**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 an 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.

**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.

Finally the for “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.

# 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 strudres examples[[1]](#footnote-1).

**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 |

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)