**CS2002 W09 Practical Report**

Logic: Unit Propagation

# 1 Overview

This practical asked for implementation of the unit propagation part of the Davis–Putnam–Logemann–Loveland (DPLL) algorithm for determining the satisfiability of formulae in conjunctive normal form (CNF). This involved searching for unit clauses and propagating their truth values through a given formula, and outputting either a sorted set of obtained unit clauses, or else signalling a contradiction.

I have attempted both a straightforward implementation that runs in polynomial time and made some efforts towards a solution that runs in linear time. In this report I provide an explanation of design, implementation and testing for both algorithms, as well as some analysis of the time complexities of each.

# 2 Design

As suggested in the specification, I began by focussing on input parsing and data structures, since these would be common to all algorithm implementations. I decided that the data types should be generalised and reusable as far as possible, and so followed an Abstract Data Type (ADT) scheme similar to the one I adopted for the “W07-C2” practical.

**2.1 Literal ADT**

The smallest data unit that unit propagation algorithms will need to work with is the logical literal. This encompasses two pieces of information: an identifier and a truth value. As in the “W07-C2” practical, I decided that in order to promote encapsulation, I would provide all ADTs via incomplete types, managing memory through constructors and destructors.

For a literal, its name and truth value should be supplied at the time of creation, and the name should remain fixed while the truth value can be modified by negation. Hence the following function headers were determined for operations on this ADT in the file “Literal.h”:

Literal new\_Literal(char\* name, bool truth);

void Literal\_free(Literal);

Literal Literal\_negate(Literal);

char\* Literal\_getName(Literal);

bool Literal\_isTrue(Literal);

Further functionality could be added when necessary, depending on the algorithm’s needs.

**2.2 Clause ADT**

Since CNF is assumed for a formula given in the input for this program, a clause will always be a disjunction of literals. Hence a complete description of a clause is given simply by a list of the literals contained within it. As advised in the specification, I decided to make no assumptions as to the number of literals contained within each clause, and so no maximum size would be defined in the ADT interface, or elsewhere. Obvious necessary functionality would be facility to add and remove literals, get the length (number of literals), and get or find the index of a particular literal, and so the following headers were declared in the file “clause.h”:

Clause new\_Clause();

void Clause\_free(Clause);

void Clause\_addLiteral(Clause this, Literal l);

void Clause\_removeLiteral(Clause this, int index);

int Clause\_getLength(Clause);

Literal Clause\_getLiteral(Clause this, int index);

int Clause\_findLiteral(Clause this, Literal l);

**2.3 Formula ADT**

Finally, in order to conveniently group together all of the clauses given in the input, I decided that an encompassing Formula ADT would be useful. This would mirror the structure of the Clause ADT, being essentially a list of clauses supporting adding and removal. The reason for including a Formula ADT would be so that, once again, no assumptions needed to be made as to the number of clauses in the input, allowing dynamic resizing of the container as necessary. So, in the file “formula.h” I would have:

Formula new\_Formula();

void Formula\_free(Formula);

void Formula\_addClause(Formula this, Clause c);

void Formula\_removeClause(Formula this, int index);

int Formula\_getLength(Formula);

Clause Formula\_getClause(Formula this, int index);

void Formula\_print(Formula);

**2.4 Input Parsing**

Beyond the ADTs needed for storing the information contained within the input, I decided another natural module would be one for input parsing itself. This would build a Formula structure by parsing the “stdin” stream, adding a new clause for each line. The only necessary public functionality here would be the following function, contained within “parser.h”, which returns a populated Formula based on the input:

Formula buildFormula();

At this design stage, I briefly outlined in pseudocode my strategy for file parsing.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | set line = “”  while (not end of file):  get next character  if character is newline:  clause = parse line  add clause to formula  set line = “”  else:  append character to line  end if  end while  return formula |

And for parsing of each line:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | while (not end of string):  get next token  if token != “-“:  truthValue = (token[0] != ‘-‘  if truthValue == false:  token = substring(token, 1, end)  end if  literal = new Literal(token, truthValue)  add literal to clause  end if  end while  return clause |
|  |  |

I added the check on line 4 to ensure that a hyphen on its own would not be treated as an empty token. Another problem that I could anticipate with the file parsing was that in C the line would have to be a fixed length char array, and hence there would be a danger of overflowing. I therefore modified my design to include the lines in bold, growing the line length as needed:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  15  16  17  18  19  20 | line = “”  **length = 0**  **capacity = 100**  while (not end of file):  **if length == capacity:**  **capacity \*= 2**  **grow line to capacity**  **end if**  get next character  if character is newline:  clause = parse line  add clause to formula  line = “”  **length = 0**  else:  append character to line  **length++**  end if  end while |
|  |  |

I decided to double the capacity of the line every time the length reaches the maximum, so as to keep the number of memory reallocations required low.

**2.5 Unit Propagation Algorithm**

I decided I would first design the algorithm without regard for time complexity, focussing on simplicity and obtaining a working solution. The idea would be to repeatedly search for a unit clause, propagate it, and keep track of any unit clauses that are thereby obtained, adding them to a set (not recording duplicates). For the propagation itself, I would follow the pseudocode outlined in the lecture notes.

# 3 Implementation

**3.1 ADT Implementation:**

I realised that, at the most basic level, both the Clause and Formula ADTs were essentially resizable lists, with Formulas being a list of Clauses, which are themselves lists of Literals. I could therefore represent both with a structure containing an array and corresponding length attribute. To enable resizing in C, I would have to make use of dynamic memory reallocation, in the functions that increase the size of the list. I decided to adopt a similar strategy to what I had already outlined for dynamically resizing line containers in the input parser. That is, I would set an initial capacity, stored in a third attribute, use the malloc function to assign the underlying array that many bytes of memory in the constructor, and then subsequently use realloc to double the capacity whenever the maximum is reached. This would be employed in the “addLiteral” and “addClause” functions.

The other aspects of the ADT constructors, getters and setters were straightforward. I was careful to use the strdup function when passing the string argument to the Literal constructor, so as not to create pointer dependencies between different parts of the program.

For Clause’s “findLiteral” function, I implemented a very simple linear search of the underlying array. This would obviously have implications for the overall time complexity of the program, but for my starting implementation I decided that improving on this was not yet a priority.

Having completed these implementations, I noted that there was a high degree of similarity between the “clause.c” and “formula.c” files. This is because, aside from one storing literals and one storing clauses, they were essentially the same data structure: a resizable. I did consider that it might be more elegant to design a more generic data structure that could be used for both, and therefore eliminate the duplication of, for example, the code for array resizing. However, I decided that since there would only ever be two of these collection structures - there is no scenario within the scope of this practical where an even higher-level container such as a collection of formulas would be needed – the small amount of repetition was permissible.

**3.2 Parser Implementation**

As in the pseudocode design, the “parser.c” file consisted of two functions: the public “buildFormula” and private “parseLine”. Since all of my data structures were dynamic, I could build them in one pass of the file, simply adding literals to clauses for each token encountered, and adding clauses to the main formula instance at each newline character.

To move through the file (which would be provided on the standard input stream) I used the C idiom while((c = getchar()) != EOF), which would continuously get characters until the end of file is reached. I stored a line buffer in a character array, growing this when necessary using realloc as outlined in the design. I was careful to add the string terminator character, ‘\0’, when the newline character is reached.

For the parsing of each line, I made use of the strtok function, getting one token at a time, separated by any number spaces. For each token, I would call the a constructor to create a corresponding Literal instance, and add it to a Clause instance that is returned at the end of the “parseLine” function. For validating tokens, strtok ensured that they would not contain spaces, but I had to decided how to deal with non-leading hyphens (since these are explicitly disallowed in the specification). I decided to use the strstr function to check whether hyphens are contained within a string, and simply ignore the token if so. This would allow otherwise correct input to be processed. I also had to cover the edge case of ignoring a single hyphen on its own, since this is meaningless under the literal form specification.

Later, after this implementation was complete, it was drawn to my attention that there would be an easier to way to implement this using the getline function, rather than getting each character one at a time. I decided at that stage (since testing was already complete) that it would be more sensible to stick with the original design and implementation even though it was more involved than it needed to be.

**3.3 Propagation Algorithm Implementation**

The desired output of the algorithm would be a collection of unit clauses obtained through propagation. I decided that I would use the Clause ADT to represent the collection of units, since there was no more information necessary to be stored for each unit clause than the single literal contained within it. Therefore, the Clause (collection of literals) structure would be more straightforward to work with than a collection of clauses that each contain one literal.

The functionality of this module would stem from the public “getUnits()” function, but I decided to factor out two helper functions: a function to propagate a single literal, as in the lecture notes’ pseudocode, and a function to add all units (not including duplicates) to the collection of unit clauses that will eventually be returned.

In the “getUnits()” function I was careful to consider both exit cases: either we obtain an empty formula, or a formula with no unit clauses. To find unit clauses to propagate, I again implemented a straightforward linear search, in keeping with my initial goal of valuing simplicity over efficiency.

The “addUnits()” function for keeping track of obtained units would mimic the effect of adding to a set data type, that is, appending to the collection only if the unit is not already present within it.

Then for the “propagate()” function, I implemented straightforward propagation of a single literal, that is, iterating through the formula, finding instances of a literal and its negation within clauses, and removing instances of the negation, and removing whole clauses that contain the literal itself. The “findLiteral()” linear search function within the Clause ADT was the primary means of accomplishing this.

One subtlety that I noted here was that, since I was removing elements from a collection while simultaneously iterating through it, I would have to be careful not to accidentally skip over a clause due to its index being decreased by one on removal of another clause. This could be easily dealt with by decrementing the for-loop counter whenever a whole clause is removed.

Finally, I added a function to use qsort within the Clause ADT to ensure lexicographical ordering. This necessitated a comparator function in the Literal ADT.

# 4 Testing

Before analysing the complexity of my basic solution, or moving to look at a linear time version, I conducted thorough testing of the solution obtained so far.

**4.1 ADT Unit Testing**

I decided to adopt a unit testing style to ensure the ADT interfaces are usable as expected. I decided that my unit tests would attempt to cover as many of the use cases of the interfaces as possible, rather than simply the way that they are used in the propagation algorithm. This would ensure that I had robust and reusable data structures for any future unit propagation algorithms I might implement.

As in the previous practical, I made use of unit testing pseudo-framework similar to the one given in the studres examples[[1]](#footnote-1). The tests are documented via multi-line comments in the C source files, but for completeness I will tabulate them here:

|  |  |
| --- | --- |
| **Literal Tests:** | |
| getName: | Check that the correct identifier is obtained from the literal. |
| longName: | Checks that a very long string can still be used as the name for a literal. |
| checkNameCopy: | Ensure that the name stored in the literal is a copy of the string passed and is unaffected by changes to the original. |
| getTruth: | Check that the correct truth value is obtained from the literal. |
| checkNegate: | Check that a literal can be properly negated. |
| checkEqual: | Check that two literals with the same attributes are considered to be equal. |
| checkNotEqual: | Check that two literals with the same name but different attributes are not considered to be equal. |
| checkCompare: | Check that the qsort comparator function compares on the name only |

|  |  |
| --- | --- |
| **Clause Tests:** |  |
| createEmptyClause | Checks that an empty clause has length 0. |
| addOneLiteral | Checks that a single literal can be added to a clause. |
| getOutOfBounds | Checks that indexing out of the clause bounds returns null. |
| removeOutOfBounds | Checks that removing a literal with an out of bounds index has no effect. |
| addTwoLiterals | Checks that two literals can be added to a clause. |
| addTenLiterals | Checks that a clause can be filled to initial capacity. |
| addElevenLiterals | Checks that a clause can be filled beyond its initial capacity. |
| addFiveThousandLiterals | Checks that a very large number of literals can be added to a clause. |
| addTwoLiteralsAndRemoveOne | Checks that removing the first literal shifts the second one back to position 0. |
| addTenLiteralsAndRemoveTwo | Checks that removing two literals results in two shifts back. |
| findLiteralInMiddle | Search the list for a literal somewhere in the middle. |
| findLiteralAtStart | Find a literal in position 0. |
| findLiteralAtEnd | Find a literal in the end position. |
| findMissing | Checks that a search for a literal that is not present in the clause yields -1. |
| checkSortLetters | Checks that a list of literals with single letter names are correctly sorted. |
| checkSortNumbers | Checks that a list of literals with single digit names are correctly sorted. |
| checkSortLonger | Checks that literals with longer, randomised names are correctly sorted. |

|  |  |
| --- | --- |
| **Formula Tests:** | |
| createEmptyFormula | Checks that an empty formula has length 0. |
| getOutOfBounds | Checks that trying to get a clause with an out of bounds index returns null. |
| removeOutOfBounds | Checks that removing a clause with an out of bounds index has no effect. |
| addOneClause | Checks that a single clause can be added to a formula. |
| addTwoClauses | Checks that two different clauses can be added to a formula. |
| addTenClauses | Checks that a formula can be filled to its initial capacity. |
| addElevenClauses | Checks that a formula's initial capcacity can be exceeded. |
| addFiveThousandClauses | Checks that a very large number of clauses can be added to the formula. |
| addAndRemoveClause | Checks that adding and removing a clause returns the formula to its initial state. |
| addTwoAndRemoveOneClause | Checks that adding two clauses and removing one leaves one remaining. |

**4.2 Further ‘Stacscheck’ Testing**

In order to test the actual input parsing and unit propagation parts of the program, as well as its running as a whole, I decided to write further tests that could be run using the school’s automated testing system. I contrived specific inputs to deal with each particular functionality, and I shall document the reasoning behind these here.

**Test 1 – Empty (Parsing)**

An empty input file contains no clauses at all, and so should result in a blank output.

**Test 2 – Whitespace (Parsing)**

A file that contains a blank line should be treated as having an empty clause, and hence signal a contradiction.

**Test 3 – Escaped (Parsing)**

Characters that are preceded by a backslash should be treated by the program as they are, and not as escape characters. Substituting the given “units1” test input for “\n”, “\t” and “\r” symbols should produce equivalent output.

**Test 4 – Many Literals (Parsing)**

An input file should be allowed to contain an arbitrarily large number of literals: 91 different ASCII characters were used as identifiers here.

**Test 5 – Many Clauses (Parsing)**

Similarly, an input file should be allowed to contain an arbitrarily large number of clauses (even if these are duplicates): 1501 clauses were demonstrated here.

**Test 6 – Extra Spaces (Parsing)**

Additional inline space characters should be ignored.

**Test 7 – Hyphens (Parsing)**

Literals that consist of a single hyphen ‘-’ or contain a hyphen anywhere other than at the start should be ignored. The contrived input here should therefore reduce to blank lines (contradiction).

**Test 8 – Hyphens (Parsing)**

Literal identifiers should be allowed to be arbitrarily long. Hence a single literal, 10001 characters long, as the input file should give output identical to the input (consists of one unit clause).

**Test 9 – Empty Line (Parsing)**

An empty line in an otherwise non-contradictory input should signal a contradiction.

**Test 10 – Lunch (Unit Propagation)**

A test for the unit propagation algorithm adapted from the “Did I go to Lunch?” example in the lecture slides. Expected output is as given in lectures.

**Test 11 – Online Example (Unit Propagation)**

A test for the unit propagation algorithm taken from “Example 1” given in a draft paper about unit propagation found online[[2]](#footnote-2).I noted that unit propagation here does not eliminate all the clauses, nevertheless the algorithm should successfully terminate when it finds there are no more units left to propagate.

**Test 12 – Whodunnit Example (Unit Propagation)**

A test for the unit propagation algorithm adapted from the “Who Is Guilty?” example in the lecture slides. Since this example was for full DPLL, I added a unit clause ‘k’ so that propagation could immediately take place.

**Test 13**, **14,** **15 – Worst Case Examples (Unit Propagation)**

Inputs contrived in increasing length so that the number of passes through the formula is maximised (worst case complexity). Literal names were taken from an online alphabetical word list[[3]](#footnote-3). The time taken for the medium and long inputs here was more significant than any other input, and I will discuss this further in my analysis of complexity.

**4.3 Testing Results**

Unit testing results:

A screenshot of a cell phone

Description automatically generated

“Stacscheck” testing results:

A close up of a newspaper

Description automatically generated

# 5 Analysis of Complexity

In this section I have undertaken an analysis of the time complexity of the implementation achieved thus far. I have considered each part of the of the program in isolation, and from this, aimed to bring together formulas for the worst-case time complexity in terms of *n,* the number of literals in the input.

It is important to note here that different clause groupings are possible for the same *n*. For example, if we have 9 literals, some possibilities are:

a

b

c

d

e

f

g

h

I **Ex. 3**

a b c

d e f

g h i

**Ex. 1**

a b c d e f g h i

**Ex. 2**

(Note that I have differentiated between the 9 literals here for clarity, but repeats are admissible). Ex. 2 then represents the worst case for the length of the formula, Ex. 3 representing the worst case for the length of a clause.

**5.1 Building Data Structures**

My algorithm for input parsing effectively contains two nested loops, with the “parseLine()” function being called for every line of the input, and various functions being called from there. In particular we have:

* strtok, strstr, strcmp: not dependent on *n* and so can be considered to be O(1)*.*
* “new\_Literal()”: O(1)
* “Clause\_addLiteral()”: O(1)
* “Clause\_getLength()”: O(1), since length is saved as an attribute of the ADT.

Then, outside the “parseline()” function we have further functions:

* realloc: again, not dependent on *n* and so can be considered to be O(1)*.*
* getchar():O(1)*.*
* “Formula\_addClause()”: O(1)

Regardless of different grouping of literals such as Ex. 1-3, on different lines, this set of constant time functions will be executed once for each, meaning that the nested loops are equivalent to one large loop through all of the literals in the input, and hence we have an O(*n*) linear time complexity for building the ADTs.

**5.2 Unit Propagation Complexity**

The loop structure of the unit propagation algorithm is more complex, owing to the fact that the outermost loop is a while construct, that has multiple exit conditions (no clauses, no unit clauses, or obtaining an empty clause). Observing that there can never be more unit clauses in total than there are clauses, and there can never be more clauses than literals (unless there is an empty clause, in which case we immediately terminate), an upper bound on the number of times this loop executes is *n*. In practice it will be some number *k*, the number of literals from the total that will be eventually obtained as unit clauses.

This outer loop effectively contains two nested loops: one of which searches for a unit clause, and the other that propagates it. The linear search clearly has no worse than O(*n*) complexity, and so can be set aside pending examination of the propagation itself.

Similar to the case for building data structures, the “propagate()” function is a traversal of the formula, which contains several traversals of clauses – via the “findLiteral()” and “removeLiteral()” functions that are equivalent to linear searches – and so this can be considered as a whole to be essentially a traversal of the whole collection.

More concretely, if the clauses have lengths *c1,…ck* so that = *n,* the the worst case for the number of comparison is *ci*, and since there are either 2 or 3 of these (depending on whether we are removing a clause), we have a constant multiple of = *n* comparisons overall. Note however that *n* will decrease with every iteration of the outermost loop, as unit clauses are removed, but it can be contrived that only one is removed per iteration for a worst case (see Tests 13-15).

Pulling this information together, it is clear that the worst case complexity is O(*n*2), since we will make a multiple of *n* comparisons on the first iteration, of (*n*-1) comparisons on the second, down to the number k, which in the worst case is a significant fraction of *n* (in the test, every clause becomes a unit clause, and there are two literals per clause so *k* » *n*/2).Hence the number of comparisons is triangular in essence, and is dominated by *n*2 for large *n*.

*n* + (*n* – 1) + (*n* – 2) + … + 2 + 1 = (*n*2 – *n*)/2

and *n*/2+ (*n*/2 – 1) + (*n*/2 – 2) + … + 2/2 + 1 = (n2/4– *n*/2)/2

so *n* + (*n* – 1) + (*n* – 2) + … + (n/2 + 1) + (n/2) = (*n*2 – *n*)/2 – (n2/4– *n*/2)/2

= (3*n*2)/8 – *n*/4

The search (linear in the number of clauses) to actually find the unit clauses happens at the same loop depth as the propagation and so does not increase the complexity order.

# 6 Linear Time Solution Efforts

After some research and thought, I decided to attempt another design for the problem that would be closer to a worst-case linear time complexity. Since the basic part of the practical was complete at this point. I duplicated all of the code and tests, and made my modifications within another directory: “linear”, as opposed to “basic”.

**6.1 Design Changes**

The essential inefficiency in the original design is that, once a new unit to be propagated has been identified, a full pass of all of the remaining literals (via iterating over the clauses and then through each) is required to determine where to make modifications. This caused by the fact that literal removal is linear in the length of each clause, clause removal is linear in the length of the formula, and “findLiteral” is linear in the length of each clause. I therefore outlined the following improvements to eliminate these inefficiencies.

* Make removal of literals constant by swapping with the end literal and decrementing length, rather than shifting everything down a place.
* Make removal of clauses constant by nullifying the clause (adding a method to set the struct pointer to NULL). This also allows an identified clause to be removed using only a reference to it, and not needing to consult the whole formula. Care would then be needed to be taken for skipping over NULL clauses.
* Eliminate the need for passing through all the literals in all the clauses to find clauses to modify, by storing in each Literal ADT a smaller Formula ADT that has references only to the clauses containing that literal[[4]](#footnote-4).

The propagation algorithm could then be as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | units = []  tempunits = []  newunits = []  add all unit clauses in formula to tempunits  while tempunits.length != 0  for each unit x in temp  for each Clause in x’s clause list  modify clause  if clause.length == 1  add clause to new units  add all tempunits to units  tempunits = newunits  return units |
|  |  |

This algorithm should run in linear time (further discussion later), since it does not need to make a pass of the whole formula for each literal it propagates. However, I anticipated a problem with building the mappings that it would rely upon. If each Literal ADT is to store references to all the clauses it appears in, then the literals within clauses should be unique. For example, if we have,

Clause X = a -b

Clause Y = a -c

Then the ‘a’ in clauses X and Y should be references to the same Literal instance, otherwise the mapping lists may be different (a1 containing only X and a2 containing only Y). Under my basic implementation these would be two technically different ADT instances, which would give true under the “equals()” function but be distinct in memory. Tackling this would have to be done at the parsing stage, when the structures are built.

One solution would be to create a set of all Literals as they are encountered, only creating a new ADT instance for a clause if it is not already present in the set. Unfortunately, in C, creating this set as a simple array (or repurposed “Clause”) of literals would result in a quadratic time complexity for parsing, since a traversal performing the “contains” functionality would be need for every literal encountered, and, as demonstrated above, the number of distinct literals could be *n* in the worst case, if every literal appears just once.

I decided to implement this solution anyway, since it would be theoretically possible to eliminate the quadratic complexity in practice by using a HashSet-like structure. Since the specification says that it is specifically the “exhaustive unit propagation” part of the functionality that should be linear, I decided that I could reasonably exclude the building of the structures from the complexity analysis.

**6.2 Implementation**

I decided to try to reuse as far as possible the code that was already written for this part of the practical. This meant using my Formula ADT whenever a list of clauses was required, and the Clause ADT whenever a list of literals (or units) was required. One issue with this approach that I quickly discovered was the problem of creating circular dependencies. When adding a Formula member to my literal structure, I completed a circle of pre-processor #include directives.

**A screenshot of a cell phone

Description automatically generated**

This was causing linking errors, but after some research I determined that a way to fix this was to add forward declarations of the Clause and Formula incomplete types in the Literal.h file, eliminating the need for including the headers.

Following this I had a direct reference in every literal to a formula referencing all clauses that contain that literal. I also decided to include a reference to a literal’s negation, so that the unit propagation algorithm could immediately find all clauses to modify given any Literal.

The next problem I encountered was related to freeing memory. My original implementation was a tree like structure, and so freeing memory could be dealt with simply by freeing all the substructures, and then the structure itself. The circular nature of the new implementation made this model not possible. Furthermore, since under this implementation each literal is its own unique instance, more care would need to be taken to ensure that pointers were not being freed multiple times.

I decided to make public from the “parser.h” the set of literals built while creating the mappings, so that this set could be iterated through for freeing. I then modified the clause ADTs so that they do not recursively free literals, and supplied an extra method to the formula ADT to free itself without freeing its substructures.

With the ADT framework re-implemented I then created the new unit propagation algorithm according to my design. For clause removal, I made use of setting NULL flags, rather than deleting from a formula to prevent having to traverse the formula to find clauses. For literal removal I implemented the swap method outlined above.

**Testing**

I ensured that the new implementation passed all the automated tests that the previous implementation passed.

**6.3 Complexity Analysis**

Setting aside the complexity for parsing, which is not linear as discussed in the design, I provide here a brief justification as to why I believe this new algorithm is more efficient than the previous. I have ultimately determined that the algorithm is not certainly linear for the way I have implemented it.

As before, there is an outer while loop that will run as long as there are new units to propagate, and as before the upper bound on this will be *n*, with the worst case being certainly dependent on *n*. Inside however, we have traversals of subsets of the whole formula, as opposed complete traversals as before. For example, in the previous worst case example, each literal is contained in exactly one clause of length 2, other than the final unit. Hence to obtain each new unit, we need only consider a constant number of literals, which does not grow as *n* does, and we have an O(*n*) complexity.

Circumstances may change however, if we try to maximise the number of clauses each literal is contained in, as well as the length of the clauses, since the best (worst?) we might achieve is the following:

-a

-b a

-c b a

-d c b a

[…]

This is worked out by considering that we must have at least one unit in the formula to continue at all times, and that the unit can reduce the lengths of clauses by at most 1, hence we have at best (worst?) a stepwise increase in the lengths of clauses, if any increase at all is to take place.

In this case we are again considering decreasing triangular numbers, but summing them up to the length of the longest clause. The length of this longest clause will be in the order of *n*1/2, and the sum of the first *k* triangular numbers is:

(*k*(*k* + 1)(*k* + 2))/6. Hence we end up with an O(*n*3/2) complexity, by substituting the *k* s for *n*1/2s.

# 7 Evaluation and Conclusion

I found the basic part of this practical fairly straightforward, but getting beyond a basic solution was very challenging. Ultimately, I have implemented an algorithm that demonstrates the principles of what a truly linear solution might look like, but I have not convinced myself beyond doubt that my particular implementation is linear in the worst cases.

Using more complex data structures than arrays would be one path to improving the unit propagation further, such as implementing HashMaps or HashSets: an involved task using the C language. This would ideally eliminate the linear searches of clauses that affects the worst-case complexity.

Looking further I could consider other parts of the DPLL algorithm and how they might be implemented, trying to understand why to date there are no algorithms that can determine satisfiability of CNF formulae in any better than worst-case exponential time complexity.

1. [https:studres.cs.st-andrews.ac.ukCS2002ExamplesC\_SPL10IntegerListTestList.c](https://studres.cs.st-andrews.ac.uk/CS2002/Examples/C_SP/L10/IntegerList/TestList.c) [↑](#footnote-ref-1)
2. <https://www2.cs.sfu.ca/~mitchell/cmpt-827/2019-Fall/Notes/417_Notes_5_Linear_Time_SAT.pdf> [↑](#footnote-ref-2)
3. <https://github.com/openethereum/wordlist/blob/master/res/wordlist.txt> [↑](#footnote-ref-3)
4. As mentioned here: <https://en.wikipedia.org/wiki/Unit_propagation> [↑](#footnote-ref-4)