関数型言語Haskellによる高速SATソルバの開発

鶴田 唯一[1]　楢崎 修二[2]

概要：与えられた命題論理式の充足可能性を判断するSAT問題を解くSATソルバは近年、高速化が目覚しい。実用的な問題へのSATソルバの利用も可能になってきた。しかしSATソルバの多くは従来の手続き型言語やオブジェクト指向言語で記述されている。
一方、純関数型言語Haskellは参照透明性、高度な型システム、各種最適化を実現したコンパイラ、各種の並行・並列機能などの特徴を持ち、研究にも実用にも広く使われている。しかし、Haskell上のSATソルバは十分研究開発されていない。本研究では現在の高速SATソルバが利用するCDCL、リスタート、VSIDSといったアルゴリズムを実装したHaskell上の純関数的な高速SATソルバを開発し、既存ソルバとの比較を行う。

キーワード：関数型言語、探索と推論、充足性問題、SAT

1. はじめに

与えられた命題論理式を充足させるような変数の割り当てがあれば割り当て例を返し、そうでなければ充足不能という解を返す問題、すなわち命題論理式の充足性を判別するSAT問題は、近年のSATソルバの高速化により応用分野が広がりつつある。

実際、1万を超える変数数の論理式の判定が1秒と掛からずに行えるようになったため、これまで個々の問題の持つ特徴を用いたヒューリスティックに基づく探索を必要としてきたいわゆる探索問題は、問題を論理変数群で符号化（SAT符号化という）することで、論理式として表現し、それをSATソルバで解くことが速度の点からも現実的な手法となりつつある。

探索は特に人工知能の基礎技術であるため、SATの理解・応用は人工知能の研究において重要と言える。問題の性質をうまく論理表現に制約として表現する必要性もあるが、一方で現実的な問題群が持つ性質から得られた統計的なヒューリスティックの導入は近年のSATソルバの高速化を達成するための一因にもなっている。これからも現実的な大規模な問題を解くにあたり、問題に応じた符号化やSATソルバの改良は必要な基礎技術であると思われるため、SATソルバをブラックボックス化するのではなく手の入れられるツールボックスとして整備することが必要である。

このような観点からは、多くのSATソルバはC言語で記述されている現状は満足の行くものではない。本研究では、現在注目を浴びつつある関数型言語の代表であるHaskell言語上に高速なSATソルバを実現することを目的とする。Haskellは高度な最適化を行うコンパイラを持つ言語である。ただし、Haskellは非破壊的な代入を（基本的には）許さない純関数型言語であるため、単純に既存の手続き型またはオブジェクト指向言語での実装を移植するというわけにはいかない。本論文では、性能評価実験を通してHaskell上でのSATソルバの実装方針を検討する。

論文の構成は以下のとおりである。2章および3章でまずSAT問題およびHaskellについて説明する。4章では、現在の高速SATソルバについて説明する。5章で我々が開発中のhSATについて述べ、6章で実験により処理速度を評価する。7章はまとめである。

2. SAT

SATとは与えられた命題論理式を充足させるような変数の割り当てがあれば割り当て例を返し、そうでなければ充足不能という解を返す問題、すなわち命題論理式の充足性を判別する問題である[1]。基本的には2値を取る変数群に適切な値を割り当てる問題であるため、変数がN個あれば、2^Nの大きさを持つ探索空間上の問題である。当然、判定の高速化のためには探索問題の削減が必要である。自然数を二進数に符号化することで様々な問

[1] 長崎大学, Nagasaki University
b609321@nagasaki-u.ac.jp
[2] 長崎大学工学研究科,Nagasaki University
narazaki@cs.cis.nagasaki-u.ac.jp

© 2013 Information Processing Society of Japan

題を SAT 問題として解くことができる。

3. Haskell

Haskell は純関数型言語である。特徴は副作用を持たない関数からプログラムを構成する純関数型言語であり、入出力や非決定性を持つプログラムを純関数型言語の枠組みで記述するための圏論のモナドを基にした Monad、柔軟な関数の合成を可能とするクラスへの拡張を持ち開かれた体系を持つ型システム、幅広いライブラリの存在、高度な最適化を行うコンパイラの存在といった特徴を持つ言語である。

4. 現在の高速 SAT ソルバ

4.1 代表的な高速化手法

近年なされた SAT ソルバの高速化は (1) 矛盾節からの節学習、(2) 決定変数の選択戦略、(3)SAT 問題の構造の理解などの手法の発明による。以下に簡単な説明を行う。

4.1.1 矛盾節からの節学習（CDCL）

原始的なソルバの動作は、式が充足するまで値が割り当てられていない変数に順次値を割り当てるというものである。高速化を行うためには割り当てるべき変数をどのように決定するか、また、矛盾が生じた場合にどこから割り当てを破棄してやり直すかを決める戦略が重要になる。矛盾節からの節学習では、矛盾が生じた場合にその矛盾を発生させた割り当ての組を節とし、その否定したものを元々の論理式に追加することで、再びそのような割り当てが起きないようにする。探索中に得られた知識を制約として追加するため、効率良く探索空間を縮小することができる [2]。

4.1.2 決定変数の選択戦略

対象とする論理式が乗法標準形の形式であるとする。その中に変数一つのみからなる節がある場合（単位節という）、次に値を割り当てるべき変数（およびその値）はその節を構成する変数に一意に定まる。この変数に他の（反対の）値を割り当てると、ただちに論理式が充足不能となるためである。従って、この変数には割り当て方に関する選択の余地はない。一方、単位節を一つも含まない論理式では、ある変数をなんらかの手法で任意に選び、真または偽の値を仮に割り当てて、割り当てを進め、もし矛盾が発生した場合はバックトラックにより、反対の値を割り当てることが必要になる。この選ばれた変数を決定変数と呼ぶ。決定変数の選択によってはバックトラックを早めに起こすこと、あるいは単位節を早く（または多く）生成することができ、従って探索空間を削減することができる。例えば、節を構成する変数が少ない（従って単位節に早く変換できる）節に含まれる変数を決定変数に選ぶというヒューリスティックは場合によっては有効である。ただし、節数が多くなった場合はどの変数がよく使われているかを検査するコストも無視できない。このようなことから、カウンタやその近似値を用いて決定変数を決めるヒューリスティックがよく用いられる。代表的なアルゴリズムである VSIDS[8] では、このアルゴリズムを使用しなかった場合に比べ数倍から十数倍の高速化が実現できる。

4.1.3 変数間の構造の理解

個々の問題が作る論理式は変数間の制約と考えることができる。節学習はその制約を探索に利用しようとするものだが、それ以外にもリスタート、リテラルブロック距離などの手法が提案されている。リスタートは変数番号を振りなおし初期状態から再探索を行うものである [6]。変数番号を変えることで、多面的に変数間の関係を獲得することができる。リテラルブロック距離は変数間の依存関係の重みを近似獲得し、決定変数の選択に利用しようとするものである [3]。

4.2 代表的な実装

以下に代表的な実装を紹介する。

4.2.1 minisat

minisat は特に実装技術に注意が払われて開発された現在の高速 SAT ソルバの代表である [5]。1 万変数を超える問題を 1 秒程度で解くことができ、実用規模の問題を解くことが可能である。実装上の特徴は C++で記述されているが、わずか 2000 行程度のコンパクトなものであること、それまで提案されていた多くの手法が導入された非常に高速なソルバであるということである。そのため、その後の多くの高速 SAT ソルバ（例えばなど）の開発ベースとして用いられている。

4.2.2 funsat

funsat は現在 hackage に登録されている唯一の SAT ソルバである [4]。Haskell プログラムの速度やプログラミング時の柔軟性などから minisat 並の速度は追求されておらず、「中規模な問題を対象にした」ものとなっている。高速化のために CDCL、リスタート、VSIDS を実装している。

haskell の持つ機能は使っているが、高速化のため Monad ベース、すなわち状態遷移を繰り返す極めて手続き的なプログラムとなっている (正確には状態を保持する ST Monad、例外を処理する Exception Monad を組み合わせたものになっている)。そのため、一般にわかりやすいものとは言えず、ライブラリの並列化や、コンパイラの最適化手法の改善を利用したり、恩恵を受けることが困難なものとなっている。速度を得る現実的な路線を取っており、純粋な関数型言語のフレームワーク内での実装にこだわったものにはなっていない。

© 2013 Information Processing Society of Japan

```
type Literal = Int                          -- 変数リテラルはIntにより表現
type LiteralPack = Data.Vector.Unboxed Literal  -- Unboxed Vectorを用いる（可変長）
data Var = Var { literal :: Literal, ... }          -- その他、決定変数かどうかのフラグなど
type DepList = Data.IntMap.IntMap (Var, [Int])  -- 節学習のための割当て依存関係の記録
type Clause = LiteralPack                   -- 第0要素はリテラル数を記録する
type ClausePack = Data.Vector.Vector Clause             -- CNF形式の論理式(節集合)はVectorで管理
data Sat = Sat { clauses :: ClausePack, dependency :: DepList, ...} -- 求解に必要な全情報
```

図 1 hSAT の主要データ型

```
-- |最終割当ての結果を表す
data AssignResult = Conflicting Sat | NeedToDecision Sat

-- |充足すればSatを、そうでなければNothingを返す
solve :: [Sat] -> Maybe Sat
solve sats@(sat:_)
  | isSatisfied sat = sat
  | decisionLevel sat == 0 = Nothing
  | Conflicting sat' <- selectAndAssign sat = solve (backtrack sat' sats)
  | NeedToDecision sat' <- selectAndAssign sat = solve $ assignVar v (asLiteral v) sat' : sats
  where
    v = selectDecisionVar sat'

-- |単位節がなくなるまで単位節中の変数（割当てるべき値が一意に決まる）の割当てを繰り返す
selectAndAssign :: Sat -> AssignResult
selectAndAssign sat
  | conflicted sat = Conflicting sat
  | hasUnitClause = selectAndAssign $ assignVar (asVar l) l sat
  | otherwise = NeedToDecision sat
  where
    l = findUnitCaluse sat
```

図 2 hSAT の処理の流れ

5. hSAT

以上から我々は Monad に過度に依存することなく順関数型言語での高速 SAT ソルバの実現を目指して、新規に SAT ソルバの開発を行うことにした（以下 hSAT と呼ぶ)。hSAT の特徴を以下に挙げる。

- Haskell のみで記述
  実用規模の問題を解くには 10 万変数を超えるような SAT 問題を実用的な時間で解けることが望ましい。しかし、Haskell ではそのような性能を持つ SAT ソルバは未だ実装されていない。いくつかのベンチマークによれば、現在の水準の最適化コンパイラ (例えば gcc) による C 言語プログラムと比べ、もっとも広く使われている Haskell 用最適化コンパイラである ghc が生成するオブジェクトではそれに比べて 2〜10 倍程度遅いものとなっている。従って現実的な目標としては 1 万変数程度の問題が実用的な時間で解けることを目指すことにする。funsat も同様に「中規模の問題を解く」ことを開発目標として挙げている。
- 純関数型の枠組で記述
  先に述べたように、移植性に加え、SAT ソルバ自身の理解や改良、問題に応じたヒューリスティックの導入・利用方法のカスタマイズを容易に行うため、純関数型の枠組みで実装することにする。従って funsat と違い Monad は用いない。また、以上から直接得られる恩恵として、種々の可視化ツール、デバッグツールの開発も容易となる。

5.1 開発方針

まず基本方針として、Vector ライブラリを利用することにした。Haskell において、複数のオブジェクトを保持する最も手軽なコンテナは List 型であるが、List 型は要素アクセスに O(n) の手間が掛かる。また、fold/map に関してよく知られた最適化を行わない。一方 Vector クラスは標準ライブラリではないもの List 型の持つ処理速度上の問題点を解決したものである。大規模な SAT 問題では 1000 を超えるリテラルや 10 万を超える節の保持が必要となるため、Vector 型の使用はそれだけで速度を大きく改善する。その他、標準ライブラリ中の IntSet, IntMap などを利用する。

これにより、変数リテラルは Int で、節は Vector Int で、論理式は Vector Clause とする方針が決定される。実際にはそれぞれ必要な情報を付加したデータ型となる。簡単ではあるが図 1 に基本となるデータ型の定義を示す。

hSAT では基本アルゴリズムとして CDCL を使用することにする。CDCL を実装し（図 2 参照)、速度を評価してデータ型や関数設計に関してもっともよいと思われるものを選んだ。

© 2013 Information Processing Society of Japan

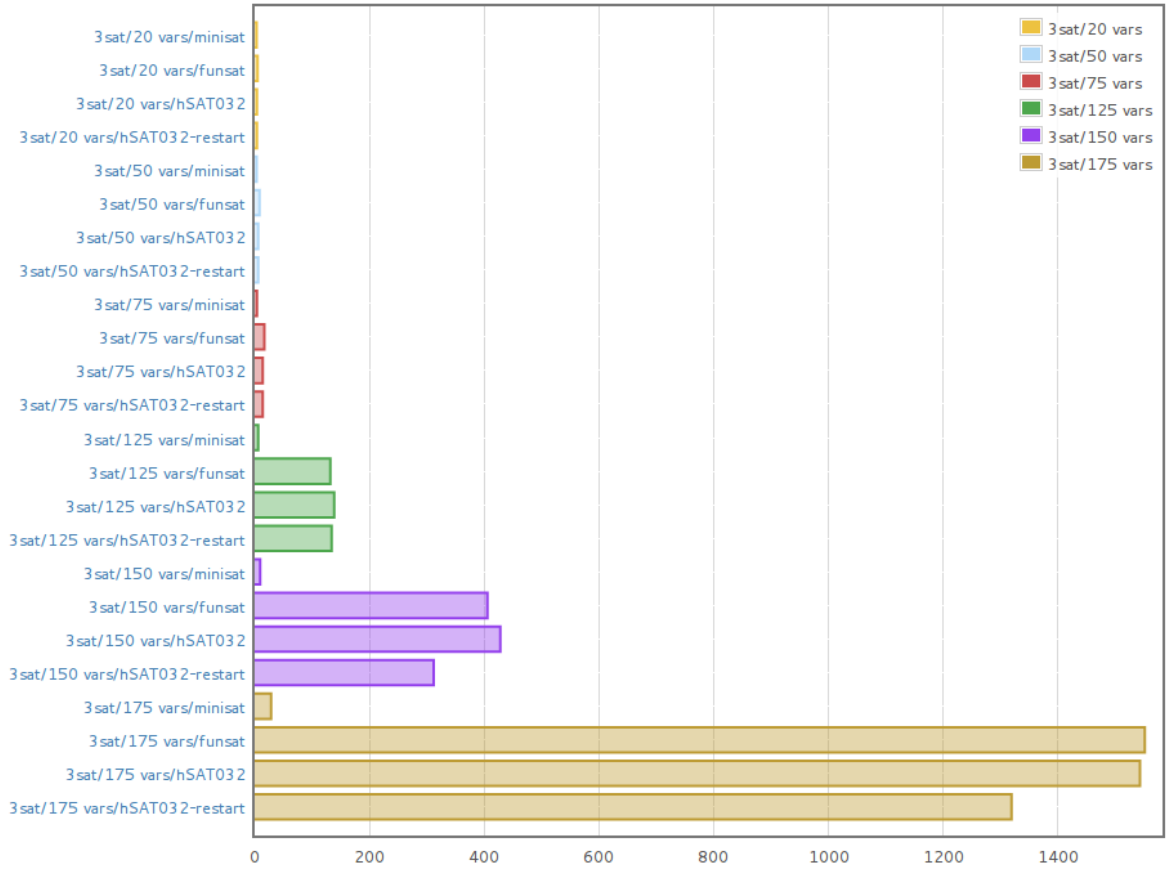

図 3 3-SAT 問題を用いた 3 種類のソルバの比較

特に節に関しては予備実験で様々な定義を試み、現在の節を構成するリテラル集合 (Unboxed Vector Int) と、節を構成するリテラル数を記録するものを選んでいる。リテラルもリテラル数もどちらも Int で表現できるので長さが（リテラル数+1）の Unboxed Vector Int で表現できる。そして、Sat データ型に処理を進めるために必要な全情報（節数、割当て情報、バックトラック情報など）を保持することにした。[*1]

リスタート, VSIDS は現在実装中であり、評価や調整が必要である。

6. 性能評価

本章では、以下の設定で hSAT の実行速度を評価する。対象として funsat, hSAT を用いる。それぞれ x86_64 上の linux、ghc 7.6.1 でコンパイルしたものを使った。なお、funsat は公式リポジトリのものは 7.6.1 ではコンパイルできなかったので、適宜修正を行い、funsat と hSAT の両者の最適化オプションを揃えてコンパイルした。minisat のバージョンは 2.2 である。

計測にあたっては、criterion[9] を用いた。criterion は、異常値の自動削除、キャッシュ効果の自動クリアなどの機能を有するベンチマークツールである。

なお、今回は消費メモリについての調査は行なっていない。

6.1 3-SAT 問題での評価

まず、SAT ソルバの評価を行う大会で用いられたベンチマーク問題の中から 3-SAT 問題を使って評価を行う [10]。3-SAT は現実の問題を基にした SAT 問題と違い、統計的なヒューリスティックが使えず、基礎となる節や式の変換のコストを反映されがちな問題である。従って minisat であっても、少ないリテラル数であっても計算時間が掛かる傾向が見られる（わずか 175 変数でも平均 28 ミリ秒掛かっている）。出現する変数の個数を順に 20, 50, 75, 125, 150, 175 とし、それぞれ 100 種類の問題に関する平均計算時間を測定したものものを図 3 に示す。

minisat に比べると、Haskell 上のソルバはどちらも非常に遅い。変数数の増加によりその比は大きくなることから計算量のオーダーが違うものと思われる。hSAT と funsat は同じような計算時間である。ただし、リスタートアルゴリズムを導入すると 1〜2 割ほどの高速化を達

[*1] なお、開発初期において Monad 及び破壊的操作を許すデータ型を用いて、手続き型に近い実装での評価も行った。意外なことに順関数型の枠組みでの実装の方が性能がよかったため、現在の方針を取ることにした。性能が上がらなかった理由は破壊操作版では過度の Vector 内要素の書き換えが発生するが、この操作に対する十分な最適化が行われていないためではないかと考えられるが詳細については調査が必要である。

© 2013 Information Processing Society of Japan

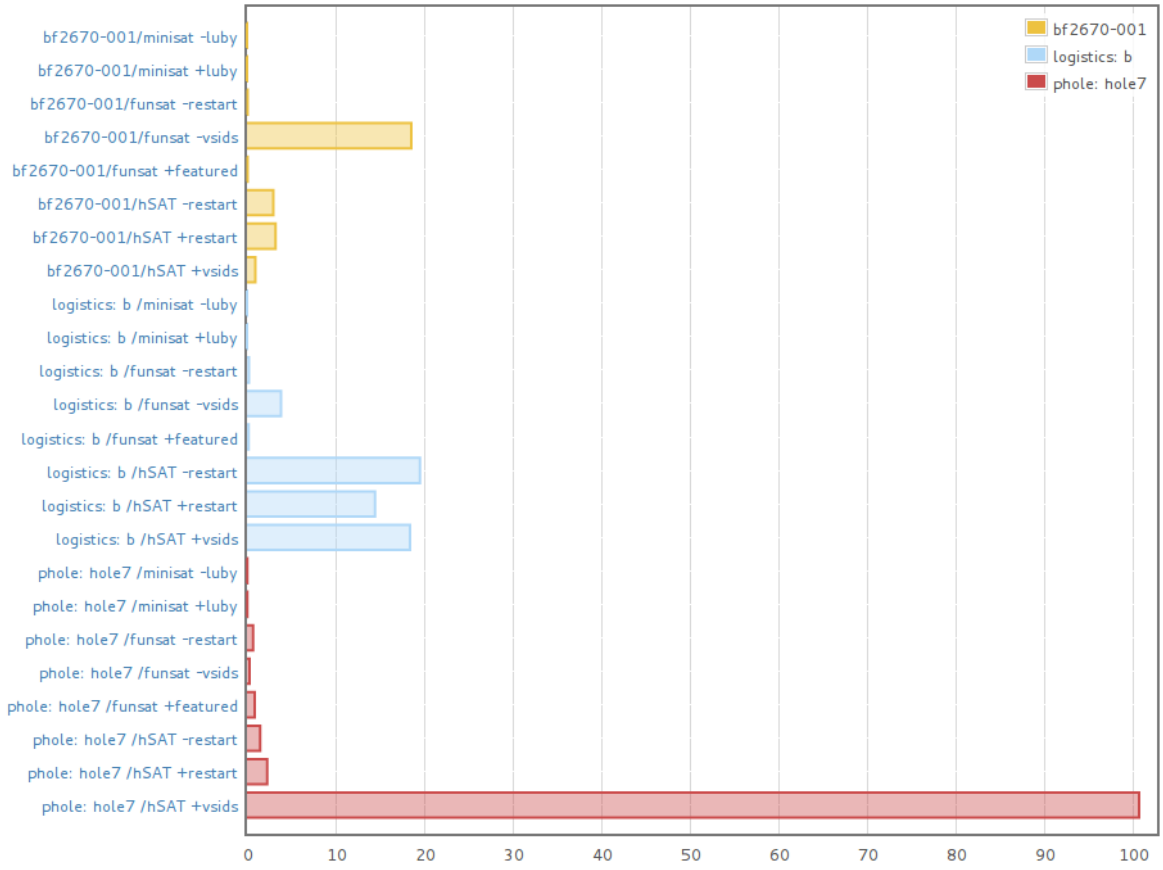

図 4 現実的な問題での minisat, funsat における各種アルゴリズムの効果

成できた。175 変数以上では実行時間は実用的とは言いがたく、また消費するメモリ量も実用的な範囲を超えるので、3-SAT 問題に関しては funsat よりも hSAT の方が優秀であると考えられる。

6.2 現実的な問題による評価と VSIDS の導入

続いて現実的な問題を SAT に変換した問題での評価を行う。用いた問題は funsat のパッケージに同梱されているベンチマーク問題から選択したものである（いずれも充足可能な問題である）。変数数は 500〜2000 となり、3-SAT 問題と比較すると規模も大きく種々の高速化手法の利用が必要とされる。結果を図 4 に示す。

この実験では 3 つのソルバの差が大きくなっており、Haskell による SAT ソルバは minisat と比べると現実的な問題規模では実用性にまだまだ問題があると言わざるを得ない。そして、この実験では hSAT は funsat に逆転され、問題によっては極端に遅いことがわかった。

理由を調べるためソルバが使用するアルゴリズムに注目する。minisat, funsat は実行時オプションにより使用するアルゴリズムを変えることができる。図中で、ソルバ名の後の「+」で始まるオプションは未使用を、「-」は使用を意味する。luby はリスタートに関する Luby アルゴリズムを、featured はデフォルト設定の全アルゴリズムの使用を、vsids は決定変数選択に関する VSIDS アルゴリズムをそれぞれ意味する。いずれのアルゴリズムも高速化に貢献しているが、中でも VSIDS の利用は効果が大きいことがわかる。

上記の結果を踏まえ、hSAT に VSIDS を導入した。VSIDS は変数出現状況に関する統計的な近似を用いた手法であるため、厳密な計数を行う場合に比べ、3-SAT 問題では不利になるが、現実的で規模の大きな問題に対しては VSIDS の利用が望ましい。VSIDS は確率的に使用されるべき一種の混合戦略であるため、望ましい使用確率を実験により求めることにした。結果を図 5 に示す。期待した通り、速度の改善が見られた。ただし問題によって効果には差がある。10 倍早くなる（この場合は funsat に匹敵する性能と言える）問題もあれば、むしろ悪化するものもある。このように、最適な確率は問題に依存しており、平均的には改善は期待できるが大幅な改善は困難かもしれない。

7. おわりに

我々が現在開発中の Haskell 上の SAT ソルバ hSAT の性能評価を行った。minisat とは大きな差があるが、3-SAT 問題においては既存の Haskell 上の SAT ソルバ funsat に上回る性能を出すことができた。現実的な問題

© 2013 Information Processing Society of Japan

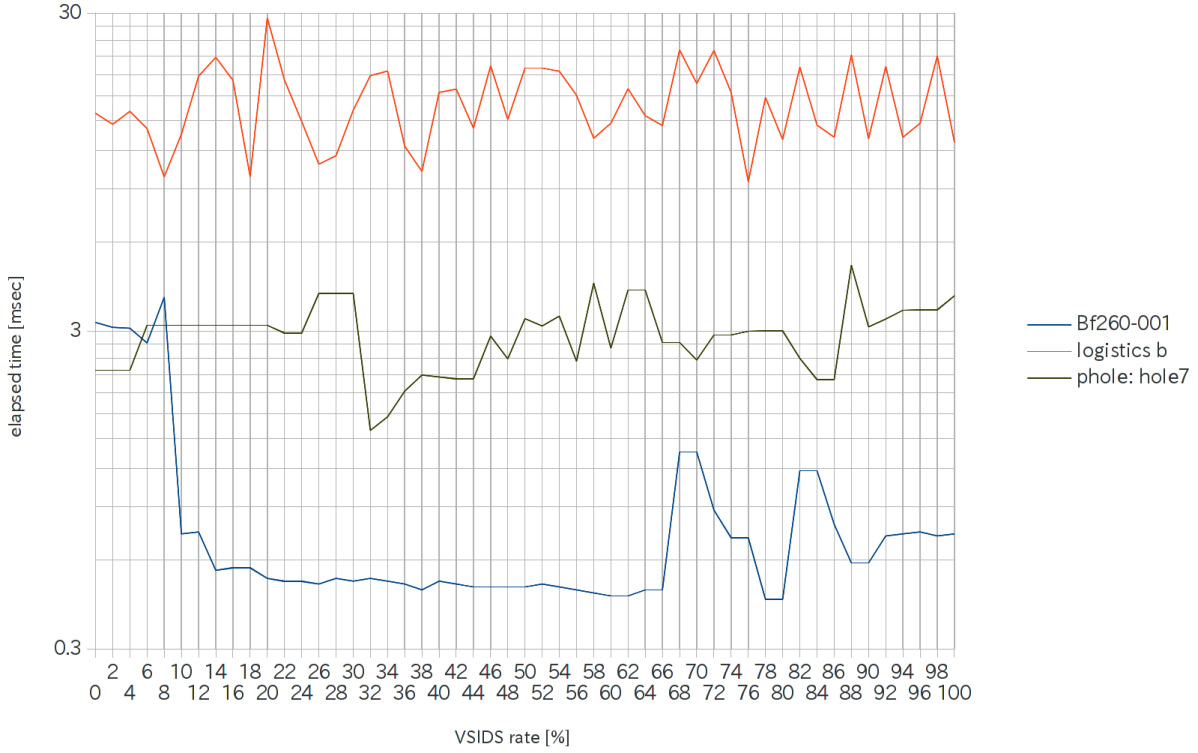

図 5 hSAT における VSIDS アルゴリズム利用確率の影響

においては差が大きく開き課題があることがわかったが、実験結果に基づきリスタートおよび VSIDS の実装を調整することで、若干ではあるがその差を縮めることができた。

以下に、さらなる性能改善のための課題を挙げる。

7.1 リスタート戦略の再実装など

リスタートはその間隔やランダム性が性能に影響する。Luby アルゴリズムがよく使われるものであるが、今回の実験では実装することができず簡易的なものになっている。今後は Luby アルゴリズムの実装について検討したい。また、同様にリテラルブロック距離といった他の手法の導入も検討する予定である。

7.2 DPH による並列化

純関数型ではデータの依存関係がないため並列化しやすいということは関数型言語の開発当初から言われてきた。ghc では基本同期命令を使った明示的な並列実行機構とともに、並行に関数を実行する並行コンテナデータ型を使った自動並行機能もライブラリとして整備されつつある (dph[7])。現在のマルチコア CPU や GPU での並列度が高いハードウェアを利用するにあたり dph は非常に魅力的である。既存 Haskell コードの並列化は、基本的には使用するデータ型を対応するものに変更するだけでできるため、hSAT の高速化もそれほど困難ではないと思われる。特に funsat と違い Monad を用いていないため、大きな変更はないはずである。ただし、現行の ghc 処理系では十分なチューニングが行われておらず (特に並列 GC の起動による待ち時間が馬鹿にならない)、むしろ平行化によって速度低下が見られたため、今回の実験評価では対象としていない。今後、Haskell 処理系の改善が行われた際には、速度評価を改めて行ないたいと考えている。

現在主流の CPU では 4〜8 スレッドの同時実行が可能である。従って理想的には数倍の速度向上が期待できる。今回の実験結果から、我々は今回用いたベンチマーク問題において 10 倍の高速化が達成できれば funsat とほぼ同等の性能と言えるのではないかと考えている。並列化は有力な手段となると思われる。

参考文献

[1] 井上克己; 田村直之, SAT ソルバーの基礎, 人工知能学会誌, 25(1), 57-67, (2010).
[2] 鍋島英知, 栄剛秀, 高速 SAT ソルバーの原理, 人工知能学会誌, 25(1), 68-76, (2010).
[3] Audemard, G, Simon, L.: Predicting Learnt Clauses Quality in Modern SAT Solvers, Proc. of 21 Int. Joint Conf. on AI (IJCAI-09), 399-404, (2009).
[4] Denis, Bueno: funsat 0.6.2(online)、入手先 ⟨http://hackage.haskell.org/funsat⟩, 入手先 ⟨http://github.com/dbueno/funsat⟩ (2011.02.27).
[5] Een, N and Sorensson, N: An extensible SAT-solver, 6th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT2003), 502-518, (2003).
[6] Luby, Michael and Sinclair, Alistair and Zuckerman, David: Optimal speedup of Las Vegas algorithms, In-

© 2013 Information Processing Society of Japan

formation Processing Letters, (1993).
[7] Marlow, Simon: Parallel and Concurrent Programming in Haskell CEFP 2011 (LNCS 7241), 339-401, (2012).
[8] Moskewicz, MW *et al.*: Chaff: Engineering an efficient SAT solver, Proceedings of DAC 2001, 530-535, (2001).
[9] O'Sullivan, Bryan: The criterion package 0.6.2.1: 入手先 ⟨http://hackage.haskell.org/criterion⟩, (2013.02.03)
[10] SATLIB - Benchmark Problems (Uniform Random 3-SAT), 入手先 ⟨http://www.cs.ubc.ca/ hoos/SATLIB/benchm.html⟩ (2000.08.11).

© 2013 Information Processing Society of Japan