6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Proving by Assignment Trees that SAT Solvers are Non-Polynomial in Unit Propagation Framework with 1 Selection and Cache*

Gábor Kusper

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at/
e-mail: gkusper@risc.uni-linz.ac.at

Abstract

We introduce the Unit Propagation Framework with 1 Selection and Cache. This framework is a SAT solver, which gets as a parameter a strategy function, which tells us how to solve SAT. This framework is suitable to simulate SAT solver algorithms, which use unit propagation to solve SAT problems and the decision, which unit to propagate, is made about one selected (usually a minimal) clause. SAT solvers may use cache, too. Any such algorithm can be described by a strategy function. A strategy function tells us which unit to propagate. We show that General Unicorn-SAT (GUS), developed by the author, and the Davis, Putnam, Logemann and Loveland procedure (DPLL) can be simulated in this framework by giving the suitable strategy function. We show that DPLL and GUS with cache use the same number of unit propagations, $2^n + 2^{n-1} - 2$, where n is the number of variables, to show that CC, the set of all clear (full) clauses, is unsatisfiable. Furthermore we introduce the Limitation Lemma, which says that there is no SAT solver in this framework, which can show that CC is unsatisfiable using fewer unit propagations than DPLL. Hence, SAT solvers are non-polynomial in this framework. To show this we use assignment trees that are suitable to represent a run of this framework.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: SAT Solver

Key Words and Phrases: SAT, DPLL, GUS, UPFw1SC

*Supported by the Upper Austrian Government (scholarship), FWF SFB F013 (P1302) and Austro-Hungarian Action Foundation.

1. Introduction

Propositional Satisfiability is the problem of determining, for a formula of the propositional calculus, if there is an assignment of truth values to its variables for which that formula evaluates the true. By SAT we mean the problem of propositional satisfiability for formulae in conjunctive normal form (CNF).

SAT is the first, and one of the simplest, of the many problems which have been shown to be **NP**-complete [1]. It is dual of propositional theorem proving, and many practical **NP**-hard problems may be transformed efficiently to SAT. Thus, a good SAT solver algorithm would likely have considerable utility. SAT solver algorithms are based on either unit propagation or resolution or combination of these techniques, see section 6 in the survey [2].

We compare two algorithms, the General Unicorn-SAT algorithm (GUS) [3], developed by the author, and the well known Davis, Putnam, Logemann and Loveland procedure (DPLL) [4]. Both of them are based on systematic unit propagation until we find a model or we show that there is no model. We did several experiments running the two algorithms on clause sets having specific properties. We found that they use the same number of unit propagation on unsatisfiable clause sets if we use cache in GUS. We investigated the so-called CC clause set, which contains all n-clauses, if we have n variables. We found that the two algorithms use the same number of unit propagation to show that CC unsatisfiable. We tried to construct better algorithms. We found that it is impossible if we use only unit propagation. The goal of this paper is to prove this experimental result.

We introduce the Unit Propagation Framework with 1 Selection and Cache. This framework is a SAT solver, which gets as a parameter a strategy function, which tells us how to solve SAT. This framework is suitable to simulate SAT solver algorithms, which use unit propagation to solve SAT problems and the decision, which unit to propagate, is made about one selected (usually a minimal) clause. SAT solvers may use cache, too. Any such algorithm can be described by a strategy function. A strategy function tells us which unit in to propagate. We show that GUS and DPLL can be simulated in this framework by giving the suitable strategy function. We show that DPLL and GUS with cache use the same number of unit propagations, $2^n + 2^{n-1} - 2$, where n is the number of variables, to show that CC is unsatisfiable. Furthermore we introduce the Limitation Lemma, which says that there is no SAT solver in this framework, which can show that CC is unsatisfiable using fewer unit propagations than DPLL. Since there is an unsatisfiable clause set, which can be solved only in non-polynomial time, SAT solvers are non-polynomial in this framework.

To show this we use assignment trees that are suitable to represent a run of this framework. This framework systematically propagates assignments from the strategy set created by the strategy function, which is the parameter of this framework. We either find a model for the input clause set or there is no model. An assignment tree contains the propagated assignments. We give algorithms to turn assignment trees to cached unit assignment trees. We do this, because the number of edges

in a cached unit assignment tree is equal to the number of used unit propagation steps. We show that if an assignment tree represent a run of this framework on CC, and we turn it to a cached unit assignment tree, then it has at least $2^n + 2^{n-1} - 1$ nodes. Hence, it has $2^n + 2^{n-1} - 1$ edges. Therefore, any SAT solver simulated in this framework uses exponentional number of unit propagation steps on CC.
This does not mean that we solved the famous **N** = **NP**? problem. This means that if we want to construct a polynomial SAT solver (which seems impossible), then it is not enough to use only unit propagation, decision based on one clause and cache.

2. Definitions

Let V be a *set of Boolean variables*. The negation of a variable v is denoted by !v. Given a set U, we denote !U = {!u | u ∈ U}. *Literals* are the members of the set V ∪ !V. *Positive literals* are the members of the set V. *Negative literals* are their negations. If w denotes a negative literal !v, then !w denotes the positive literal v.

A *clause* is a finite set of literals that does not contain simultaneously any literal together with its negation. A clause is interpreted as the disjunction of its literals. An *assignment* is a finite set of literals that does not contain simultaneously any literal together with its negation. An assignment is interpreted as the conjunction of its literals. Informally speaking, if an assignment A contains a literal v, it means that v has the value *True* in A. Note that both, a clause and an assignment, is defined by the same definition. A *clause set* or *formula* (*formula in CNF form*) is a finite set of clauses. A clause set is interpreted as the conjunction of its clauses. An *assignment set* is a finite set of assignments. An assignment set is interpreted as the disjunction of its assignment. If C is a clause, then !C is an assignment. If A is an assignment, then !A is a clause. If S is a clause set, then !S is an assignment set. If T is an assignment set, then !T is a clause set.

The empty set is denoted by {}. The *length* of a set U is its cardinality, denoted by |U|. Let n be the *number of variables*, i.e., n = |V|. If C is a clause and |C| = k, then we say that C is a *k-clause*. Special cases are *unit clauses* or *unit* which are *1-clauses*, and *clear* or *full clauses* which are *n-clauses*. Note that any unit clause is at the same time clause and assignment. The clause set *CC* or *set of clear clauses* is the set of all clear clauses.

The clause C *subsumes the clause* B if C is a subset of B. We say that the clause set T is the *equivalent clear clause set* of the clause set S if T is a subset of CC and T is the largest set that has the property that each clause in it is subsumed by a clause in S. We say that the clause sets S and T are *equivalent*, denoted by S ≡ T, if they have the same equivalent clear clause set. Note that if S ≡ T then S and T are logically equivalent. We say that a clause C *is subsumed by the clause set* S if for all clear clauses, which are subsumed by C, there is a clause in S which subsumes it.

If we say that a *literal* v *occurs* in a set, we mean that this set contains v or it contains a set, which contains v. However, if we say that a *variable* v *occurs* in a

set, we mean that this set contains the literal v, or it contains the literal !v, or it contains a set, which contains the literal v, or the literal !v. We denote by Var(U) the *set of variables occurring in* the set U. We say that two clauses *differ in some variables* if these variables occur in both clauses but as different literals. We say that *resolution can be performed on* two clauses if they differ only in one variable. Note that this is not the usual notion of resolution, because we allow resolution only if it results in a non-tautologous resolvent. For example resolution cannot be performed on {v, w} and {!v, !w} but can be performed on {v, w} and {!v, z}. If resolution can be performed on two clauses, say A and B, then the *resolvent*, denoted by Res(A, B), is the union of them excluded the variable they differ in.

Definition 2.1. (Unit and Hyper-Unit Propagation) UP(S,{c}):= {C \ {!c} |C ∈ S ∧ c ∉ C}, HUP(S, A) := {C \ !A | C ∈ S ∧ C ∩ A = {}}, where S is a clause set, c is a literal and A is an assignment.

Clauses of a clause set with minimal length are called *minimal clauses* of this clause set. If C is a clause and c is a literal and c occurs in C then the *resolution-mate of* clause C by literal c, denoted by rm(C, c), is the union of C and {!c} excluded {c}. Note that resolution can be performed always on C and rm(C, c), and Res(C, rm(C, c)) = C \ {c}, i.e., we obtain a shorter clause. The *sub-model* generated from the clause C and from the literal c, denoted by sm(C, c), is !rm(C, c). The name "sub-model" comes from the observation that in a resolution-free clause set an assignment created from a shortest clause in this way is a part of a model [5], i.e., a sub-model. Note that rm(C, c) is a clause but sm(C, c) is an assignment.

We may use *sub-model propagation* as a synonym of hyper-unit propagation if the propagated assignment is a sub-model. An assignment M is a *model* for a clause set S if HUP(S, M) is the empty clause set. A clause set is *satisfiable* (*sat*) if there is a model for it. A clause set is *unsatisfiable* (*unsat*) if it is not satisfiable. A clause set is *trivially satisfiable* if it is empty and it is *trivially unsatisfiable* if it contains them empty clause. An assignment A is a *failing assignment* if HUP(S, A) is unsatisfiable. Note that if A is a failing assignment then !A is subsumed S. A function F is a *SAT solver algorithm* if it has the type "function SATsolver(S : clause set) : assignment" and F(S) = {} if the clause set S is unsatisfiable, and F(S) = M if the clause set S is satisfiable and M is a model for S.

We recall two basic SAT solver algorithms, the well know *DPLL* and *GUS*.

Algorithm 2.2. (DPLL)

```
function DPLL(S: clause set): assignment    // We assume that S is not empty.
begin
  A := {};
  Let C be a minimal clause in S;
  while (C is unit) do                      // Boolean Constraint Propagation.
    S := UP(S, C);
    A := A ∪ C;
    if (S is empty) then return A;
    Let C be a minimal clause in S;
  od
  if ({} ∈ S) then return {};               // S is unsatisfiable.
  Let c be a literal in C;
  B := DPLL(S ∪ {c});                       // Either c is part of a model,
  if B != {} then return A ∪ B;
  B := DPLL(S ∪ !{c})                       // or !c is part of a model,
  if B != {} then return A ∪ B;
  return {};                                // or there is no model.
end
```

Algorithm 2.3. (GUS)

```
function GUS(S: clause set): assignment     // We assume that S is not empty.
begin
  Let C be a minimal clause in S;
  while (C is not empty) do
    Let c be a literal in C;
    B := GUS(HUP(S, sm(C, c)));             // Sub-model propagation.
    if B != {} then return sm(C, c) ∪ B;    // Either sm(C, c) is part of a model,
    C := C \ {c};                           // or C \ {c} is subsumed by S.
  od
  return {};                                // {} is subsumed by S, i.e., it is unsat.
end
```

3. Representation

In this paper we use three kinds of representation to represent clauses, assignments, clause sets and assignment sets: logical, set based and literal matrix. Let us see the same formula in all three representations.

Example 3.1.

Logical representation:	(a ∨ b) ∧ (!a ∨ c ∨ d) ∧ (!b ∨ c ∨ !d).
Set based representation:	{{a, b}, {!a, c, d}, {!b, c, !d}}.
Literal matrix representation:	++xx
	−x++
	x− + −

We believe that this example is self-explanatory. In literal matrix representation + means positive literal occurrence, − is negative literal occurrence and x is no variable occurrence. We believe that set based representation is the best for giving

formal definitions, lemmas and proofs. The literal matrix representation is the most readable and concise and therefore very good to give examples.

4. Strategy Function

The aim of this paper is to construct a common framework of SAT solver algorithms, which use unit propagation to solve SAT problems. We can characterize such SAT solvers by a strategy function, which tells us in any "situation" which unit to propagate. What does any "situation" mean? Usually SAT solver algorithms use a decision function that is based on some selected clauses and on a heuristic and it makes a decision which unit to propagate. We do not deal with heuristics, because we investigate SAT solvers running on CC, and since CC is unsatisfiable, it makes any heuristic obsolete. Hence, "situation" is the set of selected clauses. The selected clauses are selected from the input problem and the decision is based on them. DPLL and GUS select only one, a minimal, clause. We deal with SAT solvers, which select only one clause. We characterize these algorithms by a strategy function.

Definition 4.1. (General Strategy Set) Let S be a clause set. Let T be an assignment set. If $\{\} \notin$ T, Var(T) $\subseteq$ Var(S) and !T $\cup$ S is unsatisfiable, then we say T is a general strategy set generated by S.

This means that a T assignment set is a general strategy set generated by S if the negation of T together with S is unsatisfiable, i.e., T subsumes any model for S. Although, we did not define the notion of subsume to assignments and assignment set, but it is the same as the one defined for clauses and clause sets.

Note that the negation of an assignment is a clause, and the negation of an assignment set is a clause set. We do not allow the empty assignment in T, because the negation of the empty assignment is the empty clause and a clause set, which contains the empty clause, is unsatisfiable, i.e., $\{\{\}\}$ would be always a general strategy set.

Set of variables of T should be the subset of set of variables of S. This condition makes sense, because there is no use to propagate a literal (as a unit) if it does not occur in S. Furthermore we need this condition to be able to count children generated by a strategy set. See it later.

Note that if S is unsatisfiable then $\{\}$ is a general strategy set generated by S.

Definition 4.2. (General Strategy Function) Let STF be a function of type "function STF(S: clause set): assignment set". If we have for all clause set S that STF(S) is a general strategy set generated by S, then we say STF is a general strategy function.

It would be more elegant to use lambda notation in this definition, but we decided to use human readable definitions. The same definition using lambda notation: STF = Lambda S. Let T be a general strategy set generated by S.
In these definitions clause set S has the intended meaning: set of selected clauses.

We will see that Unit Propagation Framework with 1 Selection does not see the whole input clause set, it see only one selected clause. The notion of general strategy set is to general for our purposes. Therefore, we do specialization.

Definition 4.3. (Strategy Set) Let C be a clause. Let T be an assignment set. If $\{\} \notin T$, $Var(T) \subseteq Var(C)$ and $!T \cup \{C\}$ is unsatisfiable, then we say T is a strategy set generated by C.

In this definition clause C has the intended meaning: the selected clause.

Note that if C is the empty clause, i.e., $\{C\}$ is unsatisfiable, then $\{\}$ is a strategy set generated by C.

Definition 4.4. (Strategy Function) Let STF be a function of type "function STF(C: clause): assignment set". If we have for all clause C that STF(C) is a strategy set generated by C, then we say STF is a strategy function.

These specialized definitions assume that there is only one selected clause. We could call them one selection strategy set and function, respectively. But to be sort we use the name strategy set and function, respectively.

We will see that we use the strategy set to break down a big problem to smaller ones.

Let us see how can we characterize by a strategy function a SAT solver. We give the strategy function of DPLL and GUS.

Algorithm 4.5. (Strategy Function of DPLL)

```
function DPLL_STF(C: clause):assignment set
begin
  if (C is empty) then return {};
  if (C is unit) then return {C};
  Let c be a literal in C;
  return {{c}, {!c}};
  end
```

We can see that DPLL_STF is indeed a strategy function. If C is a unit, then DPLL_STF(C) = {C} and $!\{C\} \cup \{C\}$ is unsatisfiable. If C is not a unit and C is not empty, then !DPLL_STF(C) is unsatisfiable, because it contains a unit and its negation. If C is empty then the clause set from which it is selected is unsatisfiable. Hence, DPLL_STF is a strategy function.

We show that DPLL_STF is a suitable strategy function to characterize DPLL. To show this, we have to show that if DPLL selects the minimal clause C from the input clause set, then DPLL_STF(C) returns the same assignments, up to undeterministicness of "Let c be a literal in C", that would be propagated by DPLL. If C is the empty clause, then DPLL do not propagate any assignments, because the input clause set is unsatisfiable. Therefore, DPLL_STF(C) returns the empty assignment set. If C is a unit then DPLL propagates it. Therefore, DPLL_STF(C) returns the assignment set {C}. If C is not empty and not a unit then DPLL select a literal c from C and propagates first {c} than, if it is a failing assignment, {!c}. Therefore DPLL_STF(C) select a literal c from C and returns the assignment set

{{c}, {!c}}. Hence, DPLL_STF is a suitable strategy function to characterize DPLL.

Algorithm 4.6. (Strategy Function of GU)

```
function GUS_STF(C: clause): assignment set
begin
   T := {};
   while C is not empty do
     Let c ∈ C;
     T := T ∪ {sm(C, c)};
     C := C \ {c};
   od
   return T;
end
```

We show that this function is a strategy function. We know that Res(C, !sm(C, c)) = C \ {c} for any c ∈ C. Therefore, in the clause set !GUS_STF(C) ∪ {C} we can obtain the empty clause by using resolution |C| times. Hence, !GUS_STF(C) ∪ {C} is unsatisfiable for any clause C.

We show that GUS_STF is a suitable strategy function to characterize GUS. To show this, we have to show that if GUS selects the minimal clause C from the input clause set, then GUS_STF(C) returns the same assignments, up to undeterministicness of "Let c be a literal in C", that would be propagated by GUS. If C is the empty clause, then GUS do not propagate any assignments, because the input clause set is unsatisfiable. Therefore, GUS_STF(C) returns the empty assignment set. If C is a unit then GUS propagates it. Therefore, GUS_STF(C) returns the assignment set {C}. If C is not empty and not a unit then GUS select a literal c from C and propagates first sm(C, c) than, if it is a failing assignment, select an unselected literal d from C, i.e., d is in C \ {c}, and propagates sm(C \ {c}, d), etc..., until there is no more unselected literal in C or we find a model. Therefore GUS_STF(C) returns the assignment set {sm(C, {c_1}), sm(C \ {c_1}, c_2),..., sm(C \ {c_1, c_2, c_n-1}, c_n)}, where C = {c_1, c_2,..., c_n}. Hence, GUS_STF is a suitable strategy function to characterize GUS.

5. Unit Propagation Framework with 1 Selection

Now let us introduce the Unit Propagation Framework with 1 Selection. This is a common framework of SAT solver algorithms that use unit propagation and decision based on one selected clause. This framework selects a minimal clause from the input clause set. We say that this framework see only one selected clause. We say that the selected clause is a known clause. Known clauses are the selected one and the ones we known by expansion rule. The expansion rule states that if HUP(S, A) is unsatisfiable, i.e., A is a failing assignment, then !A is subsumed by S. This case we say that the clause !A is known. The set of know clauses is always subsumed by the input clause. In this framework we systematically propagate

assignments from the strategy set generated by the selected clause. We either find a model or if all assignment from the strategy set is a failing one, then there is no model, because the negation of strategy set together with its generator is unsatisfiable, hence, the input problem is unsatisfiable.

We introduce first the simplified version of the framework, which returns true if the input clause set is satisfiable, otherwise false. This version is more concise but says nothing about the model.

Algorithm 5.1. (Unit Propagation Framework with 1 Selection, Simplified Version)

```
function UPFw1S(I: clause set, STF : strategy function): Boolean
begin
  if (I = {}) then return True;    // If the input clause set, I, is trivially sat,
                                   then return true.
  if ({} ∈ I) then return False;   // If the input clause set trivially unsat,
                                   the return false.
  Let C be a minimal clause in I;  // 1 selection.
  T := STF(C);                     // T is a strategy set generated by C.
  for each A ∈ T do                // We propagate each assignment from T.
    if (UPFw1SC( HUP(I, A), STF)) then return True;   // If HUP(I, A) is sat I,
                                                        then I is sat.
  od
  return False;   // If non of the assignments is a model, then there is model.
end
```

We can see that we use the strategy function to create a strategy set. First we select a minimal clause from the set. One could ask why do we select a minimal one? It is because GUS does so and we want to simulate GUS and for DPLL it is all the same which literal do we choice. If one does not like this explanation, then we will see that we always run the framework on CC, where each clause has the same length, i.e., all of them are minimal.

The next point is to show that this framework is a SAT solver. To be able to do this we give a version which returns a model if the input clause set is satisfiable, otherwise the empty assignment. We introduce a new variable: K, the set of known clauses, to show that the framework is correct.

Algorithm 5.2 (Unit Propagation Framework with 1 Selection)

```
function UPFw1S(I: clause set, STF : strategy function): assignment
```

```
                                        // We assume that I is not empty
begin
   if ({} ∈ I) then return {};          // I: input clause set
   Let C be a minimal clause in I;      // selection
   K := {C};                            // K: set of known clauses
   T := STF(C);                         // T is a strategy set generated
                                        by C
   for each A ∈ T do                    // We propagate each assignment
                                        from T.
     J := HUP(I, A);                    // propagate A on I
     if (J is empty) then return A;     // A is a model for I
     M := UPFw1S(J, STF);               // if J is sat, M is a model for J;
                                        otherwise, M is empty
     if (M is not empty) then return A ∪ M;  // then A ∪ M is a model for I
     K := K ∪ {!A};                     // else HUP(I, A) is unsat; by
                                        expansion rule we know !A
   od
   return {};                           // K = {C} ∪ !T, i.e., K is unsat,
                                        i.e., I is unsat
end
```

We show that UPFw1S is a SAT solver. This algorithm assumes that I, the input problem, is not empty. Set of known clauses K is always subsumed by I, because K contains only the selected clause, C, and clauses added by the expansion rule. We try each assignment A from the strategy set T until we find a model or there is no more assignment to try. To decide whether A is a part of a model or not we have to decide whether HUP(I, A) is satisfiable or not. This means we have to recursive call the function with the new input clause set. We may use the same strategy function or use another one. For sake of simplicity we use always the same strategy function. If there is no more assignment to try then K is unsatisfiable, because K = {C} ∪ !T, which is by definition of strategy set is unsatisfiable, and since I subsumes K, we know that I is also unsatisfiable. Hence, if I is a not empty clause set and STF is a strategy function, then UPFw1S(I, STF) is a model for I if it is satisfiable or the empty assignment if it is unsatisfiable. Hence, UPFw1S is a SAT solver.

Let us see how can we simulate DPLL and GUS in this framework.

Algorithm 5.3 (Simulated DPLL)

```
function SimulatedDPLL(I: clause set): assignment
   begin
return UPFw1S(I, DPLL_STF);
end
```

Algorithm 5.4 (Simulated GUS)

```
function SimulatedGUS(I: clause set): assignment
begin
   return UPFw1S(I, GUS_STF);
end
```

We see that it is very easy to simulate a SAT solver in unit propagation framework. We have to give the suitable strategy function and call the framework with the input problem and with the strategy function.

Now if we count UP steps, then we will see that SimulatedGUS uses more UP steps than SimulatedDPLL. But our guess is that GUS is as good as DPLL. Let us analyze GUS. But first we need some denotation power.

6. 6 k-Clause Sets

In this paper clear clauses (or full clauses or n-clauses) play an important role. We have a constant, called CC, which is the set of all clear clauses. We run the Unit Propagation Framework with 1 Selection on CC. We do this, because CC is the worst case for the type of algorithms we can simulate. It is the worst, because it is unsatisfiable and each clause in it has maximal length. Furthermore if we propagate any assignment on CC, then we obtain a clause set which has also the property that it contains all possible combinations but for less variables. We give definitions for such clause sets.

Definition 6.1. ([[U]])
Let U be a set of variables. Then [[U]] is the set of all clause C that has Var(C) = U.

Note that for any U we have that [[U]] is unsatisfiable, [[{}]] = {{}} and [[V]] = CC. Let us see some examples.

Example 6.2 Assume that we have 3 variables a, b, and c.

[[{a}]]	[[{b}]]	[[{c}]]	[[{a,b}]]	[[{a,b,c}]]
+xx	X+x	xx+	++x	+ + +
−xx	x−x	xx−	+−x	+ + −
			−+x	+ − +
			−−x	+ − −
				− + +
				− + −
				− − +
				− − −

Definition 6.3. ($[[2^k]]$) *Let k be a natural number or zero that has $n \geq k$. Let U be a set of variables that has $|U| = k$. Then $[[2^k]] = [[U]]$.*

Note that for any k we have that $[[2^k]]$ is unsatisfiable, $[[2^0]] = [[1]] = [[\{\}]]$ and $[[2^n]] = CC$.

We use $[[2^k]]$ as a synonym for [[U]] if |U| = k and it is irrelevant which variables are contained in U. This notation is useful, while it shows how many clauses are

contained in it. But it is not well-defined, since [[2]] may denote [[{a}]] and also [[{b}]]. Let us see some more examples.

Example 6.4. Assume that we have 3 variables a, b, and c.

[[2]]	[[2]]	[[2]]	[[8]]
+xx	x+x	xx+	+ + +
−xx	x−x	xx−	+ + −
			+ − +
			+ − −
			− + +
			− + −
			− − +
			− − −

Lemma 6.5. *Let U be a set of variables. Let A be an assignment. Then HUP([[U]], A) = [[U \ Var(A)]].*

Proof: Let k be the number of variables in U. We know that [[U]] contains all possible k-clauses on variables of U. If U ∩ Var(A) is empty then HUP([[U]], A) = [[U]]. Otherwise, by definition of hyper-unit propagation, we know that we select those k-clauses, which do not contain any literal from A, and delete from them those literals, which negation are in A. This means we obtain k-|A| length clauses, and since [[U]] contains all possible k-clauses on variables of U, HUP([[U]], A) will contain all possible clauses on variables of U \ Var(A). Hence, HUP([[U]], A) = [[U \ Var(A)]].

This is an important property. We know that if T is a strategy set generated by the clause C then Var(T) ⊆ Var(C) and T is not empty. These properties make sure that if we break down a big problem by a strategy set we obtain smaller ones. We show that if we run a SAT solver on CC then any heuristic is useless. To show this we should understand why do we use heuristics. If there is a model for a clause set then we want to find it as fast as possible, meanin using less number of unit propagation steps. If we have a choice, for example which literal to select, then the heuristic tells us which one to choice. This is useful if the input clause set is satisifiable. But if the input clause set is unsatisfiable, as for example CC, then we have to visit each branch. Hence, any heuristic is useless.

7. Unit Propagation Framework with 1 Selection and Cache

Simulated DPLL and GUS use 1 UP step to solve [[2]] and 4 UP steps to solve [[4]]. This is what we are expected. But for [[8]] simulated DPLL uses 10 UP steps, while simulated GUS uses 11 UP steps. This is not what we are expecting. Our guess is that GUS is as good as or than DPLL. How could be DPLL better than

GUS for [[8]]? Let us analyze simulated GUS. We will see that if we add a cache mechanism to our framework, we can overcome the problem and we can show that DPLL and GUS uses the same number of UP steps to show that CC is unsatisfiable.

Example 7.1. (Running Simulated GUS on [[4]]) Level is the level of recursion of UPFw1S. I, K, T, A are variables of UPFw1S. We count number of UP steps in the last column. The word "empty" states for empty set.

Level	I	K	T	A	action	UPs
1	++ +− −+ −−	++	−+ +x	−+	HUP by A	2
2	empty				return()	2
1	++	++	+x	+x	HUP by A	3
	+− −+ −−	+−				
2	x+	x+	x+	x+	HUP by A	4
	+−					
3	empty				return()	4
2	x+	x+	empty		return()	4
	+−	+−				
1	++ +− −+ −−	++ +− −x	empty		return()	4

We see that the result, empty assignment, is correct since [[4]] is unsatisfiable. The number of UP steps is 4. We have selected always the "first" clause of I to put in K, although, our algorithm says only we have to select a minimal length clause. But in CC each clause has the same length. So if we select another clause then still we need 4 UP steps. We can also see very nicely that K is unsatisfiable when T is empty.

Example 7.2. (Running Simulated GUS on [[8]]) Level is the level of recursion of UPFw1S. I, K, T, A are variables of UPFw1S. We count number of UP steps in the last column. The word "empty" states for empty set. In the example above we already saw how to solve [[2]] and [[4]]. Therefore, we do not repeat these steps; just give a note, how many UP steps are needed to solve them.

Level	I	K	T	A	action	UPs
1	[[8]]	$+++$	$--+$ $-+$x $+$xx	$--+$	HUP by A	3
2	empty				return()	3
1	[[8]]	$+++$ $++-$	$-+$x $+$xx	$-+$x	HUP by A	5
2	[[2]]	Can be solved using 1 UP steps, see Example 7.1				6
1	[[8]]	$+++$ $++-$ $+-$x	$+$xx	$+$xx	HUP by A	7
2	[[4]]	Can be solved using 4 UP steps, see Example 7.1				11
1	[[8]]	$+++$ $++-$ $+-$x $-$xx	Empty		return()	11

If we look at this example carefully we see that after propagating $--+$ we propagate $-+x$ on the same clause set. This means we propagate $--xx$ twice, although, second time we might know the result of propagating $--xx$. To overcome this problem we add a cache mechanism to our framework and obtain the Unit Propagation Framework with 1 Selection and Cache. Furthermore we have to make the strategy function of GUS deterministic. The following example shows that not all the possible outputs of GUS_STF are suitable to use the advantage of cache.

Example 7.3. GUS_STF({{a, b, c}}) = {{!a, !b, c}, {!a, b}, {a}}.
GUS_STF({{a, b, c}}) = {{a, !b, !c}, {b, !c}, {c}}.

The first output is produced if we select always the "last" literal to generate sub-model. The second one is produced if we select always the "first" literal to generate sub-model. For our goal, use the advantage of our cache mechanism, the first output is more suitable. So let us redefine our GUS_STF function.
We need a total ordering on literals to make the strategy function of GUS deterministic. Let us assume that we have a total ordering on literals. Now we can select the least and the greatest literal of a clause.

Algorithm 7.4. Deterministic GUS Strategy Function)

```
function GUS_STF(C: clause): assignment set
begin
  T := {};
  while C is not empty do
    Let c ∈ C be the greatest literal in C;      // formal line was: Let c ∈ C;
    T := T ∪ {sm(C, c)};
    C := C \ {c};
  od
  return T;
end
```

We did not give a new name to this algorithm, because we use only this variant from this point on. Now GUS_STF is deterministic, because we have a total ordering on literals.
Now we build in the cache in our algorithms. HUP becomes a trinary function, because we need to know the assignment, which has been propagated. Our cache contains (assignment M, clause set J) pairs. We obtain the clause set J by hyper-unit propagating assignment M on the input clause set. Since arty of HUP is changed we shell also redefine UPFw1S with the new name UPFw1SC.

Algorithm 7.5. (HUP with Cache)

```
function HUPwC(I: clause set, A, B: assignment): clause set
begin                                  // I: input clause set, A:
                                       assignment to propagate, B:
                                       propagated assignment
  M := B;                              // M is the key, B is the first part
                                       of the key
  while (A is not empty) do            // we propagate each literal as unit
                                       from A
    Let a ∈ A be the least literal in A; // it is okay, because the total ordering
                                       on literals
    M := M ∪ {a};                      // we expand the key
    Let J be the value for key M from  // we look up the cache with key M
    the cache;
    if (J is undefined) then           // if there is now value for key M
      J := UP(I, {a});                 // we do UP and put the
      Put into the cache (M, J),       // result in the cache
      where M is the key, J is the value;
    fi
    I := J;                            // otherwise we use the value
                                       from the cache
    A := A \ {a};                      // we have processed a, we can delete it
  od
  return I;
end
```

HUPwC does the same as HUP. It propagates all the literals of the assignment as units, but not always. If we find the result of the unit propagation in the cache,

then we use that value. If not, then we put the result of unit propagation in the cache. Now we can introduce the Unit Propagation Framework with 1 Selection and Cache.

Algorithm 7.6. (Unit Propagation Framework with 1 Selection and Cache)

```
function UPFw1SC(I: clause set, B: assignment, STF : strategy function):
assignment
begin                                          // B: assignment already propagated
   if ({} ∈ I) then return {};                 // I : input clause set
   Let C be a minimal clause in I;             // selection
   K := {C};                                   // K : set of known clauses
   T := STF(C);                                // T is a strategy set generated by C
   for each A ∈ T do                           // We propagate each assignment
                                               from T.
     J := HUPwC(I, A, B);                      // propagate A on I
     if (J is empty) then return A;            // A is a model for I
     M := UPFw1SC(J, A ∪ B, STF);              // if J is sat, M is a model for J
                                               // otherwise, M is empty
     if (M is not empty) then return A ∪ M;    // then A ∪ M is a model for I
     K := K ∪ {!A};                            // else J, i.e., HUP(I, A) is unsat.
                                               By expansion rule we know !A
   od
   return {};                                  // K = {C} ∪ !T, i.e., K is unsat,
                                               i.e., I is unsat.
end
```

UPFw1SC is the same as UPFw1S but it uses HUPwC instead of HUP. Now we can rewrite SimulatedDPLL and SimulatedGUS using the new framework to simulate them.

Algorithm 7.7. (Simulated DPLL)

```
function SimulatedDPLL(I: clause set): assignment
begin
   return UPFw1SC(I, {}, DPLL_STF);
end
```

Algorithm 7.8. (Simulated GUS)

```
function SimulatedGUS(I: clause set): assignment
begin
   return UPFw1SC(I, {}, GUS_STF);
end
```

Let us see again the run of simulated GUS on [[8]]. Now we expect that it uses the same number of UP steps as simulated DPLL.

Example 7.9. (Running Simulated GUS on [[8]] in UPFw1SC) Level is the level of recursion of UPFw1SC. I, K, T, A are variables of UPFw1SC. We count number of UP steps in the last column. The word "empty" states for empty set.

In the example above we already saw how to solve [[2]] and [[4]]. Therefore, we do not repeat this steps, just give a note, how many UP steps needed to solve them.

Level	I	K	T	A	action	UPs
1	[[8]]	$+++$	$--+$ $-+x$ $+xx$	$--+$	HUP by A	3
2	empty				return()	3
1	[[8]]	$+++$ $++-$	$-+x$ $+xx$	$-+x$	HUP by A	4
2	[[2]]	Can be solved using 1 UP steps, see Example 7.1				5
1	[[8]]	$+++$ $++-$ $+-x$	$+xx$	$+xx$	HUP by A	6
2	[[4]]	Can be solved using 4 UP steps, see Example 7.1				10
1	[[8]]	$+++$ $++-$ $+-x$ $-xx$	empty		return()	11

We see that it needs 10 UP steps to show that [[8]] is unsatisfiable. The difference to the first example can be seen in the third row. Here the number of used UP steps is 4 instead of 5, since in or cache the result of $--$xx is already there, so we have to propagate instead of $-+$x only x+x. One may ask the following question. We have the strategy set containing $--+$, $-+$x, +xx, but it is a set. So we could propagate these assignments in any other order. Will we use the same number of UP steps if we propagate the assignments in some other order? Yes, we will. For example if we propagate first $-+$x then we use 2 UP step, but in our cache there is the result of propagating $--$xx. So if we propagate now $--+$, then we have to propagate only x$-+$ instead, i.e., we need 4 UP steps to propagate both. This means that without lost of generality we can fix an order of the assignments in the strategy set.

8. GUS vs. DPLL

Now let us see that simulated DPLL and simulated GUS use the same number of UP steps to show that CC is unsatisfiable. We know that CC = $[[2^n]]$. We prove this by induction on the natural number k, $n \geq k > 0$. We know the strategy function of DPLL and GUS, so we know how do we break down $[[2^n]]$ to smaller problems. We know by the induction hypothesis how many UP steps are needed to solve the smaller problems. We count the UP steps used to break down the problem, minus the number of UP steps that are not performed because of the cache, and the UP steps needed to solve the smaller ones. It will turn out that for both algorithms this number is $2^n + 2^{n-1} - 2$.

Lemma 8.1. (Used UP steps in Simulated DPLL) Simulated DPLL uses $2^{k} + 2^{qmboxk-1} - 2$ UP steps to show that $[[2^{k}]]$, $n \geq k > 0$, is unsatisfiable.

Proof by Induction : We show that this is true for $k = 1$. We know that $[[2]]$ contains only unit clauses. It is all the same which clause do we select, the DPLL strategy gives back the selected clause as an assignment, because it is a unit. We propagate it and we obtain a clause set, which contains the empty clause. This means we know a unit and its negation, i.e., the set of known clauses is unsatisfiable. We used 1 UP step, since we propagated only a unit, to show that $[[2]]$ is unsatisfiable. We cannot show that $[[2]]$ is unsatisfiable using less then 1 UP step, since we may select only one clause in this framework.

Assume that for some $n \geq k > 1$ we can show that $[[2^{k-1}]]$ is unsatisfiable using $2^{k-1} + 22^{k-2} - 2$ UP steps. We show that $[[2^{k}]]$ is unsatisfiable using $2^{k} + 2^{k-1} - 2$ UP steps.

Let us consider $[[2^{k}]]$, $n \geq k > 1$. We know that $[[22^{k}]]$ contains only k-clauses, i.e., no unit clauses. It is all the same which clause do we select, the DPLL strategy gives back a literal from it as a unit and its negation. We propagate the unit and obtain $[[2^{k-1}]]$. We know by the assumption that $[[2^{k-1}]]$ can be solved using $2^{k-1} + 2^{k-2} - 2$ UP steps. Since $[[2^{k-1}]]$ is unsatisfiable we add the negation of the unit to known clauses. Now we propagate the negation of the unit and obtain $[[2^{k-1}]]$. We know by the assumption that $[[2^{k-1}]]$ can be solved using $2^{k-1}+2^{k-2}-2$ UP steps. Since $[[2^{k-1}]]$ is unsatisfiable we add the negation of the negation of the unit, i.e., the unit to known clauses. This means, the set of known clauses is unsatisfiable. We used altogether $2^{*}(2^{k} + 2^{k-2} - 2) + 2 = 2^{k} + 2^{k-1} - 2$ UP steps to show that $[[2^{k}]]$ is unsatisfiable.

Now we try to prove the same lemma for simulated GUS.

Lemma 8.2. (Used UP steps in Simulated GUS) Simulated GUS uses $2^{k} + 2^{k-1} - 2$ UP steps to show that $[[2^{k}]]$, $n \geq k > 0$, is unsatisfiable.

Proof by Induction: We know that $[[2]]$ contains only unit clauses. It is all the same which clause do we select, the GUS strategy gives back the selected clause as an assignment, because it is a unit. We propagate it and we obtain a clause set, which contains the empty clause. This means we know a unit and its negation, i.e., the set of known clauses is unsatisfiable. We used 1 UP step, since we propagated only a unit, to show that $[[2]]$ is unsatisfiable. We cannot show that $[[2]]$ is unsatisfiable using less then 1 UP step, since we may select only one clause in this framework.

Assume that for some $n \geq k > 1$ and for all $k > m > 1$ we can show that $[[2^{m}]]$ is unsatisfiable using $2^{m} + 2^{m-1} - 2$ UP steps. We show that $[[2^{k}]]$ is unsatisfiable using $2^{k} + 2^{k-1} - 2$ UP steps.

Let us consider $[[2^{k}]]$, $n \geq k > 2$. We know that $[[2^{k}]]$ contains only k-clauses. Without lost of generality assume we select {a1, a2,..., a(k-1), ak} from $[[2^{k}]]$. Then

GUS strategy gives back {{!a1, !a2,..., !a(k-1), ak}, {!a1, !a2,..., a(k-1)},...., {!a1, a2}, {a1}}. First we propagate {!a1, !a2,..., !a(k-1), ak} and obtain a clause set which contains the empty clause, i.e., we can add !{!a1, !a2,..., !a(k-1), ak} to set of know clauses. We propagate {!a1, !a2,..., !a(k-2), a(k-1)} and obtain [[2]]. Note, that from the cache we already know the result of propagating {!a1, !a2,..., !a(k-2)}, so we need only 1 UP step to propagate {!a1, !a2,..., !a(k-2), a(k-1)}. We know by the assumption that [[2]] can be solved using 1 UP step. Since [[2]] is unsatisfiable we add !{!a1, !a2,..., a(k-1)} to the set of known clauses. We do the same for the rest of the strategy set. The two last step are: We propagate {!a1, a2} and obtain $[[2^k]]$. Note, that from the cache we already know the result of propagating {!a1}, so we need only 1 UP step to propagate {!a1, a2}. We know by our assumption that $[[2^{k-2}]]$ can be solved using $2^{k-2}+2^{k-3}-2$ UP steps. Since $[[2^{k-2}]]$ is unsatisfiable we add !{!a1, a2} to the set of known clauses. We propagate {a1} and obtain $[[2^{k-1}]]$. We know by the assumption that $[[2^{k-1}]]$ can be solved using $2^{k-1}+2^{k-2}-2$ UP steps. Since $[[2^{k-1}]]$ is unsatisfiable we add !{a1} to the set of known clauses. This means, the set of known clauses is unsatisfiable. Without cache we would use k+(k-1)+...+1 UP steps to break the problem to smaller ones. But with cache we need for this purpose only k+(k-1) = 2*k-1 UP steps, because each assignment form the strategy set has the form negative literals and a positive literal. After propagating the first k long assignment one can find in the cache the result of propagating the negative literals, so (k-1) times we propagate only the one positive literal. We used altogether $2*k-1+2+1-2+4+2-2+...+2^{k-2}+2^{k-3}-2+2^{k-1}+2^{k-2}-2 = 2*k-1+(k-1)*(-2)+2+1+4+2+...+2^{k-2}+2^{k-3}+2^{k-1}+2^{k-2} = 1+2+1+4+2+...+2^{k-2}+2^{k-3}+2^{k-1}+2^{k-2} = 1+1+2+2+4+4+....+2^{k-3}+2^{k-3}+2^{k-2}+2^{k-2}+2^{k-1} = 2^k+2^{k-1}-2$ UP steps to show that $[[2^k]]$ is unsatisfiable.

Now we see that DPLL and GUS use the same number of UP steps on CC.

Lemma 8.3. Simulated DPLL and simulated GUS use the same number of UP steps to show that CC is unsatisfiable.

Proof: We know, by Lemma 8.1 and 8.2, that both of them uses 2^n+2^(n-1)-2 UP steps to show that $[[2^n]]$ is unsatisfiable. Hence, simulated DPLL and simulated GUS use the same number of UP steps to show that CC is unsatisfiable.

One might ask the question. Is there a better algorithm? No, there is not. This is what Limitation Lemma says. It is not easy to prove this. Namely we have only one property, that a negation of a strategy set together with its generator is unsatisfiable. We have to investigate the consequences of this property.

9. Limit for Children

The set of all clear clauses, CC, is our universe. We can convert any clause to a set of clear clauses. If C is a clause and T is the set of clear clauses subsumed by C,

then {C} and T are equivalent, furthermore T is well-defined. For example we can convert ++x to the set of $+++$ and $++-$. Since we can convert any clause (and clause set) to a set of clear clauses, i.e., to a subset of CC, we can understand CC as our universe. There is only one problem with this point of view. It is expensive. To find the equivalent clear clause set of a clause set may take exponentional time.

But it is easy to give upper limit for the number of subsumed clear clauses. We count the subsumed clear clauses for each clause in the set. This is an upper limit, because we count some clear clauses more than once, if they are subsumed by more clauses. Furthermore we know that any unsatisfiable clause set subsumes all the 2^{n} clear clauses. Hence, a strategy set subsumes at least 2^{n-1} clear clauses. From these we can prove an important lemma, the Limit for Children Lemma. This says that if we break down $[[2^{n}]]$ to smaller problems, called children, by a strategy set and the smaller problems are $[[2^{k1}]]$, $[[2^{k2}]]$,..., $[[2^{km}]]$, then $2^{k1} + 2^{k2} + ... + 2^{km} \geq (2^{k})1$. We will use this property in the proof of Limitation Lemma.

Definition 9.1. (Number of Subsumed k-Clauses) Let U be a set of variables.

(a) NokC(C, U) := $2^{|U|-|C|}$, if C is a clause or an assignment and $Var(C) \subseteq U$.

(b) NokC(S, U) := $|T|$, if S is a clause or assignment set and $Var(S) \subseteq U$ and $T \subseteq [[U]]$ and $T \equiv S$.

The motivation of this definition is that we know that a clause C subsumes a clause D if C is a subset of D. The question is how many k-clauses, with the same k variables, does C subsume. The answer is easy, 2^{k-l}, where l is the length of C. We know that from example ++x subsumes two 3-clauses: $+++$ and $++-$.

Now we give several lemmas about NokC. The most interesting one is Lower Limit for Subsumed k-Clauses by a Strategy Set Lemma. It states that a strategy set subsumes at least (2^{k})-1 k-clause if it is generated by a k-clause. This lemma is important, because it is a consequence of the definition of strategy set. We need this kind of lemmas to be able to prove the Limitation Lemma, because we know nothing else just that our framework breaks down a bigger problem to smaller ones by a strategy set.

Lemma 9.2. Let C be a clause or assignment. Then NokC(C, Var(C)) = NokC(!C, Var(C)). Let S be a clause or assignment set. Then NokC(S, Var(S)) = NokC(!S, Var(S)).

Proof: The number of subsumed k-clauses depends on the occurring variables, but not on the occurring literals.

Lemma 9.3. Let U be a set of variables. Let k be the number of variables in U. Then NokC([[U]], U) = 2^{k}.

Proof: U contains k variable. [[U]] contains all possible k-clauses on variables of U. Since a literal is either positive or negative, there are 2^{k} such k-clauses.

Lemma 9.4. Let S be an unsatisfiable clause set. Let k be the number of variables in S. Then NokC(S, Var(S)) = 2^k.

Proof: We know that any unsatisfiable clause set is equivalent. We know that [[U]] is unsatisfiable for any U. From these we know that S is unsatisfiable if S ≡ [[Var(S)]]. Hence, S is unsatisfiable if NokC(S, Var(S)) = 2^k.

Lemma 9.5. (Upper Limit for Subsumed k-Clauses) Let S be a clause set or an assignment set. Let k be the number of variables in S. Then Sum({NokC(C, Var(S)) | C ∈ S}) ≥ NokC(S, Var(S)).

Proof: Let B, C ∈ S. If there is no k-clause in [[Var(S)]], which is subsumed by B and C simultaneously, then it is easy to see that we have Sum({NokC(C, Var(S)) | C ∈ S}) = NokC(S, Var(S)). Otherwise, we have Sum({NokC(C, Var(S)) | C ∈ S}) > NokC(S, Var(S)), because we count some k-clause twice or more.

Lemma 9.6 (Lower Limit for Subsumed k-Clauses by a Strategy Set) Let S be clause set. Let C ∈ S. Let k be the number of variables in S. Let T be a strategy set generated by C. Then NokC(T, Var(S)) ≥ (2^k) − 1.

Proof: Since T is a strategy set generated by C, we know, by definition of strategy set, that !T ∪ {C} is unsatisfiable, i.e., !T ∪ {C} ≡ [[Var(S)]]. This means, we have NokC(!T, Var(S)) + NokC(C, Var(S)) ≥ NokC([[Var(S)]], Var(S)) = 2^k. Since NokC(C, Var(S)) ≥ 1, we have NokC(T, Var(S)) ≥ (2^k)-1.

This is an important lemma. We can use this lemma to show a bit more complicated lemma, the Limit for Children Lemma. It states that if we break down [[2^k]] to its "children" and the children are [[2^{k1}]], [[2^{k2}]],..., [[2^{km}]], then we have the following inequality: $2^{k1} + 2^{k2} + \ldots + 2^{km} \geq (2^k) - 1$. We need this property to prove the Limitation Lemma. First we give the definition of children, generated by a strategy set.

Definition 9.7. (Children Generated by a Strategy Set) *Let S be a clause set. Let C ∈ S. Let T be a strategy set generated by C. Then Children(S, T) := {HUP(S, A) | A ∈ T}.*

The following lemma describes the main idea needed to prove Limit for Children Lemma. This lemma states the following: if we propagate the l-length assignment A on [[2^k]], then we obtain [[2^m]], where m = k − l. And since A subsumes 2^m peaces of k-clauses it makes possible to give a lower limit for children.

Lemma 9.8. Let U be a set of variables. Let A be an assignment. Then NokC(HUP([[U]], A), U \ Var(A)) = NokC(A, U).

Proof: NokC(HUP([[U]], A), U \ Var(A)) = NokC([[U \ Var(A)]], U \ Var(A)) = $2^{|U|-|A|}$ = NokC(A, U).

Theorem 9.9. (Limit for Children) Let U be a set of variables. Let C ∈ [[U]]. Let T be a strategy set generated by C. Let k be the number of variables in U. Then Sum({NokC(C, Var(C)) | C ∈ Children([[U]], T)}) ≥ (2^k) − 1.

Proof: Sum({NokC(C, Var(C)) | C ∈ Children([[U]], T)}) = Sum({NokC(C, Var(C)) | C ∈ {HUP([[U]], A) | A ∈ T}}) = Sum({NokC(HUP([[U]], A), U \ Var(A)) | A ∈ T}) = Sum({NokC(A, U) | A ∈ T}) ≥ NokC(T, U) ≥ $(2^{k}) - 1$.

This is a very important property of strategy sets. If we break down $[[2^{k}]]$ to smaller problems by a strategy set and the smaller problems are $[[2^{k1}]]$, $[[2^{k2}]]$,..., $[[2^{km}]]$, then $2^{k1} + 2^{k2} + \ldots + 2^{km} \geq (2^{k}) - 1$. We will use this property in the proof of Limitation Lemma.

10. Assignment Trees

We need a data structure to represent a run of the unit propagation framework on a clause set. We use for this propose a tree. The tree contains the assignments to propagate, therefore, we call this data structure assignment tree, or for short, A-tree. A node of an A-tree contains an assignment to propagate and the set of its children nodes. The set containing the assignments of children nodes is a strategy set generated by a clause from the clause set, which we obtain by propagating the assignment of the parent node on the input clause set. Usually the root of an A-tree contains the empty assignment. It depends on the algorithm we simulate. The root contains the assignment from the second parameter of UPFw1SC, which is the empty assignment in case of simulated DPLL and GUS.

Definition 10.1. (A-tree) The ordered pair $\langle A, Cs \rangle$ is an A-tree if A is an assignment and Cs is a set of A-trees.

Example 10.2. The following A-tree represents a run of simulated GUS on [[4]], see Example 7.1: ⟨{}, {⟨{!a, b}, {}⟩, ⟨{a}, {⟨{b}, {}⟩}⟩}⟩.

We can see that the root of the tree contains the empty assignment, since simulated GUS calls the framework by the empty assignment as the second parameter. The root has two children, since on the first recursion level we propagate two assignments. The two children are these two assignments. The "first" child (first in the order of appearance) has no children, because if we propagate {!a, b} on [[4]] we obtain a clause set which contains the empty clause, i.e., we have found a conflict and have to backtrack. The "second" child has one child.

Example 10.3 The following A-tree represents a run of simulated GUS on [[8]], see Example 7.9: {⟨{!a, !b, c}, {}⟩, ⟨{!a, b}, {⟨{c}, {}⟩}⟩, ⟨{a}, {⟨{!b, c}, {}⟩, ⟨{b}, {⟨{c}, {}⟩}⟩}⟩}⟩}.

The following algorithm is a variant of UPFw1SC, which builds the A-tree, which represent the run of it. In the parameter list the word "OUT", like in ADA, means that this is an output parameter.
We use the following notation: If A denotes an ordered pair, then A#1 denotes the

first element and A#2 denotes the second element of it. From now on we may use this notation.

Algorithm 10.4. (Unit Propagation Framework with 1 Selection and Cache)

```
function UPFw1SC(I: clause set, A, B: assignment, STF : strategy function,
OUT AT : A-tree): assignment
begin                                          // B: assignment already
                                               propagated
   AT := ⟨A, {}⟩;                              // new row, each tree has at
                                               least 1 node
   if ({} ∈ I) then return {};                 // I : input clause set
   Let C be a minimal clause in I;             // selection
   K := {C};                                   // K : set of known clauses
   T := STF(C);                                // T is a strategy set generated
                                               by C
   for each A ∈ T do                           // We propagate each assignment
                                               from T.
     J := HUPwC(I, A, B);                      // propagate A on I
     if (J is empty) then return A;            // A is a model for I
     M := UPFw1SC(J, A, A ∪ B, STF, N);        // if J is sat, M is a model for J;
                                               // otherwise, M is empty
     AT#2 := AT#2 ∪ {N};                       // new row, we add the a new
                                               child to the children nodes
     if (M is not empty) then return A ∪ M;    // then A ∪ M is a model for I
     K := K ∪ {!A};                            // else J, i.e., HUP(I, A) is unsat.
                                               By expansion rule we know !A
   od
   return {};                                  // K = {C} ∪ !T, i.e., K is unsat,
                                               i.e., I is unsat
end
```

We did not give a new name to this algorithm, because it differs only in the parameter list and in two rows from the previous variant. Now we have 5 parameters. We have 2 new parameters: A and AT. In AT we generate the A-tree, which represent the run of this algorithm. To be able to do this we need also A, the assignment propagated on one recursion level above. The two new rows do not influence the run of the algorithm, but we need them to build the A-tree.

Now we have to rewrite the two algorithms, which simulate DPLL and GUS, respectively.

Algorithm 10.5. (Simulated DPLL)

```
function SimulatedDPLL(I: clause set, OUT AT : A-tree): assignment
begin
   return UPFw1SC(I, {}, {}, DPLL_STF, AT);
end
```

Algorithm 10.6. (Simulated GUS)

```
function SimulatedGUS(I: clause set, OUT AT : A-tree): assignment
begin
   return UPFw1SC(I, {}, {}, GUS_STF, AT);
end
```

We see that the second and third parameters are the same. It is because, on the first level there is nothing to propagate and nothing has been propagated. Both of them have a new OUT parameter. This is the A-tree, which is given back by the framework. This A-tree represents a run of the framework in the following meaning. We know that any SAT solver in this framework differs only in the strategy function we use. This means we can represent a run by storing the result of the strategy function, which is a strategy set. This is what we are doing.

The set of assignments of the children of the root is the first strategy set. The set of assignments of the children of one of the children of the root is the first strategy set on the second recursion level after propagating the assignment of this child of the root, etc.

Now we have to formally give the link between A-trees and strategy sets. The following algorithm does algorithmically what is described above.

Algorithm 10.7. (A-Tree to Assignment Set)

```
function AT2AS(AT : A-tree) : assignment set
begin
   S := AT#2;
   T := {};
   for each A ∈ S do
     T := T ∪ {A#1};
   od
   return T;
end
```

Lemma 10.8. (Correctness of AT2AS) Let S be an unsatisfiable clause set. Let AT be an A-tree, which represent a run of the Unit Propagation Framework with 1 Selection and Cache on S. Then for some C ∈ S we have that AT2AS(AT) is a strategy set generated by C.

Proof: We know that the framework select a clause from S. Let C be this clause. We show that AT2AS(AT) is a strategy set generated by C. We know that the framework creates an assignment set, which is a strategy set generated by C. Let T be this assignment set. We know that the framework constructs AT. First AT is ⟨B, {}⟩, where B is the second parameter of the framework. Then we touch each assignment from the strategy set. Let A be an assignment from T. We propagate it on the input clause, and we call recursively the framework with the second parameter A. This parameter then becomes the assignment part of the root of the A-tree given back by the framework. The framework collects these sub-trees in the second element of AT, i.e., add them to children nodes. Since S is unsatisfiable the framework touches all elements of T. Since AT2AS(AT) collects the assignment part of the children nodes of the root of AT, it gives back T. We know that T is

a strategy set generated by C. Hence, AT2AS(AT) is a strategy set generated by C.

Since we will represent a run of the framework by an A-tree, we shall find a shorter phrase instead of saying "which represent a run of the Unit Propagation Framework with 1 Selection and Cache on S".

Definition 10.9 (ATree) *Let S be a clause set. Then let ATree(S) be an A-tree, which represent a run of the Unit Propagation Framework with 1 Selection and Cache on S.*

In other words, ATree(S) := AT, where AT is given by the fifth parameter of UPF1SC(S, {}, {}, STF, AT), where STF is some strategy function. We can see that ATree(S) is not well-defined, but has an important property, represents a run of the framework. We need only this property, therefore, this definition is good for us.

Lemma 10.10. (Link Between A-Trees and Children) Let S be an unsatisfiable clause set. If AT = ATree(S) and T = AT2AS(AT), then {ATree(N) | N ∈ Children(S, T)} = AT#2.

Proof: Since S is unsatisfiable, we know that S != {}, hence, AT#2 != {}. We have already seen that AT2AS(AT) collects the assignment part of the children of the root of AT. If we propagate these assignments we obtain Children(S, T), since T is the set of these assignments. We know that our framework calls itself recursively for each element of Children(S, T), if S is unsatisfiable. Therefore, {ATree(N) | N ∈ Children(S, T)} = AT#2. Note that this equality means that both of them represent a set of runs of the framework on the same clause sets. Since we examine only this property, we can interchange them. Hence, {ATree(N) | N ∈ Children(S, T)} = AT#2.

Now we give algorithm to turn an A-tree to a cached unit A-tree. We do this, because the number of edges of a cached unit A-tree is equal to the number of used UP steps. First we give the definitions of unit A-tree and cached unit A-tree, than we show how to convert an A-tree into these data structures.

Definition 10.11. (Unit A-Tree) *(a) ⟨{}, Cs⟩ is a unit A-tree if Cs is a set of unit A-trees.*
(b) ⟨{a}, Cs⟩ is a unit A-tree if {a} is an assignment and Cs is a set of unit A-trees.

Definition 10.12. (Cached Unit A-Tree) *(a) ⟨{}, Cs⟩ is a cached unit A-tree if Cs is a set of cached unit A-trees and for all C, D ∈ Cs, C != D we have C#1 != D#1.*
(b) ⟨{a}, Cs⟩ is a unit A-tree if {a} is an assignment and Cs is a set of unit A-trees and for all C, D ∈ Cs, C != D we have C#1 != D#1.

Algorithm 10.13. (A-Tree to Unit A-Tree)

(a) AT2UAT(⟨{}, Cs⟩) := ⟨{}, {AT2UAT(C) | C ∈ Cs}⟩.

(b) AT2UAT(⟨{a}, Cs⟩) := ⟨{a}, {AT2UAT(C) | C ∈ Cs}⟩.

(c) AT2UAT(⟨{a1, a2,..., ak}, Cs⟩) := ⟨{a1}, AT2UAT(⟨{a2,..., ak}, Cs⟩)⟩.

This algorithm is given in function programming style. It follows the definition of the data structure A-tree. It has 3 variants. The (a) variant is used if the assignment is the empty one. The (b) variant is used if the assignment is unit. And the (c) variant is used to turn non-unit assignments to series of unit ones. All 3 variants are recursive. Eventually the recursion is halted by empty or unit assignment leaves.

Example 10.14.

AT2UAT(⟨{}, {}⟩) = ⟨{}, {}⟩.

AT2UAT(⟨{a1, a2}, {}⟩) = ⟨{a1}, {⟨{a2}, {}⟩}⟩.

AT2UAT({}, {⟨{!a, !b, c}, {}⟩, ⟨{!a, b}, {⟨{c}, {}⟩}⟩, ⟨{a}, {⟨{!b, c}, {}⟩, ⟨{b}, {⟨{c}, {}⟩}⟩}⟩}) = ⟨{}, {⟨{!a}, {⟨{!b}, {⟨{c}, {}⟩}⟩}⟩, ⟨{!a}, {⟨{b}, {⟨{c}, {}⟩}⟩}⟩, ⟨{a}, {⟨{!b}, {⟨{c}, {}⟩}⟩, ⟨{b}, {⟨{c}, {}⟩}⟩}⟩}.

The last example shows how to convert the A-tree, which represents a run of simulated GUS on [[8]], to unit A-tree. We show that AT2UAT is correct.

Lemma 10.15. (Correctness of AT2UAT) If T is an A-tree then AT2UAT(T) is a unit A-tree.

Proof: We know from the definition of A-tree that T is either (A, {}) or (A, Cs), where A is an assignment and Cs is a set of A-Trees. In either case, we break down A by the (c) variant of AT2UAT, till we can use the (a) or (b) variants, by taking a literal from it and forming a unit A-tree from the literal and calling AT2UAT recursively by the rest of A. For this we need finitely many steps, because assignments are finite set of literals. The (a) and (b) variants of AT2UAT follows the recursive definition of A-tree.

The main point of the next lemma is that if we have an A-tree, which represent a run of the framework on [[2^k]], and we turn this A-tree to an unit A-tree and we convert it back to a strategy set, then if use this strategy set then of course we obtain the same result but the intermediate steps are different. Namely we unfold each HUP step to UP steps. Hence, each child is [[2^(k-1)]].

Lemma 10.16. Let k be a natural number with $n \geq k > 0$. If UAT = AT2UAT(ATree([[2^k]])) and T = AT2AS(UAT), then for all N ∈ Children([[2^k]], T) we have N = [[2^{k-1}]].

Proof: Since UAT is a unit A-tree, we know that each element of T is a unit assignment. Let N ∈ Children([[2^k]], T). Then we know that N = HUP([[2^k]], A), where A ∈ T. Since A is a unit, we know that N = [[$2^{k - 1}$]].

Now we go one step further and convert unit A-trees to cached unit A-tree. We do this, because the number of edges of a cached unit A-tree equals to the number of UP steps used by a run of Unit Propagation Framework with 1 Selection and Cache, if we obtain this cached unit A-tree from an A-tree, which represent this run of the framework.

Definition 10.17. (Unit A-Tree to Cached Unit A-Tree)
(a) UAT2CUAT(⟨A, {H1, ⟨{a}, T⟩, H2, ⟨{a}, S⟩, H3}⟩) := UAT2CUAT(⟨A, {H1, ⟨{a}, T ∪ S⟩, H2, H3}⟩).
(b) UAT2CUAT(⟨A, Cs⟩) := ⟨A, {UAT2CUAT(C) | C ∈ Cs}⟩.

This algorithm is given in function programming style. It follows the definition of the data structure unit A-tree. It has 2 variants. The (a) variant is used to simulate the effect of using cache. If a node has two or more children whit the same unit assignment, then we make out of them one, which inherits the children of them. The (b) variant is used if the (a) variant cannot be used anymore on this level. The (b) variant follows the recursion in the unit A-tree data structure. All 2 variants are recursive. Eventually the leaves stop the recursion.

Example 10.18.
UAT2CUAT(⟨{}, {}⟩) = ⟨{}, {}⟩.
UAT2CUAT(⟨{}, {⟨{a}, {⟨{b}, {}⟩}⟩, ⟨{a}, {⟨{c}, {}⟩}⟩}⟩) = ⟨{}, {⟨{a}, {⟨{b}, {}⟩, ⟨{c}, {}⟩}⟩}⟩.
UAT2CUAT(⟨{}, {⟨{!a}, {⟨{!b}, {⟨{c}, {}⟩}⟩}⟩, ⟨{!a}, {⟨{b}, {⟨{c}, {}⟩}⟩}⟩, ⟨{a}, {⟨{!b}, {⟨{c}, {}⟩}⟩, ⟨{b}, {⟨{c}, {}⟩}⟩}⟩}⟩) = ⟨{}, {⟨{!a}, {⟨{!b}, {⟨{c}, {}⟩}⟩, ⟨{b}, {⟨{c}, {}⟩}⟩}⟩, ⟨{}, {⟨{a}, {⟨{!b}, {c}⟩}⟩, ⟨{}, {⟨{b}, {c}⟩}⟩}⟩}⟩.

The last example shows how to convert the unit A-tree, which represents a run of simulated GUS on [[8]], to cached unit A-tree. Now we prove that UAT2CUAT is correct.

Lemma 10.19 (Correctness of UAT2CUAT) If T is an unit A-tree then UAT2CUAT(T) is a cached unit A-tree.

Proof: We know from the definition of unit A-tree that T is either ({}, Cs) or ({a}, Cs), where {a} is a unit assignment and Cs is a set of unit A-Trees. In either case, we simulate the effect of using cache by the (a) variant of UAT2CUAT. The (b) variant of UAT2CUAT follows the recursive definition of unit A-tree.

11. Limitation Lemma

Our experience shows that if we have k > 1 then we have to break $[[2^k]]$ down to two or more smaller problems. We know that the cache mechanism reduce the number of children. But still there are at least two children.

Lemma 11.1 Let k be a natural number with $n \geq k > 1$. If CUAT = UAT2CUAT (AT2UAT(ATree($[[2^k]]$))) and T = AT2AS(CUAT), then |Children($[[2^k]]$, T)| ≥ 2.

Proof: Assume |Children([[2^k]], T)| = 1. This means that |T| = 1. Let A ∈ T. Then T = {A}. We know that A is a k-clause. We also know that T is a strategy set generated by some C ∈ [[2^k]]. Let C ∈ [[2^k]] such that T is a strategy set generated by C. We know that C is also a k-clause. We have k > 1. But this case {C} ∪ !T is not unsatisfiable, because we can construct an assignment, which is a model for {C} ∪ !T. Let a, b ∈ C such that a != b. Since C is a k-clause and k > 1 we can find such a and b. If a ∈ !A then let D := {a}, otherwise, let D := {!a, b}. D is a model for {C} ∪ !T, since T = {A}, and A is a k-clause on the same variables as C. Therefore, |Children([[2^k]], T)| != 1 and since [[2^k]] is not empty, we know that |Children([[2^k]], T)| > 0. Hence, |Children([[2^k]], T)| ≥ 2.

Now we can prove a very important lemma about the number of nodes of cached unit A-tree. Since the number of edges of a cached unit A-tree gives the number of used UP steps, and there is a well-known correspondence between the number of nodes and edges in a tree, this lemma will be a big step forward to Limitation Lemma.

Definition 11.2 (Number of Nodes of an A-tree)
(a) NoNs(⟨A, {}⟩) := 1.
(b) NoNs(⟨A, C⟩) := 1 + Sum({NoNs(B) | B ∈ C}).

Note that NoNs(ATree([[2^0]])) = 1.

Lemma 11.3 Let S be an unsatisfiable clause set. If AT = ATree(S) and T = AT2AS(AT), then NoNs(AT) = 1 + Sum({NoNs(ATree(N)) | N ∈ Children(S, T)}).

Proof: By Lemma Link Between A-Trees and Children we know that {ATree(N) | N ∈ Children(S, T)} = AT#2. Since S is unsatisfiable, we know that S != {}, hence, AT#2 != {}. Hence, by definition of number of nodes of an A-tree, NoNs(AT) = Sum({NoNs(ATree(N)) | N ∈ Children(S, T)}).

Lemma 11.4 (Number of Nodes in Cached Unit A-Trees) Let k be a natural number with n ≥ k > 0. Let A = UAT2CUAT(AT2UAT(ATree([[2^k]]))). Then NoNs(A) ≥ $2^k + 2^{k-1} - 1$.

Proof by Induction: For k = 1 it is trivial, since T has 2 nodes.
Assume that for k = n – 1 this lemma holds. We show that it holds also for k = n.

Let T be AT2AS(A). We know that Sum({NokC(N, Var(N)) | N ∈ Children([[2^n]], T)}) ≥ $(2^n) - 1$. Assume that Children([[2^n]], T) = {[[2^{k1}]], [[2^{k2}]],..., [[2^{km}]]}. Then we have $2^{k1} + 2^{k2} + \ldots + 2^{km} \geq (2^n) - 1$.

We know that in any cached unit A-tree a child of a node is ATree([[2^{k-1}]]) if the parent node is ATree([[2^k]]), and we know that each node has at least two children if k > 1. From this we obtain that ki = k – 1, m ≥ i ≥ 1, and m ≥ 2.

From this we obtain that NoNs(A) = 1 + Sum({NoNs(ATree(N)) | N ∈ Children([[2^n]], T)}) = 1 + NoNs(ATree([[2^{k1}]])) + NoNs(ATree([[2^{k2}]])) +... + NoNs(ATree([[2^{km}]])) = 1 + NoNs(ATree([[2^{k-1}]])) + NoNs(ATree([[2^{k-1}]])) +... + NoNs(ATree([[2^{k-1}]])) ≥ 1 + NoNs(ATree([[2^{k-1}]])) + NoNs(ATree([[2^{k-1}]])). From this by the induction hypothesis we obtain that NoNs(A) ≥ 1 + NoNs(ATree([[2^{k-1}]])) + NoNs(ATree([[2^{k-1}]])) ≥ $1 + 2^{k-1} + 2^{k-2} - 1 + 2^{k-1} + 2^{k-2} - 1 = 2^k + 2^{k-1} - 1$. Hence, NoNs(A)≥ $2^k + 2^{k-1} - 1$.

Now we are at the stage that we can prove the limitation lemma. We know that any tree, which has m nodes, has m-1 edges. Since we represent a run of the unit propagation framework with one selection and cache by the corresponding A-tree, meaning ATree([[2^k]]), and since the number of used UP steps is the number of edges in the corresponding cached unit A-tree, meaning UAT2CUAT (AT2UAT(ATree([[2^k]]))), from the lemma above it follows that any SAT solver in this framework uses at least $2^k + 2^{k-1} - 2$ UP steps to show that [[2^k]] is unsatisfiable.

Theorem 11.5. (Limitation Lemma) There is no SAT solver algorithm in the Unit Propagation Framework with 1 Selection and Cache, which can show that CC is unsatisfiable using fewer UP steps than DPLL or GUS.

Proof: We know that DPLL and also GUS uses $2^n + 2^{n-1} - 2$ UP steps to show that CC is unsatisfiable. We have to show that any SAT solver in the unit propagation framework with one selection and cache uses $2^n + 2^{n-1} - 2$ UP steps or more. Since any run of the framework can be represented by the corresponding A-tree, and the number of used UP steps is the number of edges in the corresponding cached unit A-tree, by Lemma Number of Nodes in Cached Unit A-Trees, any possible run of the framework on CC uses at least $2^n + 2^{n-1} - 2$ UP steps. Hence, there is no SAT solver algorithm in the unit propagation framework with one selection and cache, which can show that CC is unsatisfiable using fewer UP steps than DPLL or GUS.

Theorem 11.6. (SAT solvers are non-polynomial in UPFw1SC) SAT solver algorithms are non-polynomial in the Unit Propagation Framework with 1 Selection and Cache.

Proof: We know, by Limitation Lemma, There is no SAT solver algorithm in the Unit Propagation Framework with 1 Selection and Cache, which can show that CC is unsatisfiable using fewer UP steps than $2^n + 2^{n-1} - 2$. Hence, SAT solver algorithms are non-polynomial in the Unit Propagation Framework with 1 Selection and Cache.

With this proof we conclude this paper. We will discuss in a later paper, how can we break the Limitation Lemma. An idea could be to allow more then one selection. Another idea could be to analyze how do we construct the set of known

clauses. This analysis will show that if we pass the set of known clauses along the recursive calls of the framework, then we can save some UP steps. But these "tricks" require more careful design of the strategy functions and the framework.

References

[1] Cook, S.: The complexity of theorem proving procedures. In *Proc. 3rd Ann. ACM Symp. on Theory of Computing*, pages 151–158, New York, 1971.

[2] Gu, J., Purdom, P. W., Franco, J. and Wah, B. W.: Algorithms for the Satisfiability (SAT) Problem: A Survey. In *Discrete Mathematics and Theoretical Computer Science: Satisfiability (SAT)*. American Mathematical Society, 1997.

[3] Kusper, G.: Solving the SAT Problem by Hyper-Unit Propagation. In *RISC Technical Report 02-02*, 2002.

[4] Davis, M., Logemann, G. and Loveland, D.: A machine program for theorem proving. In *Communications of the ACM*, 5: pages 394—397, 1962.

[5] Kusper, G.: Solving the resolution-free SAT problem by hyper-unit propagation in linear time. *Proceedings of SAT2002*, 2002.

Postal address

Gábor Kusper
Department of Computer Science,
Eszterházy Károly College,
Leányka u. 6-8, H-3300 Eger,
Hungary