly better.
  - The largest satisfiable instance of the Boolean Pythagorean Triples problem can be solved using DDFW [Divide and Distribute Fixed Weights] local search in ~one CPU minute. Other algorithms time out.
  - SLS solvers perform well in the search for new matrix multiplication schemes expressed as a SAT problem via the Boolean Brent equations.
- Can we further improve the performance of LS on a class of problems where it already performs best? We look at problems involving XOR constraints, of which matrix multiplication is one instance.

When SLS outperforms CDCL

- Random k-SAT and satisfiable, hard-combinatorial problems.



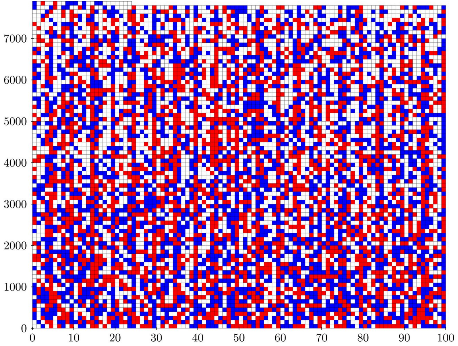
(a) DDFW on Boolean Pythagorean Triples



- While CDCL is the dominating SAT solving paradigm, there are problems on which Stochastic Local Search performs significantly better.
  - The largest satisfiable instance of the Boolean Pythagorean Triples problem can be solved using DDFW [Divide and Distribute Fixed Weights] local search in ~one CPU minute. Other algorithms time out.
  - SLS solvers perform well in the search for new matrix multiplication schemes expressed as a SAT problem via the Boolean Brent equations.
- Can we further improve the performance of LS on a class of problems where it already performs best? We look at problems involving XOR constraints, of which matrix multiplication is one instance.

When SLS outperforms CDCL

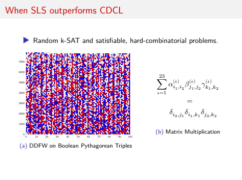
- Random k-SAT and satisfiable, hard-combinatorial problems.



(a) DDFW on Boolean Pythagorean Triples

$$\sum_{\iota=1}^{23} \alpha^{(\iota)}_{i_1,i_2}\beta^{(\iota)}_{j_1,j_2}\gamma^{(\iota)}_{k_1,k_2} = \delta_{i_2,j_1}\delta_{i_1,k_1}\delta_{j_2,k_2}$$

(b) Matrix Multiplication



- While CDCL is the dominating SAT solving paradigm, there are problems on which Stochastic Local Search performs significantly better.
  - The largest satisfiable instance of the Boolean Pythagorean Triples problem can be solved using DDFW [Divide and Distribute Fixed Weights] local search in ~one CPU minute. Other algorithms time out.
  - SLS solvers perform well in the search for new matrix multiplication schemes expressed as a SAT problem via the Boolean Brent equations.
- Can we further improve the performance of LS on a class of problems where it already performs best? We look at problems involving XOR constraints, of which matrix multiplication is one instance.

Solving XOR in CNF form

$$(x_1 \oplus \ldots \oplus x_k) \mapsto ?$$

Solving XOR in CNF form

$(x_1 \oplus \ldots \oplus x_k) \mapsto ?$

- To solve problems involving XOR constraints, we have to pick an encoding into CNF.
- Most straightforwardly, we can use a direct encoding – `XOR_d`. But this produces exponentially many clauses.
- The usual linear approach is Tseitin encoding. It recursively breaks off fixed-size chunks and encodes them directly.
- But we pay for linearity. Tseitin encoding introduces auxiliary variables (y, underlined). These interact poorly with the SLS algorithm.

Solving XOR in CNF form

$$(x_1 \oplus \ldots \oplus x_k) \mapsto ?$$

$$\texttt{XOR\_d}(x_1, \ldots, x_k) = \bigwedge_{\text{even } \#-} (\pm x_1 \vee \ldots \vee \pm x_k)$$

Solving XOR in CNF form
$(x_1 \oplus \ldots \oplus x_k) \mapsto ?$
$\texttt{XOR\_d}(x_1, \ldots, x_k) = \bigwedge_{\text{even } \#-} (\pm x_1 \vee \ldots \vee \pm x_k)$

- To solve problems involving XOR constraints, we have to pick an encoding into CNF.
- Most straightforwardly, we can use a direct encoding – `XOR_d`. But this produces exponentially many clauses.
- The usual linear approach is Tseitin encoding. It recursively breaks off fixed-size chunks and encodes them directly.
- But we pay for linearity. Tseitin encoding introduces auxiliary variables (y, underlined). These interact poorly with the SLS algorithm.

Solving XOR in CNF form

$$(x_1 \oplus \ldots \oplus x_k) \mapsto ?$$

$$\texttt{XOR\_d}(x_1, \ldots, x_k) = \bigwedge_{\text{even } \#-} (\pm x_1 \vee \ldots \vee \pm x_k)$$

$$\texttt{XOR\_T\_n}(x_1, \ldots, x_k) = \texttt{XOR\_d}(x_1, \ldots, x_{n-1}, -y) \wedge \texttt{XOR\_T\_n}(y, x_n, \ldots, x_k)$$

Solving XOR in CNF form

$(x_1 \oplus \ldots \oplus x_k) \mapsto ?$

$\texttt{XOR\_d}(x_1, \ldots, x_k) = \bigwedge_{\text{even } \#-} (\pm x_1 \vee \ldots \vee \pm x_k)$

$\texttt{XOR\_T\_n}(x_1, \ldots, x_k) = \texttt{XOR\_d}(x_1, \ldots, x_{n-1}, -y) \wedge \texttt{XOR\_T\_n}(y, x_n, \ldots, x_k)$

- To solve problems involving XOR constraints, we have to pick an encoding into CNF.
- Most straightforwardly, we can use a direct encoding – `XOR_d`. But this produces exponentially many clauses.
- The usual linear approach is Tseitin encoding. It recursively breaks off fixed-size chunks and encodes them directly.
- But we pay for linearity. Tseitin encoding introduces auxiliary variables (y, underlined). These interact poorly with the SLS algorithm.

Solving XOR in CNF form

$$(x_1 \oplus \ldots \oplus x_k) \mapsto ?$$

$$\texttt{XOR\_d}(x_1, \ldots, x_k) = \bigwedge_{\text{even } \#-} (\pm x_1 \vee \ldots \vee \pm x_k)$$

$$\texttt{XOR\_T\_n}(x_1, \ldots, x_k) = \texttt{XOR\_d}(x_1, \ldots, x_{n-1}, \underline{-y}) \wedge \texttt{XOR\_T\_n}(\underline{y}, x_n, \ldots, x_k)$$

Solving XOR in CNF form

$(x_1 \oplus \ldots \oplus x_k) \mapsto ?$

$\texttt{XOR\_d}(x_1, \ldots, x_k) = \bigwedge_{\text{even } \#-} (\pm x_1 \vee \ldots \vee \pm x_k)$

$\texttt{XOR\_T\_n}(x_1, \ldots, x_k) = \texttt{XOR\_d}(x_1, \ldots, x_{n-1}, \underline{-y}) \wedge \texttt{XOR\_T\_n}(\underline{y}, x_n, \ldots, x_k)$

- To solve problems involving XOR constraints, we have to pick an encoding into CNF.
- Most straightforwardly, we can use a direct encoding – `XOR_d`. But this produces exponentially many clauses.
- The usual linear approach is Tseitin encoding. It recursively breaks off fixed-size chunks and encodes them directly.
- But we pay for linearity. Tseitin encoding introduces auxiliary variables (y, underlined). These interact poorly with the SLS algorithm.

Flipping Tseitin

$$\texttt{XOR\_T\_2}(x_1, x_2, x_3, x_4, x_5) =$$

$$\texttt{XOR\_d}(x_1, x_2, \underline{y_1}) \wedge \texttt{XOR\_d}(-\underline{y_1}, x_3, \underline{y_2}) \wedge \texttt{XOR\_d}(-\underline{y_2}, x_4, x_5) =$$
$$(x_1, x_2, \underline{y_1}) \wedge (-x_1, -x_2, \underline{y_1}) \wedge (-x_1, x_2, -\underline{y_1}) \wedge (x_1, -x_2, -\underline{y_1}) \wedge$$
$$(-\underline{y_1}, x_3, \underline{y_2}) \wedge (\underline{y_1}, -x_3, \underline{y_2}) \wedge (\underline{y_1}, x_3, -\underline{y_2}) \wedge (-\underline{y_1}, -x_3, -\underline{y_2}) \wedge$$
$$(-\underline{y_2}, x_4, x_5) \wedge (\underline{y_2}, -x_4, x_5) \wedge (\underline{y_2}, x_4, -x_5) \wedge (-\underline{y_2}, -x_4, -x_5)$$

Flipping Tseitin

- At its core, SLS proceeds by flipping variables in falsified clauses according to a probabilistic heuristic.
- Consider solving the following XOR with 5 original and 2 auxiliary variables.
  - Suppose we start with the all-true assignment. The high-level constraint is *already* satisified but the solver does not "see" this because it operates on a low-level CNF representation.
  - The solver takes steps flipping variables, even going backwards in a sense by falsifying the high-level constraint in order to solve its CNF encoding.
  - Generally, for a given assignment to original variables there is precisely one assignment satisfying the auxiliary variables. Searching for it is unnecessary work.

Flipping Tseitin

$$\texttt{XOR\_T\_2}(x_1, x_2, x_3, x_4, x_5) =$$

$$\texttt{XOR\_d}(x_1, x_2, \underline{y_1}) \wedge \texttt{XOR\_d}(\underline{-y_1}, x_3, \underline{y_2}) \wedge \texttt{XOR\_d}(\underline{-y_2}, x_4, x_5) =$$
$$(x_1, x_2, \underline{y_1}) \wedge (-x_1, -x_2, \underline{y_1}) \wedge (-x_1, x_2, \underline{-y_1}) \wedge (x_1, -x_2, \underline{-y_1}) \wedge$$
$$(\underline{-y_1}, x_3, \underline{y_2}) \wedge (\underline{y_1}, -x_3, \underline{y_2}) \wedge (\underline{y_1}, x_3, \underline{-y_2}) \wedge (\underline{-y_1}, -x_3, \underline{-y_2}) \wedge$$
$$(\underline{-y_2}, x_4, x_5) \wedge (\underline{y_2}, -x_4, x_5) \wedge (\underline{y_2}, x_4, -x_5) \wedge (\underline{-y_2}, -x_4, -x_5)$$

Flipping Tseitin

$$\texttt{XOR\_T\_2}(x_1, x_2, x_3, x_4, x_5) =$$

$$\texttt{XOR\_d}(x_1, x_2, \underline{y_1}) \wedge \texttt{XOR\_d}(\underline{-y_1}, x_3, \underline{y_2}) \wedge \texttt{XOR\_d}(\underline{-y_2}, x_4, x_5) =$$
$$(x_1, x_2, \underline{y_1}) \wedge (-x_1, -x_2, \underline{y_1}) \wedge (-x_1, x_2, \underline{-y_1}) \wedge (x_1, -x_2, \underline{-y_1}) \wedge$$
$$(\underline{-y_1}, x_3, \underline{y_2}) \wedge (\underline{y_1}, -x_3, \underline{y_2}) \wedge (\underline{y_1}, x_3, \underline{-y_2}) \wedge (\underline{-y_1}, -x_3, \underline{-y_2}) \wedge$$
$$(\underline{-y_2}, x_4, x_5) \wedge (\underline{y_2}, -x_4, x_5) \wedge (\underline{y_2}, x_4, -x_5) \wedge (\underline{-y_2}, -x_4, -x_5)$$

- At its core, SLS proceeds by flipping variables in falsified clauses according to a probabilistic heuristic.
- Consider solving the following XOR with 5 original and 2 auxiliary variables.
  - Suppose we start with the all-true assignment. The high-level constraint is *already* satisified but the solver does not "see" this because it operates on a low-level CNF representation.
  - The solver takes steps flipping variables, even going backwards in a sense by falsifying the high-level constraint in order to solve its CNF encoding.
  - Generally, for a given assignment to original variables there is precisely one assignment satisfying the auxiliary variables. Searching for it is unnecessary work.

Flipping Tseitin

$$\downarrow$$

$$\texttt{XOR\_T\_2}(x_1, x_2, x_3, x_4, x_5) =$$

$$\texttt{XOR\_d}(x_1, x_2, \underline{y_1}) \wedge \texttt{XOR\_d}(-\underline{y_1}, x_3, \underline{y_2}) \wedge \texttt{XOR\_d}(-\underline{y_2}, x_4, x_5) =$$

$$(x_1, x_2, \underline{y_1}) \wedge (-x_1, -x_2, \underline{y_1}) \wedge (-x_1, x_2, -\underline{y_1}) \wedge (x_1, -x_2, -\underline{y_1}) \wedge$$

$$(-\underline{y_1}, x_3, \underline{y_2}) \wedge (\underline{y_1}, -x_3, \underline{y_2}) \wedge (\underline{y_1}, x_3, -\underline{y_2}) \wedge (-\underline{y_1}, -x_3, -\underline{y_2}) \wedge$$

$$(-\underline{y_2}, x_4, x_5) \wedge (\underline{y_2}, -x_4, x_5) \wedge (\underline{y_2}, x_4, -x_5) \wedge (-\underline{y_2}, -x_4, -x_5)$$

Flipping Tseitin

- At its core, SLS proceeds by flipping variables in falsified clauses according to a probabilistic heuristic.
- Consider solving the following XOR with 5 original and 2 auxiliary variables.
  - Suppose we start with the all-true assignment. The high-level constraint is *already* satisified but the solver does not "see" this because it operates on a low-level CNF representation.
  - The solver takes steps flipping variables, even going backwards in a sense by falsifying the high-level constraint in order to solve its CNF encoding.
  - Generally, for a given assignment to original variables there is precisely one assignment satisfying the auxiliary variables. Searching for it is unnecessary work.

Flipping Tseitin

$$\texttt{XOR\_T\_2}(x_1, x_2, x_3, x_4, x_5) =$$

$$\texttt{XOR\_d}(x_1, x_2, \underline{y_1}) \wedge \texttt{XOR\_d}(-\underline{y_1}, x_3, \underline{y_2}) \wedge \texttt{XOR\_d}(-\underline{y_2}, x_4, x_5) =$$
$$(x_1, x_2, \underline{y_1}) \wedge (-x_1, -x_2, \underline{y_1}) \wedge (-x_1, x_2, -\underline{y_1}) \wedge (x_1, -x_2, -\underline{y_1}) \wedge$$
$$(-\underline{y_1}, x_3, \underline{y_2}) \wedge (\underline{y_1}, -x_3, \underline{y_2}) \wedge (\underline{y_1}, x_3, -\underline{y_2}) \wedge (-\underline{y_1}, -x_3, -\underline{y_2}) \wedge$$
$$(-\underline{y_2}, x_4, x_5) \wedge (\underline{y_2}, -x_4, x_5) \wedge (\underline{y_2}, x_4, -x_5) \wedge (-\underline{y_2}, -x_4, -x_5)$$

Flipping Tseitin

- At its core, SLS proceeds by flipping variables in falsified clauses according to a probabilistic heuristic.
- Consider solving the following XOR with 5 original and 2 auxiliary variables.
  - Suppose we start with the all-true assignment. The high-level constraint is *already* satisified but the solver does not “see” this because it operates on a low-level CNF representation.
  - The solver takes steps flipping variables, even going backwards in a sense by falsifying the high-level constraint in order to solve its CNF encoding.
  - Generally, for a given assignment to original variables there is precisely one assignment satisfying the auxiliary variables. Searching for it is unnecessary work.

Flipping Tseitin

$$\downarrow$$

$$\texttt{XOR\_T\_2}(x_1, x_2, x_3, x_4, x_5) =$$

$$\texttt{XOR\_d}(x_1, x_2, \underline{y_1}) \wedge \texttt{XOR\_d}(-\underline{y_1}, x_3, \underline{y_2}) \wedge \texttt{XOR\_d}(-\underline{y_2}, x_4, x_5) =$$

$$(x_1, x_2, \underline{y_1}) \wedge (-x_1, -x_2, \underline{y_1}) \wedge (-x_1, x_2, -\underline{y_1}) \wedge (x_1, -x_2, -\underline{y_1}) \wedge$$
$$(-\underline{y_1}, x_3, \underline{y_2}) \wedge (\underline{y_1}, -x_3, \underline{y_2}) \wedge (\underline{y_1}, x_3, -\underline{y_2}) \wedge (-\underline{y_1}, -x_3, -\underline{y_2}) \wedge$$
$$(-\underline{y_2}, x_4, x_5) \wedge (\underline{y_2}, -x_4, x_5) \wedge (\underline{y_2}, x_4, -x_5) \wedge (-\underline{y_2}, -x_4, -x_5)$$

Flipping Tseitin

- At its core, SLS proceeds by flipping variables in falsified clauses according to a probabilistic heuristic.
- Consider solving the following XOR with 5 original and 2 auxiliary variables.
  - Suppose we start with the all-true assignment. The high-level constraint is *already* satisified but the solver does not "see" this because it operates on a low-level CNF representation.
  - The solver takes steps flipping variables, even going backwards in a sense by falsifying the high-level constraint in order to solve its CNF encoding.
  - Generally, for a given assignment to original variables there is precisely one assignment satisfying the auxiliary variables. Searching for it is unnecessary work.

XNF

```
p xnf 3 2
1 2 3 0
x 1 2 0
```

$$(x_1 \lor x_2 \lor x_3) \land$$
$$(x_1 \oplus x_2)$$

XNF
p xnf 3 2
1 2 3 0
x 1 2 0
$(x_1 \lor x_2 \lor x_3) \land$
$(x_1 \oplus x_2)$

- We propose to experiment with native XOR representations more widely. In the spirit of DIMACS CNF, an *XNF* format could be used.
- Worth noting that we later found out XNF is already implemented in CryptoMiniSAT.

xnfSAT: Stochastic Local Search with native XOR

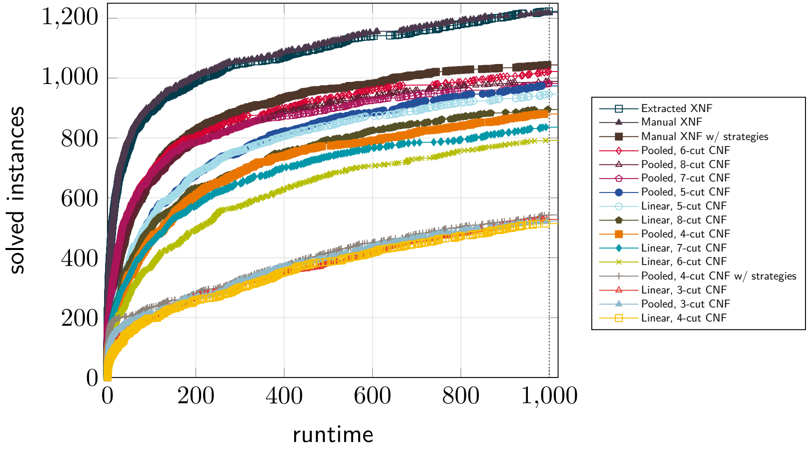
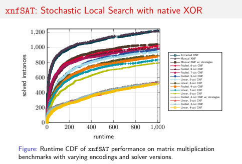


Figure: Runtime CDF of `xnfSAT` performance on matrix multiplication benchmarks with varying encodings and solver versions.



- To solve XNF, we present xnfSAT, an SLS solver supporting native XOR.
- Its performance on matrix multiplication benchmarks significantly improves upon the best CNF-based solver, YalSAT.
- To go with xnfSAT, we implemented a tool to extract XOR gates from CNF files.

We build on YalSAT

Algorithm `YalSAT`, a `WalkSAT`-based solver

for clause in input file **do**
  parse and store clause to data structure
end for
preprocess formula
$\alpha \leftarrow$ complete initial assignment of truth values
while there exists a clause falsified by α **do**
  $C \leftarrow \texttt{pickUnsatClause}()$
  $x \leftarrow \texttt{pickVarIn}(C)$
  $\alpha \leftarrow \alpha$ with x flipped
  update solver state
end while

- `xnfSAT` is based on `YalSAT`. Instructive to understand its outline.
- Unsurprisingly, supporting XOR needs no modifications to the high-level structure.
- We adapt parsing (XNF), preprocessing and variable selection (`pickVarIn`).

We build on `YalSAT`

Algorithm `YalSAT`, a `WalkSAT`-based solver

for clause in input file **do**
    parse and store clause to data structure
end for
preprocess formula
$\alpha \leftarrow$ complete initial assignment of truth values
while there exists a clause falsified by α **do**
    $C \leftarrow$ `pickUnsatClause`$()$
    $x \leftarrow$ `pickVarIn`(C)
    $\alpha \leftarrow \alpha$ with x flipped
    update solver state
end while

We build on YalSAT
Algorithm YalSAT, a WalkSAT-based solver
for clause in input file do
parse and store clause to data structure
end for
preprocess formula
α ← complete initial assignment of truth values
while there exists a clause falsified by α do
C ← pickUnsatClause()
x ← pickVarIn(C)
α ← α with x flipped
update solver state
end while

- `xnfSAT` is based on `YalSAT`. Instructive to understand its outline.
- Unsurprisingly, supporting XOR needs no modifications to the high-level structure.
- We adapt parsing (XNF), preprocessing and variable selection (`pickVarIn`).

We build on `YalSAT`

Algorithm `YalSAT`, a `WalkSAT`-based solver

for clause in input file **do**
    parse and store clause to data structure
end for
preprocess formula
$\alpha \leftarrow$ complete initial assignment of truth values
while there exists a clause falsified by α **do**
    $C \leftarrow$ `pickUnsatClause`()
    $x \leftarrow$ `pickVarIn`(C)
    $\alpha \leftarrow \alpha$ with x flipped
    update solver state
end while

We build on YalSAT
Algorithm YalSAT, a WalkSAT-based solver
for clause in input file do
parse and store clause to data structure
end for
preprocess formula
α ← complete initial assignment of truth values
while there exists a clause falsified by α do
C ← pickUnsatClause()
x ← pickVarIn(C)
α ← α with x flipped
update solver state
end while

- `xnfSAT` is based on `YalSAT`. Instructive to understand its outline.
- Unsurprisingly, supporting XOR needs no modifications to the high-level structure.
- We adapt parsing (XNF), preprocessing and variable selection (`pickVarIn`).

We build on YalSAT

Algorithm `YalSAT`, a `WalkSAT`-based solver

```
for clause in input file do
    parse and store clause to data structure
end for
preprocess formula
α ← complete initial assignment of truth values
while there exists a clause falsified by α do
    C ← pickUnsatClause()
    x ← pickVarIn(C)
    α ← α with x flipped
    update solver state
end while
```


We build on YalSAT

Algorithm YalSAT, a WalkSAT-based solver
for clause in input file do
parse and store clause to data structure
end for
preprocess formula
α ← complete initial assignment of truth values
while there exists a clause falsified by α do
C ← pickUnsatClause()
x ← pickVarIn(C)
α ← α with x flipped
update solver state
end while

- `xnfSAT` is based on `YalSAT`. Instructive to understand its outline.
- Unsurprisingly, supporting XOR needs no modifications to the high-level structure.
- We adapt parsing (XNF), preprocessing and variable selection (`pickVarIn`).

Parsing and clause storage

- XOR constraints are stored as buffers of variable indices, forgetting negations.
- Track the *parity* of each so that $\bigoplus_i x_i$ is satisfied iff $\Sigma x_i + \text{parity} \equiv 1 \mod 2$.

Parsing and clause storage

- XOR constraints are stored as buffers of variable indices, forgetting negations.
- Track the *parity* of each so that $\bigoplus_i x_i$ is satisfied iff $\Sigma x_i + \text{parity} \equiv 1 \mod 2$.

Parsing and clause storage

- XOR constraints are stored as buffers of variable indices, forgetting negations.
- Track the *parity* of each so that $\bigoplus_i x_i$ is satisfied iff $\Sigma x_i + \text{parity} \equiv 1 \mod 2$.
- For example, $(x_1 \oplus -x_2)$ is $(x_1 \oplus x_2, \texttt{parity} = 1)$.

Parsing and clause storage

- XOR constraints are stored as buffers of variable indices, forgetting negations.
- Track the *parity* of each so that $\bigoplus_i x_i$ is satisfied iff $\Sigma x_i + \text{parity} \equiv 1 \mod 2$.
- For example, $(x_1 \oplus -x_2)$ is $(x_1 \oplus x_2, \texttt{parity} = 1)$.

Preprocessing

- During preprocessing, to remove a propagating unit from an XOR constraint we simply flip the parity.

Preprocessing

- During preprocessing, to remove a propagating unit from an XOR constraint we simply flip the parity.

Preprocessing

- During preprocessing, to remove a propagating unit from an XOR constraint we simply flip the parity.
  - Have $(x_1 \oplus x_2 \oplus x_3, \texttt{parity} = 0)$

Preprocessing

Preprocessing

- During preprocessing, to remove a propagating unit from an XOR constraint we simply flip the parity.
  - Have $(x_1 \oplus x_2 \oplus x_3, \texttt{parity} = 0)$

Preprocessing

- During preprocessing, to remove a propagating unit from an XOR constraint we simply flip the parity.
  - Have $(x_1 \oplus x_2 \oplus x_3, \texttt{parity} = 0)$
  - Then $x_1 = 1$ propagates

Preprocessing

- During preprocessing, to remove a propagating unit from an XOR constraint we simply flip the parity.
  - Have $(x_1 \oplus x_2 \oplus x_3, \texttt{parity} = 0)$
  - Then $x_1 = 1$ propagates

Preprocessing

- During preprocessing, to remove a propagating unit from an XOR constraint we simply flip the parity.
  - Have $(x_1 \oplus x_2 \oplus x_3, \texttt{parity} = 0)$
  - Then $x_1 = 1$ propagates
  - Then have $(x_2 \oplus x_3, \texttt{parity} = 1)$

Preprocessing

- During preprocessing, to remove a propagating unit from an XOR constraint we simply flip the parity.
  - Have $(x_1 \oplus x_2 \oplus x_3, \texttt{parity} = 0)$
  - Then $x_1 = 1$ propagates
  - Then have $(x_2 \oplus x_3, \texttt{parity} = 1)$

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$

probSAT-like variable selection
$x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$

- Clause and variable selection heuristics are *the most* important parts of LS solvers. `YalSAT` and consequently `xnfSAT` use `probSAT`-like selection extended with weights.
- While long OR clauses with many satisfied literals are hard to break, XOR constraints can always be broken. Length is not a good measure of the importance of an XOR constraint.

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i

probSAT-like variable selection
$x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
Let $B(x_i)$ be the clauses falsified on flipping x_i

- Clause and variable selection heuristics are *the most* important parts of LS solvers. `YalSAT` and consequently `xnfSAT` use `probSAT`-like selection extended with weights.
- While long OR clauses with many satisfied literals are hard to break, XOR constraints can always be broken. Length is not a good measure of the importance of an XOR constraint.

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;

probSAT-like variable selection
$x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
Let $B(x_i)$ be the clauses falsified on flipping x_i
and w be a clause weighing function;

- Clause and variable selection heuristics are *the most* important parts of LS solvers. `YalSAT` and consequently `xnfSAT` use `probSAT`-like selection extended with weights.
- While long OR clauses with many satisfied literals are hard to break, XOR constraints can always be broken. Length is not a good measure of the importance of an XOR constraint.

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;
  - Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$

probSAT-like variable selection
$x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
Let $B(x_i)$ be the clauses falsified on flipping x_i
and w be a clause weighing function;
Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$

- Clause and variable selection heuristics are *the most* important parts of LS solvers. `YalSAT` and consequently `xnfSAT` use `probSAT`-like selection extended with weights.
- While long OR clauses with many satisfied literals are hard to break, XOR constraints can always be broken. Length is not a good measure of the importance of an XOR constraint.

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;
  - Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$
- While $w(x_1 \vee \ldots \vee x_n) \sim n$, we set $w(x_1 \oplus \ldots \oplus x_m)$ constant. Why?

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;
  - Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$
- While $w(x_1 \vee \ldots \vee x_n) \sim n$, we set $w(x_1 \oplus \ldots \oplus x_m)$ constant. Why?

- Clause and variable selection heuristics are *the most* important parts of LS solvers. `YalSAT` and consequently `xnfSAT` use `probSAT`-like selection extended with weights.
- While long OR clauses with many satisfied literals are hard to break, XOR constraints can always be broken. Length is not a good measure of the importance of an XOR constraint.

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;
  - Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$
- While $w(x_1 \vee \ldots \vee x_n) \sim n$, we set $w(x_1 \oplus \ldots \oplus x_m)$ constant. Why?
  1. Simplicity.

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;
  - Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$
- While $w(x_1 \vee \ldots \vee x_n) \sim n$, we set $w(x_1 \oplus \ldots \oplus x_m)$ constant. Why?
  1. Simplicity.

- Clause and variable selection heuristics are *the most* important parts of LS solvers. `YalSAT` and consequently `xnfSAT` use `probSAT`-like selection extended with weights.
- While long OR clauses with many satisfied literals are hard to break, XOR constraints can always be broken. Length is not a good measure of the importance of an XOR constraint.

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;
  - Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$
- While $w(x_1 \vee \ldots \vee x_n) \sim n$, we set $w(x_1 \oplus \ldots \oplus x_m)$ constant. Why?
  1. Simplicity.
  2. A literal x is *critical* in C if flipping x breaks C.

probSAT-like variable selection

$x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
Let $B(x_i)$ be the clauses falsified on flipping x_i
and w be a clause weighing function;
Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$
While $w(x_1 \vee \ldots \vee x_n) \sim n$, we set $w(x_1 \oplus \ldots \oplus x_m)$ constant. Why?
1. Simplicity.
2. A literal x is *critical* in C if flipping x breaks C.

- Clause and variable selection heuristics are *the most* important parts of LS solvers. `YalSAT` and consequently `xnfSAT` use `probSAT`-like selection extended with weights.
- While long OR clauses with many satisfied literals are hard to break, XOR constraints can always be broken. Length is not a good measure of the importance of an XOR constraint.

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;
  - Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$
- While $w(x_1 \vee \ldots \vee x_n) \sim n$, we set $w(x_1 \oplus \ldots \oplus x_m)$ constant. Why?
  1. Simplicity.
  2. A literal x is *critical* in C if flipping x breaks C.
     - In $(x_1 \vee x_2 \vee x_3)$, x_1 critical.

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;
  - Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$
- While $w(x_1 \vee \ldots \vee x_n) \sim n$, we set $w(x_1 \oplus \ldots \oplus x_m)$ constant. Why?
  1. Simplicity.
  2. A literal x is *critical* in C if flipping x breaks C.
     - In $(x_1 \vee x_2 \vee x_3)$, x_1 critical.

- Clause and variable selection heuristics are *the most* important parts of LS solvers. `YalSAT` and consequently `xnfSAT` use `probSAT`-like selection extended with weights.
- While long OR clauses with many satisfied literals are hard to break, XOR constraints can always be broken. Length is not a good measure of the importance of an XOR constraint.

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;
  - Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$
- While $w(x_1 \vee \ldots \vee x_n) \sim n$, we set $w(x_1 \oplus \ldots \oplus x_m)$ constant. Why?
  1. Simplicity.
  2. A literal x is *critical* in C if flipping x breaks C.
     - In $(x_1 \vee x_2 \vee x_3)$, x_1 critical.
     - In $(x_1 \vee x_2 \vee x_3)$, nothing critical.

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;
  - Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$
- While $w(x_1 \vee \ldots \vee x_n) \sim n$, we set $w(x_1 \oplus \ldots \oplus x_m)$ constant. Why?
  1. Simplicity.
  2. A literal x is *critical* in C if flipping x breaks C.
     - In $(x_1 \vee x_2 \vee x_3)$, x_1 critical.
     - In $(x_1 \vee x_2 \vee x_3)$, nothing critical.

- Clause and variable selection heuristics are *the most* important parts of LS solvers. `YalSAT` and consequently `xnfSAT` use `probSAT`-like selection extended with weights.
- While long OR clauses with many satisfied literals are hard to break, XOR constraints can always be broken. Length is not a good measure of the importance of an XOR constraint.

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;
  - Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$
- While $w(x_1 \vee \ldots \vee x_n) \sim n$, we set $w(x_1 \oplus \ldots \oplus x_m)$ constant. Why?
  1. Simplicity.
  2. A literal x is *critical* in C if flipping x breaks C.
     - In $(x_1 \vee x_2 \vee x_3)$, x_1 critical.
     - In $(x_1 \vee x_2 \vee x_3)$, nothing critical.
     - In a satisfied XOR, all literals are critical.

probSAT-like variable selection

- Clause and variable selection heuristics are *the most* important parts of LS solvers. `YalSAT` and consequently `xnfSAT` use `probSAT`-like selection extended with weights.
- While long OR clauses with many satisfied literals are hard to break, XOR constraints can always be broken. Length is not a good measure of the importance of an XOR constraint.

probSAT-like variable selection

- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;
  - Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$
- While $w(x_1 \vee \ldots \vee x_n) \sim n$, we set $w(x_1 \oplus \ldots \oplus x_m)$ constant. Why?
  1. Simplicity.
  2. A literal x is *critical* in C if flipping x breaks C.
     - In $(x_1 \vee x_2 \vee x_3)$, x_1 critical.
     - In $(x_1 \vee x_2 \vee x_3)$, nothing critical.
     - In a satisfied XOR, all literals are critical.
- For efficiency, $break_w$ tables are cached. Tracking critical literals allows for fast updates.

probSAT-like variable selection
- $x_i \leftarrow \texttt{PickVarIn}(x_1 \vee \ldots \vee x_k)$ with probability $\sim \frac{1}{break_w(x_i)}$
  - Let $B(x_i)$ be the clauses falsified on flipping x_i
  - and w be a clause weighing function;
  - Then $break_w(x_i) = \Sigma_{C \in B(x_i)} w(C)$
- While $w(x_1 \vee \ldots \vee x_n) \sim n$, we set $w(x_1 \oplus \ldots \oplus x_m)$ constant. Why?
  1. Simplicity.
  2. A literal x is *critical* in C if flipping x breaks C.
     - In $(x_1 \vee x_2 \vee x_3)$, x_1 critical.
     - In $(x_1 \vee x_2 \vee x_3)$, nothing critical.
     - In a satisfied XOR, all literals are critical.
- For efficiency, $break_w$ tables are cached. Tracking critical literals allows for fast updates.

- Clause and variable selection heuristics are *the most* important parts of LS solvers. `YalSAT` and consequently `xnfSAT` use `probSAT`-like selection extended with weights.
- While long OR clauses with many satisfied literals are hard to break, XOR constraints can always be broken. Length is not a good measure of the importance of an XOR constraint.

How fast can CNF get? Pooled encoding

$$\texttt{XOR\_l\_n}(x_1, \ldots, x_k) = \texttt{XOR\_d}(x_1, \ldots, x_{n-1}, -y) \wedge \texttt{XOR\_l\_n}(y, x_n, \ldots, x_k)$$

How fast can CNF get? Pooled encoding
$\texttt{XOR\_l\_n}(x_1, \ldots, x_k) = \texttt{XOR\_d}(x_1, \ldots, x_{n-1}, -y) \wedge \texttt{XOR\_l\_n}(y, x_n, \ldots, x_k)$

- To make sure solving XNF strongly outperforms CNF, we performed heavy tuning on the CNF formula. We tried two variants of the Tseitin encoding.
  - First, the linear encoding seen earlier, produced with a stack.
  - Then, a *pooled* encoding produced with a queue. To our knowledge, this encoding is novel.
- We also tried various *cutting numbers* – sizes of the directly-encoded chunks.

How fast can CNF get? Pooled encoding

$$\texttt{XOR\_l\_n}(x_1, ..., x_k) = \texttt{XOR\_d}(x_1, ..., x_{n-1}, -y) \wedge \texttt{XOR\_l\_n}(y, x_n, ..., x_k)$$

$$\texttt{XOR\_p\_n}(x_1, ..., x_k) = \texttt{XOR\_d}(x_1, ..., x_{n-1}, -y) \ \wedge \ \texttt{XOR\_p\_n}(x_n, ..., x_k, y)$$

How fast can CNF get? Pooled encoding
XOR_l_n(x_1, ..., x_k) = XOR_d(x_1, ..., x_{n-1}, -y) ∧ XOR_l_n(y, x_n, ..., x_k)
XOR_p_n(x_1, ..., x_k) = XOR_d(x_1, ..., x_{n-1}, -y) ∧ XOR_p_n(x_n, ..., x_k, y)

- To make sure solving XNF strongly outperforms CNF, we performed heavy tuning on the CNF formula. We tried two variants of the Tseitin encoding.
  - First, the linear encoding seen earlier, produced with a stack.
  - Then, a *pooled* encoding produced with a queue. To our knowledge, this encoding is novel.
- We also tried various *cutting numbers* – sizes of the directly-encoded chunks.

How fast can CNF get? Pooled encoding

$$\texttt{XOR\_l\_n}(x_1, ..., x_k) = \texttt{XOR\_d}(x_1, ..., x_{n-1}, -y) \wedge \texttt{XOR\_l\_n}(y, x_n, ..., x_k)$$

$$\texttt{XOR\_p\_n}(x_1, ..., x_k) = \texttt{XOR\_d}(x_1, ..., x_{n-1}, -y) \;\wedge\; \texttt{XOR\_p\_n}(x_n, ..., x_k, y)$$

↑ cutting number

How fast can CNF get? Pooled encoding
$\texttt{XOR\_l\_n}(x_1, ..., x_k) = \texttt{XOR\_d}(x_1, ..., x_{n-1}, -y) \wedge \texttt{XOR\_l\_n}(y, x_n, ..., x_k)$
$\texttt{XOR\_p\_n}(x_1, ..., x_k) = \texttt{XOR\_d}(x_1, ..., x_{n-1}, -y) \wedge \texttt{XOR\_p\_n}(x_n, ..., x_k, y)$
↑ cutting number

- To make sure solving XNF strongly outperforms CNF, we performed heavy tuning on the CNF formula. We tried two variants of the Tseitin encoding.
  - First, the linear encoding seen earlier, produced with a stack.
  - Then, a *pooled* encoding produced with a queue. To our knowledge, this encoding is novel.
- We also tried various *cutting numbers* – sizes of the directly-encoded chunks.

How fast can CNF get?

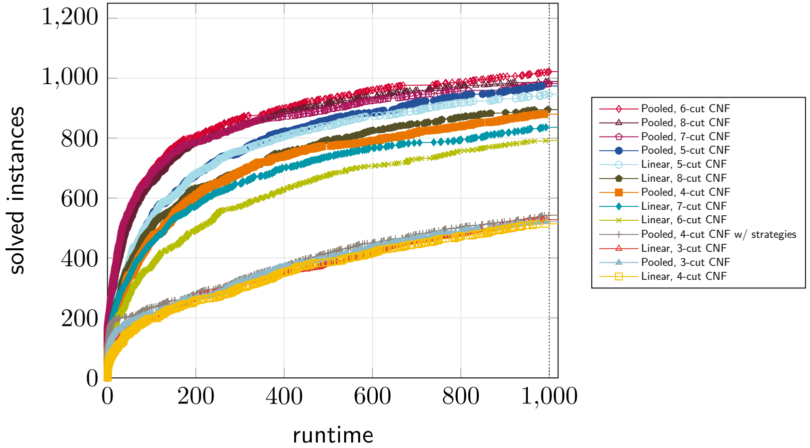
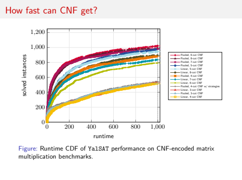


Figure: Runtime CDF of `YalSAT` performance on CNF-encoded matrix multiplication benchmarks.



- On these instances, pooled encodings are better across the board.
- Interestingly, performance initially increases with cutting number and plateaus at 6.

Not as fast as XNF!

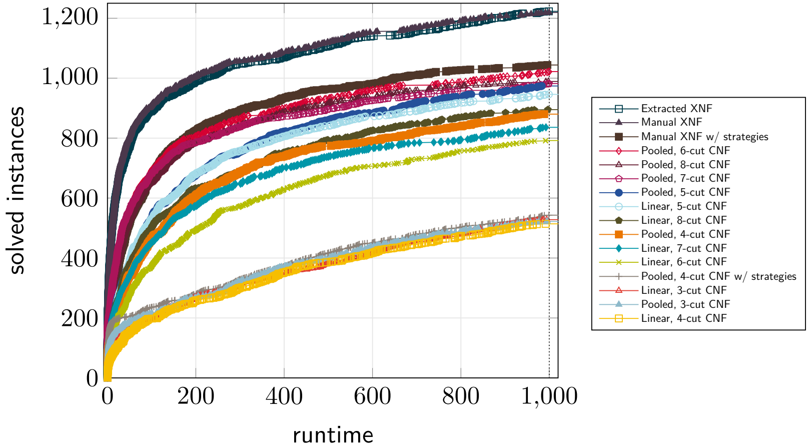
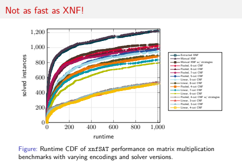


Figure: Runtime CDF of `xnfSAT` performance on matrix multiplication benchmarks with varying encodings and solver versions.



- Here, XNF solving improves over even highly tuned CNF, without having to spend any computational power on optimising the XOR encoding.
- Within a 1000s timeout, our solver operating on XNF can find between 200 and 700 more solutions compared to CNF-based runs in various configurations.

Conclusion

- Implemented SLS with native XOR constraints in `xnfSAT`.
- Observed strong performance improvements on matrix multiplication benchmarks where SLS already outperformed CDCL.
- Propose to experiment with native XOR more widely and to standardise the XNF format.
- `https://github.com/Vtec234/xnfSAT`
- `https://github.com/arminbiere/cnf2xnf`

Conclusion

- Implemented SLS with native XOR constraints in xnfSAT.
- Observed strong performance improvements on matrix multiplication benchmarks where SLS already outperformed CDCL.
- Propose to experiment with native XOR more widely and to standardise the XNF format.
- https://github.com/Vtec234/xnfSAT
- https://github.com/arminbiere/cnf2xnf