Algorithms for Computational Logic
SAT Algorithms

Emmanuel Hebrard (adapted from) João Marques Silva

Outline

1. Algorithms
2. Tree Search
3. Clause Learning
4. Search Techniques
5. Conclusions
1 Algorithms

2 Tree Search
   - The DPLL Solver

3 Clause Learning
   - The CDCL Solvers
   - Clause Learning, UIPs & Minimization

4 Search Techniques
   - Restarts
   - Search Heuristics
   - Clauses Deletion

5 Conclusions

---

How to Solve SAT?

- **Tableau**: Deductive/syntactic system
- **DP**: Resolution
  - Davis & Putnam procedure in 1960 \[DP60\]
- **DPLL**: Semantic system, tree search for a model, with unit propagation
  - Davis, Logemann & Loveland procedure in 1962 \[DLL62\]
- **CDCL**: Conflict-Driven Clause Learning
  - Marques Silva & Sakallah in 1999
  - Moskewicz, Madigan, Zhao, Zhang & Malik in 2001
- Local search and heuristics
Resolution is a powerful proof system, but DP is exponential in memory.

DPLL is memory efficient, but tree search is a weak proof system.

- The length of the shortest refutation is at least as long as in resolution.
- There are cases where it is exponentially larger.

CDCL is memory efficient, very efficient in practice, and as powerful as resolution as a proof system.

Outline

1. Algorithms
2. Tree Search
   - The DPLL Solver
3. Clause Learning
   - The CDCL Solvers
   - Clause Learning, UIPs & Minimization
4. Search Techniques
   - Restarts
   - Search Heuristics
   - Clauses Deletion
5. Conclusions
The DPLL Algorithm

\[ \mathcal{F} = (x \lor y) \land (a \lor b) \land (\overline{a} \lor b) \land (\overline{a} \lor \overline{b}) \]

Level Dec. Unit Prop.
0 \( \emptyset \)
1 \( x \)
2 \( y \)
3 \( a \rightarrow b \rightarrow \perp \)

Level Dec. Unit Prop.
0 \( \emptyset \)
1 \( x \)
2 \( y \)

Level Dec. Unit Prop.
0 \( \emptyset \)
1 \( x \)
2 \( \overline{y} \)

Level Dec. Unit Prop.
0 \( \emptyset \)
1 \( x \)
2 \( \overline{y} \)

\[ \overline{a} \rightarrow \overline{b} \rightarrow \perp \]

\[ \overline{a} \rightarrow \overline{b} \rightarrow \perp \]

Implementation

- Data structures

  - trail:
    - [trail] is the current level in the search tree
    - \( \text{trail}(i) \) is the number of true literals at level \( i \)
    - Stack: push(), back(), pop-back() in \( O(1) \)

- Functions

  - unassign-back()
Backtracking

- Backtrack to decision level 3
- Backtrack to decision level 2

DPLL: Pseudocode

Algorithm: DPLL

while satisfiability = UNKNOWN do
  if unit-propagate() then
    if |unit-literals| = n then  satisfiability ← SAT // a model is found
    else
      trail.push(|unit-literals|) // save current level
      assign(select-lit()) // add a new true literal
  else
    if |trail| = 0 then  satisfiability ← UNSAT // search tree exhausted
    else
      d ← unit-literals[trail.back()] // retrieve previous decision
      while |unit-literals| > trail.back() do  unassign-back() // backtrack
to-propagate ← trail.back()
      trail.pop-back()
      assign( ̅d) // branch out of previous decision
What is a CDCL SAT Solver?

- Extend DPLL SAT solver with:
  - Clause learning & non-chronological backtracking
    - Exploit UIPs
    - Minimize learned clauses
    - Opportunistically delete clauses
  - Search restarts
  - Lazy data structures
    - Watched literals
  - Conflict-guided branching
    - Activity-based branching heuristics
    - Phase saving
  - ...

References:
[DP60, DLL62]
[MSS96, BS97, Z97]
[MSS96, SSS12]
[SB09, VG09]
[MSS96, MSS99, GN02]
[CSK98, BMS00, H07, B08]
[MMZZM01]
[MMZZM01]
[PD07]
How Significant are CDCL SAT Solvers?

Results of the SAT competition/race winners on the SAT 2009 application benchmarks, 20mn timeout

- Limmat (2002)
- Zchaff (2002)
- Berkmin (2002)
- ForKlitt (2003)
- Stengs (2003)
- SatEH (2005)
- Minisat 2 (2006)
- Precosat (2007)
- Ruil (2007)
- Minisat 2.1 (2008)
- Precosat (2009)
- Glucose (2009)
- Clusi (2009)
- Cryptominisat (2010)
- Lingeling (2010)
- Minisat 2.2 (2010)
- Glucose 2 (2011)
- Glomeminisat (2011)
- Contrasat (2011)
- Lingeling 587f (2011)
- GRASP
- DPLL

Clauses Learning

Level Dec. Unit Prop.
0 $\emptyset$
1 $a$
2 $b \rightarrow \bar{f} \rightarrow g$
3 $c$
4 $d \rightarrow \bar{h} \rightarrow i$
5 $e \rightarrow j \rightarrow k \rightarrow l$

$\varphi = (\bar{b} \lor \bar{f}) \land (\bar{a} \lor f \lor g) \land (\bar{d} \lor \bar{c} \lor \bar{b} \lor \bar{h}) \land (h \lor \bar{g} \lor i) \land (\bar{e} \lor j) \land (\bar{e} \lor k) \land (\bar{j} \lor \bar{k} \lor \bar{g} \lor \bar{l}) \land (\bar{f} \lor \bar{g}) \land (a \land b \land c \land d \land \bar{e}) \Rightarrow \bot \land (a \land b \land e) \Rightarrow \bot \land (g \land j \land k) \Rightarrow \bot \land (l \land \bar{g}) \Rightarrow \bot$
Cut of the Implication Graph

- Any cut that separate the decisions from the fail in the decision graph
- Cuts correspond to clauses
  - $\varphi \vdash (a \land b \land c \land d \land e) \implies \bot$: $\varphi \vdash (\overline{a} \lor \overline{b} \lor \overline{e} \lor \overline{d} \lor \overline{e})$
  - $\varphi \vdash (a \land b \land e) \implies \bot$: $\varphi \vdash (\overline{a} \lor \overline{b} \lor \overline{e})$
  - $\varphi \vdash (g \land j \land k) \implies \bot$: $\varphi \vdash (\overline{g} \lor \overline{j} \lor \overline{k})$
  - $\varphi \vdash (g \land l) \implies \bot$: $\varphi \vdash (\overline{g} \lor \overline{l} \lor \overline{k})$
- DPLL (bactracks) equivalent to learning that one decision must be changed
- CDCL learn non-trivial cuts

Level Dec. Unit Prop.
0 $\emptyset$
1 $a$
2 $b \rightarrow \overline{f} \rightarrow g$
3 $c$
4 $d \rightarrow \overline{h} \rightarrow i$
5 $e \rightarrow j \rightarrow \overline{l}$

Learnt clause prevent the algorithm from repeating the same mistake later on

Consider what DPLL would do next:
- Explore branch $a \land b \land c \land \overline{e}$
- Explore branch $a \land b \land \overline{e} \land e$
- Explore branch $a \land \overline{b} \land c \land e$
- Explore branch $a \land \overline{b} \land \overline{e} \land e$

Adding the clause $(\overline{a} \lor \overline{b} \lor \overline{e})$ makes sure that the solver does not explore the last three branches
**Clause Learning and Resolution**

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>(¬a ∨ ¬b)</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>(¬b ∨ ¬z)</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>(¬a ∨ ¬b)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(¬b ∨ ¬z)</td>
</tr>
</tbody>
</table>

- **Analyze conflict**
  - Reasons: x and z
    - Decision variable & literals assigned at lower decision levels
  - Create new clause: (¬x ∨ ¬z)

- Can relate **clause learning** with resolution
  - Learned clauses result from (selected) resolution operations

**Computing a Cut**

- Computing a minimum cut is polynomial (e.g., with Edmonds–Karp algorithm)
  - But costly and more importantly, might often return the failed clause (not asserting!)

- Computing a cut by exploring the implication graph up from the fail
  - At any time the list of open nodes is a valid cut
  - removing a literal from the current cut and replacing it by its parents is a resolution step
Unique Implication Point (UIP)

A *Unique Implication Point* is a node of the current decision level such that any path from the decision variable to the conflict node must pass through it.

- The decision variable is a UIP
- There might be other UIPs

Learn clause \((\overline{w} \lor \overline{x} \lor \overline{y} \lor \overline{z})\)
- But \(a\) is an UIP:
  - Learn clause \((\overline{w} \lor \overline{x} \lor \overline{y} \lor \overline{z})\)
  - But \(a\) is an UIP: learn clause \((\overline{w} \lor \overline{x} \lor a)\)
Clause Learning and Backjumping

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>

Clause $(\overline{x} \lor \overline{z})$ is asserting at decision level 1 (it unit propagates at previous level)

- We want to learn UIP-clauses (clauses containing a UIP) because they are asserting
  - A learned clause is asserting if and only if it contains exactly one literal of the current level because literals from older levels are all falsified
  - A learned clause must contain at least one literal of the current level (since unit propagation did not detect an inconsistency at the previous level)

- Backjump to the highest level of any literal but the UIP

Implementation

- Functions
  - `unit-propagate()` return the failed clause if there is an inconsistency (null otherwise)
  - `backjump(Clause c)` conflict analysis and backjump
Algorithm: CDCL

```plaintext
while satisfiability = UNKNOWN do
    c = unit-propagate()
    if c = Null then
        if |unit-literals| = n then
            satisfiability ← SAT
        else
            trail.push(|unit-literals|)
            assign(select-lit())
    else
        if |trail| = 0 then
            satisfiability ← UNSAT
        else
            backjump(c)
```

Algorithm: Backjump

```plaintext
Input: Conflict clause c
learnt ← analyze-conflict(c)
l ← arg max_{l \in learnt} (level(p) | p \neq l)
lvl ← max_{p \in learnt} (level(p) | p \neq l)
while |unit-literals| > trail[lvl] do
    unassign-back()
while |trail| > lvl do
    trail.pop-back()
add(learnt)
assign(l)
```

Implementing `analyze-conflict`

- We first need to encode the conflict graph
- The parents of a literal \( l \) node are the \( k - 1 \) falsified literals of the clause that unit-propagated \( l \)
- For every variable \( x \), store \( \text{reason}[x] \) the clause responsible for \( x \)'s unit propagation
  - Encoding of the conflict graph
- Which cut(s) should we keep?
  - First UIP clauses
First UIP:
- Learn clause ($\overline{w} \lor \overline{y} \lor \overline{a}$)

But there can be more than 1 UIP

Second UIP:
- Learn clause ($\overline{x} \lor \overline{z} \lor a$)

In practice smaller clauses more effective
- Compare with ($\overline{w} \lor \overline{x} \lor \overline{y} \lor \overline{z}$)

Multiple UIPs used in GRASP
- First UIP learning used in Chaff
  
  [MSS96]
  
  [MMZZM01] and in most modern solvers

Why are First UIP Good?

- Mainly empirical evidences
- Can be seen as a way to detect “hubs”
- How to effectively vaccinate a population against a contagious disease if you have only a limited number of doses?
  - Pick a person randomly, ask her to name a friend, give a vaccine shot to the friend
  - Repeat until there is no dose
- People nominated as friends are more likely to know many people, and hence be super-spreaders
- The decision at failure level is always a UIP (random)
- Other UIPs are “friends” (linked via unit propagation)
Computing First UIPs

- Not all traversal orders reach the first UIP clause
  - E.g., resolve $c$ then resolve $a$
- Solution: resolve literals in reverse chronological order (of unit propagation)
- The first UIP literal is not resolved until all its descendants are
  - By definition, once all its descendants are resolved, it is the only literal of the current level and the exploration can stop

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$w$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$x$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$y$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$z$</td>
<td>$r$ $a$ $c$</td>
</tr>
</tbody>
</table>

Level Dec. Unit Prop.

Implementation

- Data structures
  - level [Variable : $x$] $\mapsto$ int
  - reason [Variable : $x$] $\mapsto$ Clause
    - Change `assign(Literal: $l$)` and `unassign-back(Literal: $l$)`

- Functions
  - `analyze-conflict(Clause: $c$) $\mapsto$ Clause`
    - analyze conflict on clause $c$ and returns a first UIP clause
  - `backjump(Clause: $c$) $\mapsto$ Boolean`
    - returns `false` if the search tree is exhausted and `true` otherwise
Algorithm: First UIP

Input: $c$

1. $\text{seen} \leftarrow \emptyset$
2. $\text{learnt} \leftarrow ()$
3. $\text{reason} \leftarrow c$
4. $n_{\text{cur}} \leftarrow 0$
5. $l \leftarrow \text{None}$
6. $i \leftarrow |\text{unit-literals}| - 1$

repeat

1. foreach $p \neq l \in \text{reason} \setminus \text{seen}$
   1. add $p$ to $\text{seen}$
   2. if $\text{level}[p] = |\text{trail}|$ then
      1. $n_{\text{cur}} \leftarrow n_{\text{cur}} + 1$
   3. else
      1. add $p$ to $\text{learnt}$

while unit-literals[$i$] is not in seen do

1. $i \leftarrow i - 1$
2. $l \leftarrow \text{unit-literals}[i]$
3. $\text{reason} \leftarrow \text{reason}[l]$
4. $n_{\text{cur}} \leftarrow n_{\text{cur}} - 1$

until $n_{\text{cur}} > 0$

add the last explore literal $l$ to $\text{learnt}$

Clause Minimization I

- Learn clause $(\overline{x} \lor \overline{y} \lor \overline{z} \lor \overline{b})$
- Apply self-subsuming resolution (i.e. local minimization) [SB09]
- Learn clause $(\overline{x} \lor \overline{y} \lor \overline{z} \lor \overline{b})$
- Apply self-subsuming resolution (i.e. local minimization)
Clause Minimization II

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td>a, c</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>e, d, ⊥</td>
</tr>
</tbody>
</table>

- Learn clause \((\overline{w} \lor \overline{x} \lor \overline{c})\)
- Learn clause \((\overline{w} \lor \overline{x} \lor \overline{c})\)
- Cannot apply self-subsuming resolution
  - Resolving with reason of \(c\) yields \((\overline{w} \lor \overline{x} \lor \overline{a} \lor \overline{b})\)
- Can apply recursive minimization
- Learn clause \((\overline{w} \lor \overline{x})\)

Marked nodes: literals in learned clause

Trace back from \(c\) until marked nodes or new decision nodes
  - Learn clause if only marked nodes visited

Outline

1. Algorithms
2. Tree Search
   - The DPLL Solver
3. Clause Learning
   - The CDCL Solvers
   - Clause Learning, UIPs & Minimization
4. Search Techniques
   - Restarts
   - Search Heuristics
   - Clauses Deletion
5. Conclusions
Let \texttt{sat-sol} be a \textit{randomized} SAT solver, and \( x \) be a SAT instance.

- The \textit{duration} of a run of \texttt{sat-sol}(\( x \)) depends on the random seed.
- SAT solvers are Las-Vegas algorithms: guaranteed correctness, unknown runtime.
  - Their runtime distribution can be leveraged to improve their efficiency!
- This is true of all exact solvers (MIP, CSP, etc.).

### Heavy tails

- Runtime distributions are rarely Gaussian.
- Often \textit{Heavy tailed}.
- The average may be greatly skewed to the right.
Example: pigeon hole

- Pigeon hole formula $PHP^{n\to n-1}$:

\[
(x_{1,1} \lor x_{1,2} \lor \ldots \lor x_{1,n-1}) \land \\
\ldots \\
(x_{n,1} \lor x_{n,2} \lor \ldots \lor x_{n,n-1}) \land \\
\bigwedge_{1 \leq i < j \leq n} (\overline{x_{i,1}} \lor \overline{x_{j,1}}) \land \\
\bigwedge_{1 \leq i < j \leq n} (\overline{x_{i,2}} \lor \overline{x_{j,2}}) \land \\
\ldots \\
\bigwedge_{1 \leq i < j \leq n} (\overline{x_{i,n-1}} \lor \overline{x_{j,n-1}})
\]

Pigeon 1 needs a hole
Pigeon n needs a hole
Hole 1 can contain at most 1 pigeon
Hole 2 can contain at most 1 pigeon
Hole $n-1$ can contain at most 1 pigeon

- DPLL on the Pigeon hole formula takes exponential time

\[
(x_{1,1} \lor x_{1,2} \lor \ldots \lor x_{1,n-1} \lor \overline{x_1}) \land \\
\ldots \\
Pigeon 1 needs a hole
\]

- Variable $x_1$, if $\text{true}$, allows Pigeon 1 to have its own hole, making the problem easy
- If Variable $x_1$ is set to $\text{false}$, the problem is not satisfiable, and it takes a time exponential in $n$ to prove it
- If we suppose that the solver branch on $x_1$ first and uniformly randomly pick the value $\text{true}$ or $\text{false}$:
  - It will solve the problem in under a second with probability $\frac{1}{2}$
  - It will solve the problem in $\Theta(2^n)$ time with probability $\frac{1}{2}$
  - In expectation: $\Theta(2^{n-1})$ time!
Search Restarts

- What if we **restart** the solver if no solution is found after 1s?

- Chances of taking more than 10 seconds is $\frac{1}{2^{10}}$

- Search restarts can reduce the runtime expectation when the runtime distribution is heavy tailed

---

**Search Techniques**

- When a time limit $\tau$ is reached, we stop and resume search from the start

- Let $t$ be a random variable equal to the runtime of the solver

\[
T = p(t \leq \tau) \cdot \mathbb{E}_p[t \mid t \leq \tau] + (1 - p(t \leq \tau)) \cdot (\tau + T)
\]

\[
T = \mathbb{E}_p[t \mid t \leq \tau] + \frac{(1 - p(t \leq \tau))\tau}{p(t \leq \tau)}
\]

- Simple Markov Decision Process with two states ("solved" and "not solved")
  - There is a stationary (constant) policy $\pi^*$ that minimizes the runtime $T(\pi^*)$
When the expectation of the runtime is unknown, the *Luby’s universal strategy* guarantees a runtime of $T(\tau^*) \log T(\tau^*)$

$$\tau_i = \begin{cases} 
2^{k-1}, & \text{if } i = 2^{k-1} - 1 \\
\tau_{i-2^{k-1}+1}, & \text{if } 2^{k-1} \leq i < 2^k - 1 
\end{cases}$$

In practice, the geometric sequence $\tau_i = f^i$ works well
Maximum Degree

Minimum Domain
Minimum Domain / Degree

\[ \{1, 2, 3, 4, 5, 6\} \{1, 2, 3, 5, 6\} \{1, 2, 3\} \{2, 3\} \]

\[
\begin{array}{c}
\x_0 \x_1 \x_2 \\
\x_3 \x_4 \x_5 \\
\x_6
\end{array}
\]

\[
\begin{array}{c}
\{5, 6\} \{4, 5\} \\
\{1, 2, 3, 4, 6\} \{1, 2, 3, 4\} \{1, 2, 3\}
\end{array}
\]

\[
\begin{array}{c}
\text{domain: } 2 \\
\text{degree: } 3 \\
\text{domain size: } 4
\end{array}
\]

\[
\begin{array}{c}
\text{domain: } 3 \\
\text{degree: } 3 \\
\text{domain size: } 3
\end{array}
\]
Branching Strategies in SAT

- Same principles in SAT and all other tree-search methods:
  - **Variable ordering**: on which variable should we branch first?
    - The one on which we will fail on both subtrees, to get out of the *unsatisfiable* branch
    - Otherwise, on the one that will minimize the size of the subtrees
  - **Value ordering**: on which variable should we branch first?
    - The one most likely to lead to a solution
    - If the current subtree is not satisfiable, it does not matter (much), both branches must be explored
  - Most of the time is spent getting out of *unsatisfiable subtrees*: the variable ordering is more important than the value ordering
    - When solving an optimization problem top-down, finding good quality solutions quickly is important
    - Interaction with clause-learning

- **Variable State Independent Decaying Sum (VSIDS)**
- Assigns a weight to variables involved in conflicts: *activity score*
- Variants exist:
  - Increment weight of the literals in the learned clause
  - Increment weight of the literals in the learned clause and all variables resolved during conflict analysis
- The activity score $A(i)$ of a variable $x_i$ is the *decayed* sum of the weight increments:
  - Let $b_j(i)$ be equal to 1 if variable $x_i$’s activity was incremented in the $j$-th fail, and let $0 < \gamma \leq 1$ be a constant, and $k$ the number of fails
  $$ A(i) = \sum_{j=1}^{k} \gamma^{k-j} b_j(i) $$
When backjumping with an asserting clause, we undo potentially useful search

Suppose that the variables between the conflict and assertion levels encode a (relatively) independent problem: its solution is lost

Phase saving: branch using the previous value

- If the previous solution still stands, it will be found efficiently

Synergy with clause learning

- Intuitively, we want to learn clauses that constrain variables in an unsatisfiable core: recently learned clauses are still asserting if we use phase saving

A SAT solver typically fail (tenth of) thousands times per second

- Learn a new clause on every fail

- Learned clauses tend to be long

Unit propagation via watched literal is efficient, but still accounts for most of the run time

Moreover, not all clauses are equally useful, some never unit propagate

Can we reliably predict which clauses are more promising and forget the rest?
Some intuitive criteria:

- **Length**: long clauses unit propagate (probably) less often.
- **Activity**: clauses with less active literals have (historically) unit propagated more often.

Deleting long and inactive learned clauses is useful.

Clause deletion is very important, but difficult to parameterized (how often?, how many?)

Length and activity are not perfect predictors.

Some clauses are long but useful.

In general, a clause of length \( L \) can be satisfied in \( 2^L - 1 \) ways.

The clause \( x_1 \lor \ldots \lor x_{100} \) from the direct encoding of the CSP variable \( x \in \{1, \ldots, 100\} \) can be satisfied in only 100 ways (the variable takes exactly one of the 100 values).

- The unit literal \( x_i \) unit propagates \( \overline{x_j} \) for all \( j \neq i \) via pairwise or sequential clauses.

Clauses involving *inter-dependent* literals are more likely to unit propagate: the implicit relation on dependent variables is tighter.
We want something *efficient*

Idea: variables that unit propagated at the same level tend to be more linked together

**Literal Block Distance** $lbd(0)$

Let $level[l]$ be the decision level at which literal $l$ was inferred.

$$lbd(c) = |\{level[l] | l \in c\}|$$

Solver “Glucose” was the first to use this idea of “Glue clauses” and was very successful [Audemard & Simon]
What is a CDCL SAT Solver?

- **Extend DPLL SAT solver with:**
  - Clause learning & non-chronological backtracking
    - Learn First-UIP clauses
    - Minimize learned clauses
    - Opportunistically delete clauses (LBD)
  - Search restarts
  - Lazy data structures
    - Watched literals
  - Conflict-guided branching
    - Activity-based branching heuristics
    - Phase saving
  - ...

[DP60, DLL62]  
[MSS96, BS97, Z97]  
[MSS96, SSS12]  
[SB09, VG09]  
[MSS96, MSS99, GN02]  
[CSK98, BMS00, H07, B08]  
[MMZZM01]  
[MMZZM01]  
[PD07]