KRF

Knowledge Representation Framework Project

Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

Computational Engines in Artificial Intelligence

This series contains technical reports and tutorial texts from the project on
the Knowledge Representation Framework (KRF).

The present report, PM-krf-022, can persistently be accessed as follows:

Project Memo URL:	http://www.ida.liu.se/ext/caisor/pm-archive/krf/022/
AIP (Article Index Page):	http://aip.name/se/Sandewall.Erik.-/2010/021/
Date of manuscript:	2011-01-04

Related information can also be obtained through the following www sites:

KRFwebsite:	http://www.ida.liu.se/ext/krf/
AIP naming scheme:	http://aip.name/info/
The author:	http://www.ida.liu.se/~erisa/

Introduction

The computational methods of Artificial Intelligence include a number of 'engines' which are essentially configurable algorithms. This means that they perform a certain computation in a systematic way, but the way that the computation is performed is subject to adaptations and adjustments that are specific to the application at hand. The resolution method for logical deduction, which has been presented earlier in the course, is one example of such a computational engine.

The present lecture note is intended to contain descriptions of a few major such engines, in particular SAT solvers, constraint programming, and answer set logic. However at present it is only a 'stub' (in the sense of Wikipedia) or a placeholder, that is, it contains the beginnings of the intended contents, awaiting that the full contents will materialize.

Chapter 1

Satisfiability Problems and SAT Solvers

In general, a *satisfiability problem* has the following form: A logic formula is given, and it is desired to identify a set of literals using the same vocabulary as is used in this formula, such that if those literals are assigned the value `true` and all others are assigned the value `false`, then the value of the given formula is `true`.

There is an important special case of the satisfiability problem, namely, the *Boolean satisfiability problem,* SAT, where the set of candidate literals is held fixed. This case is obtained if the formula is restricted to propositional logic, or if it is expressed in relational logic (only predicates, no functions) and the domain of objects is selected as fixed. There are also important uses of non-Boolean satisfiability problems, for example, the representation of Partial-Order Planning as a non-Boolean satisfiability problem in the lecture note 'Reasoning about actions and action planning.' [1]

Satisfiability problems are solved using *search* where one gradually extends or modifies a working set of literals until it satisfies the given restriction. Since the working set only contains literals, i.e. variable-free atomic formulas, we shall write these literals on successive lines and without their surrounding square brackets, when we show examples of working sets of literals. For ease of reading we shall also use infix notation of predicates when this is natural, for example `A = B` and `A before B` instead of the expressions `[= A B]` and `[before A B]`

1.1 The SAT Problem

The Boolean Satisfiability Problem (SAT) is defined as follows in its basic form.

Given: a set of propositional clauses

Question: does there exist an assignment of truthvalues to the proposition symbols whereby all the clauses are true?

[1] It is in fact possible to restrict and rewrite some planning problems as SAT problems, but this will not be addressed here.

In other words, does there exist an assignment whereby at least one of the literals in each clause is true?

There are two major approaches to the SAT problem: (1) The DPLL Engine: search the space of truthvalue assignments in a systematic, depth-first way, and (2) Stochastic Local Search: Pick one assignment randomly, then change the value of one proposition symbol at a time. Each of these have the character of computational engines.

1.2 The DPLL Engine

The DPLL Engine is based on a particular algorithm whose full name is the the Davis-Putnam-Logemann-Loveland algorithm, after its inventors. It obtains its strengths by the various plug-in methods that have been developed for it, and it is use of these additional methods that give it the character of a computational engine rather than just an algorithm.

The basic algorithm is as follows. Given a set A of clauses:

- Pick one of the proposition symbols in these clauses, e.g. p, and construct one modification of A for each of the two truth-values. Obtain A(p) by removing all clauses containing p and by removing -p in all clauses where they occur. Similarly for A(-p).
- Repeat the same operation with another proposition symbol, obtaining e.g. A(p,-q), and proceed recursively obtaining a search tree.
- If a branch obtains a descendant of A containing both {a} and {-a} for some literal a, then close that branch, i.e. do not expand the tree further from that point. This is called a *conflict.*
- If you can find a branch where all the proposition symbols have a value, then you have found an assignment. If all branches become closed then no assignment can exist whereby all clauses are true.

For an example of this algorithm, please see the e-slides ('powerpoint') for Lecture 13 in the course TDDC65.

The crucial issue in executing this procedure is to decide which proposition symbol to use next, during the search in a particular direction of the search tree. The following are some major *decision strategies,* i.e. rules for how to make this choice.

- Maximum Occurrence in clauses of Minimum Size (MOMS) as a goodness measure for selecting prop symbol.
- Dynamic Largest Individual Sum (DLIS): choose the literal occurring the most frequently in the clauses at hand.
- Variable State Independent Decaying Sum (VSIDS): keep track of the 'weight' of each literal, allow it to 'decay' i.e. it is gradually reduced over time, but if a literal is used for closing a branch then it is 'boosted' (its value is increased) for use elsewhere in the search tree.

There are also several other decision strategies, some of them fairly complex. One interesting method is *Clause Learning and Randomized Restart* which works as follows. The basic algorithm is modified so that if you arrive to a conflict, then analyze the situation and identify what clauses contributed to the conflict. Extract one or more additional "learned" clauses that are added to the given ones. Also, identify the level in the search tree that one has to return to. Proceed from there.

Then, from time to time you let the search process do a randomized restart, i.e. it restarts the search process from the root of the search tree, but retains the learned clauses. The purpose of these techniques is to let the process "learn" more direct ways of arriving to the desired result in the sense of the closing of a branch in the search tree.

In addition there are the following important implementation considerations that contribute to keeping down the search.

- Do a (modified) depth-first search, not a breadth-first search of the tree of possible assignments.
- Implement iteratively rather than using recursion.
- Literals in unit clauses are immediately set to true (as a preprocessing step and during the computation), except if you have a conflict (in which case close that branch).
- Proposition symbols that only occur positively in all the given clauses are immediately set to true and those clauses are removed. Conversely for those prop symbols that only occur negatively.

1.3 Statistical Local Search Techniques

The basic method in this approach to pick an initial assignment randomly, and then to change the value of one proposition symbol at a time, in such a way as to gradually approach a solution to the given SAT problem.

A number of techniques of this kind exist. We only consider one, called GSAT (G for Greedy). The basic idea in GSAT is: Start with an randomly chosen assignment. Calculate, for each proposition symbol, the increase or decrease in the number of clauses that become true if the value of that prop symbol is reversed. Pick the one that gives the best increase. Repeat this process until a satisfying assignment has been found or a maximum number (max-flips) has been reached. If max-flips has been reached, then try another randomly chosen assignment. Repeat until success or until a maximum number (max-tries) has been reached.

1.4 State of the Art for SAT Solvers

SAT solvers have been strikingly successful, both within Artificial Intelligence and in other areas. In principle, they provide a method for combinatorial reasoning and search which is able to handle very large sets of clauses. In comparison with the use resolution theorem-proving, SAT solvers use a more primitive representation, but they have the advantage of a number of very efficient implementation techniques.

Chapter 2

Constraint Programming

Constraint programming is an additional and important software technique that exhibits some similarities with SAT solving, but also many differences.

In general, a constraint programming problem specifies:

- A set of "variables"
- A domain of possible values for each variable
- A set of constraints ("relations") on these variables

An assignment of values to the variables that satisfies the restrictions is a solution to the constraint programming problem.

This is similar to SAT solving in the sense that one searches for an assignment of values to variables, but SAT solving is concerned with assignment of truth-values whereas constraint programming can be applied to any kinds of values, for example integers. Another difference is that constraint programming requires the facilities of a programming language, so that it is realized by extending a programming language with facilities for stating and solving constraint programming problems. Logic programming was the original host language for constraint programming. In this case we talk of constraint logic programming.

A tight integration of constraint programming in its host language requires that it should use as much as possible the data structures, declarations, and operators on data that are provided by that language. Therefore, constraint programming is the most easily hosted by languages with an interpretive character, e.g. functional programming languages (including Lisp and Scheme) and even Java, besides logic programming languages. However, constraint programming packages do exist even for C++.

The Wikipedia article on Constraint Programming is a useful source of examples and additional information.