Inf2D 09: CW1 (12.5%)

Michael Herrmann

University of Edinburgh, School of Informatics

02/02/2017

Credits: The content of this lecture was prepared by Michael Rovatsos and follows R&N
Overview

- Constraint Satisfaction Problems (CSP)
- WalkSAT
- DPLL
- A bit of Haskell
Aim

- Understand inference and satisfiability problems in Propositional Logic
- Familiarize yourself with some algorithms for solving inference/SAT problems
- Implement satisfiability algorithms using Haskell
Datatypes (t.b.r.)

- **Atom**: a.k.a. “variable”, e.g. $P$.
- **Literal**: a.k.a. “symbol”: either atoms (for example $P$) or negations of atoms (for example $\neg P$).
- **Clause**: A Clause is a disjunction of literals, for example $P \lor Q \lor R \lor \neg S$.
- **Formula**: a.k.a. “sentence”: conjunction of clauses, for example $(P \lor Q) \land (R \lor P \lor \neg Q) \land (\neg P \lor \neg R)$.
- **Assignment**: assignment of a truth value (True or False) to a particular Atom.
- **Model**: A (partial) Model is a (partial) assignment of truth values to the Atoms in a Formula (a list of Assignments).
- **Node**: The Node is the datatype describing a node in the search space
  - Formula
  - list of unassigned Atoms
  - Model containing the current variable assignments.
Coursework Overview

- Task 1: General Helper Functions (10 marks)
- Task 2: WalkSAT (20 marks)
- Task 3: DPLL (30 Marks)
- Task 4: Improving the efficiency of DPLL (15 Marks)
- Task 5: Evaluation (15 Marks)
- Task 6: Report (10 Marks)
Functions defined in this task will be useful in the remaining tasks.

- `lookupAssignment::Symbol->Model->Maybe Bool`
- `negateSymbol::Symbol->Symbol`
- `isNegated::Symbol->Bool`
- `getUnsignedSymbol::Symbol->Symbol`
- `getSymbols::[Sentence]->[Symbol]`
Task 2: WalkSAT (20 marks)

- Incomplete, local search algorithm
  - Evaluation function: The min-conflict heuristic of minimizing the number of unsatisfied clauses
  - Balance between greediness and randomness
The WalkSAT algorithm

```
function WALKSAT(clauses, p, max-flips) returns a satisfying model or failure
    inputs: clauses, a set of clauses in propositional logic
            p, the probability of choosing to do a “random walk” move
            max-flips, number of flips allowed before giving up

    model ← a random assignment of true/false to the symbols in clauses
    for i = 1 to max-flips do
        if model satisfies clauses then return model
        clause ← a randomly selected clause from clauses that is false in model
        with probability p flip the value in model of a randomly selected symbol from clause
        else flip whichever symbol in clause maximizes the number of satisfied clauses
    return failure
```

Algorithm checks for satisfiability by randomly flipping the values of variables
Determine if an input propositional logic sentence (in CNF) is satisfiable.

Improvements over truth table enumeration:

1. Early termination
2. Pure symbol heuristic
3. Unit clause heuristic
A clause is true if one of its literals is true,
- e.g. if $A$ is true then $(A \lor \neg B)$ is true.

A sentence is false if any of its clauses is false,
- e.g. if $A$ is false and $B$ is true then $(A \lor \neg B)$ is false, so sentence containing it is false.
Pure symbol heuristic

- Pure symbol: always appears with the same “sign” or polarity in all clauses.
  - e.g., in the three clauses \((A \lor \neg B), (\neg B \lor \neg C), (C \lor A)\)  
    \(A\) and \(B\) are pure, \(C\) is impure.

- Make literal containing a pure symbol true.
  - e.g. (for satisfiability) Let \(A\) and \(\neg B\) both be true
Unit clause: only one literal in the clause, e.g. \((A)\)

- The only literal in a unit clause must be true.
  - e.g. \(A\) must be true.
- Also includes clauses where all but one literal is false,
  - e.g. \((A, B, C)\) where \(B\) and \(C\) are false since it is equivalent to \((A, \text{false}, \text{false})\) i.e. \((A)\).
Tautology Deletion

- **Tautology**: both a proposition and its negation in a clause.
  - e.g. \((A, B, \neg A)\)
- Clause bound to be true.
  - e.g. whether \(A\) is true or false.
  - Therefore, can be deleted.
- Can be extended across clauses ...
Task 4: Improving DPLL (15 Marks)

- Implement an improved variable selection heuristic, i.e. a better choice function for DPLL.
- Improvement is possible in various dimensions:
  - time, space, exploitation of bias or inhomogeneities
  - specify what is improved (most likely time, i.e. efficiency)
  - any drawbacks?
- Partial credit will be given for any relevant attempt at a solution for this part.
Study performance of WalkSAT for different $p$

Compare WalkSAT, DPLL, DPLLv2 based on sample inputs provided

*Compare WalkSAT, DPLL, DPLLv2 for different $m/n$. 
Hard satisfiability problems

![Graphs showing the probability of satisfiability and runtime for different clause-to-symbol ratios for two algorithms: DPLL and WalkSAT.](image)
You should write a brief report

- Either as a comment in your code (in the section provided at the end of the file)
- If you want to add figures, then submit the report as an additional pdf file (add a note in the section provided at the end of the file)
- You will answer specific questions discussing the results and your experiences in implementing the algorithms.
Read the assignment sheet carefully before starting.

Code clarity is important. Comment your code adequately.

Reuse functions as much as possible.

You can add your own helper functions, but you should explain why they are necessary in your comments.

Test code and make sure they run before submitting.
Help and support

- Coursework clinics: Fridays from 11am (FH 1.B31)
- Ask you tutor
- Piazza
- e-mail to lecturer
9th March 2017 @ 4pm

[Official start: 9th February]

Good Luck!!!
Haskell Refresher
Purely functional! : “Everything is a function”

Main topics:

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

More on Haskell:
[https://wiki.haskell.org/Haskell](https://wiki.haskell.org/Haskell)
Unlike other programming languages like Java, Haskell has type inference.

However, type declarations ensures that you are specific about the input arguments of your function and the output values.

Example:

pLogicEvaluate :: Sentence -> Model -> Bool

The pLogicEvaluate function takes arguments of type Sentence and Model and returns a Bool type.
Type Synonyms

type Symbol = String

type Model = [(Symbol, Bool)]

- The type Symbol is a synonym for String, and type Model is a synonym for a list of (Symbol, Bool) tuples.
- Types synonyms are good for code clarity.
Important role in Haskell.

A function is recursive when one part of its definition includes the function itself again.

It is important to have a termination condition to avoid infinite loop.

```haskell
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```
Currying

- The process of creating intermediate functions when feeding arguments into a complex function.
- Note: all functions in Haskell really only take one argument

Example:
2 * 3 in Haskell:
- (*) function takes first argument 2, and returns an intermediate function (2*)
- The new function (2*) takes one argument, 3, and completes the multiplication

- Applying only one parameter to a function that takes two parameters returns a function that takes one parameter
Higher-Order Functions

- Functions are just like any other value in Haskell.
- Functions can take functions as parameters and also return functions.

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]
\[
\text{map} \ _ \ [] = []
\]
\[
\text{map} \ f \ (x:xs) = f \ x : \ \text{map} \ f \ xs
\]

- Map takes a function and list and applies that function to every element in the list.
List Processing Functions (map, filter, foldl, etc.)

- **map**: takes a function and list and applies that function to every element in the list.

  \[
  \text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
  \]

- **filter**: takes a predicate (function that returns true or false) and list and then returns the list of all elements that satisfy the predicate.

  \[
  \text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
  \]

- **foldl**: takes a binary function, an accumulator and a list. It 'folds' up the items in the list and return a single value.

  \[
  \text{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a
  \]
List Comprehension

- Build more specific sets out of general sets
- Example: to create a list of integers that are multiples of 2 and greater than than 20:

\[ \{ x \times 2 \mid x \leftarrow [1..25], x \times 2 \geq 20 \} \]

<table>
<thead>
<tr>
<th>Output function</th>
<th>Elements bound to x</th>
<th>Condition or Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
</tbody>
</table>
The Maybe monad represents computations which might “go wrong” by not returning a value.

If a value is returned, it uses Just a, where a is the type of the value.

If no value is available, it returns Nothing.

Example:

```haskell
safeDiv :: Double -> Double -> Maybe Double
safeDiv x y
  | y == 0 = Nothing
  | otherwise = Just (x/y)
```