Informatics 2D. Coursework 1:
Propositional Inference and Satisfiability

20th February 2017

1 Introduction

The objective of this assignment is to help you understand inference procedures using Propositional Logic. You will implement, use and evaluate the inference problems and their algorithms using Haskell.

You should download the following file from the Inf2D coursework page:

http://www.inf.ed.ac.uk/teaching/courses/inf2d/coursework/Inf2dAssignment1.tar.gz

Use the command tar xvf Inf2dAssignment1.tar.gz to extract the files contained within. This will create a directory named Inf2dAssignment1/ containing all files provided for the assignment.

You will submit a version of the Inf2d.hs file containing your implemented functions and a small report (report.pdf). Remember to add your matriculation number at the top of the Inf2d.hs file and in the report.

The deadline for the assignment is:
Thursday, 9th of March 2017 at 3 pm.

Submission details can be found in the last Section.

If your Haskell is quite rusty, you should revise the following topics:

- Recursion
- Currying
- Higher-order functions
- List Processing functions such as map, filter, foldl, sortBy, etc.
- The Maybe monad

The following webpage has lots of useful information if you need to revise Haskell. There are also links to descriptions of the Haskell libraries that you might find useful:

Haskell resources: http://www.haskell.org/haskellwiki/Haskell

In particular, to read this assignment sheet, you should recall the Haskell type-definition syntax. For example, a function foo which takes an argument of type Int and an argument of type String, and returns an argument of type Int, has the type declaration foo :: Int → String → Int. Most of the functions you will write have their corresponding type-definitions already given.
2 Important Information

There are the following sources of help:

- Attend lab sessions
- Email the course lecturers
- Read all emails sent to the Inf2D mailing list regarding the assignment.

Also ...

- Comment your Haskell code.
- Make sure your code compiles successfully before submitting.
- Your code will be tested using the libraries and supporting files provided in the assignment folder.
- Don’t change the names or type definitions of the functions you are asked to implement.

3 Getting Started

In this assignment, you will be implementing some inference procedures for propositional logic. Specifically, you will be implementing the WalkSAT algorithm for deciding propositional entailment, as well as the DPLL (Davis-Putnam-Logemann-Loveland) algorithm for checking satisfiability of a sentence in propositional logic. This assignment will guide you through the implementation by providing you function signatures which you must complete for your inference programs to work.

You can test your implementation by using the support code provided in the assignment folder.

4 Preliminaries

In this assignment, we will focus on propositional inference for sentences in CNF form. A set of datatypes and functions can be found in the Prop.hs file.

The following datatypes are provided for you to use:

1. **Variable**: The Variable is the datatype used to describe “atomic” sentences or propositions. These are basically represented as a string.

2. **Symbol**: Symbols correspond to either variables (for example $P$) or negations of variables (for example $\neg P$). In our implementation each symbol is represented by a custom datatype, and functions such as `variableOf` and `polarity` have been provided to access the information about a symbol.

3. **Clause**: A Clause is a disjunction of symbols, for example $P \lor Q \lor R \lor \neg S$. In our implementation this is represented as a list of symbols.

4. **Sentence**: A Sentence is a conjunction of clauses, for example $(P \lor Q) \land (R \lor P \lor \neg Q) \land (\neg P \lor \neg R)$. This is the CNF form of a propositional sentence. In our implementation this is represented as a list of Clauses, so it is a list of lists of symbols.

5. **Assignment**: The Assignment custom datatype corresponds to an assignment of a truth value (True or False) to a particular Variable. The functions `assignedVariable` and `assignedValue` can be used to access the details of an assignment.
6. **Model**: A (partial) Model is a (partial) assignment of truth values to the Variables in a Sentence. In our implementation this is a list of Assignments. Among the functions already provided for you is the function assign that allows us to (re)assign a truth value to an Variable in a particular Model, and the function valueOf that returns the value assigned to a given Variable (if any).

7. **Node**: The Node is the datatype describing a node in the search space of DPLL. It consists of the Sentence being investigated, a list of unassigned Variables and the Model containing the current variable assignments.

A sentence is satisfiable if there is at least one assignment of truth values (Model) to the variables of the sentence such that the whole sentence is true.

In Haskell, we will represent symbols as Strings. So the symbol $P$ is represented as "P" and the negation symbol $\neg P$ is represented as "-P".

Clauses (disjunction of symbols) will be represented as a list of symbols. For example, the clause $P \lor Q$ is represented as ["P","Q"], and $A \lor \neg B \lor C$ is represented as ["A","-B","C"].

Conjunctions are represented as lists of clauses. So we express $A \land B$ as [["A"],["B"]].

Putting all these together, we can represent the sentence $(P \lor Q) \land (\neg P \lor R)$ in Haskell as

```
["P","Q"], ["-P","R"]
```

Since every sentence of propositional logic is logically equivalent to a conjunction of clauses, the CNF representation in Haskell should be adequate for the needs of this assignment.

Models represent assignments to symbols. A model in the Haskell implementations for this assignment is represented as a list of tuples of Strings and Booleans. For example, in a domain with symbols $A$, $B$ and $C$, one model of the world where $A$ is assigned True, $B$ is assigned False and $C$ is True is represented as

```
[("A",True),("B",False),("C",True)]
```

5 **Task 1: General Helper Functions (10 Marks)**

In this part of the assignment, you are asked to create helper functions that will be used in several other methods. A set of datatypes and functions is already provided in the Prop.hs file.

- **satisfiesSymbol :: Model → Symbol → Bool**
  This function returns True if the given Symbol is satisfied by the Model, i.e. if the Model contains an assigned value for the Variable of the Symbol, such that the Symbol becomes True. [2 marks]

- **satisfiesClause :: Model → Clause → Bool**
  This function returns True if the given Clause is satisfied by the Model, i.e. if any of the Symbols of the Clause is satisfied by the Model. [2 marks]

- **satisfiesSentence :: Model → Sentence → Bool**
  This function returns True if the given Sentence is satisfied by the Model, i.e. if all of the Clauses of the Sentence are satisfied by the Model. [2 marks]

- **falsifiesSymbol :: Model → Symbol → Bool**
  This function returns True if the given Literal is falsified by the Model, i.e. if the Model contains an assigned value for the Variable of the Symbol, such that the Symbol becomes False. [2 marks]

- **falsifiesClause :: Model → Clause → Bool**
  This function returns True if the given Clause is falsified by the Model, i.e. if all of the Symbols of the Clause are falsified by the Model. [1 mark]

- **falsifiesSentence :: Model → Sentence → Bool**
  This function returns True if the given Sentence is falsified by the Model, i.e. if any of the Clauses of the Sentence is falsified by the Model. [1 mark]
6 Task 2: WalkSAT (20 marks)

The WalkSAT algorithm takes a sentence, a probability \( 0 \leq p \leq 1 \), and a boundary of maximum flips \( \text{maxflips} \) and returns a model that satisfies the sentence or failure. The algorithm begins by assigning truth values to the variables randomly, i.e. generating a random model. Then it proceeds to flip the value of an variable from one of the unsatisfied clauses until it is able to find a model that satisfies the sentence or reaches the maximum number of flips.

Figure 1: WalkSAT algorithm for checking satisfiability of a sentence in propositional logic

In order to select which variable from the currently selected clause to flip, it follows two strategies:

1. Either flip any variable randomly.
2. Or flip the variable that maximizes the number of satisfied clauses in the sentence.

The choice to flip randomly is followed with probability \( p \).

A part of the algorithm has already been implemented for you. For your assignment you are required to implement the following functions that complete the WalkSAT algorithm:

1. \textbf{unsatClauses :: Model \to Sentence \to [Clause]} 
   This function returns the list of Clauses in the Sentence that are not (yet) satisfied by the Model.

2. \textbf{maxSatClauses :: Model \to Sentence \to [Variable] \to Variable} 
   This function returns an Variable from the list of Variables. If the value of the Variable in the Model is flipped (using the already provided function \text{flipSymbol}), then the maximum number of Clauses in the Sentence will be satisfied, compared to the number of Clauses satisfied by flipping the values of any of the other Variables in the list. For this function you may assume the list of variables given as an argument always has at least one element.

If you implement these functions as well as the functions from Part 3 correctly, you will be able to use the WalkSAT algorithm fully. If you are using the Haskell toplevel (GHCi, see also Section 9), you can use the following function, which has been provided for you, to apply the algorithm:

\textbf{WalkSAT :: Sentence \to Float \to Int \to IO (Maybe (Model,Int))} 
This function runs WalkSAT for the given sentence, probability (given as a Float), and \text{maxflips} (given as an Int). The returning value is a \texttt{Monad} whose description goes beyond the scope of this assignment. The result of the algorithm is printed out by the function. It is either \texttt{Nothing} (in the case of failure) or \texttt{Just} the model that was found and the remaining number of flips.
Figure 2: DPLL algorithm for checking satisfiability of a sentence in propositional logic

7 Task 3: DPLL (30 Marks)

The DPLL (Davis-Putnam-Logemann-Loveland) algorithm is a more efficient algorithm than truth-table entailment for checking satisfiability (i.e. entailment). It uses clauses in CNF representation as described in section 4. In this section, you will implement the DPLL algorithm (figure 2). You should refer to R&N Chapter 7 Section 6.1 or NIE Chapter 6 (Logical Agents) Section 6.1.

7.1 Tautology Deletion (5 Marks)

- removeTautologies :: Sentence → Sentence
  This function outputs a simplified sentence if tautologies can be found in one or more clauses in the input.

7.2 Pure Symbol Heuristic (5 Marks)

- findPureSymbol :: [Variable] → Sentence → Model → Maybe (Variable, Bool)
  A pure Symbol is a symbol that always appears with the same “sign” in all clauses. This function can also ignore clauses which are already known to be true in the model. The function takes a list of symbols, a list of clauses and a model as inputs. It returns Just a tuple of a symbol and the truth value to assign to that symbol. If no pure symbol is found, it should return Nothing.

7.3 Unit Clause Heuristic (5 Marks)

- findUnitClause :: Sentence → Model → Maybe (Variable, Bool)
  A unit clause is a clause with just one symbol. It is also a clause in which all symbols but one are already assigned false by the model. The findUnitClause function takes a list of clauses and a model as inputs. It returns Just a tuple of a symbol and the truth value to assign to that symbol. If no unit clause is found, it should return Nothing.

7.4 Early Termination (5 Marks)

- earlyTerminate :: Sentence → Model → Bool
  The early termination function checks if a sentence is true or false even with a partially completed
model. For example, given a sentence \([\{P, \neg Q\}, \{P, \neg R\}]\) and the model \([\{P\}, True]\),
the earlyTerminate function should return true since the partially completed model (without
assignments for \(Q\) and \(R\)) still results in the sentence evaluating to true.

### 7.5 DPLL Satisfiability (10 Marks)

- **dpll :: [Clause] \rightarrow [Symbol] \rightarrow Bool**
The `dpll` function checks the satisfiability of a sentence in propositional logic. It takes as input a
list of clauses in CNF and a list of symbols for the domain. It returns true if there is a model which
satisfies the propositional sentence. Otherwise it returns false. You can use the earlyTerminate,
findPureSymbol and findUnitClause functions to complete this function. Also note that DPLL
operates over partial models, and so it is important to implement it using a backtracking search
algorithm as shown in figure 2.

- **dpllSatisfiable :: [Clause] \rightarrow Bool**
  This function serves as the entry point to the `dpll` function. It takes a list clauses in CNF as input
  and returns true or false. It uses the `dpll` function above to determine the satisfiability of its
  input sentence.

### 8 Task 4: Improving the efficiency of DPLL (15 Marks)

In this section you should implement an improved variable selection heuristic, i.e. a better choice function
for `dpll`. Partial credit will be given for any relevant attempt at a solution for this part.

You should implement the following function:

- **variableSelectionHeuristic :: Node \rightarrow Variable**
  Your implemented function will be given as an argument to `dpll` instead of the firstVariable
  function used above. You should try to implement a heuristic which, on average, improves DPLL’s
  speed and reduces the number of visited branches in the search space. Extra credit will be given
  for any non-obvious heuristics meeting the stated improvement criteria.

- **Note that the function dpllSatisfiablev2 that executes your implemented dpll algorithm using
  your improved variable selection heuristic has already been provided for you. The means to evaluate
  your heuristic method are provided in the following section.**

### 9 Task 5: Evaluation (15 Marks)

To complete your assignment, you need to execute your implemented code and evaluate it on some
non-trivial SAT problems. Moreover, you should write a report about the implemented algorithms.

For this part of the assignment you must evaluate the two implemented algorithms, WalkSAT and DPLL,
compare and discuss the results.

Within the assignment tarball, a Makefile and three files containing Main functions are provided. Using
these you can produce executable files of your code in order to evaluate your algorithms. In particular:

- **make walksat**
  creates the walksat executable that can be used to evaluate the WalkSAT algorithm.

- **make dpll**
  creates the dpll executable that can be used to evaluate DPLL.

- **make dpllv2**
  creates the dpllv2 executable that can be used to evaluate DPLL with your implemented variable
  selection heuristic (DPLLv2).
Moreover, the two folders named **Sat-test** and **Sat-dev-test** respectively are provided with sample SAT problems for evaluation. Each `.cnf` file within contains information for a single CNF sentence. You can use the aforementioned executables with any of those files as arguments. For example:

- `./dpll Sat-test/Sat01.cnf` will give you results for DPLL on the first SAT problem provided.
- `./dpllv2 Sat-test/Sat01.cnf` will give you results for DPLL with the added heuristic on the first SAT problem provided.

Note that the `walksat` executable requires 3 parameters: the file containing the problem, the probability $p$ and the `maxflips` parameter. For example:

- `./walksat Sat-test/Sat02.cnf 0.5 100` will give you results for the second SAT problem using WalkSAT with probability 0.5 and maxflips 100.

The Sat-dev-test folder contains 3 satisfiable (S1, S2, S3) and 3 unsatisfiable (U1, U2, U3) problems that are provided for your convenience to allow you to test your implementation.

The Sat-test folder contains 10 SAT problems that you should use in your evaluation. You should fill in the tables of values at the end of the Assignment.hs file for these problems.

- The first table involves the evaluation of WalkSAT with different parameters. More specifically,
  - You should set `maxflips` to 600 and the probability to 0, 0.5 and 1. You are, however, strongly encouraged to try more combinations and experiment further.
  - You should run the algorithm at least 5 times for each SAT problem and record the worst (i.e. the highest number of flips) successful result (unless all runs return a failure in which case simply note “Fail” on the table). The second table involves the comparison between DPLL and DPLL with your implemented heuristic (DPLLv2). You should record the number of visited branches in the search space as reported in each case by the two DPLL algorithms.

10 Task 6: Report (10 Marks)

You should write a brief report, as a comment in your code (in the section provided at the end of the file), which discusses the results and your experiences in implementing the algorithms. Your discussion should include details about the following:

1. WalkSAT and DPLL (6 Marks)
   (a) The results you obtained. Explain briefly what they mean and whether they were expected or not.
   (b) How do the different values in the parameters of WalkSAT affect its performance? What do you think are the ideal values of these parameters for the given set of problems, if any, and why?
      **Bonus marks** (up to 10) can be earned for an evaluation similar to the two figures on the slides “Hard satisfiability problems” in Lecture 07.
   (c) A brief comparison of WalkSAT and DPLL. What are their main differences?

2. Improving the efficiency of DPLL (4 Marks)
   (a) Where you got your idea for the heuristic, e.g. intuition or some source such as a textbook, research paper or a website. Please clearly specify any external source you use.
   (b) How you implemented your heuristic.
(c) Why is your heuristic expected to improve DPLL’s performance (either in theory or based on your intuition, whichever you used)? Was there an improvement in the speed with which a solution was found for the provided SAT problems?

(d) Apart from the implemented tautology removal and the component analysis mentioned in R&N p.261, can you suggest another step that can be used in the simplification stage of DPLL in order to simplify the sentence and improve DPLL’s efficiency? Briefly explain why your suggestion should achieve the desired result.

The length of your report should not exceed two A4 pages of printed plain text.

11 Notes

Please note the following:

- To ensure maximum credit, make sure you have commented your code and that it can be properly compiled before submitting.

- All the algorithms are expected to terminate within 1 minute or less in this assignment. If any of your algorithms take more than 1 minute to terminate, you should make a comment in your code with the name of the algorithm, and how long it takes on average. You may lose marks if your algorithms do not terminate in under 2 minutes of runtime for any of the problems.

- You are free to implement any number of custom functions to help you in your implementation. However, you must comment these functions explaining their functionality and why you needed them.

- Do not change the names or type definitions of the functions you are asked to implement. Moreover, you may not alter any part of the code given in the rest of the files for the assignment.

- You are strongly encouraged to create your own unit tests to test your individual functions (these do not need to be included in your submitted file).

- Ensure your algorithms follow the pseudocode provided in the books and lectures. Implementations with increased complexity may not be awarded maximum credit.

12 Good Scholarly Practice

Please remember the University requirement as regards all assessed work for credit. Details about this can be found at:

http://www.ed.ac.uk/academic-services/students/undergraduate/discipline/academic-misconduct

and at:

http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct

Furthermore, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work on a public repository then you must set access permissions appropriately (generally permitting access only to yourself, or your group in the case of group practicals).

13 Submission

You should submit your copy of the file Inf2d.hs that contains your implemented functions using the following command:

submit inf2d 1 Inf2d.hs report.pdf
Your file must be submitted by 3 pm on the 9th of March 2016.