Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

# A comparison of SL- and unit-resolution search rules for stratified logic programs

by

**Victor Lagerqvist**

LIU-IDA/LITH-EX-G--10/013--SE

2010-06-08


Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

# A comparison of SL- and unit-resolution search rules for stratified logic programs

**by**

**Victor Lagerqvist**

LIU-IDA/LITH-EX-G--10/013--SE

2010-06-08

Supervisor: Ulf Nilsson
Examiner: Ulf Nilsson

**Abstract**

There are two symmetrical resolution rules applicable to logic programs — SL-resolution which yields a top-down refutation and unit-resolution which yields a bottom-up refutation. Both resolution principles need to be coupled with a search rule before they can be used in practice. The search rule determines in which order program clauses are used in the refutation and affects both performance, completeness and quality of solutions. The thesis surveys exhaustive and heuristic search rules for SL-resolution and transformation techniques for (general) logic programs that makes unit-resolution goal oriented.

The search rules were implemented as meta-interpreters for Prolog and were benchmarked on a suite of programs incorporating both deterministic and nondeterministic code. Whenever deemed applicable benchmark programs were permuted with respect to clause and goal ordering to see if it affected the interpreters performance and termination.

With the help of the evaluation the conclusion was that alternative search rules for SL-resolution should not be used for performance gains but can in some cases greatly improve the quality of solutions, e.g. in planning or other applications where the quality of an answer correlates with the length of the refutation. It was also established that A* is more flexible than exhaustive search rules since its behavior can be fine-tuned with weighting, and can in some cases be more efficient than both iterative deepening and breadth-first search.

The bottom-up interpreter based on unit-resolution and magic transformation had several advantages over the top-down interpreters. Notably for programs where subgoals are recomputed many times. The great disparity in implementation techniques made direct performance comparisons hard however, and it is not clear if even an optimized bottom-up interpreter is competitive against a top-down interpreter with tabling of answers.

# Preface

## What

This document is the culmination of a bachelor thesis in computer science at the Theoretical computer science laboratory at Linköping University. It is structured in four major parts:

- The theory, which explicates two restrictions of resolution applicable to general logic programs.
- The implementation, which surveys a collection of interpreters embodying the theory.
- The evaluation, which tests each interpreter on a benchmark suite.
- The discussion, which attempts to bring clarity over the usefulness of each interpreter.

## Why

When I first opened my copy of Sterling and Shapiro's The Art of Prolog a rainy[1] summer day I was immediately hooked on the elegance and simplicity of the language. One of the features I found intriguing was that the interpreter had to make choices during the refutation of a goal clause. How the choices are made affects the order in which solutions are derived and hence also the performance. If an interpreter is able to make intelligent choices the burden on the programmer to correctly order clauses efficiently is relaxed. Since it is an interesting problem it should not be to surprising that it is NP-hard [1]. We shall not let such trifles stop us however!

[1] Sunny.

# Acknowledgements

I would like to thank my supervisor, Ulf Nilsson, for patiently explaining the numerous pitfalls of combining negation, magic transformations and fixpoints. A dangerous blend which is sometimes hard to swallow. Moreover I am also indebted to Paulo Moura for his work on Logtalk and for clearing my confusion over some of Logtalk's finer concepts.

# Contents

# Chapter 1

# Introduction

*"A serious and good philosophical work could be written consisting entirely of jokes."*
– Ludwig Wittgenstein

A logic program is a finite collection of if-then rules of the form $B_1 \wedge ... \wedge B_n \rightarrow A$. The classically schooled logician will immediately emit that there are two standard inference rules for extracting knowledge from such programs. The first and perhaps most natural, modus ponens, states that if $B_1 \wedge ... \wedge B_n$ is true then $A$ must also be true. The second, modus tollens, states that if $A$ is not true, i.e. $\neg A$, then $\neg(B_1 \wedge ... \wedge B_n)$ is true. These classical rules are both instances of a single powerful inference rule which goes by the name *resolution*. It was formulated in a cryptic paper by John Allan Robinson in 1965 and first applied to logic programs by Robert Kowalski in 1971 [2], although it had informally been used by both Alain Colmerauer in natural language applications and in the Planner programming language embedded in Lisp [3].

The first Prolog system was implemented by Philippe Roussel in the fall of 1972 [3]. It used a simple depth-first backtracking approach in conjunction with SL-resolution with the computation rule of always selecting the leftmost literal. The arguments for this strategy back then were more or less the same that are used today: it uses little memory, has a small overhead and the burden of avoiding infinite branches in the resolution tree is placed on the programmer. Another breakthrough occurred when the semantics of logic programs was thoroughly explicated by Kowalski and Van Emden in 1976 [4] with the notion of fixpoints. An equivalent operational proof procedure by the name hyper-resolution[1] was also introduced. Hyper-resolution

[1]Hyper-resolution will in the other chapters go by the name unit-resolution.

and SL-resolution are the dualities of each other just like modus ponens and modus tollens, and are the basis for most modern logic programming implementations.

While Prolog's depth-first search rule have its merits it also introduces problems. It is incomplete for infinite trees, rarely produces the shortest refutation and is sensitive to clause and goal ordering. Unit-resolution and alternative search rules for SL-resolution can fix these problems even if efficiency in some instances must be sacrificed.

# Problem statement

The aim of the thesis is to identify strengths and weaknesses of two different resolution rules, SL-resolution and unit-resolution, and how search rules affects performance, completeness and computed answer substitutions. Each search rule is implemented as a meta-interpreter written in Prolog + Logtalk and benchmarked on a plethora of programs to ascertain its qualities.

# Restrictions

Like all other finite set of strings this thesis is incomplete. The most glaring omission is perhaps SLG-resolution[2], i.e. SL-resolution extended with tabling. However interesting it would have been to investigate how the search rules performed when combined with tabling it could not have been squeezed into this thesis with dignity.

Alternative computation rules are also not covered due to time and space constraints. If it had been covered the most interesting part would probably to see how coroutining integrated with the various search rules.

Regarding heuristic search there is still work left. This thesis only covers a simple admissible heuristic based on clause length. While it is not likely that it is possible to get a much better heuristic based on static program analysis, statistical models could potentially give more accurate predictions even if they are not admissible. Incomplete search strategies such as beam search are left out, even though they could be useful in programs that have high branching factor and where it is good enough to find a couple of solutions.

[2]To be truthful, a bottom-up interpreter with magic transformation is closely related to SLG-resolution.

# Chapter 2

# Notation and terminology

Before we can start our excursion into logic programming we need some basic terminology. The reader is assumed to be familiar with first-order logic and notions such as *term*, *unification*, *quantifier* as well as the standard logical connectives [5]. The notion *atomic* will sometimes be used instead of *atomic formula*, even though the former is sometimes reserved for use in propositional logic.

**Definition.** A first-order formula $F$ is *universal* if it is of the form
$\forall x_1 ... \forall x_n \phi$
where $\phi$ is a quantifier free formula with no free variables besides $x_1, ..., x_n$.

This is just a matter of convenience but will be useful in the following definitions.

**Definition.** If F is a universal formula $\forall x_1 ... \forall x_n \phi$ then $\phi$ is called the *kernel* of F.

**Definition.** A universal formula F is a *Horn clause* if its kernel $\phi$ is of the form (a), (b) or (c)

**(a)** $\chi$

**(b)** $\chi \vee \neg\chi_1 \vee \neg\chi_2 ... \vee \neg\chi_n$

**(c)** $\neg\chi_1 \vee \neg\chi_2 ... \vee \neg\chi_n$

Where $\chi$ and $\chi_i$ are atomic. In the cases of (a) and (b) F is positive, in (c) negative. In logic programming notation (a), (b) and (c) are written as

**(a)** $\chi$

**(b)** $\chi \leftarrow \chi_1, ..., \chi_n$

**(c)** $\leftarrow \chi_1, ..., \chi_n$

and are called *fact/unit clause, rule* and *goal clause* respectively. In the case where $n = 0$ (c) is written as $\square$ and called *the empty goal clause*.

**Definition.** A *(definite) logic program* $P$ is a finite set of positive Horn clauses.

Hence a logic program consists of a set of assertions, the facts, and a set of rules which allows us to infer additional information. This is a serious restriction of first-order logic and it is not the case that every set of first order formulas has a logically equivalent set of Horn clauses. Despite this it is a very useful fragment of first-order logic which is well understood in both proof and model theory.

# Chapter 3

# Inference in logic programming

In first-order logic we are in general interested whether $\Sigma \models \phi$ [1], i.e. that every model of the set of formulas $\Sigma$ is a model of $\phi$. In contrast to propositional logic it is not feasible to simply enumerate all possible interpretations since there are infinitely many. Thankfully, to paraphrase George Orwell, some interpretations are more equal than others. In Herbrand interpretations all ground terms are mapped to themselves and the predicate extension is just a subset of all combinations of predicate symbols with ground terms. Universal formulas have the property that whenever there exists a model there exists a corresponding Herbrand interpretation. This forms the basis for *resolution* which is the most widely used proof procedure for logic programs.

## 3.1 Resolution

Resolution is a refutation procedure. This means that we start with a (satisfiable) set of clauses, e.g. a logic program $P$, and assert the negation of what we want to prove, e.g. a goal clause $\leftarrow G$. Through applications of a suitable resolution rule we then deduce whether this new set is unsatisfiable or not. If it is unsatisfiable it means that there is no interpretation $I$ such that $I$ is a model of $P$ and $\leftarrow G$. Or, if we turn it around, no interpretation $I$ such that $I$ is a model of $P$ but not a model of $G$. The resolution rule is truth preserving in the sense that if it is possible to deduce an unsatisfiable statement, the empty clause, then the clause set is unsatisfiable. The following definitions formalizes this intuition.

**Definition.** Two literals $\phi$ and $\chi$ are *clashing* if either $(\neg\phi)\theta = \chi\theta$ or $\phi\theta = (\neg\chi)\theta$ with most general unifier $\theta$.

[1] Equivalently it can also be stated $\Sigma \cup \{\neg\phi\} \models \square$.

**Definition.** Let $A$ and $B$ be two Horn clauses with clashing literals $\neg\phi$ and $\chi$. Then the *resolvent* of $A$ and $B$ is
$C = (A \setminus \{\neg\phi\})\theta \cup (B \setminus \{\chi\})\theta$
Where $\theta$ is the most general unifier of $\phi$ and $\chi$ and variables are renamed if necessary.

Moreover, if $A$ and $B$ are satisfiable then $C$ is satisfiable. The proof of this is not provided but may be found in almost any logic textbook, e.g. Ben-Ari [5].

**Definition.** $A$ and $B$ in the previous definition are called *input clauses.*

This resolution is called *binary* resolution. Without the factoring rule to remove subsumed literals it is not a complete proof procedure for full first-order logic, but restricted to Horn clauses it is.

**Definition.** $C$ is *derived* from $A$ and $B$ if $C$ is the resolvent of $A$ and $B$.

**Definition.** Let $P$ be a logic program and $\leftarrow G$ a goal clause. A *refutation* of $\leftarrow G$ is a finite sequence of derivations ending in the empty clause.

For a logic program $P$ and a goal clause $\leftarrow G$ we now have a simple but computationally inefficient refutation proof procedure: derive clauses until either the empty clause is derived or until no more resolvents can be formed. In the former case this means that $P \models G$ [2].

This proof procedure is still rather crude. Much efficiency can be gained if the choice of input clauses and the choice of clashing literal is somehow restricted. The two different strategies that are presented in the following sections are called selective linear-resolution and unit-resolution respectively.

## 3.2 SL-resolution

Selective linear-resolution is the proof procedure of choice for most Prolog systems. When restricted to Horn clauses it is often referred to as SLD-resolution, where SLD is an abbreviation of Selective Linear Definite. Our first stepping stone towards SL-resolution is linear resolution, where one of the input clauses in a resolution step must be a previous resolvent or the

[2] More precisely it means that there exists a computed answer substitution $\theta$ such that P $\models \forall(G\theta)$.

initial goal clause. This should not come as a surprise to any reader familiar with Prolog's execution strategy.

**Example.** Given the logic program

edge(a, b). (1)
edge(b, c). (2)
connected(X, Y) ← connected(X0, Y), edge(X, X0). (3)
connected(X, Y) ← edge(X, Y). (4)

And the goal clause ← $connected(a, c)$ (5) a linear resolution refutation could look as follows*:*

- Resolving (5) with (3) gives ← $connected(X0, c), edge(a, X0)$ (6)
- Resolving (6) with (1) gives ← $connected(b, c)$ (7)
- Resolving (7) with (4) gives ← $edge(b, c)$ (8)
- Resolving (8) with (2) gives □ (9)

**Definition.** The last goal clause in a sequence of derivations is called a *center clause.* In the previous example the center clauses during the refutation are (5), (6), (7), (8) and (9).

**Definition.** An input clause used in a resolution step together with a center clause is called a *side clause.* In the previous example the side clauses during the refutation are (3), (1), (4) and (2). Due to the restricted nature of Horn clauses a side clause is always one of the original, renamed clauses of the program.

What have we accomplished with this restriction? It is now more goal oriented than before. Once we have chosen an initial center clause the refutation must continue on that track. The next refinement restricts the choice of clashing literal.

**Definition.** A *computation rule* $\sigma$ is a possibly multivalued map from a set of goal clauses to a set of atoms, i.e. given a goal clause as input the computation rule gives an atom as output, called the *selected atom.*

This might not sound very helpful. After all, how are we going to decide which atom to pick? It turns out that the choice of $\sigma$ does not alter the completeness of SL-resolution, hence we have found a way to severely restrict

the choice of clashing literal by very simple means. The intuitive reason behind the independence of the computation rule is that if it is possible to derive the empty clause by some selection of atoms any other selection will eventually accomplish the same thing — the only difference lies in which order the atoms are resolved away. With the computation rule in place it is now possible to represent derivations as branches in a tree.

**Definition.** Let P be a logic program, $G_0$ a goal clause and $\sigma$ a computation rule. Then the SL-tree of $G_0$ using $\sigma$ is defined as the smallest tree satisfying (a) and (b):

**(a)** The root of the tree is labeled $G_0$.

**(b)** If $G_i$ is a node in the tree which resolves with $\mathrm{A} \in \mathrm{P}$ using computation rule $\sigma$ with resolvent $G_{i+1}$, then $G_{i+1}$ is a child of $G_i$ and the edge connecting them is labeled A (where the variables in A are renamed if necessary).

If $G_i$ is a leaf in the SL-tree labeled with $\square$, then $G_i$ is called a *goal node*.

**Example.** Given the logic program:

$f(1) \leftarrow f(X), g(X).$
$f(2).$
$g(2).$

And the goal clause $\leftarrow f(1)$, it might come as a surprise that the computation rule does not affect the success or failure of the refutation. After all, if the uppermost clause is selected as the side clause and Prolog's computation rule of always selecting the leftmost atom is used the refutation will continue in infinity due to the infinite branch in the resolution tree.

This disparity with the completeness of SL-resolution stems from the fact that any concrete implementation needs to be coupled with a strategy for traversing the resolution tree, called the *search rule*. That SL-resolution is complete affirms the existence of a refutation but does not offer any guideline of how to find it.

## 3.3 Unit-resolution

When a goal clause is used as the initial center clause with SL-resolution a *top-down* refutation is obtained. The duality of top-down is *bottom-up*. Before introducing the restriction of resolution which yields such a computation scheme an example which highlights the differences is provided.

**Example.** Consider the following logic program whose Herbrand base is isomorphic to the set of natural numbers:

natural(zero).
natural(s(N)) ← natural(N).

Given the goal clause $\leftarrow natural(s(zero))$ a top-down refutation would proceed by selecting $natural(s(N)) \leftarrow natural(N)$ as side clause and produce the resolvent $\leftarrow natural(zero)$, which a resolution step later would resolve with the fact $natural(zero)$ and produce the empty clause.

A bottom-up refutation would on the other hand first notice that $natural(zero)$ is true. If $natural(zero)$ is true it may be used in a resolution step with $natural(s(N)) \leftarrow natural(N)$ to give the resolvent $natural(s(zero))$ which resolves with the goal clause to produce the empty clause. Notice the difference between this and the top-down approach. Here we started with the fact $natural(zero)$ and used it in a resolution step with the *body* of the rule instead of the *head*.

With the example in mind it should be clear that the correct way to restrict resolution to obtain a bottom-up refutation is to restrict one of the input clauses to be a unit clause, i.e. a clause with exactly one positive literal. The unit-resolution tree is then defined in the same spirit as for SL-resolution:

**Definition.** Let P be a logic program and $\leftarrow G_0, ..., G_m$ a goal clause. Then the unit-tree is defined as the smallest tree satisfying (a), (b) and (c):

**(a)** The root node is labeled by the symbol 'P', and the children of the root are the unit clauses of P.

**(b)** If $U$ is a (positive) node in the tree that resolves with a program clause or another node in the tree $A \leftarrow ..., B_i, ...$ with resolvent $V = (A \leftarrow ..., B_{i-1}, B_{i+1}, ...)\theta$ then $V$ is a child of $U$ and the edge connecting them is labeled by $\theta$.

**(c)** If $U$ is a (positive) node in the tree that resolves with a goal clause or another node in the tree $\leftarrow ..., B_i, ...$ with resolvent $V = (\leftarrow ..., B_{i-1}, B_{i+1}, ...)\theta$ then $V$ is a child of $U$ and the edge connecting them is labeled by $\theta$.

In contrast to an SL-tree the unit-tree as defined here can contain many redundant nodes. The reasons are twofold. First, all logical consequences of the program is contained in the tree regardless of whether they are relevant to the goal or not. Second, since no computation rule is defined some branches

differ only in which literal was selected first. Hence, in practical applications of unit-resolution we are only interested in generating parts of the resolution tree. The interpreters in chapter 4 does for instance only use resolution on a rule if the whole body can be resolved away in a single resolution step.

The following theorem establishes that unit-resolution is as powerful as SL-resolution when restricted to logic programs.

**Theorem.** *Let $P$ be a logic program. If $P \models G$ then there exists a refutation of $\leftarrow G$ using only unit-resolution.*

The interested reader is redirected to Van Emden and R. A. Kowalski [4] for a proof.

## 3.4 Negation as failure

It is not possible infer negative conclusions from a (definite) logic program in a sound proof system. Logically speaking this is nothing weird — absence of proof is not proof of absence. There is however a conflict between this and ordinary reasoning, where it is often assumed that information is complete in the sense that anything not explicitly stated as true should be interpreted as being false. This intuition is called *closed world assumption (CWA)* and may be stated as: if $P \nvdash g$ then $\neg g$. The CWA rule also goes by the name *negation as infinite failure*. In the case of SL- and unit-resolution only finite refutations are of interest. This restriction of CWA is naturally called *negation as finite failure*.

The first step of obtaining a refutation procedure for logic programs with negation is the definition of a general logic program:

**Definition.** A universal formula $F$ is a *general clause* if its kernel $\phi$ is of the form (a), (b) or (c)

**(a)** $\chi$

**(b)** $\chi \vee \neg\chi_1 \vee \neg\chi_2 ... \vee \neg\chi_n$

**(c)** $\neg\chi_1 \vee \neg\chi_2 ... \vee \neg\chi_n$

Where $\chi$ is atomic and $\chi_i$ is a literal. In the cases of (a) and (b) $F$ is positive, in (c) negative. As in the case of Horn clauses (b) and (c) are often written as:

**(b)** $\chi \leftarrow \chi_1, ..., \chi_n$

**(c)** $\leftarrow \chi_1, ..., \chi_n$

**Definition.** A *general logic program* $P$ is a finite set of positive general clauses.

Negation as failure may be used with both SL-resolution and unit-resolution. The formal definitions of both these endeavors fall outside the scope of this document; instead a procedural interpretation is provided. The interested reader should turn to Lloyd [6] for a formal introduction to SLDNF-resolution and to Ross [7] for more information regarding bottom-up and negation.

## SL-resolution with negation as finite failure

Recall that SL-resolution uses a computation rule. In the case where the computation rule returns a positive goal it is solved as usual. If the goal $\neg G$ is encountered the traversal of the current resolution tree is temporarily suspended and construction of the resolution tree for $G$ is started. If the tree is finitely failed the goal succeeds, otherwise it fails if the empty answer substitution was computed [3].

## Unit-resolution with stratified negation

The reason why SL-resolution is easily augmented to general logic programs is that one only needs to check whether or not the goal is provable. This scheme is not sufficient for unit-resolution since the logical consequences of the program are derived in iterations. If a goal is provable we may safely assert that its negation is false. The converse is however much harder — just because a goal is not provable at a specific time does not mean that it will not become provable later on. Worse yet, there are cases where it is never possible to know whether a negated goal is true or false. Even if the situation looks grim there is no reason to despair — we can impose an additional requirement on a general logic program to prohibit this behavior.

**Definition.** Let P be a general logic program, B a predicate symbol and $P^B$ the set of clauses in P that has B in their heads. P is *stratified* iff there exists a partitioning $P_1 \cup P_2 \cup ... \cup P_n$ of P such that:

- if $A(...) \leftarrow ..., B(...), ... \in P_i$ then $P^B \subseteq (P_1 \cup ... \cup P_i)$.

[3] It should be noted that most implementations of negation as finite failure omit to check whether or not the empty answer substitution was produced, which results in an unsound refutation procedure.

- if $A(...) \leftarrow ..., \neg B(...), ... \in P_i$ then $P^B \subseteq (P_1 \cup ... \cup P_{i-1})$.

Let $A \leftarrow ..., \neg B, ... \in P_{i+1}$. Then negation can be handled as follows:

- $\neg B$ is true in $P_{i+1}$ if $B$ is false in the least Herbrand model of $P_1 \cup ... \cup P_i$.
- $\neg B$ is false in $P_{i+1}$ if $B$ is true in the least Herbrand model of $P_1 \cup ... \cup P_i$.

The effect is that rules are not allowed to contain negated recursive references. With stratified programs the search rule can be restricted to form all possible resolvents in a stratum before moving on to the next one. If the program is not stratified it would in general not be possible to deduce this since the absence of an atom in the least Herbrand model for the previous stratum would not imply that it is not a consequence of the program. A classical example that may be viewed as an unstratified general logic program is Bertrand Russel's Barber paradox:

**Example.** Barber paradox.

shaves(barber,Person) ← person(Person), not(shaves(Person,Person)).
person(barber).
person(mayor).

It should be read as "the barber shaves every person that does not shave himself". The barber shaves the mayor. But who shaves the barber? It is unstratified since the negation refers to a predicate that belongs to the same stratum.

# Chapter 4

# Implementation

## 4.1 Overview

In chapter 3 two different restrictions of resolution are presented: SL-resolution and unit-resolution. They are both theoretical principles and need to be augmented with a search rule before they can be used as automatic refutation procedures[1].

Implementing a Prolog system from scratch in a low-level language is not a trivial task. It is possible to save a lot of time if one can borrow garbage collection, term representation, unification et al. from the host language. What language could possible be more suitable for this purpose than Prolog itself? Writing an interpreter for Prolog in Prolog is really no different than writing any other program. These kind of interpreters are often referred to as *meta-interpreters*. In Sterling and Shapiro [1] a meta-interpreter for a language is defined as an interpreter for the language written in the language itself. The interpreters in this document are by this definition not meta-interpreters, but rather interpreters written in Prolog for a Prolog-like language. Where to start? We need three things. First we need to state what is a priori true. Second we need to state what it means for a conjunction to be true. Third we need to state what it means for a rule to be true.

An example of a meta-interpreter from Sterling and Shapiro [1] have the following structure:

**Program.** Meta-interpreter for (pure) Prolog

prove(true).
prove((A, B)) ← prove(A), prove(B).
prove(A) ← clause(A, B), prove(B).

[1] SL-resolution also needs a computation rule.

Where $clause(A, B)$ is true if $B$ is the body in the rule $A \leftarrow B$. If $A$ is a unit-clause then the $B$ is simply the constant *true*. Viewed with the declarative thinking cap on the interpreter is simple to understand:

- The goal *true* is provable.
- The conjunction $A, B$ is provable if both $A$ and $B$ are provable.
- $A$ is provable if there exists a clause $A' \leftarrow B$ such that $A = A'$ with most general unifier $\theta$ and $B\theta$ is provable.

There is of course nothing that prevents us from using another syntax in the interpreted language. The syntax used in the actual implementations is borrowed from O'Keefe [8] and has the following structure:

- The constant *true* is written as [].
- The unit clause $A$ is written as $rule(A, [])$.
- $B_1, ..., B_n$ is written as $[B_1, ..., B_n]$.
- $A \leftarrow B_1, ..., B_n$ is written as $rule(A, [B_1, ..., B_n])$.

Manually writing all those rule-predicates is tedious work. Therefore Prolog's $term\_expansion/2$ mechanism is used to expand programs into the correct format during compilation time. This scheme has the nice property that it is possible to calculate certain properties of clauses that can be used in heuristic search, e.g. the length of the body, without any impact on the run time performance.

## 4.2 Method

Just like with any other language Prolog programmers are faced with the usual software development hurdles of code reuse and capsulation. Module systems are available in most modern Prolog implementations but are not always compatible with one another. Logtalk[2] is an object-oriented extension to Prolog compatible with most major vendors, and can in its simplest form be used as a portable module system. For an in-depth explanation of Logtalk features and their implementation, see [11] by Paulo Moura. The actual implementation of the interpreters presented in this chapter are written in Logtalk, but for clarity and brevity the code is stripped of everything not necessary to convey the general idea behind it. In reality it often makes use of the standard library in Logtalk, is often longer and often more efficient.

[2] http://logtalk.org/

## 4.3 Exhaustive top-down search

Exhaustive search, or uninformed search, only has the capability to expand nodes and distinguish between goal nodes and non-goal nodes. This section encompasses depth-first, breadth-first and iterative deepening depth-first.

### 4.3.1 Depth-first

In depth-first search the nodes that are to be visited are kept on a stack. The result is a search which traverses the resolution tree a branch at a time until either a goal or failure node is reached. Despite its shortcomings with completeness for infinite trees it is the strategy of choice in Prolog, the reason being that it is very space efficient and has a relatively small overhead. Implementing a depth-first search in Prolog is for natural reasons very simple, as the following interpreter will demonstrate.

**Program.** Depth-first interpreter for general logic programs. [3]

```
prove(Goal) ←
    prove_conjunction([Goal]).

prove_conjunction([]).
prove_conjunction([not(Goal)|Goals]) ←
    (prove(Goal) -> fail ; prove_conjunction(Goals)).
prove_conjunction([Goal|Goals]) ←
    rule(Goal, Body),
    prove_conjunction(Body),
    prove_conjunction(Goals).
```

The interpreter used in the evaluation chapter uses a more clever scheme in order to avoid the two recursive calls in the third clause of prove_conjunction/1: instead of storing rules as tuples of a head and a body, i.e. rule(Head, Body), a triple rule(Head, Body, Tail) is used where Tail denotes the end of Body. Then the tail of Body can be used to append Body and Goals in constant time.

This depth-first implementation is known as *backtracking*. The difference between backtracking and regular depth-first is that backtracking only expands states as they are needed. If a branch leads to a dead-end control is returned to the latest choice point and the next state is tried.

[3] Based on source code from The Craft of Prolog [8].

### 4.3.2 Breadth-first

Breadth-first search uses a queue instead of a stack to save expanded nodes. Its behavior is the opposite of depth-first search — instead of exploring a single branch at time all branches are explored simultaneously. What does this mean in the case of a resolution tree? Branches in a resolution tree correspond to different choices of side clauses.

**Example.** Given the logic program:

f(X) ← g(X).
f(b).
g(a).

And the goal clause ← $f(X)$, an interpreter using breadth-first search will first give the answer X = b followed by X = a. The reason is that the empty clause may be reached earlier when the second clause of f/1 is picked.

How can this behavior be expressed in an interpreter? It is clear that a data structure that separates branches from each other is needed. The interpreter then expands the current goal of a branch, adds the new branches last in the queue before it moves on to the next branch of the same depth in the tree. The following incomplete code fragment illustrates the idea:

```
prove([Branch|_Branches]) ←
      goal_branch(Branch).
prove([Branch|Branches]) ←
      negation_branch(Branch),
      handle_negation([Branch|Branches]).
prove([Branch|Branches0]) ←
      expand_branch(Branch, New_Branches),
      append(Branches0, New_Branches, Branches),
      prove(Branches).
```

Implementing expand_branch/2 efficiently is the hardest part. Let $Branch = [G_1, G_2, ..., G_n]$. Then $New\_Branches$ is the (multi) set of branches $[H_1\theta, ..., H_m\theta, G'_2\theta, ..., G'_n\theta]$, where $H_1, ..., H_m$ is the body of the rule whose head $H$ unified with $G_1$ and $G'_2\theta, ..., G'_n\theta$ is the result of applying the most general unifier between $H$ and $G_1$, $\theta$, and renaming all unbound variables. [4]

In the actual implementation a queue is used instead of a list to prevent the needless copying of $Branches0$ in the third clause of $prove/1$. Also,

[4] This is necessary since if two branches shares a common variable, binding it in one branch will incorrectly bind it in the other too.

since a yes/no-answer hardly is satisfying, each branch contains a list of the equations that have been applied so far. When a goal node is reached the equations are solved which effects the Prolog system to print the unifiers. Evidently the breadth-first interpreter has quite a bit of overhead compared to the simple depth-first version since all branches have to be kept in memory.

### 4.3.3 Iterative deepening depth-first

One of the problems with depth-first search is that it is incomplete for infinite trees. Iterative deepening depth-first is a restriction of depth-first that contains an additional argument, the depth bound, which is used as a warning flag to indicate that the depth limit has been exceeded. If no goal node is found at the current depth the bound is increased and the search starts anew. The effect is that the search will not get stuck in infinite branches since the bound will eventually force the branch to be abandoned. We can code it very simply by augmenting *prove_conjunction*/1 in the depth-first interpreter with the depth bound.

**Program.** Iterative deepening depth-first interpreter for general logic programs. [5]

```
prove(Goal) ←
      prove([Goal], 1).
prove(Goals, Bound) ←
      bounded_prove(Goals, Bound).
prove(Goals, Bound) ←
      Bound1 is Bound + 1,
      bounded_prove(Goals, Bound1).
bounded_prove([], _Bound).
bounded_prove([not(Goal)|Goals], Bound) ←
      handle_negation([Goal|Goals], Bound).
bounded_prove([Goal|Goals], Bound, Remaining) ←
      Bound1 is Bound - 1,
      Bound1 >= 0,
      rule(Goal, Body, Goals),
      bounded_prove(Body, Bound1).
```

This version has a major weakness. If a solution is reported at bound $x$, it will be reported again at bound $x + 1$. It is not possible to abstain from finding the solution again but relatively straightforward to at least avoid

[5] Based on source code from The Craft of Prolog [8].

reporting it: add an argument to *bounded_prove*/2 which keeps track of the remaining bound, and add a check in *prove*/2 that confirms whether or not the solution has been found earlier by comparing the remaining bound to the value which the bound was previously incremented with. The remaining bound tells us how much of the bound that was needed to find the solution, and if it is smaller than the increment it must mean that the solution could not have been found previously.

## 4.4 Heuristic top-down search

The search rules presented thus far are uninformed in the sense that only the depth of nodes are taken into account — depth-first search prefers freshly expanded nodes while breadth-first search prefers nodes at lower depth. It would be preferable if the search algorithm could choose the branch closest to a goal node without fruitless expansion of branches that lie far away from a goal. In most interesting applications this knowledge cannot be estimated exactly and must be approximated with a heuristic. Let $P$ be a logic program, $G$ a goal clause, $T$ the SL-tree of $P$ and $G$ with Prolog's computation rule and $x$ a node in $T$. Then:

- $h(x)$ = estimated length of the shortest path from $x$ to a a goal node in $T$.
- $g(x)$ = the depth of $x$ in $T$.
- $f(x)$ = the cost of $x$. This can simply be $h(x)$, $g(x)$ or a combination of the two.

Branches from the current node are then evaluated with regard to their cost and are placed in the appropriate position with the help of a priority queue. The general framework is easily expressed:

**Program.** Framework for a heuristic interpreter for general logic programs.

```
prove(Goal) ←
    empty_heap(H),
    insert_goal(H, Goal, H1),
    prove_branch(H1).
prove_branch(Heap) ←
    heap_top(Heap, _Cost, Branch),
    goal_branch(Branch).
prove_branch(Heap) ←
```

```
heap_delete(Heap, Cost, Branch, Heap1),
(     negation_branch(Branch) ->
      handle_negation_branch(Heap1, Cost, Branch)
;     expand_branch(Cost, Branch, Branch_Cost_Pairs),
      heap_insert_all(Branch_Cost_Pairs, Heap1, Heap2),
      prove_branch(Heap2)
).
```

Before we can implement *expand_branch*/3 we need to pinpoint the heuristic. We can, in principle, select any function as $h(x)$ irregardless of whether it is correlated to the cost of reaching a goal node or not. The choice of $h(x)$ affects not only the performance of the search rule but also its completeness. If $h(x)$ never overestimates the cost of reaching a goal node it is said to be *admissible*. There exists a very simple admissible heuristic for SL-tree search which is based on the minimum number of resolution steps required to solve a clause. Let $A \leftarrow B_1, ..., B_n$ be a clause and assume that we do not know anything about $B_1, ..., B_n$. While we do not know anything about the upper bound of resolution steps we do know something about the lower bound — namely that at least $n$ steps are required. Even if it turns out that the actual cost is much higher it is impossible to overestimate the actual cost. The best-first and A* interpreter both use variations on this heuristic.

The problem with using the minimum number of resolution steps is that it is *too* optimistic. In general solving each $B_i$ may require much more work. One possible workaround is to estimate the required number of resolution steps for $A$ as $\Sigma steps(B_i)$ and recursively calculate $steps(B_i)$. While this is a slight improvement over the first heuristic not much is gained in practice. The goals that will take the longest time to solve are those performing any form of iteration. Assume that $B_i$ is defined recursively. Then $B_i$ likely contains one or more unit clause as base case to end the recursion, and if the heuristic is to remain optimistic it must estimate the cost of $B_i$ as 1 since it is possible that it could be solved in one step. Possible extensions to the heuristic function using run-time analysis while maintaining admissibility is discussed in chapter 6.

We are now ready to implement *expand_branch*/3. To prevent unnecessary run time overhead the earlier *rule*/3 construct is augmented with an additional argument that contains the length of the body of the rule. Each branch consists of a list of literals, the length of the list and the current depth of the branch.

**Program.** Definition of *expand_branch*/3.

```
expand_branch(_, branch([], 0, _), []).
expand_branch(Cost0, branch([Goal|Goals], Length0, Depth0), Pairs) ←
    Depth is Depth0 + 1,
    findall(Cost - branch(Body, Length, Depth),
            (
             rule(Goal, Body, Length1, Goals),
             Length is Length0 + Length1 - 1,
             f(Length0, Length1, Depth, Cost)
            ),
            Pairs).
```

Each concrete implementation must provide the definition of $f/4$ which evaluates the new branches. For clarity the branch representation is simplified. Like the breadth-first interpreter each branch also contains a list of equations that are solved when a goal node is reached.

### 4.4.1 Greedy best-first

Greedy best-first search uses $h(x)$ as the cost of $x$. The effect is that the branch which is currently believed to be closest to a goal node is always picked. As might be expected this scheme does not yield a complete search rule since it is possible to get stuck in infinite branches. The heuristic used is the length of the current goal clause. If the goal clause is $\leftarrow G_1, ..., G_n$ and $G_1$ resolves with both $A_0 \leftarrow A_1, ..., A_i$ and $B_0 \leftarrow B_1, ..., B_j$ the cost of the first clause is $i + n - 1$ and the cost of the second is $j + n - 1$.

**Program.** Definition of $f/4$ for a greedy best-first interpreter for general logic programs.

```
f(Length1, Length2, _Depth, Cost) ←
    Cost is Length1 + Length2 - 1.
```

### 4.4.2 A*

The problem with greedy best-first search is that the cost of reaching a node is not taken into account. This is easily mended by combining $h(x)$ and $g(x)$, i.e. $f(x) = h(x) + g(x)$. With this change the interpreter will not get stuck in infinite branches since the cost would rapidly eclipse the other branches.

**Program.** Definition of $f/4$ for an A* interpreter for general logic programs.

```
f(Length1, Length2, Depth, Cost) ←
    Cost is (Length1 + Length2 - 1) + Depth.
```

### Taming the beast - weighing $f(x)$

The problem with A* is that it degenerates into a breadth-first search if all branches have equal cost. It is sometimes preferable to exhaust one of the branches like a depth-first search if no better heuristic is available. The method for accomplishing this is called *weighting* and works by introducing an additional parameter into $f(x)$, $w$, which can be tuned in order to give more weight to either $h(x)$ or $g(x)$. The function then becomes:
$f(x) = (1 - w) * g(x) + w * h(x)$
If $w = 1$ then a greedy best-first search is obtained. Similarly a breadth-first search is obtained for $w = 0$. Care must be taken to ensure that $f$ is still admissible. In Pearl [10] it is established that $f$ may lose its admissibility for $w > 1/2$. In the evaluation two different weights are used to test what performance gains can be made by either decreasing or increasing $w$.

## 4.5 Bottom-up search

Bottom-up search corresponds to unit-resolution in the same way that top-down search corresponds to SL-resolution. Again: the resolution rule that we are using is only a principle. To obtain a refutation procedure we also need a search rule. The idea behind the search rule is that, given a logic program $P$:

- Start with what we know is true, i.e. all unit clauses in $P$.
- If $A \leftarrow B_1, ..., B_n \in P$ and $B_1, ..., B_n$ are known to be true then $A$ is true. [6]
- Repeat until no new information can be obtained.

The set obtained by repeating these actions with respect to $P$ is called a *fixpoint*. It is equal to the least Herbrand model of $P$ if Herbrand interpretations are extended to include non-ground atoms.

### 4.5.1 Naïve interpreter

The simplest method of constructing the fixpoint with respect to a logic program is to blindly check all rules and collect those whose bodies are satisfied in the current iteration. This method goes by the name naïve evaluation.

[6] Formally speaking this is a sequence of unit-resolution steps when using binary resolution. Alternatively we can use unit-resolution with full resolution and do it in one step.

**Program.** Naïve interpreter for logic programs.

```
prove(Goal) ←
        iterate([], Fixpoint),
        member(Goal, Fixpoint).
iterate(I, Fixpoint) ←
        next(I, I1),
        (     I1 = I ->
              Fixpoint = I
        ;     iterate(I1, Fixpoint)
        ).
next(I0, I) ←
        findall(H, (rule(H, B), satisfy(B, I0)), I1),
        append(I0, I1, I2),
        sort(I2, I).
```

The call to *sort*/2 in *next*/2 is necessary, otherwise the iteration would never run out of steam because duplicate atoms would be added in each iteration. This program is not included in the evaluation since it only serves as a stepping stone towards the semi-naïve interpreter.

### 4.5.2 Semi-naïve interpreter

The main reason for the inefficiency of the naïve interpreter is that all clauses are tried in every iteration, i.e. if $A \leftarrow B_1, ..., B_n$ is proven to be true in iteration $i$ it will be proven true yet again in iteration $i+1$. This can be avoided if the newly obtained facts are kept in a separate argument which denotes the difference between the previous iteration and the current iteration. Then a head of a clause is only collected if its body is satisfied by some member of this set.

**Program.** Semi-naïve interpreter for logic programs. [7]

```
prove(Goal) ←
        findall(F, rule(F, []), Fs),
        iterate(Fs, [], Fixpoint),
        member(Goal, Fixpoint).
iterate(I, DI, Fixpoint) ←
        next(I, DI, Next_I, Next_DI),
        (     NextDI = [] ->
```

[7] Based on BUP 1.0 by Ulf Nilsson.

```
        Fixpoint = Next_I
    ;   iterate(Next_I, Next_DI, Fixpoint)
    ).
next(I, DI, Next_I, Next_DI) ←
    findall(H,
        (
         rule(H, B),
         satisfy_one(B, DI),
         satisfy_all(B, I),
         not member(H, I)
        ),
        Tmp),
    sort(Tmp, Next_DI),
    append(I, Next_DI, Next_I).
```

### 4.5.3 Magic transformation

While the semi-naïve interpreter is a large improvement over the naïve interpreter it is still very wasteful compared to its top-down cousins. Facts are blindly generated even if there is no chance of them ever being used to resolve the goal clause. The *magic transformation* of a logic program is a method for encoding those clauses that would have been used in a top-down refutation with a special prefix.

**Example.** Given the logic program:

```
edge(a, b).
edge(b, c).
edge(i, j).
```

$\text{connected}(X_1, Y_1) \leftarrow \text{edge}(X_1, Y_1).$
$\text{connected}(X_2, Y_2) \leftarrow \text{edge}(X_2, X_2 1), \text{connected}(X_2 1, Y_2).$

One can reason as follows with regards to what clauses would have been proven true in a top-down refutation:

- The edge/2 facts are true if they are called.
- The first clause of connected/2 is true if $\text{connected}(X_1, Y_1)$ is called and $\text{edge}(X_1, Y_1)$ is true.
- $\text{edge}(X_1, Y_1)$ will be called if $\text{connected}(X_1, Y_1)$ is called.

- The second clause of connected/2 is true if connected($X_2, Y_2$) is called and its body is true.
- edge($X_2, X_2 1$) will be called if connected($X_2, Y_2$) is called.
- connected($X_2 1, Y_2$) will be called if connected($X_2, Y_2$) is called and edge($X_2, X_2 1$) is true.

A clause will not be proven true if it is not called. If a program is transformed so that this information is encoded in its clauses a bottom-up interpreter will only collect the facts that would have been proven true in a top-down interpreter. The following definition from Małuszyński and Nilsson [14] formalizes the magic completion of a logic program:

**Definition.** Let $P$ be a logic program. Then $magic(P)$ is the smallest logic program such that if $A_0 \leftarrow A_1, ..., A_n \ \in P$ then:

- $A_0 \leftarrow call(A_0), A_1, ..., A_n \in magic(P)$.
- $call(A_i) \leftarrow call(A_0), A_1, ..., A_{i-1} \in magic(P)$ for each $1 \leq i \leq n$.

Where $call(A)$ denotes the new atom $call\_A$ and should be read as "A is called".

The magic completion of the program from the previous example is:

edge(a, b) ← call_edge(a, b).
edge(b, c) ← call_edge(b, c).
edge(i, j) ← call_edge(i, j).
connected($X_0$, $Y_0$) ←
    call_connected($X_0$, $Y_0$),
    edge($X_0$, $Y_0$)
call_edge($X_0$, $Y_0$) ←
    call_connected($X_0$, $Y_0$).
connected($X_1$, $Y_1$) ←
    call_connected($X_1$, $Y_1$),
    edge($X_1$, $X_1 0$),
    connected($X_1 0$, $Y_1$).
call_edge($X_1$, $X_1 0$) ←
    call_connected($X_1$, $Y_1$).
call_connected($X_1 0$, $Y_1$) ←
    call_connected($X_1$, $Y_1$),
    edge($X_1$, $X_1 0$).

If the semi-naïve interpreter is started with the fact $call\_connected(a, c)$ this would yield the new facts $call\_edge(a, c)$ and $call\_edge(a, X_1 0$ in the first iteration. This allows other rules to fire and the fact $connected(a, c)$ will eventually be reached. The interesting thing is that only those clauses that would have been used in a top-down computation will be considered, e.g. the $edge(i, j)$ fact will never be collected since it will not be used.

Given a logic program $P$ and a goal clause $\leftarrow G$ one then calculates the fixpoint with respect to $magic(P) \cup call(G)$ with the naïve or semi-naïve interpreter. The result is a bottom-up computation which only generates the facts that a top-down computation would use. Calculating the magic completion of a logic program is straightforward:

**Program.** Magic completion of a logic program defined by a set of rule/2 clauses. [8]

```
magic_completion ←
    findall(rule(MagicH, MagicB),
            (
             rule(H, B),
             magicise(H, B, MagicH, MagicB)
            ),
            MagicRules).
magicise(Head, Body, Head, [X|Body]) ←
    magic(Head, X).
magicise(Head, Body, NewHead, [X|Left]) ←
    magic(Head, X),
    append(Left, [Y|_], Body),
    magic(Y, NewHead).
magic(X, Y) ←
    nonvar(X),
    term_to_atom(X, Atom),
    atom_concat(call_, Atom, Y).
```

## Adding stratified negation

When a body of a rule contains a negative literal $\neg G$ there are three possible outcomes:

- $G$ is already known to be true. Then $\neg G$ is false.
- $G$ is not currently true. Then the truth value of $\neg G$ is unknown.

[8] Based on BUP 1.0 by Ulf Nilsson.

- $G$ is not currently true and will never become true. Then $\neg G$ is true.

As mentioned in chapter 3 we must impose the restriction that programs are stratified in order to determine when a negative goal is true. Then a fixpoint can be generated for each stratum and a negative literal is deemed true if it is false in the previous fixpoint.

**Program.** Part of a bottom-up interpreter for general logic programs.

```
prove(Goal) ←
    findall(F, rule(F, []), Fs),
    prove(Fs, Fs, Fixpoint),
    member(Goal, Fixpoint).
prove(I, DI, Fixpoint) ←
    iterate_intermediate(I, DI, [], Pending, Fixpoint0),
    (   Pending = [] ->
        Fixpoint = Fixpoint0
    ;   satisfy_negative_literals(Pending, Fixpoint0, Satisfied),
        union(Fixpoint0, Satisfied, Fixpoint1),
        prove(Fixpoint1, Satisfied, Fixpoint)
    ).
```

The predicate *iterate_intermediate*/5 is *iterate*/3 from the semi-naïve interpreter augmented with arguments that keeps track of the pending negative goals whose truth-value have not yet been determined. The predicate *satisfy_negative_literals*/3 takes the pending literals as input and checks whether they are satisfiable in $Fixpoint0$. If a literal $not(L)$ is satisfied, i.e. $L \notin Fixpoint0$, then the new fact $not(L)$ is added to $I$ and $DI$ and the next intermediate fixpoint is calculated recursively.

**Example.** Let $P$ be the logic program:

```
man(a).
man(b).
married(a).
bachelor(X) ← man(X), not(married(X)).
```

And $G$ the goal clause $\leftarrow bachelor(X)$.

The first intermediate fixpoint is $I = [man(a), man(b), married(a)]$. Since $not(married(a))$ is known to be false the only pending negative literal is $not(married(b))$. Since $not(married(b))$ is satisfiable in $I$ it is added to the

new $I$ and $DI$ before the next fixpoint calculation starts. In the next iteration $bachelor(b)$ is true, and since there are no more negative goals pending the computation terminates.

## Magic transformation of a general logic program

Since the interpreter handles all the hard work with negative literals only small changes to the previous definition of magic completion are necessary. Everything that needs to be said about a negative literal $not(G)$ is that it is true if $G$ is called and $not(G)$ is true.

**Definition.** Let $P$ be a general logic program. Then $magic$(P) is the smallest logic program such that if $A \leftarrow L_1, ..., L_n \ \in P$ then:

- $A \leftarrow call(A), L_1, ..., L_n \in magic(P)$.
- If $L_i$ is a positive literal then $call(L_i) \leftarrow call(A), L_1, ..., L_{i-1} \in magic(P)$ for each $1 \leq i \leq n$.
- If $L_i$ is a negative literal $\neg A_i$ then $call(A_i) \leftarrow call(A), L_1, ..., L_{i-1} \in magic(P)$ for each $1 \leq i \leq n$.

The Argus-eyed reader might notice something unnerving with this definition. If a rule contains both an atom and its complement in its body, something might be lost in translation since the new magic predicate does not remember if the original literal was negative or positive.

**Program.** Consider the following general logic program:

a.
f ← not(a), a.

Its magic completion is:

a ← call_a.
f ← call_f, not(a), a.
call_a ← call_f.
call_a ← call_f, not(a).

The call to $a$ depends of $call\_a$ which in turn depends on $not(a)$. The program is no longer stratified and our previous implementation of negation is no longer guaranteed to work! There exists a multitude of solutions for this problem. The simplest workaround is to simply remove the negative

dependency in $call\_a$. The effect is that the second literal in the body of $f$ might be called even if $not(a)$ is proven false. Hence more calls will potentially be made in vain, but the transformed program is guaranteed to be stratified if the original program is. For a more efficient solution albeit mainly intended for Datalog, see for example Ramakrishnan et al. [15].

# Chapter 5

# Evaluation

## 5.1 Purpose

The purpose of these benchmarks is to identify strengths and shortcomings of the interpreters presented in chapter 4. They are not intended to provide an accurate performance comparison for a category of interpreters versus another, since the results are skewed by the fact that interpreters which are similar to the underlying Prolog system can borrow more features from it.

## 5.2 Method

A warning to the reader — this evaluation, like any other, is not completely fair. A perfectly fair evaluation would require tailor-written programs for each interpreter and evaluate each program with regards to both performance and to how much effort that must be put into writing the code. However interesting this would be it is hardly practical due to time constraints. The programs are instead kept as simple as possible, and whenever deemed interesting benchmarked in permutations of both clauses and goals in bodies.

Each benchmark starts with a short description followed by a list of goals. Each permutation of a program is presented in a table with a goal in every column. When a goal is intended to fail it is marked as (F) in the table. For simplicity only ground goals are used in most of the benchmarks.

## 5.3 Tools

The software tools used in these benchmarks are SWI-Prolog[1] 5.8.0 and Logtalk 2.39.1.

## 5.4 Metric

The only metric taken into account is the number of logical inferences it takes for an interpreter to find the first proof of the goal or reject it, where the number of logical inferences is counted in the Prolog system and not in the interpreter itself. The number of logical inferences is in SWI-Prolog defined as the number of passes via the call and redo port. What does this mean? A single resolution step amounts to one inference. If a clause fails and the next one needs to be tried this also counts as one inference.

**Example.** Number of logical inferences for a simple logic program.

f ← a, b, fail.
f ← a, b.
a.
b ← a.

To solve $f/0$ with Prolog's computation and search rule 8 inferences are necessary.

1. The uppermost clause of $f/0$ is called.
2. $a$ is called and solved.
3. $b$ is called.
4. $b$ is solved.
5. $f/0$ fails and a redo instruction is issued which calls the second clause of $f/0$.
6. $a$ is called and solved.
7. $b$ is called.
8. $b$ is solved.

[1] http://www.swi-prolog.org

Logical inferences does not perfectly correspond to CPU time for several reasons. First, failed unifications are not taken into account, i.e. it is possible to spend a lot of time doing database lookups[2] without spending more than a single inference. Second, if a predicate works as a wrapper for a procedure written in another language, e.g. *sort*/2, it only counts as a single inference.

Whence as already disclaimed direct comparisons between interpreters that use different data structures and different built-in methods are bound to be skewed. It is however still a useful metric for determining how interpreters perform with respect to different types of logic programs. Moreover it is also very simple to measure in most cases and does not fluctuate depending on the hardware.

## 5.5 The interpreters under evaluation

The interpreters are more or less those presented in chapter 4, albeit often more efficiently implemented. The naïve bottom-up interpreter is excluded since it does not have any advantages compared to the semi-naïve interpreter.

- Depth first (DFS).
- Breadth first (BFS).
- Iterative deepening depth-first (IDDFS) with an increment of 32. [3]
- Semi-naïve bottom-up with magic transformation (BUP).
- Greedy best-first (GBFS).
- A*.
- Optimal weighted A* ($A_{w1}$*) with $w = 1/5$.
- Suboptimal weighted A* ($A_{w2}$*) with $w = 99/100$.

For convenience the following table summarizes the formal properties of the search rule in use by each top-down interpreter. A search rule is *complete* if each goal node in the SL-tree is eventually reached. It is *optimal* if it reaches goal nodes in depth-descending order.

[2] Assuming that indexing is not applicable.

[3] Why 32? Lower increments makes the IDDFS-interpreter unreasonably inefficient in many cases.

| Interpreter | Complete | Optimal |
|---|---|---|
| DFS | | |
| BFS | ⊤ | ⊤ |
| IDDFS | ⊤ | |
| GBFS | | |
| A* | ⊤ | ⊤ |
| $A_{w1}$* | ⊤ | ⊤ |
| $A_{w1}$* | ⊤ | |

## 5.6 Benchmarks

The full source code for all benchmark programs and permutations may be found in appendix A.

**Benchmark 1.** Naïve reverse.
goal1(nrev("abcde", "edcba")).
goal2(nrev("abcdefghij", "jihgfedcba")).

The first benchmark is the infamous naïve reverse benchmark. It is not a very interesting metric on its own but serves to show the overhead of the interpreters compared to each other.

| Perm 1 | Goal1 | Goal2 |
|---|---|---|
| DFS | 74 | 209 |
| BFS | 558 | 1728 |
| IDDFS | 76 | 503 |
| BUP | 20689 | 152334 |
| GBFS | 941 | 2876 |
| A* | 941 | 2876 |
| $A_{w1}$* | 941 | 2876 |
| $A_{w2}$* | 941 | 2876 |

In the first goal DFS and IDDFS perform almost equally well. In the second goal the bound for IDDFS is exceeded which degrades its performance somewhat. BUP is by far the worst performing. The reason is that the implementation cannot borrow as much by the underlying Prolog system, and much of the time is spent on operations that could be performed in expected constant time in a real implementation. The heuristic interpreters perform identically which is expected for deterministic code.

In the second permutation the order of the goals in the body of nrev/2

is swapped.

| Perm 2 | Goal1 | Goal2 |
|---|---|---|
| DFS | 74 | 209 |
| BFS | 668 | 1948 |
| IDDFS | 76 | 539 |
| BUP | 22069 | 168364 |
| GBFS | 1064 | 3094 |
| A* | 1064 | 3094 |
| $A_{w1}$* | 1064 | 3094 |
| $A_{w2}$* | 1064 | 3094 |

Not much happens since all arguments are bound.

**Benchmark 2.** Graph search.

```
goal1(connected(1, 2)).
goal2(connected(1, 3)).
goal3(connected(1, 4)).
goal4(connected(1, 5)).
goal5(connected(1, 6)).
goal6(connected(1, 7)).
goal7(connected(1, 8)).
goal8(connected(1, 9)).
```

The edge/2 collection used may be found in appendix A. To keep things simple the graph is not cyclic since this could prevent the DFS interpreter from completing the queries.

| Perm 1 | Goal1 | Goal2 | Goal3 | Goal4 | Goal5 | Goal6 | Goal7 | Goal8 |
|---|---|---|---|---|---|---|---|---|
| DFS | 387 | 963 | 168 | 384 | 165 | 165 | 72 | 84 |
| BFS | 328 | 170 | 636 | 322 | 1294 | 624 | 1226 | 1226 |
| IDDFS | 389 | 965 | 170 | 386 | 167 | 167 | 74 | 86 |
| BUP | 24356 | 23110 | 27969 | 25685 | 33224 | 30078 | 40167 | 37708 |
| GBFS | 373 | 146 | 639 | 373 | 946 | 639 | 917 | 917 |
| A* | 426 | 146 | 931 | 402 | 2075 | 957 | 2194 | 2194 |
| $A_{w1}$* | 569 | 210 | 1342 | 545 | 2959 | 1265 | 2876 | 2876 |
| $A_{w2}$* | 373 | 146 | 639 | 373 | 1218 | 639 | 1188 | 1188 |

Finally some interesting results! BFS and the heuristic interpreters perform quite well since they are able to find shorter paths to the goal node. Unweighted A* does not perform as well as GBF and $A_{w2}$* since it spends time switching between multiple paths instead of just pursuing the path with the estimated lowest cost to the goal node.

| Perm 2 | Goal1 | Goal2 | Goal3 | Goal4 | Goal5 | Goal6 | Goal7 | Goal8 |
|---|---|---|---|---|---|---|---|---|
| DFS | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| BFS | 381 | 170 | 801 | 378 | 1053 | 792 | 1924 | 1406 |
| IDDFS | 709 | 478 | 1598 | 1068 | 1557 | 2391 | 6464 | 3511 |
| BUP | 5065 | 3519 | 8592 | 6563 | 10464 | 10436 | 17445 | 14484 |
| GBFS | 299 | 146 | 553 | 299 | 782 | 553 | 1623 | 952 |
| A* | 347 | 146 | 680 | 327 | 909 | 708 | 1833 | 1294 |
| $A_{w1}$* | 569 | 210 | 1176 | 535 | 1613 | 1119 | 3037 | 2148 |
| $A_{w2}$* | 299 | 146 | 553 | 299 | 782 | 553 | 1623 | 952 |

Here the goals in the first clause of *connected*/2 are swapped. DFS draws the shortest straw since the SL-tree is infinite, but the performance of IDDFS is also degraded. BFS is hardly affected but the heuristic interpreters and BUP are actually improving somewhat.

| Perm 3 | Goal1 | Goal2 | Goal3 | Goal4 | Goal5 | Goal6 | Goal7 | Goal8 |
|---|---|---|---|---|---|---|---|---|
| DFS | 45 | 36 | 54 | 45 | 99 | 54 | 63 | 63 |
| BFS | 272 | 114 | 580 | 266 | 1238 | 568 | 1170 | 1170 |
| IDDFS | 47 | 38 | 56 | 47 | 101 | 56 | 65 | 65 |
| BUP | 24358 | 23142 | 27995 | 25703 | 33248 | 30088 | 40183 | 37732 |
| GBFS | 373 | 146 | 639 | 373 | 911 | 911 | 1190 | 1190 |
| A* | 426 | 146 | 952 | 402 | 2081 | 928 | 2150 | 2150 |
| $A_{w1}$* | 569 | 210 | 1332 | 545 | 2958 | 1269 | 2881 | 2881 |
| $A_{w2}$* | 373 | 146 | 639 | 373 | 1200 | 914 | 1788 | 1788 |

Now the clauses of connected/2 are swapped, allowing DFS to terminate early and outclass the competition.

| Perm 4 | Goal1 | Goal2 | Goal3 | Goal4 | Goal5 | Goal6 | Goal7 | Goal8 |
|---|---|---|---|---|---|---|---|---|
| DFS | 45 | 36 | 60 | 45 | 75 | 60 | 108 | 84 |
| BFS | 272 | 114 | 580 | 266 | 807 | 568 | 1395 | 1045 |
| IDDFS | 47 | 38 | 62 | 47 | 77 | 62 | 110 | 86 |
| BUP | 5065 | 3519 | 8592 | 6563 | 10464 | 10436 | 17445 | 14484 |
| GBFS | 299 | 146 | 553 | 299 | 782 | 553 | 1623 | 952 |
| A* | 347 | 146 | 680 | 327 | 909 | 708 | 1833 | 1294 |
| $A_{w1}$* | 569 | 210 | 1167 | 535 | 1602 | 1104 | 3040 | 2147 |
| $A_{w2}$* | 299 | 146 | 553 | 299 | 782 | 553 | 1623 | 952 |

In the last permutation both the clauses and the goals are swapped. The result is basically the same with the difference that BUP has improved.

**Benchmark 3.** Calculating the N:th Fibonacci number recursively.
goal1(fib_rec(5, 5)).
goal2(fib_rec(20, 6765)).

This is the first of two benchmarks computing the fifth and twentieth Fibonacci number.

| Perm 1 | Goal1 | Goal2 |
|---|---|---|
| DFS | 105 | 121786 |
| BFS | 803 | 1204080 |
| IDDFS | 119 | 90217200 |
| BUP | 8112 | 129988 |
| GBFS | 1328 | 2276571 |
| A* | 1353 | 2168622 |
| $A_{w1}$* | 1378 | 2104348 |
| $A_{w2}$* | 1328 | 2281922 |

Compared to the other benchmarks IDDFS performs quite poorly since it has to repeat a lot of work due to the two recursive references in the benchmark program. BUP performs admirably well in the second goal since it never has to recompute any values. If $fib\_rec(I)$ is already known to be true a simple lookup is all that is necessary. This should be contrasted with the top-down interpreters that have to prove the second $fib\_rec/2$ goal true again and again. For larger values of $N$ BUP would be the only interpreter that would finish in a reasonable amount of time since the time complexity of $fib\_rec/2$ is reduced to $O(N)$.

**Benchmark 4.** Calculating the N:th Fibonacci number iteratively.
goal1(fib_iter(5, 5)).
goal2(fib_iter(20, 6765)).

Instead of using two recursive references an accumulator is used. This prevents duplicate goals to be calculated in vain.

| Perm 1 | Goal1 | Goal2 |
|---|---|---|
| DFS | 75 | 225 |
| BFS | 424 | 1984 |
| IDDFS | 77 | 549 |
| BUP | 3252 | 19002 |
| GBFS | 700 | 3280 |
| A* | 700 | 3280 |
| $A_{w1}$* | 700 | 3280 |
| $A_{w2}$* | 700 | 3280 |

The results are as expected. The IDDFS-interpreter in particular performs much better since it does not have to recompute quite as much.

**Benchmark 5.** Determining whether two binary trees are isomorphic.
goal1(isotree(T, IsoT)) ← tree1(T), tree1_iso(IsoT).
goal2(isotree(T, IsoT)) ← tree2(T), tree2_iso(IsoT).
goal3(isotree(T, NonIsoT)) ← tree3(T), tree3_non_iso(NonIsoT).

See appendix A for further details regarding $tree_i/1$ predicates. The first two goals are benchmarked as normal, counting inferences until the first proof, while the third is benchmarked with respect to the number of inferences before failure.

| Perm 1 | Goal1 | Goal2 | Goal3 |
|---|---|---|---|
| DFS | 168 | 171 | (F) 3224 |
| BFS | 304128 | 283981 | (F) 25938 |
| IDDFS | 10646 | 8123 | $\infty$ |
| BUP | 27673 | 26223 | (F) 18144 |
| GBFS | 5846 | 8079 | (F) 62720 |
| A* | 845106 | 772435 | (F) 76843 |
| $A_{w1}$* | 1242519 | 1170854 | (F) 77252 |
| $A_{w2}$* | 7953 | 9948 | (F) 62733 |

IDDFS as implemented here does not terminate when no solution exists. There are ways around this, but due to time constraints it is kept as is during the evaluation. DFS performs well with respect to Goal1 and Goal2 but Goal3 takes significantly more inferences to complete, the reason being that the troublesome node is found last which effects the interpreter to jump back to all previous choicepoints before it can safely say 'no'. Both A* and BFS perform poorly since they must pursue a lot of different paths. GBFS and $A_{w2}$* does not fall into this trap since it with lack of better heuristics they just pick the uppermost clause.

It is also interesting to note that BUP is actually more efficient with Goal3 than Goal1 and Goal2. The node not belonging to the first tree will cause the fixpoint calculation to terminate earlier than it would do otherwise.

| Perm 2 | Goal1 | Goal2 | Goal3 |
|---|---|---|---|
| DFS | 168 | 171 | (F) 188 |
| BFS | 266779 | 291955 | (F) 1436 |
| IDDFS | 7460 | 9095 | $\infty$ |
| BUP | 26528 | 27203 | (F) 6536 |
| GBFS | 19579 | 5321 | (F) 2813 |
| A* | 700062 | 799118 | (F) 3012 |
| $A_{w1}$* | 1097990 | 1196581 | (F) 3030 |
| $A_{w2}$* | 40728 | 24693 | (F) 2813 |

Here the goals in the first *isotree*/2 clause are swapped so luckily that the performance for Goal3 is improved for all interpreters, while BUP, GBFS and $A_{w2}$* perform worse with the first two goals. This does of course not mean that this ordering is better or worse than the original. Tree isomorphism is a problem where the ordering of clauses only affects performance for specific trees. There is no ordering that is guaranteed to give better performance in the general case.

**Benchmark 6.** Parsing a simple English sentence with a DCG.
goal1(sentence([the, corpulent, man, contains, a, decorated, pieplate], [])).
goal2(sentence([the, corpulent, man, contains, a, decorated, platepie], [])).

| Perm 1 | Goal1 | Goal2 |
|---|---|---|
| DFS | 99 | (F) 89 |
| BFS | 669 | (F) 629 |
| IDDFS | 101 | $\infty$ |
| BUP | 31616 | (F) 21620 |
| GBFS | 1176 | (F) 1174 |
| A* | 1188 | (F) 1159 |
| $A_{w1}$* | 1209 | (F) 1153 |
| $A_{w2}$* | 1194 | (F) 1192 |

The large number of clauses in the program severely impacts BUP's performance. Otherwise the results are consistent with earlier benchmarks.

**Benchmark 7.** Solving the MU-puzzle from Gödel, Escher, Bach.
test_theorem([m, i, u, i, u, i, u, i, u, i, u, i, u, i, u, i, u]).

test_non_theorem([m, i, u, i, u, i, u, i, u, i, u, i, u, i, u, x, u]).

goal1(theorem(4, T)) :- test_theorem(T).
goal2(theorem(4, T)) :- test_non_theorem(T).

The MU-puzzle is an example of a simple formal system with rules for manipulating theorems. The first goal tests a valid theorem while the second tests an invalid one.

| Perm 1 | Goal1 | Goal2 |
|---|---|---|
| DFS | 1059 | (F) 1949 |
| BFS | 15394 | (F) 15460 |
| IDDFS | 2607 | $\infty$ |
| BUP | 703850 | (F) 594338 |
| GBFS | 32447 | (F) 36696 |
| A* | 38768 | (F) 39231 |
| $A_{w1}$* | 39673 | (F) 39805 |
| $A_{w2}$* | 34403 | (F) 36641 |

Again the large number of clauses prevents BUP from competing. It still manages to reject Goal2 faster than Goal1 though, in contrast to the other interpreters.

| Perm 2 | Goal1 | Goal2 |
|---|---|---|
| DFS | 1890 | (F) 1949 |
| BFS | 15508 | (F) 15460 |
| IDDFS | 3438 | $\infty$ |
| BUP | 703890 | (F) 594370 |
| GBFS | 33936 | (F) 36309 |
| A* | 38935 | (F) 39176 |
| $A_{w1}$* | 39647 | (F) 39785 |
| $A_{w2}$* | 34965 | (F) 37029 |

Here the rule order is reversed. It degrades performance for all interpreters except BUP.

**Benchmark 8.** Solving the 4-queens puzzle.
goal1(queens(4, [2, 4, 1, 3])).
goal2(queens(4, [2, 4, 3, 1])).

The N-queens puzzle is another classic benchmark. The version used here is based on the generate and test paradigm.

| Perm 1 | Goal1 | Goal2 |
|---|---|---|
| DFS | 547 | (F) 749 |
| BFS | 6115 | (F) 6057 |
| IDDFS | 1060 | $\infty$ |
| BUP | 333167 | (F) 308980 |
| GBFS | 13169 | (F) 13220 |
| A* | 13230 | (F) 13190 |
| $A_{w1}$* | 13179 | (F) 13139 |
| $A_{w2}$* | 13281 | (F) 13238 |

Heuristics does not help here. DFS manages to find the solution much quicker even when the overhead of the interpreters is taken into account.

| Perm 2 | Goal1 | Goal2 |
|---|---|---|
| DFS | 411 | (F) 749 |
| BFS | 6096 | (F) 6057 |
| IDDFS | 925 | $\infty$ |
| BUP | 333175 | (F) 308988 |
| GBFS | 13312 | (F) 13269 |
| A* | 13317 | (F) 13274 |
| $A_{w1}$* | 13307 | (F) 13259 |
| $A_{w2}$* | 13192 | (F) 13243 |

The clause order of select/3 is swapped. It does not affect the performance however.

**Benchmark 9.** Database test.
goal1(query([ethiopia, 77, mexico, 76])).
goal2(query([france, 246, iran, 628])).

A database test for finding related geographical regions.

| Perm 1 | Goal1 | Goal2 |
|---|---|---|
| DFS | 69 | (F) 38 |
| BFS | 369 | (F) 231 |
| IDDFS | 71 | $\infty$ |
| BUP | 11186 | (F) 10010 |
| GBFS | 597 | (F) 380 |
| A* | 597 | (F) 380 |
| $A_{w1}$* | 597 | (F) 380 |
| $A_{w2}$* | 597 | (F) 380 |

Yet again the large number of clauses prevents BUP from competing. The other numbers are fairly consistent.

**Benchmark 10.** Comparison of negation performance between BUP and DFS.
goal1(member(0'e, "abcde")).
goal2(member(0'e, "abcdd")).
goal3(nonmember(0'e, "abcdd")).
goal4(nonmember(0'e, "abcde")).

The purpose of this benchmark is to compare how negation as failure affects performance when implemented in a top-down interpreter versus a bottom-up interpreter. Since all top-down interpreters handle negation in the same way only DFS is included in the benchmark.

| Perm 1 | Goal1 | Goal2 | Goal3 | Goal4 |
|---|---|---|---|---|
| DFS | 45 | (F) 29 | 73 | (F) 62 |
| BUP | 2868 | (F) 1556 | 8864 | (F) 6426 |

Negated goals are not handled as efficiently in BUP since the fixpoint construction is suspended when negative goals are pending. Goal3 takes trice times as many inferences as Goal1 in BUP, while in DFS it is not even doubled.

**Benchmark 11.** Quality of solutions.
In this benchmark the solutions produced by the various interpreters will be taken into account. This metric is of course only relevant for applications where multiple solutions exists and it is possible to rank them. A typical application fulfilling these requirements is a planner. The domain used is the blocks world problem — given an initial configuration of blocks the planner tries to find a sequence of legal actions that results in the final state. To simplify things for the depth-first interpreter the planner does not allow duplicate states. The initial state and final state used are:

initial_state([on(a, b), on(b, p), on(c, r)]).
final_state([on(a, b), on(b, c), on(c, r)]).

The first plans produced by the interpreters were:

**DFS**

[to_place(a, b, q), to_block(a, q, c), to_place(b, p, q), to_place(a, c, p), to_block(a, p, b), to_place(c, r, p), to_place(a, b, r), to_block(a, r, c), to_place(b, q, r), to_place(a, c, q), to_block(a, q, b), to_place(c, p, q), to_place(a, b, p), to_block(a, p, c), to_place(b, r, p), to_place(a, c, r), to_block(b, p, a), to_place(c, q, p), to_block(b, a, c), to_place(a, r, q), to_block(b, c, a), to_place(c, p, r), to_block(b, a, c), to_place(a, q, p), to_block(a, p, b)]

**BFS**
[to_place(a, b, q), to_block(b, p, c), to_block(a, q, b)]

**IDDFS**
[to_place(a, b, q), to_block(b, p, a), to_block(b, a, c), to_block(a, q, b)]

**BUP** (modified to halt the fixpoint calculation whenever a solution appears)
[to_place(a, b, q), to_block(b, p, c), to_block(a, q, b)]

**GBFS**
[to_place(a, b, q), to_block(b, p, c), to_block(a, q, b)])

**A***
[to_place(a, b, q), to_block(b, p, c), to_block(a, q, b)]

**$A_{w1}$***
[to_place(a, b, q), to_block(b, p, c), to_block(a, q, b)]

**$A_{w2}$***
[to_place(a, b, q), to_block(b, p, c), to_block(a, q, b)]

Not all plans are created equal. The plan produced by the depth-first interpreter is horrendously abysmal with a length of 25 steps. All heuristic interpreters find the same plan, but it should be noted that there is no guarantee that GBFS finds a good one. Iterative deepening does a respectable job with a plan consisting of only four steps. If the increment is kept relatively low it should on average find good solutions even if they are not optimal.

Like all the other benchmarks this is a simplification of reality. But even though the planner is very simple and not suitable for anything else than toy-problems it still shows that it is possible for alternative search rules to produce more satisfactory answers.

# Chapter 6

# Conclusions

This chapter will discuss the merits of each interpreter and extrapolate the interesting results from chapter 5. Care must be taken to not incorrectly induce a general statement by just staring at raw numbers. The interpreters are intended as prototypes and may as such not always produce accurate statistics. For instance it is not very interesting to measure the exact overhead that the breadth-first interpreter has over the depth-first interpreter, but it is interesting to analyze if this ratio changes depending on the logic program in question. The same is also true for the heuristic interpreters and the bottom-up interpreter.

## 6.1 DFS and IDDFS

As expected the depth-first interpreter is very fast in most applications. This is not really a virtue of the search strategy per se, but a depth-first search can be implemented very efficiently without the need to explicitly expand multiple branches. It is however very sensitive to clause order and is not complete for infinite trees. The latter problem is solved by adding iterative deepening, but this adds a whole slew of problems. For some programs the approach works just fine, but depending on how much work that needs to be redone it can in some cases be unreasonably inefficient. One can improve the performance in these instances by increasing the increment of the depth-bound, but if the increment is to large early solutions might not be reported until very late on and the search basically devolves into an inefficient depth-first search.

One potential remedy is tabling. XSB[1] is a Prolog system that successfully implements a form of resolution called SLG-resolution which can dras-

[1] http://xsb.sourceforge.net/index.html

tically improve both performance and completeness. Just like the bottom-up interpreter it will not recalculate goals in vain, but it also has the advantage that it is very easy to integrate with ordinary SL-resolution.

## 6.2 BFS

The breadth-first approach work well in some cases but horrendous in others. Throughout the benchmarks it often scores better than the heuristic interpreters, but some of its advantage is due to the fact that it is easier to implement an efficient queue in Prolog than to implement an efficient heap. Its problems are best highlighted in the *isotree*/2 benchmark where both the greedy interpreter and the weighted A* interpreter with $w$ close to 1 managed to outperform it substantially. If the SL-tree has a high branching factor and does not have any goal nodes at the lower depths chances are that it will not perform very well due to the frequent switching between branches.

## 6.3 GBFS and A*

A greedy interpreter seems to work well in most cases. If the heuristic function is valued equally for two clauses the uppermost is picked first which allows it to more or less function as a depth-first search. Just like BFS A* can be expensive if there are several branches with equal probability, but if it is weighted properly its performance approaches that of GBFS. There are no advantages to decrease the value of $w$ below $1/2$ however since $h(x)$ almost always underestimates the cost by a large degree.

In the planner benchmark the plans produced by GBFS and A* were both optimal. Can this result be extrapolated? If there is a correlation between the quality of the computed answer substitution and the length of the SL-tree path that was traversed an optimal search rule will in general give better answers. If A* is weighted too hard and loses its optimality the quality of its solution can degrade, but at the same time large performance gains can be made. How $w$ should be set ultimately depends on the application. If the branching factor is high and all branches have the same cost $w$ should be close to 1, but if there is a discernible difference between branches it is better to lower the value of $w$ so that optimality is ensured.

A possible extension to $h(x)$ is to memorize the exact cost of $x$ and use this value the next time the branch is encountered. If the interpreter caches the results between queries it could in some cases directly pick the shortest path to the goal. The downside of this approach is of course memory usage.

The *append*/3 program might for instance be called millions of times during the execution of a program. To save the cost of every goal is not exactly feasible and would probably reduce the overall performance. It is instead tempting to calculate an average value for each predicate, e.g. if *append*/3 has been called 3 times and the required inferences were 5, 4 and 10 the average would be $(5 + 4 + 10)/3 = 6$. This would provide a more accurate heuristic but with the cost that it is no longer admissible, whence optimality for A* is lost.

## 6.4 BUP

The BUP interpreter has many merits. The key differentiating factor between it and the top-down methods is that it never has to recompute goals. It also handles infinite branches more satisfactory in the cases where a fixpoint may be reached since even the complete top-down interpreters will get stuck after all finite goals have been found. The downside is that a bottom-up interpreter in general is harder to implement efficiently and if used with general logic programs the overhead is even higher. The benchmarks done by Swift and Warren [17] suggests that a top-down interpreter based on SLG-resolution can be much more efficient than a bottom-up interpreter with magic transformation. Benchmarks of this kind are of course skewed since the systems under evaluation are implemented by different teams with different goals, which is also acknowledged in the paper. For the BUP interpreter to be useful in practice the performance problems have to be sorted out. Some of the problems are easy to fix by simply using better data structures, but other issues such as scaling well to large (general) programs needs more ingenuity.

## 6.5 Concluding remarks

The question that effected the thesis was if alternative search rules and unit-resolution is a worthwhile endeavor. The answer is: "it depends". For programming, none of the surveyed search rules for SL-resolution is adequate as a general replacement. But there are more sides to the coin. We may coarsely judge an interpreter by four criterions.

1. Soundness — are all answers correct?
2. Completeness — can all correct answers be computed?
3. Performance — how much does it cost to compute an answer?

4. Ease of writing — does the interpreter allow us to write more succinct programs?

All interpreters are by this definition sound for definite logic programs, but none of them are sound for general logic programs since no check is made to see whether negated goals contains unbound variables.

An interpreter can avoid infinite branches if it is complete. The question is how useful this is. Sometimes an infinite branch denotes that there is a bug in the program and that it should be reordered, but there are exceptions. As an example we return to *connected*/2 from appendix A. That an interpreter is able to produce answers even when the program is ordered such that Prolog would go into an infinite loop is hardly of any use since it only takes a few seconds to reorder the program for maximum performance. If the *edge*/2 collection contains cycles the scenario becomes more interesting. Then the complete top-down interpreters would be able to find all connected nodes with no changes to *connected*/2, albeit with so severely degraded performance that it would be a better idea to rewrite the program. The bottom-up interpreter would on the other hand efficiently handle cyclic graphs since no goals are ever recomputed. Hence the conclusion is that completeness per se is not necessarily a property worth striving for, but if it is handled in such a way that infinite branches are rejected rather than pursued it can be quite useful.

When discussing performance we have a multitude of metrics available. Even if the evaluation only demonstrates that the depth-first interpreter is more efficient when counting logical inferences it is likely that the numbers would be even more favorable if CPU and memory usage is taken into account. It is unlikely that the alternative search rules surveyed would be faster in any real circumstance if the programmer is able to order clauses and goals after his/her whim.

Ease of writing is intertwined with performance. The cases that are most interesting is when we can use less code but still produce the same or better result. Yet again we return to the problem of calculating the transitive closure. The following logic program wonderfully captures the idea with only one line of code:

**Program.** Computing the transitive closure of a relation.

```
connected(X, Y) ← connected(X, X0), connected(X0, Y).
connected(a, b).
connected(b, a).
/* Insert facts here. */
```

When executed with an efficient bottom-up interpreter this program would do a respectable job. To produce an equivalent program performance-wise using one of the top-down interpreters one has to turn to the literature and implement e.g. the Floyd-Warshall algorithm which is significantly more job [8].

When the quality of a computed answer substitution correlates with the length of the refutation the alternative search rules begin to shine. Although a more extensive study is necessary to pinpoint the programs fulfilling this requirement the planning benchmark at least suggest that it is possible. If optimality is strictly needed then iterative deepening with an increment of 1, breadth first or unweighted A* are the best candidates. Otherwise a weigthed A* with $1/2 < w < 1$ is the best choice.

# Future work

How great is the leap from the prototypes to a useful implementation? Performance wise the best route would be to augment the Prolog system with new search rules and decide the search rule for a program through compiler directives. A more exhaustive study is of course required to judge whether it is worth the not insignificant programming effort that this solution requires. A more theoretically pleasing solution is to use a partial evaluator[2] and partially evaluate an interpreter with a logic program and a query. By the first Futamura projection we then obtain a specialized interpreter for that logic program which will run faster than the original [19]. Continuing on the same train of thought, what could be better than partially evaluating an interpreter? The partial evaluation of the partial evaluator that partially evaluates the interpreter of course! By the second Futamura projection we then obtain a *compiler* for translating a program into using another search rule if we have an *interpreter* embodying that search rule. While it is easy to get carried away into the magical land of partial evaluation an empirical investigation must be made in order to deduce how the theory aligns with the real world.

[2]The reader hungry for a practical guide to partial evaluation should chew, swallow and digest the ProMiX chapter in The Practice of Prolog [18].

# Bibliography


[1] Leon Sterling and Ehud Shapiro *The Art of Prolog.* The MIT Press, 1994.

[2] Robert A. Kowalski, *The early years of logic programming.* CACM, 1988.

[3] Alain Colmeraurer and Philippe Roussel, *The birth of Prolog.* The second ACM SIGPLAN conference on History of programming languages, 1992.

[4] M. H. Van Emden and R. A. Kowalski, *The Semantics of Predicate Logic as a Programming Language.* 1976.

[5] Mordechai Ben-Ari *Mathematical Logic for Computer Science, sixth edition.* Springer-Verlag, 2008.

[6] J. W. Lloyd *Foundations of Logic Programming, second edition.* Springer-Verlag, 1987.

[7] Kenneth A. Ross *Modular stratification and Magic Sets for Datalog Programs with Negation.* Journal of the ACM, Vol. 41, No. 6, 1994, pp.1216-1266, 1994

[8] Richard A. O'Keefe, *The Craft of Prolog.* The MIT Press, 1990.

[9] Stuart Russel and Peter Norvig, *Artificial Intelligence - A Modern Approach.* Prentice Hall, 2003.

[10] Judea Pearl, *Heuristics - Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, 1984.

[11] Paulo Jorge Lopes de Moura, *Logtalk - Design of an Object-Oriented Logic Programming Language.* 2003.

[12] Colin R Reeves, *Modern Heuristic Techniques for Combinatorial Problems.* Blackwell Scientific Publications, 1993.

[13] David Maier and David S. Warren, *Computing with Logic - Logic Programming with Prolog*. The Benjamin/Cummings Publishing Company, 1988.

[14] Jan Małuszyński and Ulf Nilsson, *Logic, Programming and Prolog, second edition*. John Wiley and Sons Ltd, 1995.

[15] Raghu Ramakrishnan, Divesh Srivastava and S. Sudarshan, *Controlling the Search in a Bottom-Up Evaluation*. University of Wisconsin-Madison, 1992.

[16] Ralph Haygood, *A Prolog Benchmark Suite for Aquarius*. University of California, 1989.

[17] Terrance Swift and David S. Warren, *Efficiently Implementing SLG Resolution*. 1994.

[18] Edited by Leon Sterling, *The Practice of Prolog*. The MIT Press 1990.

[19] Yoshihiko Futamura, *Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler*. 1971.

# Appendix A

# Benchmark code

## A.1 Benchmark 1 - Naïve reverse

```
append([], Ys, Ys) ← true.
append([X|Xs], Ys, [X|Zs]) ←
      append(Xs, Ys, Zs).

/* Permutation 1. */
nrev([], []).
nrev([X|Xs], Reversed) ←
      nrev(Xs, Reversed1),
      append(Reversed1, [X], Reversed).

/*Permutation 2. */
nrev([], []).
nrev([X|Xs], Reversed) ←
      append(Reversed1, [X], Reversed),
      nrev(Xs, Reversed1).
```

## A.2 Benchmark 2 - Graph search

```
edge(1, 3).
edge(3, 5).
edge(5, 7).
edge(7, 9).

edge(0, 2).
edge(2, 4).
```

```
edge(4, 6).
edge(6, 8).

edge(1, 0).
edge(3, 2).
edge(5, 4).
edge(7, 8).
edge(9, 8).

edge(0, 3).
edge(2, 5).
edge(4, 7).
edge(6, 9).

/* Permutation 1. */
connected(X, Z) ←
      edge(X,Y),
      connected(Y, Z).
connected(X, Y) ← edge(X, Y).

/* Permutation 2. */
connected(X, Z) ←
      connected(Y, Z),
      edge(X,Y).
connected(X, Y) ← edge(X, Y).

/* Permutation 3. */
connected(X, Y) ← edge(X, Y).
connected(X, Z) ←
      edge(X,Y),
      connected(Y, Z).

/* Permutation 4. */
connected(X, Y) ← edge(X, Y).
connected(X, Z) ←
      connected(Y, Z),
      edge(X,Y).
```

## A.3 Benchmark 5.6 - Recursive Fibonacci

Source code taken from O'Keefe [8].

```
fib_rec(1, 1).
fib_rec(2, 1).
fib_rec(N, X) ←
    N > 2,
    N1 is N - 1,
    N2 is N - 2,
    fib_rec(N1, X1),
    fib_rec(N2, X2),
    X is X1 + X2.
```

## A.4 Benchmark 4 - Iterative Fibonacci

Source code taken from O'Keefe [8].

```
fib_iter(1, 1).
fib_iter(2, 1).
fib_iter(N, X) ←
    N > 2,
    fib_iter(2, N, 1, 1, X).

fib_iter(N, N, X, _, X).
fib_iter(N0, N, X2, X1, X) ←
    N0 < N,
    N1 is N0 + 1,
    X3 is X2 + X1,
    fib_iter(N1, N, X3, X2, X).
```

## A.5 Benchmark 5 - Isomorphic trees

Source code taken from Sterling and Shapiro [1].

```
isotree(void, void).

/* Permutation 1. */
isotree(t(X, L1, R1), t(X, L2, R2)) ←
```

```
    isotree(L1, L2),
    isotree(R1, R2).
isotree(t(X, L1, R1), t(X, L2, R2)) ←
    isotree(L1, R2),
    isotree(R1, L2).

/* Permutation 2. */
isotree(t(X, L1, R1), t(X, L2, R2)) ←
    isotree(R1, R2),
    isotree(L1, L2).
isotree(t(X, L1, R1), t(X, L2, R2)) ←
    isotree(R1, L2),
    isotree(L1, R2).

/* Benchmark trees. */
tree1(t(2, t(3, t(0, t(2, void, void), t(0, t(0, void, void), t(0, t(0, void, void),
t(0, void, void)))), t(1, void, void)),t(0, t(0, t(3, void, void), t(0, void, void)),
t(2, t(4, void, void), t(3, t(2, void, void), t(3, void, void))))).

tree1_iso(t(2, t(3, t(0, t(2, void, void), t(0, t(0, void, void),
t(0, t(0, void, void), t(0, void, void)))), t(1, void, void)),
t(0, t(2, t(3, t(2, void, void), t(3, void, void)), t(4, void, void)),
t(0, t(0, void, void), t(3, void, void))))).

tree2(t(1, t(3, t(1, void, void), t(3, void, void)), t(1, t(4, t(4, void, void),
t(2, t(2, t(0, void, void), t(4, void, void)), t(4, void, void))),
t(1, t(1, t(2, void, void), t(2, void, void)), t(0, t(2, void, void), t(3, void, void))))).

tree2_iso(t(1, t(3, t(1, void, void), t(3, void, void)), t(1, t(4, t(2, t(4, void, void),
t(2, t(4, void, void), t(0, void, void))), t(4, void, void)), t(1, t(0, t(2, void, void),
t(3, void, void)), t(1, t(2, void, void), t(2, void, void))))).

tree3(t(1, t(3, t(1, void, void), t(3, void, void)), t(1, t(4, t(4, void, void),
t(2, t(2, t(0, void, void), t(4, void, void)), t(4, void, void))), t(1, t(1, t(2, void, void),
t(2, void, void)), t(0, t(2, void, void), t(3, void, void))))).

tree3_non_iso(t(1, t(3, t(1, void, void), t(3, void, void)), t(1, t(4, t(2, t(4, void, void),
t(2, t(4, void, void), t(0, void, void))), t(4, void, void)), t(1, t(0, t(2, void, void),
t(3, void, void)), t(1, t(2, void, void), t(x, void, void))))).
```

## A.6 Benchmark 6 - DCG parsing

Source code taken from Sterling and Shapiro [1], with some minor modifications to the word database.

```
sentence(A, C) ←
    noun_phrase(A, B),
    verb_phrase(B, C).

noun_phrase(A, B) ←
    noun_phrase2(A, B).
noun_phrase(A, C) ←
    determiner(A, B),
    noun_phrase2(B, C).

verb_phrase(A, C) ←
    verb(A, B),
    noun_phrase(B, C).
verb_phrase(A, B) ←
    verb(A, B).

verb([contains|A], A).
verb([eats|A], A).

noun([pieplate|A], A).
noun([surprise|A], A).
noun([man|A], A).

adjective([decorated|A], A).
adjective([corpulent|A], A).

determiner([the|A], A).
determiner([a|A], A).

noun_phrase2(A, C) ←
    adjective(A, B),
    noun_phrase2(B, C).
noun_phrase2(A, B) ←
    noun(A, B).
```

## A.7 Benchmark 7 - Solving the MU-puzzle

Source code taken from the Aquarius Prolog benchmark suite [16].

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) ←
      append(Xs, Ys, Zs).

theorem(_, [m, i]).
theorem(_, []) ← fail.
theorem(Depth, R) ←
      Depth > 0,
      D is Depth - 1,
      theorem(D, S),
      rules(S, R).

/* Permutation 1. */
rules(S, R) ← rule1(S, R).
rules(S, R) ← rule2(S, R).
rules(S, R) ← rule3(S, R).
rules(S, R) ← rule4(S, R).

/* Permutation 2. */
rules(S, R) ← rule4(S, R).
rules(S, R) ← rule3(S, R).
rules(S, R) ← rule2(S, R).
rules(S, R) ← rule1(S, R).

rule1(S, R) ←
      append(X, [i], S),
      append(X, [i,u], R).

rule2([m|T], [m|R]) ←
      append(T, T, R).

rule3([], _) ←
      fail.

rule3(R, T) ←
      append([i,i,i], S, R),
      append([u], S, T).
```

```
rule3([H|T], [H|R]) ←
    rule3(T, R).

rule4([], _) ←
    fail.

rule4(R, T) ←
    append([u,u], T, R).

rule4([H|T], [H|R]) ←
    rule4(T, R).
```

## A.8 Benchmark 8 - N-queens puzzle

Source code taken from the Aquarius Prolog benchmark suite [16].

```
queens(N, Qs) ←
    range(1, N, Ns),
    queens(Ns, [], Qs).

queens([], Qs, Qs).
queens(UnplacedQs, SafeQs, Qs) ←
    select(UnplacedQs, UnplacedQs1, Q),
    not_attack(SafeQs, Q),
    queens(UnplacedQs1, [Q|SafeQs], Qs).

not_attack(Xs, X) ←
    not_attack(Xs, X, 1).

not_attack([], _, _).
not_attack([Y|Ys], X, N) ←
    X =\= Y+N,
    X =\= Y-N,
    N1 is N+1,
    not_attack(Ys, X, N1).

/* Permutation 1. */
select([X|Xs], Xs, X).
select([Y|Ys], [Y|Zs], X) ← select(Ys,Zs,X).
```

```
/* Permutation 2. */
select([Y|Ys], [Y|Zs], X) ← select(Ys, Zs, X).
select([X|Xs], Xs, X).

range(N, N, [N]).
range(M, N, [M|Ns]) ←
    M < N,
    M1 is M+1,
    range(M1, N, Ns).
```

## A.9 Benchmark 9 - Database test

Source code taken from the Aquarius Prolog benchmark suite [16].

```
query([C1, D1, C2, D2]) ←
    density(C1, D1),
    density(C2, D2),
    D1 > D2,
    T1 is 20*D1,
    T2 is 21*D2,
    T1 < T2.

density(C,D) ←
    pop(C,P),
    area(C,A),
    D is (P*100)//A.

pop(china, 8250).
pop(india, 5863).
pop(ussr, 2521).
pop(usa, 2119).
pop(indonesia, 1276).
pop(japan, 1097).
pop(brazil, 1042).
pop(bangladesh, 750).
pop(pakistan, 682).
pop(w_germany, 620).
pop(nigeria, 613).
pop(mexico, 581).
pop(uk, 559).
```

```
pop(italy, 554).
pop(france, 525).
pop(philippines, 415).
pop(thailand, 410).
pop(turkey, 383).
pop(egypt, 364).
pop(spain, 352).
pop(poland, 337).
pop(s_korea, 335).
pop(iran, 320).
pop(ethiopia, 272).
pop(argentina, 251).
area(china, 3380).
area(india, 1139).
area(ussr, 8708).
area(usa, 3609).
area(indonesia, 570).
area(japan, 148).
area(brazil, 3288).
area(bangladesh, 55).
area(pakistan, 311).
area(w_germany, 96).
area(nigeria, 373).
area(mexico, 764).
area(uk, 86).
area(italy, 116).
area(france, 213).
area(philippines, 90).
area(thailand, 200).
area(turkey, 296).
area(egypt, 386).
area(spain, 190).
area(poland, 121).
area(s_korea, 37).
area(iran, 628).
area(ethiopia, 350).
area(argentina, 1080).
```

## A.10 Benchmark 10 - Negation test

```
eq(X, X).

member(X, [X|_]).
member(X, [_|Xs]) ←
        member(X, Xs).

nonmember(_, []).
nonmember(X, [Y|Ys]) ←
        not(eq(X, Y)),
        nonmember(X, Ys).
```

## A.11 Benchmark 11 - A planner

Source code taken from Sterling and Shapiro [1], with some small modifications in clear/2.

```
member(X, [X|_]).
member(X, [_|Xs]) ←
        member(X, Xs).

transform(State1,State2,Plan) ←
        transform(State1,State2, [State1], Plan).

transform(State,State,_,[]).
transform(State1,State2,Visited,[Action|Actions]) ←
        legal_action(Action,State1),
        update(Action,State1,State),
        not(member(State,Visited)),
        transform(State,State2,[State|Visited],Actions).

legal_action(to_place(Block,Y,Place),State) ←
        on(Block,Y,State),
        clear(Block,State),
        place(Place),
        clear(Place,State).
legal_action(to_block(Block1,Y,Block2),State) ←
        on(Block1,Y,State),
        clear(Block1,State),
```

```
        block(Block2),
        Block1 \ == Block2,
        clear(Block2,State).

clear(X,State) ← not(above(X, State)).
above(X, State) ← member(on(_, X), State).
on(X,Y,State) ← member(on(X,Y),State).

update(to_block(X,Y,Z),State,State1) ←
        substitute(on(X,Y), on(X,Z),State,State1).
update(to_place(X,Y,Z),State,State1) ←
        substitute(on(X,Y),on(X,Z),State,State1).

substitute(X,Y,[X|Xs],[Y|Xs]).
substitute(X,Y,[X1|Xs],[X1|Ys]) ←
        X = X1,
        substitute(X,Y,Xs,Ys).

block(a). block(b). block(c).
place(p). place(q). place(r).
```

# Appendix B

# Online code repository

The full Logtalk source code for the interpreters may at the time of writing be retrieved from the author's GitHub project page at:

*http://joelbyte.github.com/verdi-neruda/*

# Index

## Upphovsrätt



## Copyright