TU Wien

Institut für Informationssysteme
Abteilung für Datenbanken und Artificial Intelligence

Bachelorarbeit

Optimierung Nichtgrundierter Answer Set Programme durch Regelzerlegung

Manuel Bichler
e1127329@student.tuwien.ac.at

betreut von
Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran

November 2015

TU Wien

Institute of Information Systems
Database and Artificial Intelligence Group

Bachelor Thesis

Optimizing Non-Ground Answer Set Programs via Rule Decomposition

Manuel Bichler
e1127329@student.tuwien.ac.at

supervised by
Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran

November 2015

Acknowledgements

I would like to thank my supervisor Stefan Woltran and the research assistant Michael Morak for offering me the opportunity to work on a research project in the deep substance of the academic front line, for introducing me to other leading reasearchers in the area of ASP, and for helping me out whenever I got lost in one of the complicated matters. Their patience for me needs its own sentence of gratitude.

I would like to thank my partner, colleague and good friend Bernhard, without whose stimuli I would presumably had not been able to do this and many other work, and who supported me in any time and matter imaginable, from near and from far.

I would like to thank my parents, who I happened to manage to explain the relationship between their Sudoku books and my bachelor thesis, for personally and financially supporting me, enabling my studies and trusting in me, even in times when it is hard to do so.

And I thank all my friends and family, for they made me who I am today.

Abstract

In this work we follow a novel method originally contrived by Michael Morak and Stefan Woltran to optimize non-ground answer set programming (ASP) statements. Applying the widely-used concept of tree decompositions, we split up single statements into multiple ones with less literals and less variables each, thus exposing the structure of rules to grounders. As we show in our benchmarks, this approach can significantly reduce the workload for grounders and solvers, constituting a potential performance-boost when integrated into grounders. We extend the Morak-Woltran approach to the full ASP standard language and certain features of the Potassco system, enabling us to benchmark all ASP Competition problem instances of 2013 and 2014.

Contents

Chapter 1

Introduction

Answer Set Programming (ASP) is a declarative approach to problem solving widely used in the fields of artificial intelligence and knowledge representation. Its increasing popularity is in part the result of the increasingly efficient systems that implement the language and solve the declaratively modeled logic problems, and in part based on the high expressiveness of its modeling language.

Although state-of-the-art ASP systems have become more and more sophisticated in recent years, there is still reason to believe that their performances may be improved even further.

Most currently available ASP systems take a two-step approach: first, the declaratively modeled logic problem ("program") is *grounded*, which means that variables are instantiated with constant elements applicable to the program domain, which results in an equivalent propositional program. The second step is for a *solver* to finally solve the ground program, which may be imagined as repeatedly guessing some propositional statements as being true or false, eventually finding one or more solutions or finding that no solution exists. Systems like the *Potassco* suite *gringo/clasp* [Gebser et al., 2007] strictly separate the grounder from the solver, whereas others, such as *dlv* [Leone et al., 2006], use a higher degree of integration.

Grounding an ASP rule is in general an exponential effort in the number of variables in the rule. However, like many other real-world problems that may be represented as graph problems, many rules exhibit a certain structure which could reduce the complexity of grounding if this structure only was exploited. Using the concept of *tree decompositions*, rules with loose literal coupling, or, speaking in the language of tree decompositions, rules with small *treewidth*, can be split up into several rules with less variables each, effectively reducing the grounding time and size to be exponential in the treewidth only.

Explained in more detail, this approach performs a preprocessing of each rule in an input program. The rule is first represented as a *hypergraph*, modeling the variables as vertices and each of the rule's elements (or literals) as a hyperedge containing the element's variables. Then, a *tree decomposition* is computed on this hypergraph, utilizing already present efficient decomposition heuristics. Based on the decomposition, the rule is then split up into a set of smaller rules, semantically equal to the original rule, especially taking care of unsafe variables.

Another advantage of this approach is that it breaks, or at least weakens, the necessity of expert hand-tuning of programs in order to reduce groundings. Going along with this argument, laymen ASP programming may become less disappointing and more empowering, eventually contributing to low-threshold access to ASP and expanding its popularity.

The approach of this kind of non-ground preprocessing has, to our knowledge, so far only been investigated in [Morak and Woltran, 2012]. Morak and Woltran implemented a prototype, benchmarked it against the classical, non-preprocessed case, and presented optimistic results. Their prototype, however, only supports a limited input language, lacking the possibility of being benchmarked on certain problem instances.

Our goals are to implement a prototype, supporting all the language features of the ASP-Core-2 input language format [Calimeri et al., 2012] and selected language features of the Potassco system[Gebser et al., 2015], and to provide benchmarks for all problems of the most recent ASP competition. These benchmarks show that our rule optimization approach can indeed decrease the grounding and solving time substantially, and we thus plan to implement it into existing state-of-the-art solvers in the future.

This work is structured as follows. In chapter 2, we will present the necessary preliminaries, namely a characterization of ASP, tree decomposition and the approach by Morak and Woltran. In chapter 3, we will introduce and discuss the concrete problems we face and present solutions, especially representing a rule as a graph and rewriting it based on the decomposition, how to handle unsafe variables, how even the inner contexts of aggregates and choice rules may be decomposed, and what other preprocessing steps we will perform. Chapter 4 will then present the benchmarking results of our prototype using the instances of the fifth ASP competition. Finally, chapter 5 will close with a summary and recapitulation and will throw a glance at the upcoming scientific tasks regarding this topic.

Chapter 2

Background

In this chapter, we introduce the necessary prerequisites by giving an overview of answer set programming, tree decompositions and an already taken attempt to ASP rewriting using tree decompositions.

2.1 Answer Set Programming

In this section, we give a formal characterization of syntax and semantics of Answer Set Programming, beginning with the ground case and finishing with language extensions.

2.1.1 Ground ASP

Syntax

A *variable-free (or ground) disjunctive logic program* is a set of rules of the form

$$H_1 \vee \cdots \vee H_k \leftarrow B_1^+, \ldots, B_n^+, \text{not } B_1^-, \ldots, \text{not } B_m^-$$

where all H_i, B_i^+ and B_i^- are *atoms.* An atom is of the form $p(t_1, \ldots, t_n)$, with the predicate p having arity $n \geq 0$ and constant terms t_i. Predicates of arity 0 are called *propositions*, where parentheses may be omitted. Rules with $n = m = 0$ are called *facts* ("$\leftarrow$" may be omitted) and rules with $k = 0$ are called *constraints.* For a rule r, we denote by $H(r) = \{H_1, \ldots, H_k\}$ the set of *head atoms*, by $B^+(r) = \{B_1^+, \ldots, B_n^+\}$ the *positive body* and by $B^-(r) = \{B_1^-, \ldots, B_m^-\}$ the *negative body* of r. Likewise, we denote by $B(r) = \{B_1^+, \ldots, B_n^+, \text{not } B_1^-, \ldots, \text{not } B_m^-\}$ the *body* of r, with its elements being *literals.* A literal is therefore either an atom a or a *default-negated* atom not a (the unary connective "not" also being called *negation as failure*).

A term is a *function* of the form $f(t_1, \ldots, t_n)$ with arity $n \geq 0$ and t_i being terms themselves. Functions of arity 0 are called *constants.*

Semantics

A rule r is said to be *triggered* by a set S of atoms iff S triggers the body of r. The body of r is said to be triggered by S iff all the atoms in the positive body and none of the atoms in the negative body occur in S: S triggers r iff $B^+(r) \subseteq S$ and $B^-(r) \cap S = \emptyset$. A rule r is said to be *satisfied* by a set S of atoms iff S does not trigger it or $H(r) \cap S \neq \emptyset$. Therefore, intuitively speaking, for s to satisfy a rule, if it triggers the body, it must contain at least one of the head atoms.

A set S of atoms is said to be a *model* of a variable-free disjunctive logic program P iff it satisfies all rules of P. Given a set S of atoms, we define the *reduct* P^S of a variable-free disjunctive logic program P as $P^S = \{H(r) \leftarrow B^+(r) : r \in P, B^-(r) \cap S = \emptyset\}$. Intuitively speaking, the reduct discards rules whose negative bodies would conflict with S, keeping only the positive bodies and the heads of the resulting rules.

We finally call a set S of atoms an *answer set* of a variable-free disjunctive logic program P iff S itself is a subset-minimal model of the reduct P^S.

2.1.2 ASP with Variables

We now give a characterization of logic programs with variables. Using variables, the expressiveness of the ASP language equals first-order predicate logic.

Grounding

Generalizing the notion of ground disjunctive logic programs, in a *disjunctive logic program*, a term may no longer be only a function, but may also be a variable: A term may be a variable or a function $f(t_1, \dots, t_n)$ with arity $n \geq 0$ and t_i being terms. Functions of arity 0 are continued to be called constants.

A rule containing variables can be interpreted as an abbreviation of all possible instantiations of the rule, with the variables being replaced by domain elements, where each instantiation uses the same domain element for multiple occurrences of the same variable. This task is usually performed by a *grounder*.

We denote by $vars(l)$ the set of variables of the literal l and by $vars(t)$ the set of all variables of the term t, more formally: if v is a variable, then $vars(v) = \{v\}$; if f is an n-ary function symbol, than $vars(f(t_1, \dots, t_n)) = \bigcup_{t=t_1,\dots t_n} vars(t)$.

Example 2.1. Consider the single-rule logic program:

$$my\_grandmother(X) \vee my\_grandfather(X) \leftarrow child(X, Y), child(Y, me)$$

The intended meaning is that X is my grandmother or grandfather if Y is the child of X and I am the child of Y. Given the domain $\{me, alice, bob\}$, the grounding of this program would be:

$$\begin{aligned}
my\_grandmother(me) \lor my\_grandfather(me) &\leftarrow child(me, me), child(me, me)\\
my\_grandmother(me) \lor my\_grandfather(me) &\leftarrow child(me, alice), child(alice, me)\\
my\_grandmother(me) \lor my\_grandfather(me) &\leftarrow child(me, bob), child(bob, me)\\
my\_grandmother(alice) \lor my\_grandfather(alice) &\leftarrow child(alice, me), child(me, me)\\
my\_grandmother(alice) \lor my\_grandfather(alice) &\leftarrow child(alice, alice), child(alice, me)\\
my\_grandmother(alice) \lor my\_grandfather(alice) &\leftarrow child(alice, bob), child(bob, me)\\
my\_grandmother(bob) \lor my\_grandfather(bob) &\leftarrow child(bob, me), child(me, me)\\
my\_grandmother(bob) \lor my\_grandfather(bob) &\leftarrow child(bob, alice), child(alice, me)\\
my\_grandmother(bob) \lor my\_grandfather(bob) &\leftarrow child(bob, bob), child(bob, me) \quad \triangle
\end{aligned}$$

Safety

When dealing with variables in ASP programs, we have to introduce the notion of *safety*. A rule r of a disjunctive logiv program is said to be safe iff all variables occuring in the head or negative body of r also occur in its positive body. A program P is safe iff all of its rules are safe. From this point onward, we consider only safe logic programs.

2.1.3 Language Features

The presentation of the language features we will use in present work (which go beyond the previously defined base language) will be succeeded by a recharacterization of safety taking into account these features.

Definition of Features

The current ASP-Core-2 language standard [Calimeri et al., 2012] defines language features all of which we are going to use. All of the subsequently described features are taken from this standard, except for bitwise arithmetics, modulo, exponentiation, absolute value, pooling, intervals and meta-statements, which are extensions introduced by the Potassco suite [Gebser et al., 2015]. For more in-depth semantics of the described features, we refer to [Calimeri et al., 2012]. All these extensions can be compiled into the base language (that is, without these features), as shown in the ASP-Core-2 language standard, the Potassco user guide, and in [Gebser et al., 2012].

Strong Negation If p is a predicate literal, p and $-p$ are classical atoms.

Arithmetics If t_1 and t_2 are terms, then $t_1 + t_2$ (addition), $t_1 - t_2$ (subtraction), $t_1 * t_2$ (multiplication), t_1/t_2 (division), $t_1 \backslash t_2$ (modulo), $t_1 * *t_2$ (exponentiation), $-t_1$ (unary minus), $|t_1|$ (absolute value), $\&t_1$ (bitwise conjunction), $?t_1$ (bitwise disjunction), $\hat{t}_1$ (bitwise exclusive or) and $\sim t_1$ (bitwise complement) are also terms.

Built-in Atoms Let $B_{\prec} = \{<, \leq, >, \geq, =, \neq\}$ be the set of all binary operators. If t_1 and t_2 are terms and $\prec \in B_{\prec}$, then $t_1 \prec t_2$ is an atom additionally to the previously characterized "classical" atoms.

Aggregates If t_1 and t_2 are terms, then

$$\begin{aligned} &\#aggr\{e_1, \ldots, e_n\}\texttt{,} \\ t_1 \prec_1 &\#aggr\{e_1, \ldots, e_n\}\texttt{,} \\ &\#aggr\{e_1, \ldots, e_n\} \prec_1 t_1 \texttt{ and} \\ t_1 \prec_1 &\#aggr\{e_1, \ldots, e_n\} \prec_2 t_2 \end{aligned}$$

are *aggregates* with $\prec_1, \prec_2 \in B_{\prec}$, $\#aggr \in \{\#count, \#sum, \#max, \#min\}$ and $e_1, \ldots, e_n$ being *aggregate elements*. t_1 and t_2 are called the *guard terms* of the aggregate. An aggregate element has the form $t_1, \ldots, t_m : l_1, \ldots, l_n$ where t_i are terms and l_i are literals. We denote by $B(e) = \{l_1, \ldots, l_n\}$ the body and by $H(e) = \{t_1, \ldots, t_m\}$ the head of an aggregate element e, with $vars(B(e))$ resp. $vars(H(e))$ denoting all the variables occurring in any of the body's literals resp. head's terms.

We denote by $invars(a)$ the set of all variables occurring in any of $e_1, \ldots, e_n$ and by $outvars(a)$ the set of all variables occurring in any of the guard terms t_1 or t_2 of an aggregate a.

An aggregate may only occur in the body of a rule, a weak constraint or a choice rule (see below).

Weak Constraints and Optimize Statements A *weak constraint* has the form $:\sim b_1, \ldots, b_n.[w@l, t_1, \ldots, t_m]$ where $n \geq 0$, $m \geq 0$, b_i are literals or aggregates (just like a rule body) and w, l and t_i are terms. If the level l is 0, the part $@l$ may be omitted. We denote by $B(r) = \{b_1, \ldots, b_n\}$ the body and by $H(r) = \{w, l, t_1, \ldots, t_m\}$ the head of a weak constraint r.

The semantics of a weak constraint is for an answer set to minimize the sum of weights of the weak rules it triggers, grouped by levels (with the level 0 being the most "important"). For a more formal semantics characterization, we refer to [Calimeri et al., 2012].

A syntactic shortcut for a set of weak constraints is an *optimize statement*. An optimize statement has the form $\#opt\{e_1; \ldots; e_n\}$. where $\#opt \in \{mathit\#minimize, \#maximize\}$ and e_i is an *optimize element*. An optimize element is of the form $w@l, t_1, \ldots, t_m : b_1, \ldots, b_n$ with $m \geq 0$, $n \geq 0$, w, l and t_i being terms and b_i being literals. As with weak constraints and aggregate elements, we denote by $B(e) = \{b_1, \ldots, b_n\}$ the body and by $H(e) = \{w, l, t_1, \ldots, t_m\}$ the head of an optimize element e.

We now redefine the notion of a program: A program is a set of statements. A statement is either a rule, a weak constraint, an optimize statement, a choice rule or a meta-statement (see below).

Queries A *query* is of the form $a?$, where a is a ground or non-ground classical atom (that is, a classical atom with or without variables). The semantics of a query only influences the output form of an ASP solver, it has no influence on the answer set semantics of the rest of the program. In the present work, we will therefore not deal with its semantics.

A program may, additionally to the set of statements, contain up to one query.

Anonymous Variables A variable may be denoted by an underscore (_), which stands for a fresh variable in the respective context.

Choice Rules If t_1 and t_2 are terms, then a *choice rule* is of the form

$$\begin{aligned}\{e_1;\dots;e_m\} &\leftarrow b_1,\dots,b_n\texttt{,}\\ t_1 \prec_1 \{e_1;\dots;e_m\} &\leftarrow b_1,\dots,b_n\texttt{,}\\ \{e_1;\dots;e_m\} \prec_1 t_1 &\leftarrow b_1,\dots,b_n \texttt{ or}\\ t_1 \prec_1 \{e_1;\dots;e_m\} \prec_2 t_2 &\leftarrow b_1,\dots,b_n\end{aligned}$$

where $n \geq 0$, $m \geq 0$, $\prec_1, \prec_2 \in B_\prec$, b_i are literals or aggregates (just like a rule body) and e_i are *choice elements*. t_1 and t_2 are called the *guard terms* of the choice rule. Similar to "classical" rules, we denote by $B(r) = \{b_1,\dots,b_n\}$ the body of a choice rule r and by $H(r)$ its head (the part left to the $\leftarrow$ symbol). By $\mathit{vars}(H(r))$ we denote the set of all variables occurring in $H(r)$.

A choice element has the form $a : l_1,\dots,l_k$ where a is a classical atom, $k \geq 0$ and l_i are literals. Similar to aggregate elements, we denote by $B(e) = \{l_1,\dots,l_k\}$ the body and by $H(e) = \{a\}$ the head of a choice element e, with $\mathit{vars}(B(e))$ resp. $\mathit{vars}(H(e))$ denoting all the variables occurring in any of the body's resp. head's literals.

The intended meaning is that if the body results to true, the subset of ground choice elements whose heads are true has a cardinaliy so that the inequality conditions hold.

Pooling and Intervals Everywhere a term may occur, it is admissible to instead have a *pool* or an *interval*. A pool is of the form $t_1;\dots;t_n$ with $n > 1$ and t_i being terms, representing each term. An interval is of the form $t_1..t_2$ with t_1 and t_2 being terms, representing each integer in the range $[t_1, t_2]$. During grounding, the context of the pool or interval is expanded to each representing term. Thus, in a literal in a rule body, a pool or interval means a disjunction, and in a literal in a rule head, it means a conjunction.

Meta-Statements Potassco introduces the *meta-statement* `#show a.` (amongst others), where a is of a certain form. These meta-statements, similar to queries, only influence the output form of an ASP solver and have no influence on the answer set semantics of the rest of the program. In the present work, we will therefore not deal with their semantics.

Safety and Globality with Language Features

When dealing with safety of logic programs featuring the presented extensions (without compiling them into the base language), safety has to be redefined, since it is no longer sufficient for all variables just to occur somewhere in the positive body. Safety of the extended language is partly defined in the ASP-Core-2 language standard [Calimeri et al., 2012]. The language extensions it does not include in its safety characterization are pooling and intervals. To deal with their safety, it is sufficient to rewrite pooling as introduced in subsection 3.3.1 and to consider intervals like arithmetic terms.

For the main part of our work, the notion of *globality* is crucial. We therefore introduce it for different syntactic types. In the remainder of this section, we consider only logic programs whose optimize statements consist of only one optimize element each.

We introduce the notion of *global variables*. The set of global variables of a rule, choice rule, weak constraint or optimize statement r, denoted by $glob(r)$, is the union of all the variables occurring in the literals $\{l \in B(r) :$ `l is a literal`$\}$ and all the variables occurring in the guard terms of the aggregates $\{a \in B(r) :$ `a is an aggregate`$\}$.

The set of global variables of a head or body literal l in a statement r is $glob_r(l) = glob(l) = vars(l)$ and is therefore independent of its "context" r.

The set of global variables of an aggregate a in a statement r is $glob_r(a) = (invars(a) \cap glob(r)) \cup outvars(a)$. As can be seen, a variable in an aggregate has either "local" scope inside an aggregate element or "global" scope in the sense that it "binds" to outside the aggregate or to the aggregate guards.

The set of global variables of a rule head $H(r)$ is $glob_r(H(r)) = \bigcup_{l \in H(r)} glob(l)$.

The set of global variables of the head of a weak constraint r is $glob_r(H(r)) = \bigcup_{t \in H(r)} vars(t)$.

We denote by the head (resp. the body) of a single-element optimize statement the head (resp. the body) of its optimize element. The set of global variables of the head of a single-element optimize statement r is $glob_r(H(r)) = \bigcup_{t \in H(r)} vars(t)$.

The set of global variables of the head $H(r)$ of a choice rule r is $glob_r(H(r)) = (vars(H(r)) \cap glob(r))$. As with aggregates, a variable in a choice element has either local or global scope.

2.2 Tree Decompositions

Many real-world problems are reducible to problems on graphs. Many of these graph problems are very hard to solve in general, but become tractable when being restricted to trees. Since many real-world problems are based on domains that comprise a certain structure instead of being generally random, the graphs they reduce to might be considered more "tree-like" than randomly generated, unstructured graphs. In fact, the research in tree decompositions has helped us measure the "tree-likeliness" of graphs and provides us with structures we can utilize to solve graph problems more efficiently. [Robertson and Seymour, 1984]

The notion of a *hypergraph* is a generalization of the notion of a graph, where an edge does not only connect two vertices, but any number of vertices of the graph. A hypergraph G is a pair $G = \langle V, E\rangle$, with V denoting the set of vertices or nodes and E denoting the set of hyperedges of G. Each hyperedge e is a subset of the vertices V of the hypergraph: $e \subseteq V | e \in E$.

A *tree decomposition of a hypergraph* $G = \langle V, E\rangle$ is a (rooted) tree $D(G) = \langle N, F\rangle$ with nodes (also called *bags*) $N \subseteq 2^V$ and edges F such that the following conditions hold:

1. for each $e \in E$ there exists a node $n \in N$ such that $e \subseteq n$,
2. for each $v \in V$ the set $\{n \in N : v \in n\}$ induces a connected subtree of T, and
3. $\bigcup_{n \in N} n = V$

We denote by the *width* of a tree decomposition $D(G)$ of a hypergraph G the size of its largest bag, minus 1: $max_{n\in N}(|n|) - 1$. The *treewidth* of a hypergraph G is defined as the smallest width of all possible tree decompositions of G.

For any given decomposition node $n \in N$, we denote by $parent(n)$ the parent node of n (except for the root node) and by $desc(n)$ the set of n's children (which is empty if n is a leaf node).

Since it is in general NP-hard to find a tree decomposition of minimal width, much effort has been invested in finding good heuristics. As a result, today, heuristics exists that find almost-minimal decompositions in appropiate time. [Dermaku et al., 2008]

2.3 Rewriting Base ASP Programs

[Morak and Woltran, 2012] have already investigated and implemented a rule decomposition algorithm, which was but limited to the ASP base language (that is, without the language features introduced in subsection 2.1.3). They proposed a hypergraph representation of a rule as follows:

> A hypergraph of a non-ground logic program rule r is a pair $HG(r) = (V, E)$ such that V consists of all the variables occurring in r and E is a set of hyperedges, such that for each atom $A \in B(r)$ there exists exactly one hyperedge $e \in E$, which consists of all the variables occurring in A. Furthermore there exists exactly one hyperedge $e \in E$ that contains all the variables occurring in $H(r)$.

Their rule rewriting algorithm for a non-ground rule r functions as follows:

Let $\mathbf{Y}_n = n \cap parent(n)$ and T_n be a fresh predicate for each decomposition node $n \in N$ except the root node.

1. Compute a tree decomposition of r, trying to minimize the maximal bag size and having the edge representing $H(r)$ only in the root node of the decomposition.
2. Do a bottom-up traversal of the decomposition, n denoting the current tree node:

if n is not the root node , generate a rule of the form:

$$\begin{aligned} T_n(\mathbf{Y}_n) \leftarrow & \{l \in B(r) | vars(l) \subseteq n, vars(l) \neq \emptyset\} \\ & \cup \{\Sigma_X(X) | X \in \bigcup_{l \in B^-(r) | vars(l) \subseteq n} vars(l)\} \\ & \cup \{T_m(\mathbf{Y}_m) | m \in desc(n)\} \end{aligned}$$

The new predicates $\Sigma_X(X)$ are necessary to guarantee safety of the generated rule. For each variable $X \in \bigcup_{l \in B^-(r) | vars(l) \subseteq n} vars(l)$, we additionally generate the rule

$$\Sigma_X(X) \leftarrow b$$

With $b \in B^+(R) | X \in vars(b)$ being a positive predicate containing X.

if n is the root node , generate the rule:

$$\begin{aligned} H(r) \leftarrow & \{l \in B(r) | vars(l) \subseteq n\} \\ & \cup \{\Sigma_X(X) | X \in \bigcup_{l \in B^-(r) | vars(l) \subseteq n} vars(l)\} \\ & \cup \{T_m(\mathbf{Y}_m) | m \in desc(n)\} \end{aligned}$$

Note that this *head rule* also includes all ground literals, since they were not included in the other generated rules.

[Morak and Woltran, 2012] proofed that the answer sets of a base logic program correspond (i.e. are the same modulo the newly introduced predicates) to those of the program resulting of application of the proposed algorithm to each rule in the original program.

Chapter 3

Rule Decomposition

Our goal in this paper is to split up each statement of any answer set program into multiple ones, using a tree decomposition on the statement in order to reproduce a new statement for each bag of the decomposition. Much of this work is based on [Morak and Woltran, 2012]. Our contribution is to extend the functionality presented in section 2.3 to programs containing the language features introduced in subsection 2.1.3.

In this chapter, we will therefore first have a look at how to represent a single ASP statement as a hypergraph. Since constructing a tree decomposition on this hypergraph is performed by off-the-shelf libraries, the subsequent chapter already deals with the opposite direction: how to rewrite a statement into multiple ones based on a decomposition. Since less substantial but still relevant, we will finally describe the preprocessing steps performed before building the hypergraph representation.

3.1 Graph Representations of ASP Rules

As in the base language case, each statement (except for meta-statements, which are not decomposable) is made up of a head and a body, the body in turn is made up of body elements (literals and aggregates). These body elements, along with the head, have to be represented as edges in our hypergraph representation.

In line with the base language case, we construct for a non-ground logic program statement r its hypergraph representation $HG(r) = \langle glob(r), E \rangle$ such that E is the smallest set satisfying that $glob_r(H(r)) \in E$ and for each body element $b \in B(r)$ (literal or aggregate), $glob_r(b) \in E$.

Note that we did not define $glob_r(H(r))$ for optimize statements r that contain more than one optimize element (how could we?). We therefore chose to preprocess each rule to split up such optimize statements into multiple, semantically equivalent ones. This task is regarded in subsection 3.3.1.

Since local variables inside aggregates and choice elements are not included in the hypergraph, different body elements and heads containing local variables with the same name may end up in different bags in the decomposition and therefore finally in different rules. This is intended behavior, since those variables do not join.

Example 3.1. Consider the following single-statement program:

$$Y \leq \{p(X) : q(X)\} \leftarrow r(Y), q(Z), Z \leq \#sum\{X : p(X)\}$$

The hypergraph representation of this statement is $\langle\{Y, Z\}, \{\{Y\}, \{Z\}\}\rangle$. We would expect our tree decomposition to consist of two bags, $\{Y\}$ and $\{Z\}$, with the first one being the root (since Y is a head variable). Based on the one presented in section 2.3, the subsequent rule synthesis algorithm is expected to produce rules similar to these:

$$\begin{aligned} tmp &\leftarrow q(Z), Z \leq \#sum\{X : p(X)\} \\ Y \leq \{p(X) : q(X)\} &\leftarrow r(Y), tmp \quad \triangle \end{aligned}$$

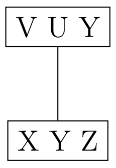
3.2 Rule Synthesis and Unsafe Variables

When it comes to generating rules based on a given decomposition, we can basically adopt the Moark-Woltran rewriting algorithm of section 2.3, since the notion of a variable has not changed introducing new language features. Regarding unsafe variables however, we have to refine the safety closure temporary predicates Σ_X, since the definition of safety is now different to that of the base ASP language.

Example 3.2. Consider the rule:

$$h(V) \leftarrow p(Z),\ \text{not}\ \ p(X),\ X = Y + Z,\ Y = U + 1,\ q(U, V).$$

The decomposition of the hypergraph representation may look like this:



Using the approach of Morak and Woltran, at the beginning of the tree traversal, we create a temporary rule containing all those body elements that only use the variables X, Y and Z, namely $p(Z)$, not $p(X)$ and $X = Y + Z$. In this body, both the variables X and Y are unsafe, so, following the Morak-Woltran algorithm, we have to include the safety-guaranteeing predicates Σ_X and Σ_Y. Trying to construct safety definition rules for these predicates leads us to a problem: There does not exist a positive body predicate containing either of those two variables, so the question to this end is: how many and which body elements do we have to include in order to make the generated rule safe? $\triangle$

In order to answer this question, we first introduce the notion of the *safety characterization* of literals, aggregates and terms.

3.2.1 Introducing Safety Characterization

Definition 3.3. A safety characterization is a triplet $\langle S, U, R\rangle$ with S denoting the set of safe variables, U denoting the set of unsafe variables and R denoting the set of *safety rules*. A safety rule is of the form $D \to d$, where D denotes the safety rule's *dependees*, which is a set of variables, and d denotes the safety rule's *depender* variable. We denote by $\mathcal{C}$ the set of all safety characterizations. △

The concept of safety rules should not be mistaken with rules in the ASP language. The former is merely used as a notational aid for performing safety analysis on the latter.

The meaning of a safety rule may be read as "if in this context, all dependee variables are safe, then the depender variable is safe as well".

Definition 3.4. A safety rule $r = D \to d$ is said to be in *normal form* (or *normal*) iff $D \neq \emptyset$ and $d \notin D$. A safety characterization $c = \langle S, U, R\rangle$ is said to be in normal form iff the following conditions hold:

normal rules: each safety rule r in R is in normal form,

variable safety: $S \cap U = \emptyset$,

unsafe dependers: for each rule $D \to d$ in R, $d \in U$ (note that combined with the variable safety condition, this implies $d \notin S$),

unsafe dependees: for each rule $D \to d$ in R, $D \subseteq U$ (note that combined with the variable safety condition, this implies $D \cap S = \emptyset$).

Note that the last two conditions imply the *unsafe rules* condition: for each safety rule $D \to d$ in R, $(D \cup \{d\}) \subseteq U$. By $\mathcal{C}_n$ we denote the set of all normal safety characterizations (clearly, $\mathcal{C}_n \subseteq \mathcal{C}$). △

Definition 3.5. The *safety characterization closure* is the function $clos : 2^{\mathcal{C}} \to \mathcal{C}_n$ that, given a set of safety characterizations, produces a normal safety characterization combining the given ones. *clos* is defined as follows:

$$clos(Cs) = norm\left(\left\langle \bigcup_{\langle S,U,R\rangle \in Cs} S, \bigcup_{\langle S,U,R\rangle \in Cs} U, \bigcup_{\langle S,U,R\rangle \in Cs} R \right\rangle\right)$$

The normalization function $norm : \mathcal{C} \to \mathcal{C}_n$ is defined in algorithm 1. △

Informally speaking, the closure combines safety characterizations in such a way that it tries to make as many variables as possible safe, triggering safety rules if their dependees are safe. The concept of combining safety rules and infering safety of variables based on rules is quite similar to aspects of ASP, and in fact the *clos* function may be implemented in a declarative way.

3.2.2 Safety Characterization of ASP Elements

Now that we have introduced the concept of safety characterizations, we may define the safety characterizations of terms, literals and aggregates (in this order), based on the definition of safety in [Calimeri et al., 2012]. We denote by $safety_r(e)$ the normal saferty characterization of the term, literal or aggregate e if it occurs in the body of the statement r.

```
input  : a safety characterization $C$
output: $C$ in normal form
1  foreach $r \in C.R$ do
2      if $r.d \in r.D$ then
3          remove r from C.R;
4      end
5  end
6  repeat
7      foreach $r \in C.R$ do
8          if $r.d \in C.S$ then
9              remove r from C.R;
10         else
11             remove from r.D all variables that also occur in C.S;
12             if r.D is empty then
13                 insert r.d into C.S;
14                 remove r from C.R;
15             end
16         end
17     end
18 until neither C nor any of C.R have changed in last pass;
19 insert into C.U all variables that occur in any rule in C.R;
20 remove from C.U all variables that also occur in C.S;
21 return C;
```

Algorithm 1: safety characterization normalization

Since when using pooling, terms with different variables may be pooled, resulting in different safety behaviour in different pooling instantiations. We therefore in this section only consider statements that do not contain pooling and restrict our rewriting process to such statements. This is why we need to preprocess any statement before handing it to the rewriting pipeline. On how to do this, see subsection 3.3.1

Terms

If t is a constant or an anonymous variable, $\mathit{safety}_r(t) = \mathit{safety}(t) = \langle \emptyset, \emptyset, \emptyset \rangle$ for any rule r.

If t is a variable, $\mathit{safety}_r(t) = \mathit{safety}(t) = \langle \{t\}, \emptyset, \emptyset \rangle$ for any rule r.

If t is a function $f(t_1, \dots, t_n)$, then $\mathit{safety}_r(t) = \mathit{safety}(t) = \mathit{clos}(\{\mathit{safety}(t_i) : 1 \leq i \leq n\})$ for any rule r.

If t is an n-ary arithmetic term or an interval term with subterms $t_1, \dots, t_n$, then $\mathit{safety}_r(t) = \mathit{safety}(t) = \left\langle \emptyset, \bigcup_{s=t_1,\dots,t_n} \mathit{vars}(s), \emptyset \right\rangle$ for any rule r (i.e. all variables are unsafe). Since the arithmetic operations discussed in subsection 2.1.3 are only unary and binary and intervals have a binary syntax, n may only be either 1 or 2.

Note that the safety characterization of a term is always rule-free, i.e. it does not contain safety rules, but only safe and unsafe variables.

Classical Literals

Recap: A classical literal is either a classical atom or a naf-negated classical atom, with a classical atom being either a predicate or a strongly negated predicate.

If l is a naf-negated classical atom, $\mathit{safety}_r(l) = \mathit{safety}(l) = \langle \emptyset, \mathit{vars}(l), \emptyset \rangle$ for any rule r.

If l is a predicate or strongly negated predicate $[-]p(t_1, \dots, t_n)$, then $\mathit{safety}_r(l) = \mathit{safety}(l) = \mathit{clos}(\{\mathit{safety}(t_i) : 1 \leq i \leq n\})$ for any rule r.

Built-in Literals

If $l = t_1 \prec t_2$ is a built-in literal with $\prec \in B_\prec \setminus \{=\}$, then $\mathit{safety}_r(l) = \mathit{safety}(l) = \langle \emptyset, \mathit{vars}(l), \emptyset \rangle$ for any rule r.

If l is a built-in literal of the form $t_1 = t_2$ with $\mathit{safety}(t_1) = \langle S_1, U_1, \emptyset \rangle$ and $\mathit{safety}(t_2) = \langle S_2, U_2, \emptyset \rangle$, then, for any rule r:

$$\begin{aligned} &\mathit{safety}_r(l) = \mathit{safety}(l) = \\ &= \mathit{norm}\left(\langle \emptyset, \mathit{vars}(l), \{\mathit{vars}(t_1) \to s_2 : s_2 \in S_2\} \cup \{\mathit{vars}(t_2) \to s_1 : s_1 \in S_1\} \rangle\right) \end{aligned}$$

The intuition here is: No variables are safe. If all the variables on the left side of the equation are being made safe in the context, then all the variables that are in the safe set in the right term's safety characterization are also being made safe by this literal, and vice versa.

Aggregates

If a is an aggregate of the form $\#aggr\{e_1, \dots, e_n\}$ (i.e. it has no guards), then $safety_r(a) = \langle \emptyset, glob_r(a), \emptyset \rangle$.

If a is an aggregate of the form $t \prec \#aggr\{e_1, \dots, e_n\}$ with $\prec \in B_\prec \setminus \{=\}$, then $safety_r(a) = \langle \emptyset, glob_r(a), \emptyset \rangle$.

If a is an aggregate of the form $t = \#aggr\{e_1, \dots, e_n\}$ and $safety(t) = \langle S, U, \emptyset \rangle$, then:

$$safety_r(a) = norm\left(\langle \emptyset, glob_r(a), \{invars(a) \cap glob(r) \to s : s \in S\} \rangle\right)$$

The intuition here is in line with that of a built-in literal.

If a is an aggregate of the form $\#aggr\{e_1, \dots, e_n\} \prec t$, then $safety_r(a) = safety_r(t \prec \#aggr\{e_1, \dots, e_n\})$.

If a is an aggregate of the form $t_1 \prec_1 \#aggr\{e_1, \dots, e_n\} \prec_2 t_2$, then $safety_r(a) = clos\left\{safety_r(t_1 \prec_1 \#aggr\{e_1, \dots, e_n\}), safety_r(\#aggr\{e_1, \dots, e_n\} \prec_2 t_2)\right\}$.

Sets of Body Elements

Definition 3.6. Given a set B of literals and aggregates and a statement r, we define $safety_r(B) = clos\{safety_r(c) : c \in B\}$ △

Theorem 3.7. A statement r is safe iff $safety_r(B(r)) = \langle glob(r), \emptyset, \emptyset \rangle$ (without proof). △

3.2.3 Choosing Elements for Domain Closure

Now that we may calculate a safety characterization for each body element of a statement, it is possible, given an unsafe variable, to choose a set of body elements of the original statement so that their combined safety characterization includes the given variable as safe. Formally:

Theorem 3.8. *Let r be a safe statement and $v \in glob(r)$ a variable. There exists a set of body elements $\Sigma_v \subseteq B(r)$, such that $safety_r(\Sigma_v) = \langle S, \emptyset, \emptyset \rangle$ and $v \in S$.*

Proof. use theorem 3.7 and let $\Sigma_v = B(r)$. △

What the proof of theorem 3.8 does not imply is that it might be possible to find a real subset $\Sigma_v \subset B(r)$:

Definition 3.9. Let r be a safe statement and $v \in glob(r)$ a variable. We denote by $cand_r(v) \subseteq 2^{B(r)}$, or "candidate bodies for saving v in r", the set of subset-minimal elements of $\{B \subseteq B(r) : safety_r(B) = \langle S, \emptyset, \emptyset \rangle, v \in S\}$. △

These candidate bodies can be inductively calculated.

Now we are able to use algorithm 2 for constructing safety closure rules and alter Morak and Woltran's rule rewriting algorithm from section 2.3. More precisely, we replace the "if n is not the root node" part by the following:

```
input  : statement $r$, temporary rule $t$ whose body includes a subset of $B(r)$ and
         a number of temporary child join predicates
output: safety closure rules
1 $R := \emptyset$;
2 while $t$'s safety characterization contains unsafe variables do
3     $v :=$ choose one of $t$'s safety characterization's unsafe variables;
4     $B :=$ choose the element of $cand_r(v)$ with the least number of variables;
5     $p :=$ create a temporary predicate name;
6     insert $p(v) \leftarrow B$ into $R$;
7     insert $p(v)$ into the body of $t$;
8 end
9 return $R$;
```

Algorithm 2: safety closure construction

if n **is not the root node** , generate the rule t of the form:

$$\begin{aligned} T_n(\mathbf{Y}_n) \leftarrow &\{l \in B(r) | vars(l) \subseteq \chi(n), vars(l) \neq \emptyset\} \\ &\cup \{T_m(\mathbf{Y}_m) | m \in desc(n)\} \end{aligned}$$

Then, we run algorithm 2 with actual arguments r and t and produce the returned rules (which, in turn, may be decomposed as well). Note that algorithm 2 adds the new temporary domain closure predicates to t's body.

Note that in line 4 of algorithm 2, we chose that body for safety closure that has the least number of variables, hoping to reduce grounder workload. A probably more promising approach is discussed in section 5.2.

Example 3.10. Reconsider example 3.2. When processing the leaf bag containing X, Y and Z, after the rule $t = T_1(Y) \leftarrow p(Z),\ \text{not}\ p(X),\ X = Y + Z$ is generated, algorithm 2 is called. In line 3, the algorithm may choose one of the unsafe variables X and Y.

If it chooses Y, then the candidate bodies for saving Y only consists of the body $\{Y = U + 1,\ q(U, V)\}$ and the domain closure rule $tmpsafe(Y) \leftarrow Y = U + 1,\ q(U, V)$ ist being generated, updating t to $T_1(Y) \leftarrow p(Z),\ \text{not}\ p(X),\ X = Y + Z,\ tmpsafe(Y)$. Rechecking the condition in line 2, algorithm 2 will find that t is now already safe (without explicitly having saved X!) and returns.

If, however, the algorithm first chooses X, a domain closure rule will be generated that contains the whole body of our original rule r, except for not $p(X)$. Instead of reducing the grounding, this decision would contrarily blow it up. As if that was not enough, rechecking the condition in line 2, algorithm 2 will find that Y is still unsafe and will generate a second domain closure rule equal to the one above. X may now be considered saved "twice", whereas once would be enough. $\triangle$

As you can see in this example, which unsafe variable to choose first may be crucial to the slimness of the resulting program. To this end, if applicable, a variable shoud be chosen that is not a depender in any safety rule of t's safety characterization.

3.3 Preprocessing Steps

Before feeding each statement of an ASP program to the decomposition pipeline, the program needs to be preprocessed, in part because the pipeline can't handle pooling (discussed in the next section), and in part in order to do some additional optimization (discussed in the subsequent section).

3.3.1 Pooling and Optimize Splitting

As already discussed in subsection 3.2.2, in order to perform our safety analysis on a statement, it must not contain pools. Also in section 3.1, we found that we need optimize statements to contain no more than one optimize element. Although the latter has nothing to do with pooling, those two problems are similar, since in both, the statement may be split up into multiple ones (and the instantiations are syntactically seperated by the semicolon character).

We chose first to perform optimize splitting: If a statement is an optimize statement with n optimize elements in its head, it it split into n new optimize statements, each being a copy of the original one, with the optimize elements being replaced by only one.

Formally, the optimize statement $\#\mathit{opt}\{e_1, \dots, e_n\}$ becomes the set of optimize statements $\{\#\mathit{opt}\{e_i\} : 1 \leq i \leq n\}$.

Afterwards, we perform pooling splitting using the notational aid of *positions* in the syntax tree of a statement. By *global position* we mean the position of a term that has global context, i.e. if a new variable were at this position, it would globally bind (as opposed to a position inside of an aggregate or choice element): If a statement r contains a pooling term $t_1, \dots, t_n$ at global position p, it becomes a set of n statements being a copy of r, each replacing the term at position p by an other t_i. If a statement r contains a pooling term $t_1, \dots, t_n$ at non-global position p inside an aggregate or choice element, the element itself is copied inside the aggregate or choice head, not resulting in more statements. This algorithm is then performed recursively on all newly generated statements.

Example 3.11. Consider following single-statement program:

$$\#\mathit{minimize}\{X : p(X);\ 1, X : q(X, (2; 3 + (4; 5)))\}$$

After optimize splitting, it reads:

$$\#\mathit{minimize}\{X : p(X)\}$$
$$\#\mathit{minimize}\{1, X : q(X, (2; 3 + (4; 5)))\}$$

After the first pooling splitting, it may read:

$$\#\mathit{minimize}\{X : p(X)\}$$
$$\#\mathit{minimize}\{1, X : q(X, (2)\}$$
$$\#\mathit{minimize}\{1, X : q(X, (3 + (4; 5)))\}$$

After the recursive pooling splitting, it reads:

$$\begin{aligned}&\#\mathit{minimize}\{X : p(X)\}\\&\#\mathit{minimize}\{1, X : q(X, (2)\}\\&\#\mathit{minimize}\{1, X : q(X, (3 + (4)))\}\\&\#\mathit{minimize}\{1, X : q(X, (3 + (5)))\} \quad \triangle\end{aligned}$$

Note that it makes no difference in which order the pooling terms are being split. In the example above, first splitting the term $4; 5$ would have resulted in the same set of statements.

Example 3.12. Consider following single-statement program:

$$\mathit{res}(W) \leftarrow W = \#\mathit{sum}\{X : X = (1; 2; 3);\ Y : p(Y)\} < (-5; 5)$$

We choose to first perform the splitting of the global term $-5; 5$ (again, making no difference in the end if we chose otherwise), resulting in:

$$\begin{aligned}&\mathit{res}(W) \leftarrow W = \#\mathit{sum}\{X : X = (1; 2; 3);\ Y : p(Y)\} < (-5)\\&\mathit{res}(W) \leftarrow W = \#\mathit{sum}\{X : X = (1; 2; 3);\ Y : p(Y)\} < (5)\end{aligned}$$

After the recursive call, the program reads:

$$\begin{aligned}&\mathit{res}(W) \leftarrow W = \#\mathit{sum}\{X : X = (1);\ X : X = (2);\ X : X = (3);\ Y : p(Y)\} < (-5)\\&\mathit{res}(W) \leftarrow W = \#\mathit{sum}\{X : X = (1);\ X : X = (2);\ X : X = (3);\ Y : p(Y)\} < (5) \quad \triangle\end{aligned}$$

3.3.2 Aggregates and Choice Rules

In our decomposition algorithm, statements are split considering their body elements as atomic. Thus, long aggregate element or choice element bodies are not examined by the algorithm, leading to a potential optimization bottleneck. Because often, the body of an aggregate element or choice element contains a set of literals that can be extracted into a new rule, allowing for our statement decomposition algorithm to split this rule. This is best explained by an example:

Example 3.13. Consider the following statement:

$$\begin{aligned}&\leftarrow \#\mathit{sum}\{X, Y : p(Z), q(Z, U), q(U, T), q(T, S), f(V + 1, W), X = 2 * W;\ X : p(X)\},\\&\quad Y = 1,\ Z = 2,\ V = 3,\ S = 4\end{aligned}$$

This statement may be rewritten into the semantically equivalent statements

$$\begin{aligned}&\leftarrow \#\mathit{sum}\{X, Y : \mathit{tmp}(Z, S), f(V + 1, W), X = 2 * W;\ X : p(X)\},\\&\quad Y = 1,\ Z = 2,\ V = 3\ S = 4\\\mathit{tmp}(Z, S) &\leftarrow p(Z), q(Z, U), q(U, T), q(T, S)\end{aligned}$$

The latter set of statements is optimizable by our decomposition algorithm, whereas the former is not. △

In order to perform this kind of optimization, we use algorithm 3, called on each aggregate element and choice element in our program. Note that since we use safety analysis in this algorithm, this optimization has to be performed after pooling splitting.

```
input  : statement $r$, aggregate or choice element $e$ inside $r$
output: if optimizable, new temporary rule (and alters $e$); if not, nothing
1  // first, calculate those literals $B$ that may be extracted:
2  $x :=$ the set of unsafe variables $U$ in $\mathit{safety}_r(B(e)) = \langle S, U, R\rangle$;
3  $B := \{l \in B(e) : \mathit{vars}(l) \cap x = \emptyset\}$;
4  repeat
5      $s := \mathit{safety}(B)$ ;
6      foreach $l \in B$ do
7          if $s.U \cap \mathit{vars}(l) \neq \emptyset$ then
8              remove $l$ from $B$ ;
9          end
10     end
11 until $s.U = \emptyset$;
12 if $B = \emptyset$ then
13     /*not optimizable*/
14     return;
15 else
16     // calculate the join variables for the new predicate:
17     $v := \mathit{glob}(\leftarrow B) \cap (\mathit{glob}(r) \cup \mathit{vars}(H(e)) \cup \mathit{vars}(B(e)))$;
18     $p :=$ create a temporary predicate name;
19     insert $p(v)$ into the body of $e$;
20     return new rule $p(v) \leftarrow B$;
21 end
```
Algorithm 3: aggregate and choice splitting algorithm

3.4 Putting it All Together

Now that we have the algorithmic foundations, we can combine them to build a preprocessing application: Whenever a statement is to be optimized we follow these steps:

1. First perform splitting of optimize statements, followed by pooling splitting, both as described in subsection 3.3.1.

2. Afterwards, a preparational safety calculation may be done, since in the following steps, the safety characterizations of term, literals and aggregates may be needed

(subsection 3.2.2). Those safety characterizations may altertively also be later calculated on demand.

3. Split aggregates and choice rules (in any order), as described in subsection 3.3.2.

4. From each resulting statement, build a hypergraph (section 3.1) and feed it into a tree decomposition library.

5. Perform a bottom-up traversal on the returned tree decomposition, generating a rule for each bag, as defined in the Morak-Woltran algorithm, considering our changes regarding unsafe variables from subsection 3.2.3. If the bag contains unsafe variables, we need to generate additional safety closure rules.

Chapter 4

Benchmarks

With our prototype implementation *lpopt* in the programming language C++, we have benchmarked the performance of gronding and solving *lpopt*-preprocessed programs (that is, the output of *lpopt*) against non-preprocessed ones.

We have chosen to use the instance sets of the fifth answer set programming competition 2014 [1], providing for most problem classes the old encoding of the ASP competition 2013 and the new one of 2014.

In appendix A, some more technical details and the benchmark results are given. Out of the 49 encodings, *lpopt* was able to syntactically rewrite 41, indicating that even such extensively hand-tuned programs as those of the international ASP competition can be further decomposed in an automated manner. Those problem classes that show the most interesting differences between *lpopt*-preprocessing and non-preprocessing performance are now discussed.

4.1 Pessimistic Results

The 2014 KnightTourWithHoles The groundings of the preprocessed encoding are about five times larger than those of the non-preprocessed ones, and grounding takes about 50% longer. An explanation of this phenomenon gives a look at the differences between the encodings (in *diff*-like formatting):

```
< other(X,Y,XX,YY) :- valid(X,Y,XX,YY), move(A,B,XX,YY), X!=A.

> temp_decomp_16_0(A,XX,YY) :- move(A,B,XX,YY).
> temp_decomp_16_2_safe(X) :- valid(X,Y,XX,YY).
> temp_decomp_16_1(X,XX,YY) :- X!=A, temp_decomp_16_0(A,XX,YY),
                               temp_decomp_16_2_safe(X).
> other(X,Y,XX,YY) :- valid(X,Y,XX,YY), temp_decomp_16_1(X,XX,YY).
```

The problem here is the safety closure rule: in order to save X, the 4-arity *valid* predicate is copied into a new rule. In this encoding, this predicate is intensional, meaning that

[1] `https://www.mat.unical.it/aspcomp2014/`

the grounder does not know for which variable assignments is it true or false. Having an intensional predicate as domain closure blows up the grounding size, because all possible variable assignments have to be taken along up our decomposition. We consider this topic at section 5.2.

The 2013 Weighted-sequenceProblem shows about 5 to 10 times slower grounding and about three times as big a grounding when using preprocessing. Also in this case, we have intensional domain closure predicates.

4.2 Optimistic Results

The most outstanding result is the 2014 PermutationPatternMatching problem. Grounding time of the heaviest instance is 980 seconds without preprocessing and 17 seconds with preprocessing, the same instance not finishing solving without preprocessing (time-out 300 seconds) and finishing withing 110 seconds when being preprocessed. Let us have a look at the differences between the original and the preprocessed encoding:

```
> kval(K) :- p(K,P), patternlength(L), K <= L.
< temp_decomp_0_1_safe(K) :- p(K,P).
< temp_decomp_0_0(K) :- patternlength(L), K<=L, temp_decomp_0_1_safe(K).
< kval(K) :- p(K,P), temp_decomp_0_0(K).

> pair(K1,K2) :- kval(K1), kval(K2), p(K1,P1), p(K2,P2), P1 <= P2.
< temp_decomp_1_1_safe(P2) :- p(K2,P2).
< temp_decomp_1_0(K1,P2) :- kval(K1), p(K1,P1), P1<=P2,
                            temp_decomp_1_1_safe(P2).
< pair(K1,K2) :- kval(K2), p(K2,P2), temp_decomp_1_0(K1,P2).

> { geq(K,I) } :- kval(K), t(I,E).
< temp_decomp_2_0(I) :- t(I,E).
< {geq(K,I)} :- kval(K), temp_decomp_2_0(I).

> :- kval(K), t(I,E), geq(K,I+1), not geq(K,I).
< temp_decomp_3_1_safe(I) :- t(I,E).
< temp_decomp_3_0(I) :- kval(K), geq(K,I+1), not geq(K,I),
                        temp_decomp_3_1_safe(I).
< :- t(I,E), temp_decomp_3_0(I).

> :- kval(K), t(I,E), geq(K-1,I), not geq(K,I+1).
< temp_decomp_4_0(I) :- t(I,E).
< :- kval(K), geq(K-1,I), not geq(K,I+1), temp_decomp_4_0(I).

> :- pair(K1,K2), solution(K1,E1), solution(K2,E2), E2 < E1.
< temp_decomp_7_0(E1,K2) :- pair(K1,K2), solution(K1,E1).
< :- solution(K2,E2), E2<E1, temp_decomp_7_0(E1,K2).
```

One can see that the original instance is not as well hand-tuned as the encodings of other ASP Competition problem classes. Six rules could be split up. Although a few safety closure predicates had to be introduced, they only depend on the extensional predicates *p* and *t*. For a discussion on this matter, see section 5.2. *p* and *t* do not occur in any rule head of the encoding, they only occur as facts in the instance program parts.

Another positive example is the 2013 StableMarriage problem. When using preprocessing, the grounding size increases dramatically from 5,700 to almost 280,000 or from 10,000 to 650,000 variables, respectively, meaning a multiplication from 50 to 65. The grounder is thirty times faster when using preprocessing, and the sat-supported solver three times faster.

The only rule that differs (from the six-rule-encoding) it the following:

```
< :- match(M,W1), manAssignsScore(M,W,Smw), W1!=W, manAssignsScore(M,W1,Smw1),
     Smw>Smw1, match(M1,W), womanAssignsScore(W,M,Swm),
     womanAssignsScore(W,M1,Swm1), Swm>=Swm1.

> temp_decomp_6_0(Swm1,W) :- match(M1,W), womanAssignsScore(W,M1,Swm1).
> temp_decomp_6_2_safe(Swm) :- womanAssignsScore(W,M,Swm).
> temp_decomp_6_1(Swm,W) :- Swm>=Swm1, temp_decomp_6_0(Swm1,W),
                            temp_decomp_6_2_safe(Swm).
> temp_decomp_6_3(M,W) :- womanAssignsScore(W,M,Swm), temp_decomp_6_1(Swm,W).
> temp_decomp_6_4(M,Smw,W1) :- match(M,W1), manAssignsScore(M,W,Smw),
                               W1!=W, temp_decomp_6_3(M,W).
> :- manAssignsScore(M,W1,Smw1), Smw>Smw1, temp_decomp_6_4(M,Smw,W1).
```

Again, the only safety closure body consists of the extensional predicate *womanAssignsScore*.

The last benchmark problem to discuss is NoMistery, both 2013 and 2014. The grounding times are in both cases about half, the grounding sizes as well. The few solving time points one can see are mostly below that non-preprocessed ones, in part way below, in one extreme case even 9 seconds instead of 174 (sat-supported). Consider the differences in the 2014 encoding:

```
< action(drive(T,L1,L2)) :- fuelcost(Fueldelta,L1,L2), truck(T).
> temp_decomp_9_0(L1,L2) :- fuelcost(Fueldelta,L1,L2).
> action(drive(T,L1,L2)) :- truck(T), temp_decomp_9_0(L1,L2).

< del(fuel(T,Fuelpre),S) :- drive(T,L1,L2,S), fuel(T,Fuelpre,S-1).
> temp_decomp_22_0(S,T) :- drive(T,L1,L2,S).
> del(fuel(T,Fuelpre),S) :- fuel(T,Fuelpre,S-1), temp_decomp_22_0(S,T).

< fuel(T,Fuelpre-Fueldelta,S) :- drive(T,L1,L2,S), fuelcost(Fueldelta,L1,L2),
                                 fuel(T,Fuelpre,S-1), Fuelpre>=Fueldelta.
```

```
> temp_decomp_23_0(Fueldelta,S,T) :- drive(T,L1,L2,S), fuelcost(Fueldelta,L1,L2).
> fuel(T,Fuelpre-Fueldelta,S) :- fuel(T,Fuelpre,S-1), Fuelpre>=Fueldelta,
                                 temp_decomp_23_0(Fueldelta,S,T).

< preconditions_d(T,L1,L2,S) :- step(S), at(T,L1,S-1), fuel(T,Fuelpre,S-1),
                                fuelcost(Fueldelta,L1,L2), Fuelpre>=Fueldelta.
> temp_decomp_32_1_safe(Fueldelta) :- fuelcost(Fueldelta,L1,L2).
> temp_decomp_32_0(Fueldelta,S,T) :- step(S), fuel(T,Fuelpre,S-1),
                                     Fuelpre>=Fueldelta,
                                     temp_decomp_32_1_safe(Fueldelta).
> preconditions_d(T,L1,L2,S) :- at(T,L1,S-1), fuelcost(Fueldelta,L1,L2),
                                temp_decomp_32_0(Fueldelta,S,T).
```

Also here, the only safety closure predicate only uses the extensional predicate *fuel-cost*.

Chapter 5

Conclusion

5.1 Summary

In this work, we have extended the algorithm introduced in [Morak and Woltran, 2012], which preprocesses non-ground answer set programs, splitting up rules into smaller ones employing tree decompositions, thus reducing the number of variables and predicates per rule. Therefore reducing the number of instantiation candidates, this approach reduces grounding time and size, effectively speeding up ASP systems. Our extension to this approach consists of additionally supporting various features to the ASP language, as those defined in the ASP-Core-2 standard and selected ones of the Potassco ASP system. We have discussed the reuirements for and characterization of such a system, implemented a prototype preprocessing program and benchmarked it, in part showing astounding results, and mainly showing that further work in this area is required.

Considering that, up to now, many program encodings have to be extensively hand-tuned in order to avoid the grounding bottleneck, with our approach, this is no longer needed. When splitting up rules by (economically expensive) expert hands, a program's readability suffers — since one of the advantages of declarative problem frameworks is the one-to-one relationship between intention and representation.

We also see a big advantage in this work regarding laymen-written programs and automatically produced programs, since in these cases, experts are not available that could rewrite a program. Those programs may include long rule bodies with relatively loose literal coupling by variables (low treewidth), where out preprocessing optimization comes in handy.

5.2 Future Work

As already touched in subsection 3.2.3, choosing a body candidate for safety domain closure is not optimal in out algorithm. To this end, we refer to the terminology of *extensional* and *intensional* predicates, originating in the declarative programming language *Datalog* [Abiteboul et al., 1995]: Extensional relations are considered those being defined in a constant database, i.e. the declarative program does not compute extensional facts. Intensional relations are those not represented in the database, i.e. being computed by the program. In ASP, in line with this definition, extensional predicates are considered those that only appear as facts and not in rule heads, yielding to the grounder exactly knowing whether an occurrence of such a predicate is true or false. Intensional predicates are considered all other.

When it comes to choosing a safety domain closure set of literals, it is therefore best to choose as little intensional predicates as possible (preferrably only extensionals), since the grounder then already exactly knows which terms to instantiate for the unsafe variable, not postponing a guess to the solver. This also leads to smaller and faster groundings.

Future academic research in the area of non-ground rule splitting is therefore encouraged to distinguish between extensional and intensional predicates. Since grounders like *gringo* already make this distinction, this goal is best reached by incorporating our rule-splitting mechanism into grounders instead of shipping standalone programs.

Similar to this problem is the question how to treat body elements that are contained in more than one bag of the tree decomposition. Our algorithm, as defined above, puts such an element in each and every decomposition rule it is contained in, although, for semantic correctness, it would be sufficient to have it in only one: if it is an extensional predicate, it should be produced in the lowermost tree node, leading to the grounder being able to restrict the instantiation domains in all following parent nodes. If it is an intensional predicate, it should contrarily be produced in the topmost tree node, leading to deferring solver guessing to influence as little other predicates as possible.

In our aggregate and choice element splitting algorithm 3, in line 3 we begin with constructing a body B, and then “thinning it out” to be safe. An other possibility would be to add safety domain closures like in the statement splitting algorithm, also considering taking literals outside of the aggregate or choice rule head. Again, this would be most applicable if the chosen domain closure literals were extensional.

In order for non-ground statement splitting to perform even better, it should be integrated into a grounder, resulting in the benefit that the safety domain closure predicates would not be needed, since the gronder could directly instantiate unsafe variables with the respective constant terms. This would speed up grounding and reduce its generated size, since those predicates would no longer exist.

Appendix A

Benchmark Results

problem class name	13avail	14avail	13opt	14opt
AbstractDialecticalFrameworks	y	y	y	y
BottleFillingProblem	y	y	y	n
ComplexOptimizationOfAnswerSets	y	y	y	y
ConnectedMaximim-densityStillLife	y	y	y	y
CrossingMinimization	y	y	y	y
GracefulGraphs	y	y	y	n
GraphColouring	y	y	n	y
HanoiTower	y	y	y	y
IncrementalScheduling	y	y	y	y
KnightTourWithHoles	y	y	y	y
Labyrinth	y	y	y	y
MaximalCliqueProblem	y	y	n	n
MinimalDiagnosis	y	y	y	y
Nomistery	y	y	y	y
PartnerUnits	y	y	y	y
PermutationPatternMatching	y	y	y	y
QualitativeSpatialReasoning	y	y	n	n
RicochetRobots	y	y	y	y
Sokoban	y	y	y	y
Solitaire	y	y	y	y
StableMarriage	y	y	y	y
StrategicCompanies	y	n	n	n
ValvesLocationProblem	y	y	y	y
Visit-all	y	y	y	y
Weighted-sequenceProblem	y	y	y	y

Above table lists the problem classes of the ASP competition 2014, each entry containing the information whether a 2013 and/or a 2014 encoding is available (13avail/14avail) and whether, for each year, our preprocessing algorithm *lpopt* outputs a program that is syntactically different from the input (13opt/14opt).

The benchmarks have been run on a 3.5GHz AMD Opteron Processor 6308 with 192 GB of RAM to its disposal. We used the Potassco software suite, namely *gringo* verison 4.5.3 as the grounder and *clasp* version 3.1.3 as the solver.

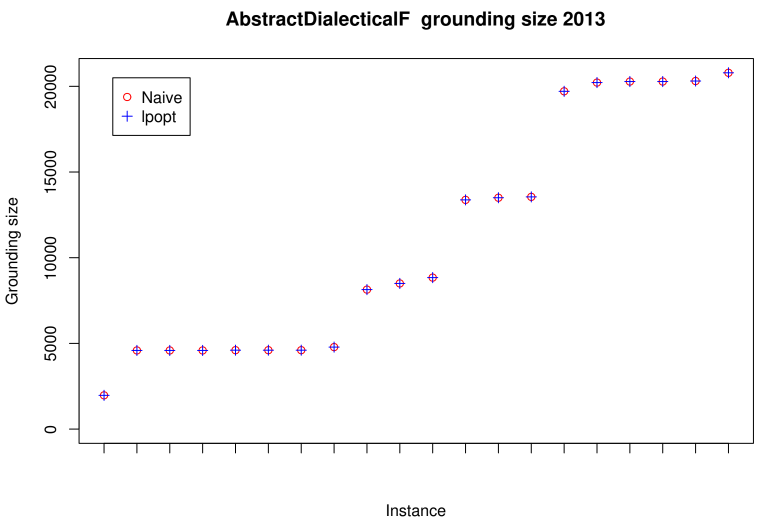
Since on each instance, *lpopt* takes no longer than 0.01 seconds, this time is not reported. We only benchmarked those encodings for which *lpopt* found an optimization (column "..opt" in the table).

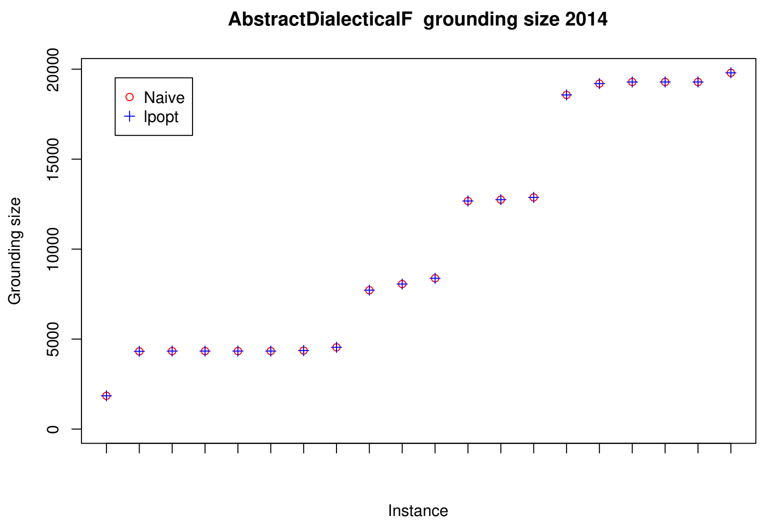
In each problem class, the ASP competition 2014 chose 20 problem instances.

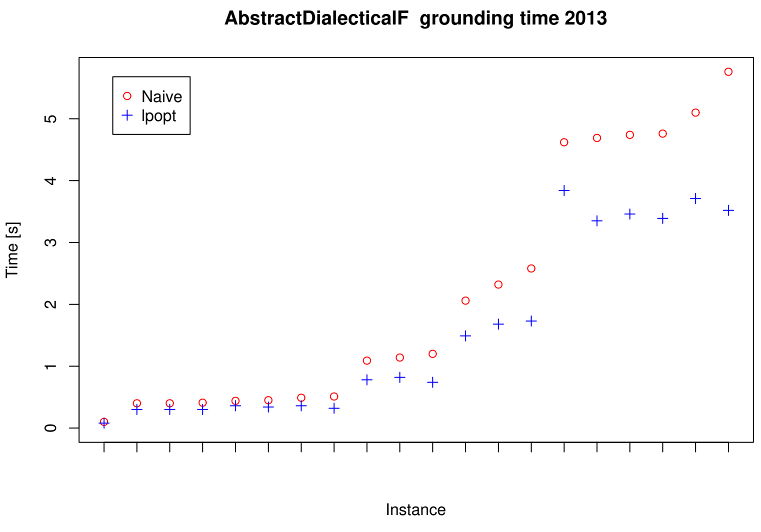
For discussion of the benchmark results, refer to chapter 4.

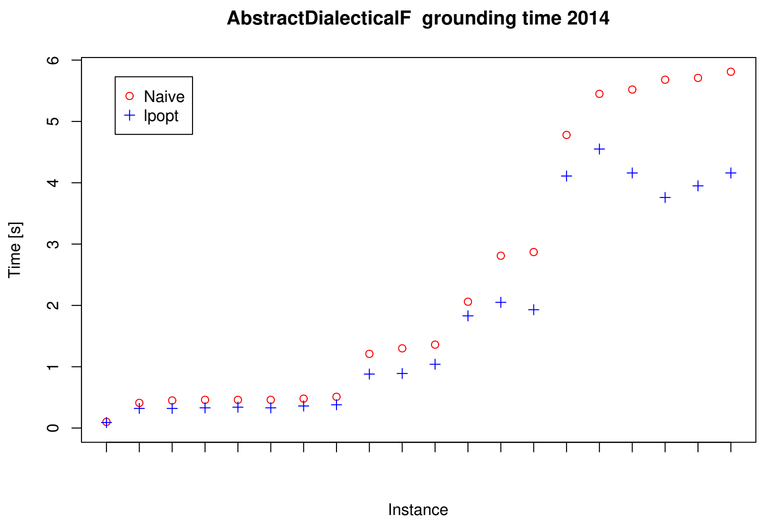
The grounding size has been measured by the number of variables as reported by running `clasp --solve-limit=0 -s2`. This number includes both the number of rule bodies and the number of atoms. In no instance, there was a timeout for *gringo*, but we set the timeout for *clasp* to 300 seconds. Instances where *clasp* runs into this timeout are not printed in the respective plots.

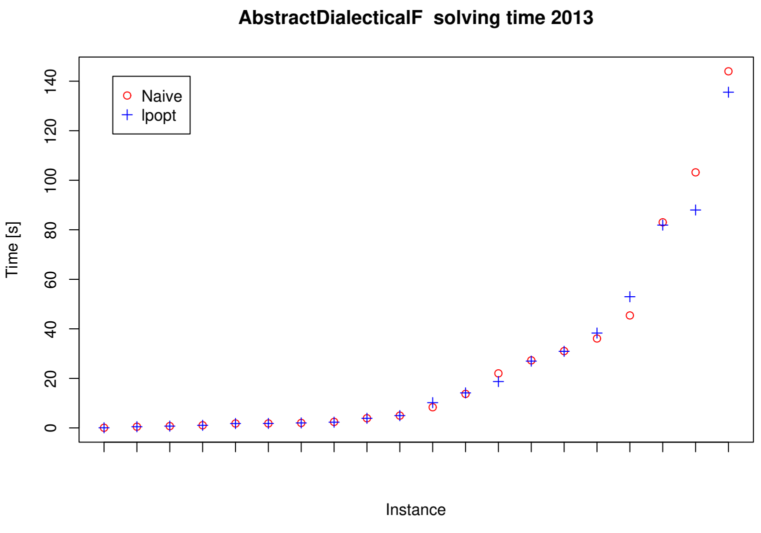
On each instance, we ran *clasp* two times: once without arguments and once with the `--sat-prepro` argument, enabling SatELite-like preprocessing (except for KnightTour-WithHoles).

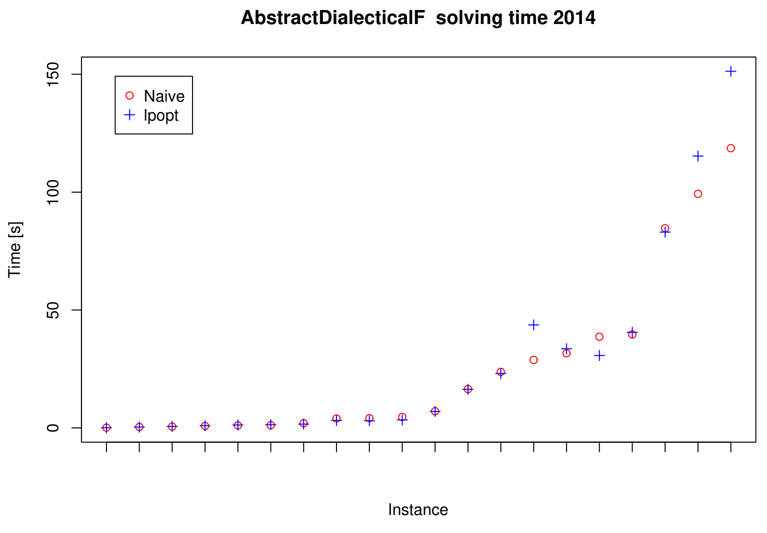

AbstractDialecticalF grounding size 2013
Naive
Ipopt
Grounding size
Instance
0
5000
10000
15000
20000

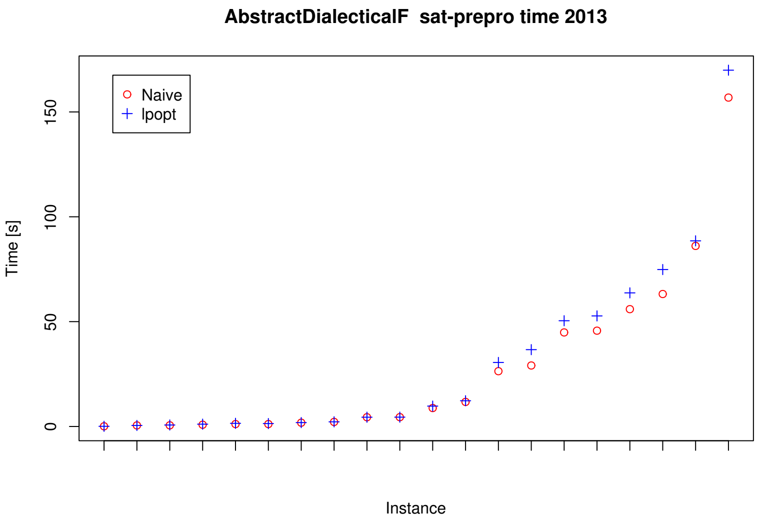

AbstractDialecticalF grounding size 2014
Naive
Ipopt
Grounding size
Instance
0
5000
10000
15000
20000

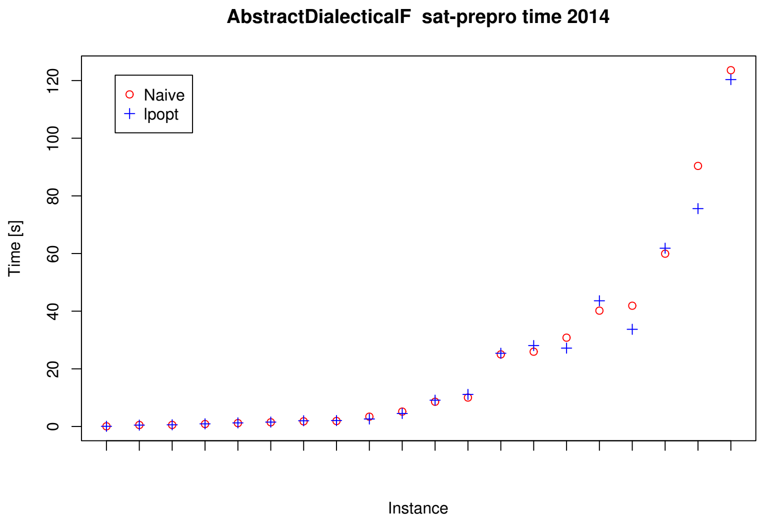

AbstractDialecticalF grounding time 2013
Naive
Ipopt
Time [s]
Instance
0
1
2
3
4
5

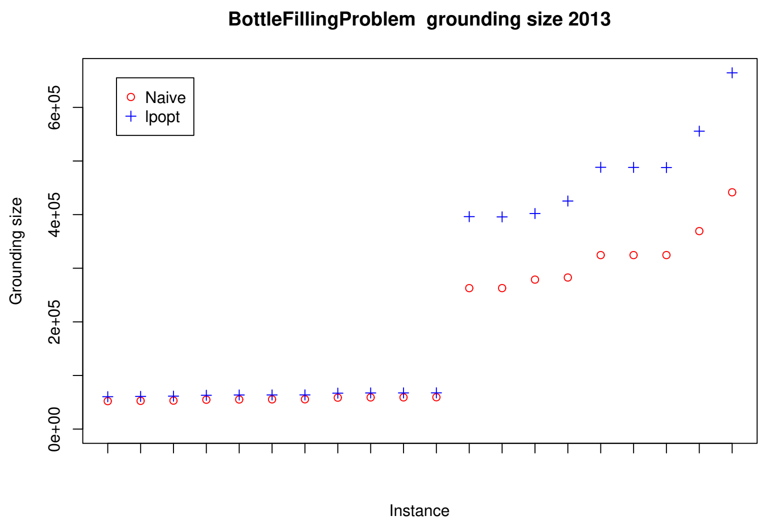

AbstractDialecticalF grounding time 2014
Naive
Ipopt
Time [s]
Instance
0
1
2
3
4
5
6

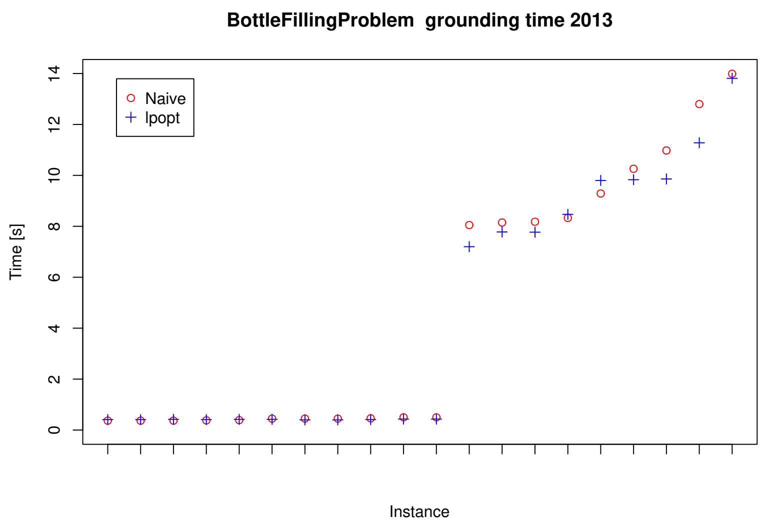

AbstractDialecticalF solving time 2013
Naive
Ipopt
Time [s]
Instance
0
20
40
60
80
100
120
140

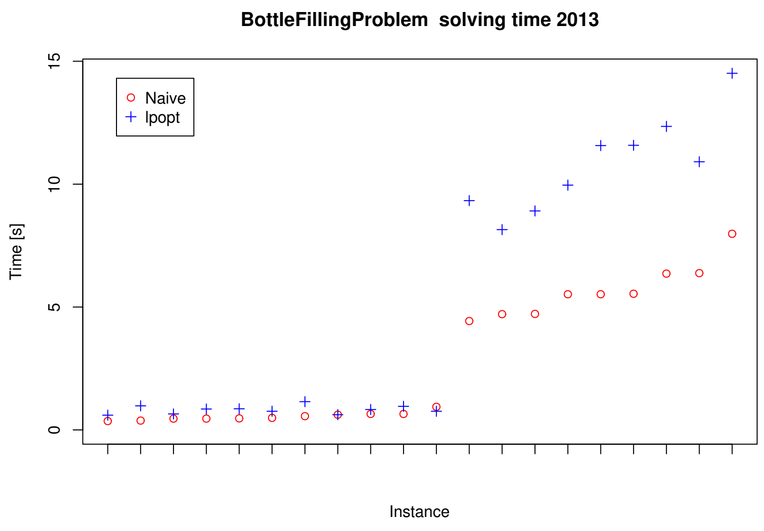

AbstractDialecticalF solving time 2014
Naive
Ipopt
Time [s]
Instance
0
50
100
150

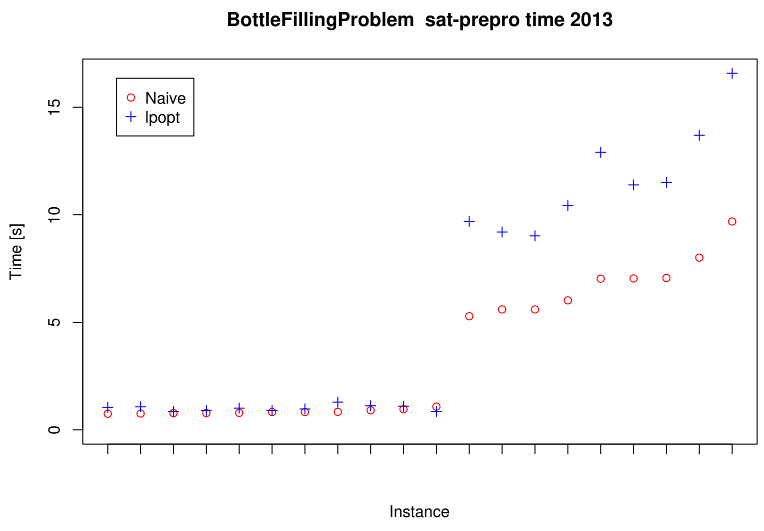

AbstractDialecticalF sat-prepro time 2013
Naive
Ipopt
Time [s]
Instance
0
50
100
150

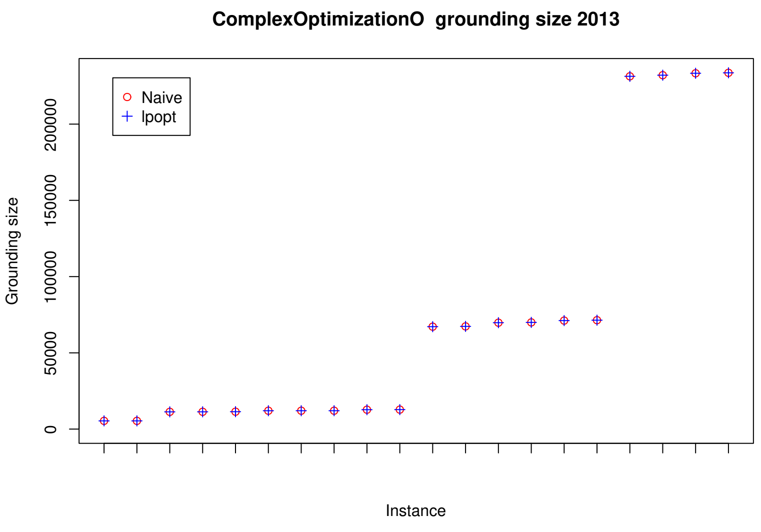

AbstractDialecticalF sat-prepro time 2014
Naive
Ipopt
Time [s]
Instance
0
20
40
60
80
100
120

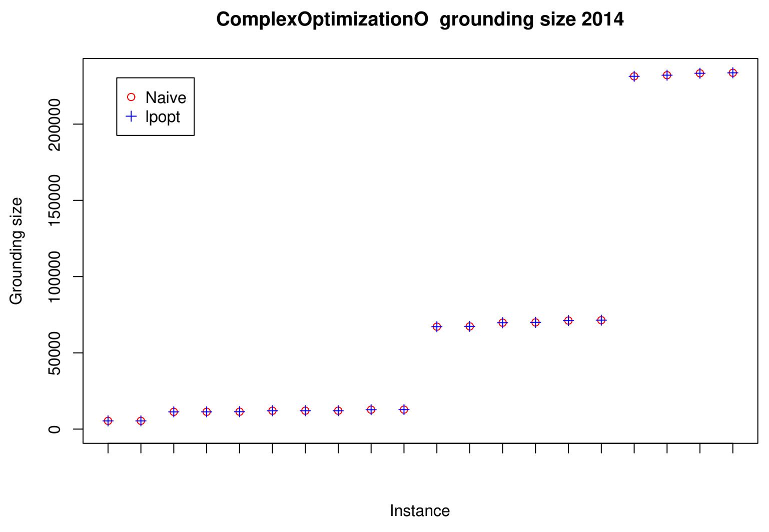

BottleFillingProblem grounding size 2013
Naive
Ipopt
Grounding size
0e+00
2e+05
4e+05
6e+05
Instance

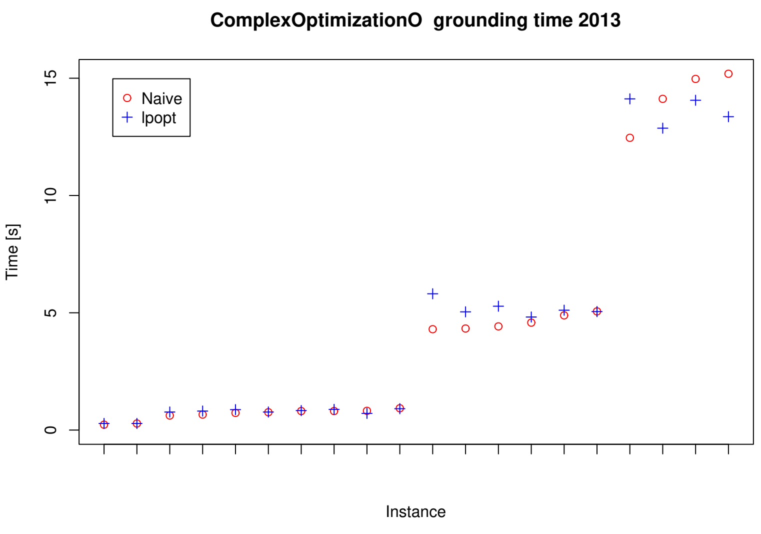

BottleFillingProblem grounding time 2013
Naive
Ipopt
Time [s]
0
2
4
6
8
10
12
14
Instance

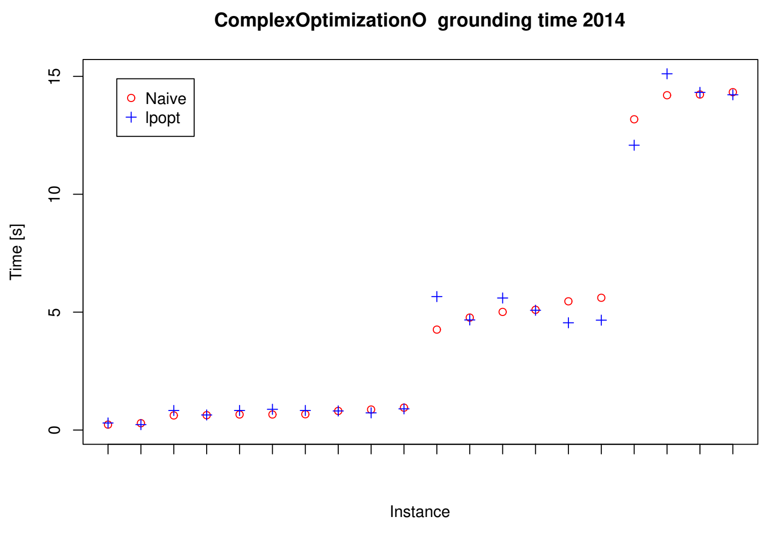

BottleFillingProblem solving time 2013
Naive
Ipopt
Time [s]
0
5
10
15
Instance

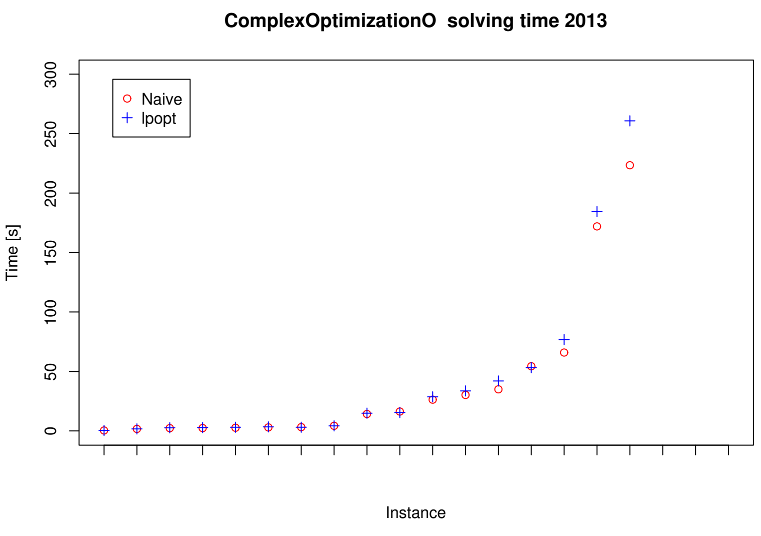

BottleFillingProblem sat-prepro time 2013
Naive
Ipopt
Time [s]
0
5
10
15
Instance

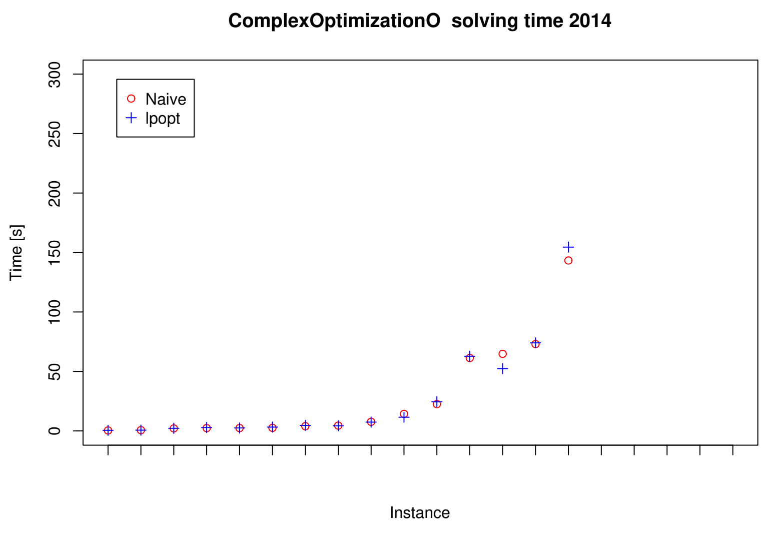

ComplexOptimizationO grounding size 2013
Grounding size
0
50000
100000
150000
200000
Instance
Naive
Ipopt

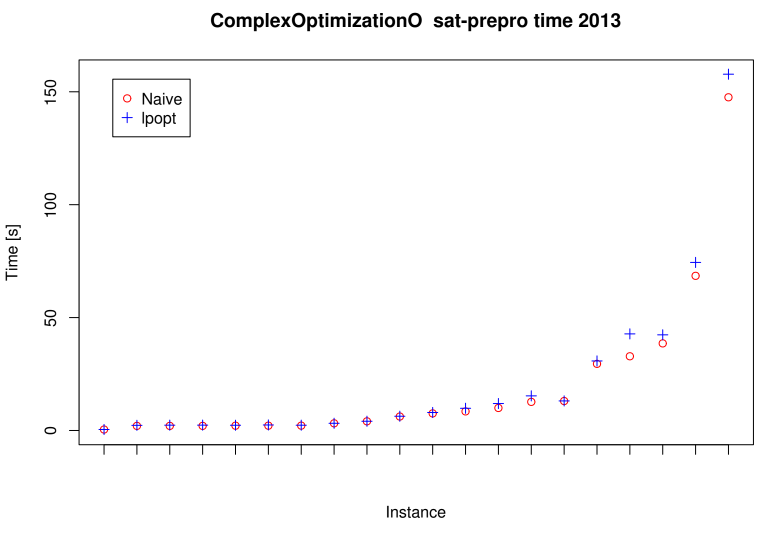

ComplexOptimizationO grounding size 2014
Grounding size
0
50000
100000
150000
200000
Instance
Naive
Ipopt

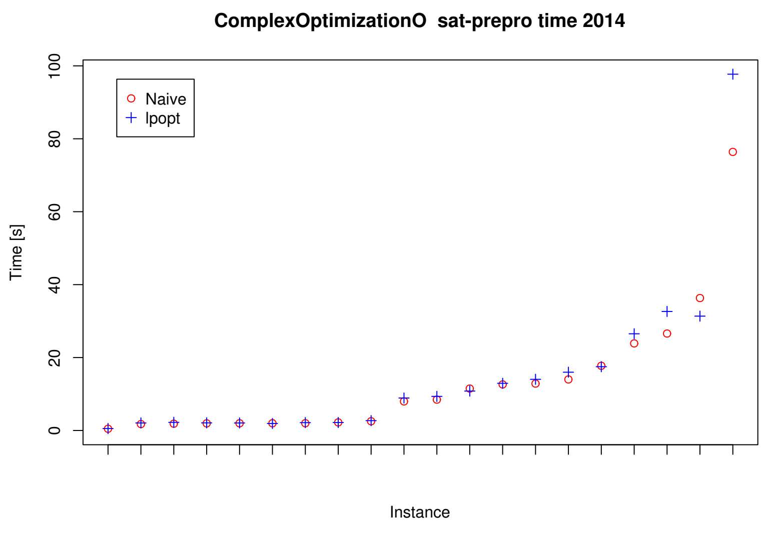

ComplexOptimizationO grounding time 2013
Time [s]
0
5
10
15
Instance
Naive
Ipopt

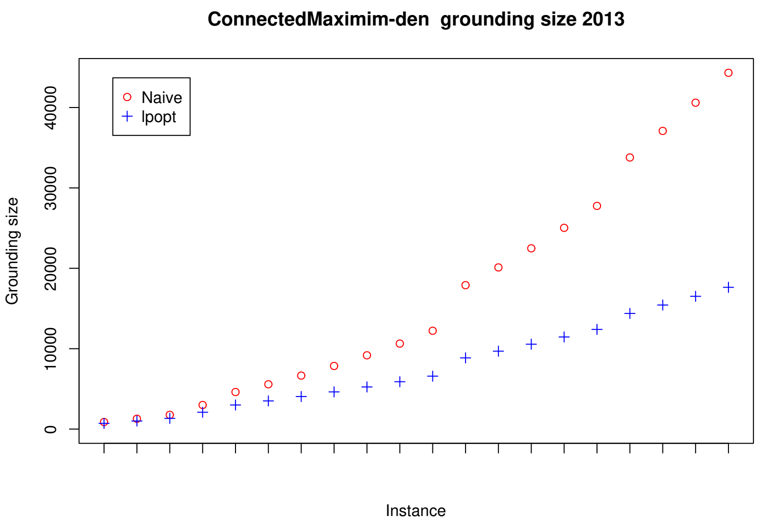

ComplexOptimizationO grounding time 2014
Time [s]
0
5
10
15
Instance
Naive
Ipopt

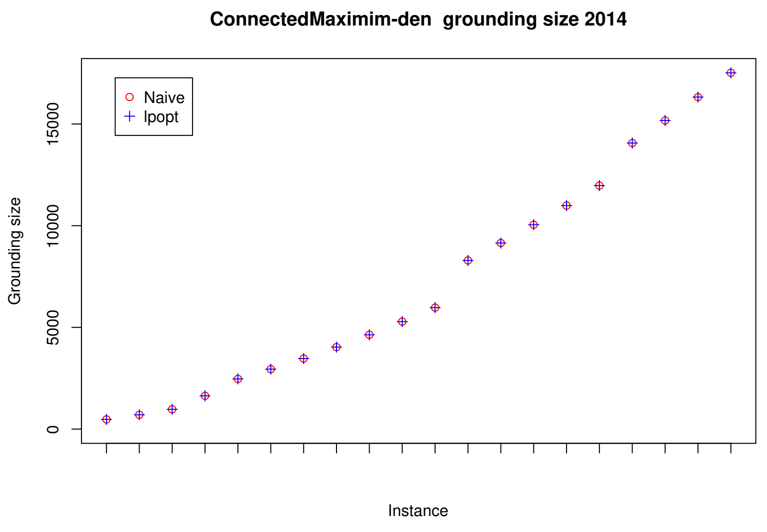

ComplexOptimizationO solving time 2013
Time [s]
0
50
100
150
200
250
300
Instance
Naive
Ipopt

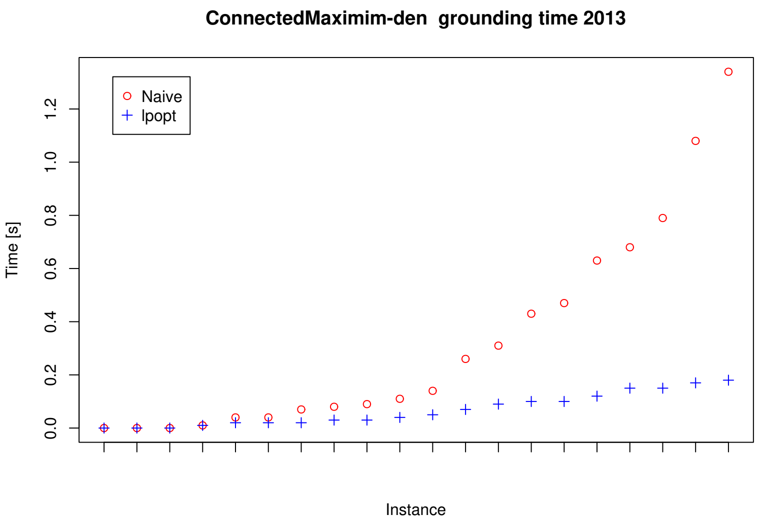

ComplexOptimizationO solving time 2014
Time [s]
0
50
100
150
200
250
300
Instance
Naive
Ipopt

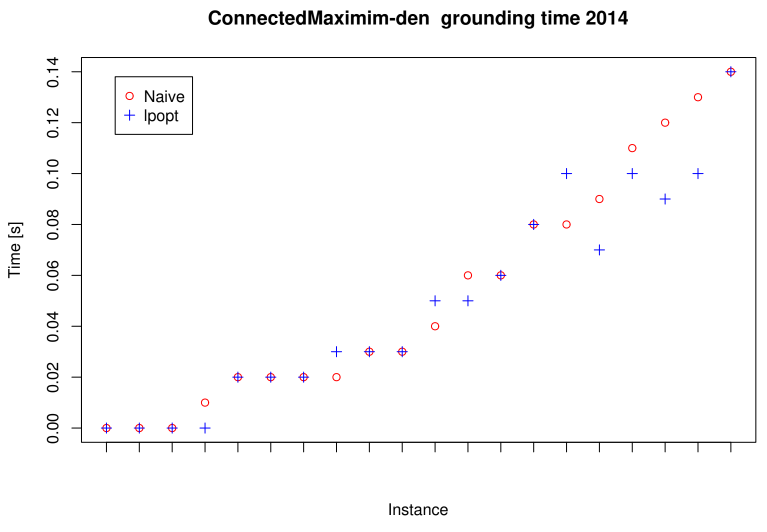

ComplexOptimizationO sat-prepro time 2013
Time [s]
0
50
100
150
Instance
Naive
Ipopt

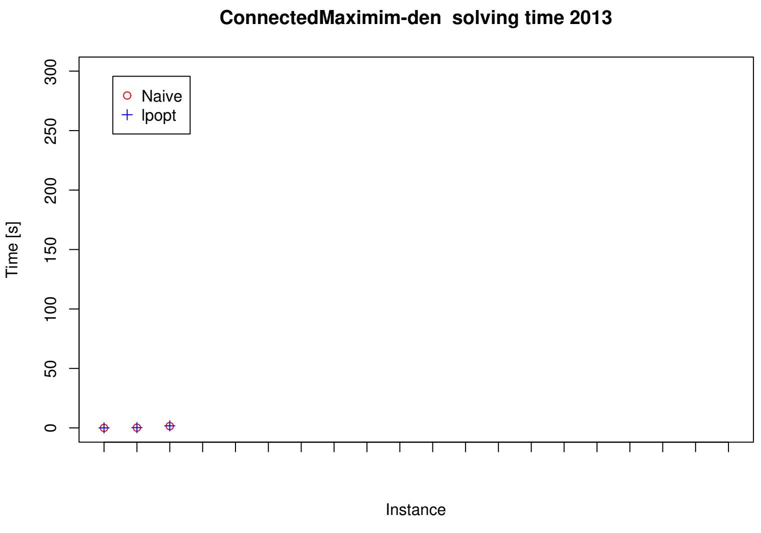

ComplexOptimizationO sat-prepro time 2014
Time [s]
0
20
40
60
80
100
Instance
Naive
Ipopt

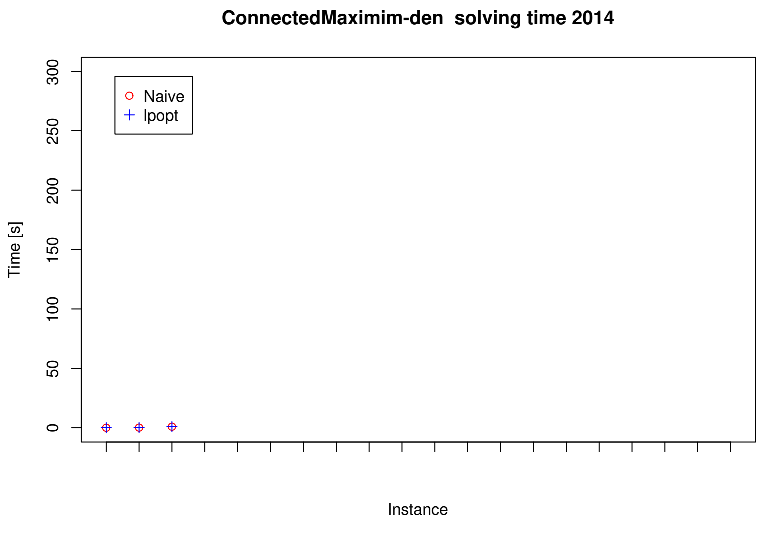

ConnectedMaximim-den grounding size 2013
Naive
Ipopt
Grounding size
0
10000
20000
30000
40000
Instance

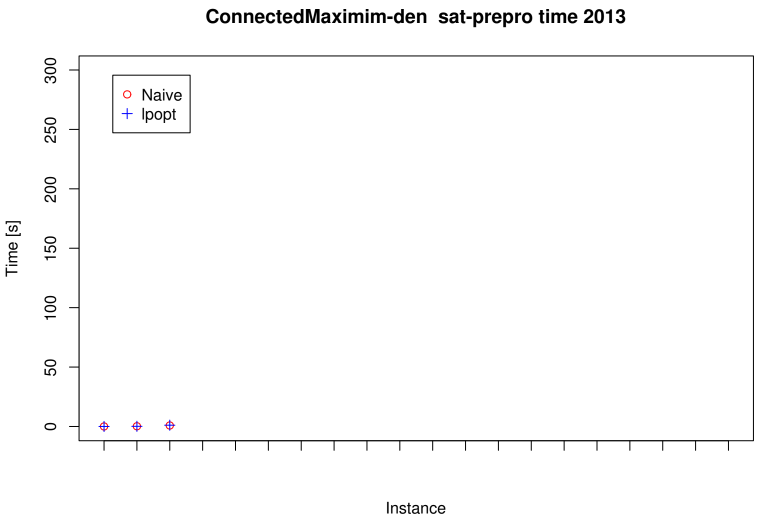

ConnectedMaximim-den grounding size 2014
Naive
Ipopt
Grounding size
0
5000
10000
15000
Instance

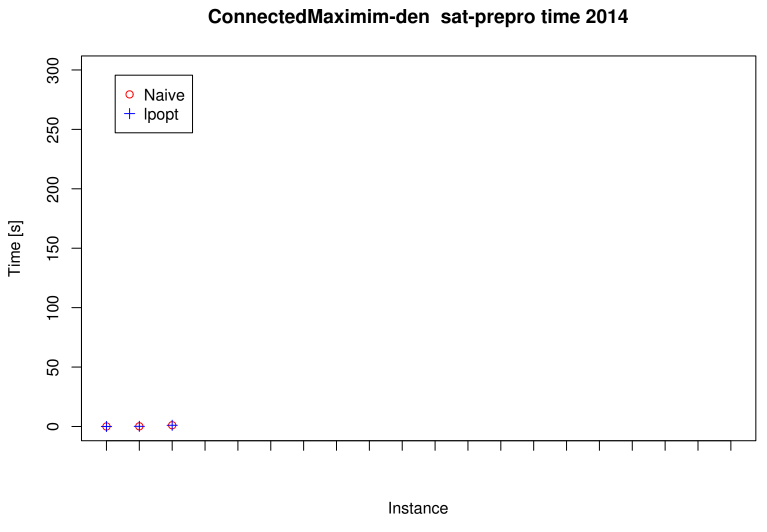

ConnectedMaximim-den grounding time 2013
Naive
Ipopt
Time [s]
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Instance

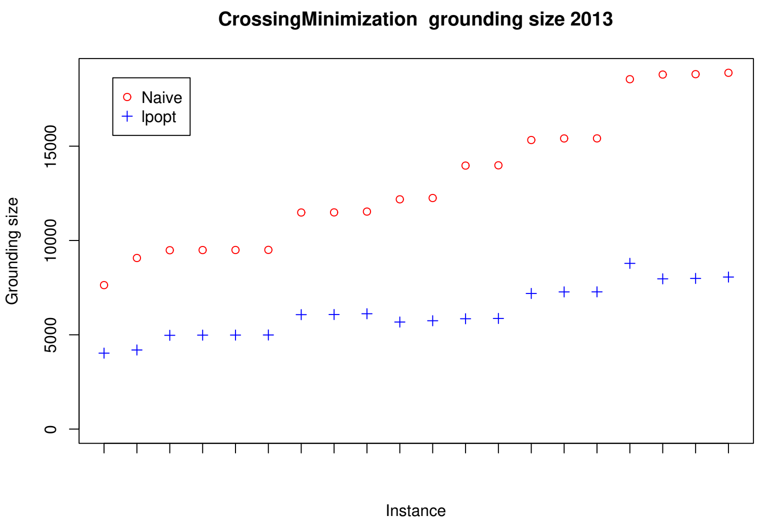

ConnectedMaximim-den grounding time 2014
Naive
Ipopt
Time [s]
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
Instance

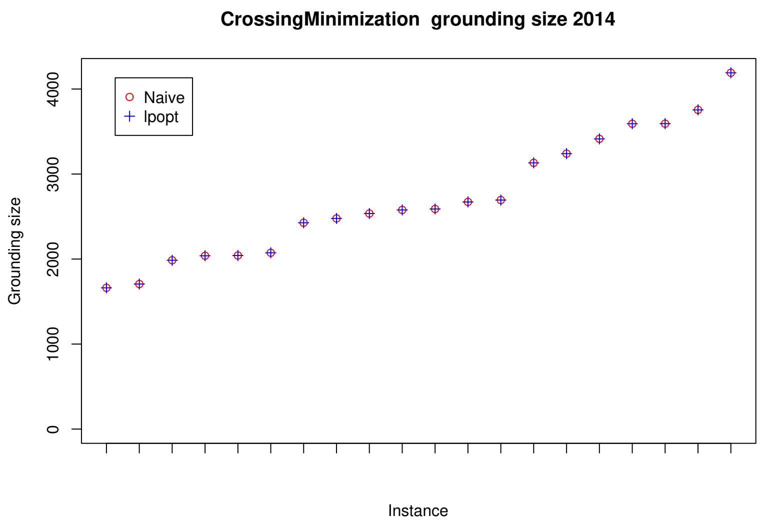

ConnectedMaximim-den solving time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

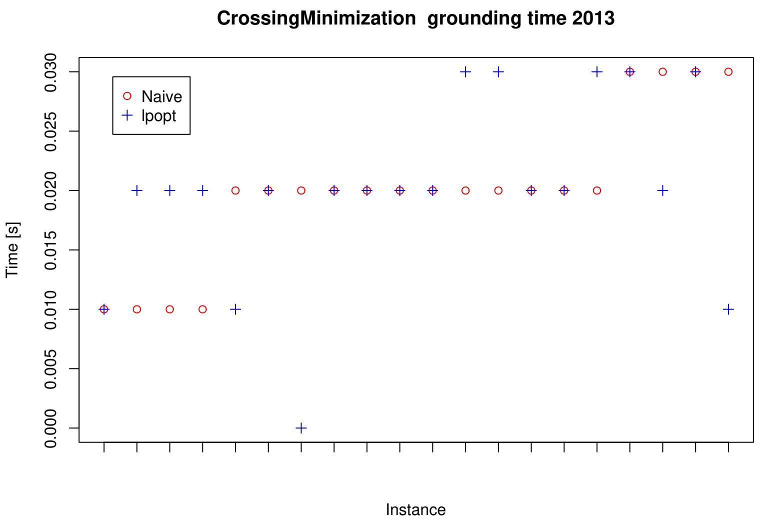

ConnectedMaximim-den solving time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

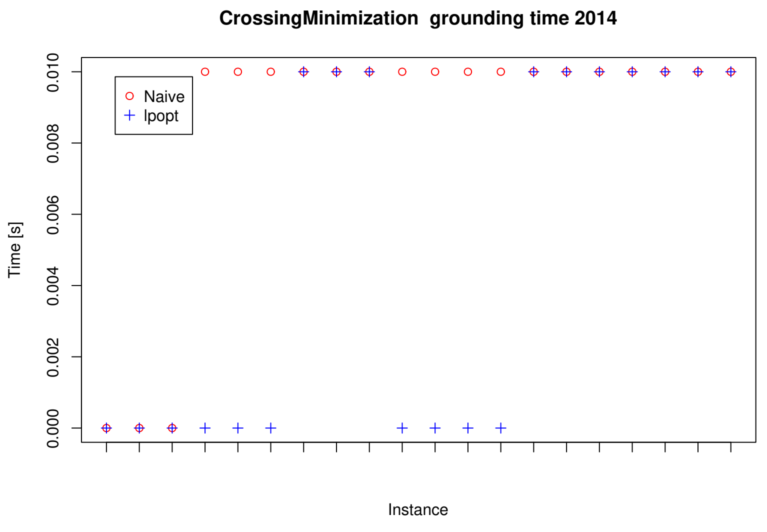

ConnectedMaximim-den sat-prepro time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

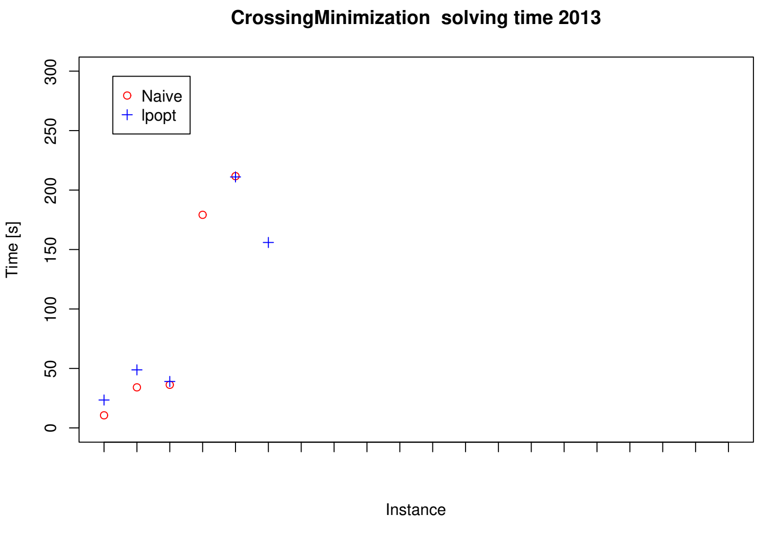

ConnectedMaximim-den sat-prepro time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

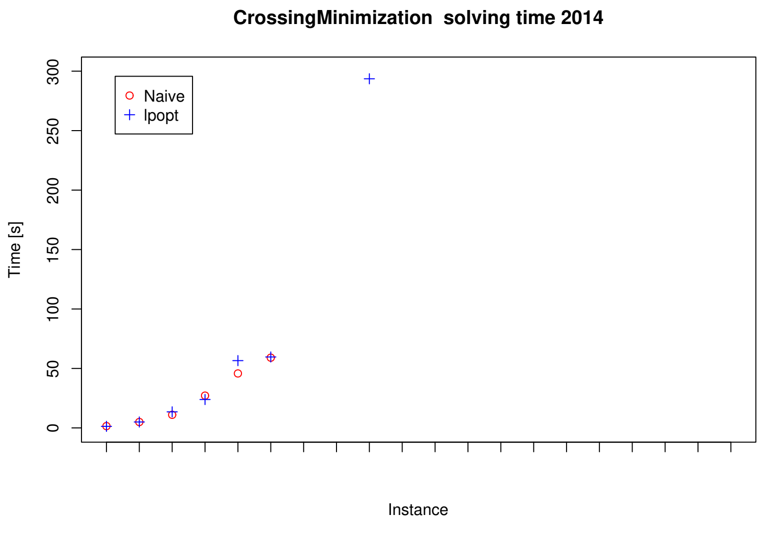

CrossingMinimization grounding size 2013
Naive
Ipopt
Grounding size
0
5000
10000
15000
Instance

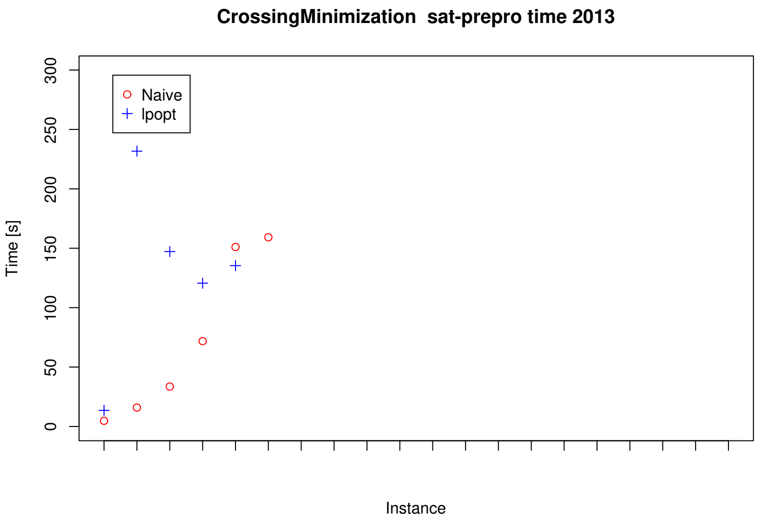

CrossingMinimization grounding size 2014
Naive
Ipopt
Grounding size
0
1000
2000
3000
4000
Instance

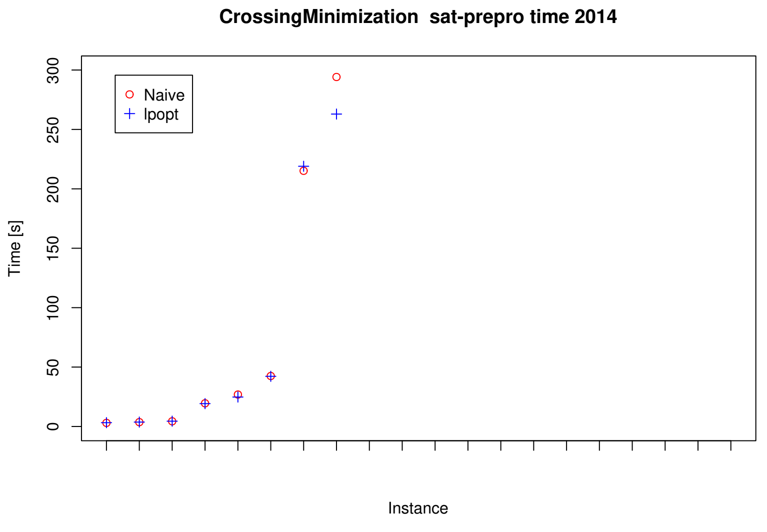

CrossingMinimization grounding time 2013
Naive
Ipopt
Time [s]
0.000
0.005
0.010
0.015
0.020
0.025
0.030
Instance

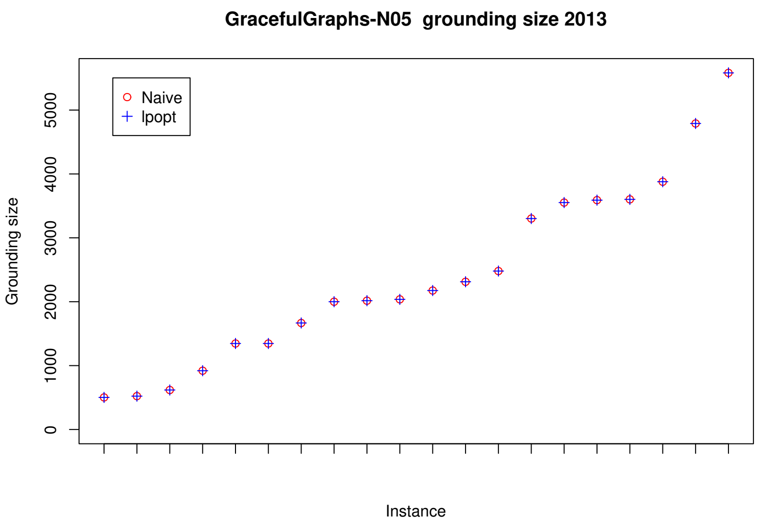

CrossingMinimization grounding time 2014
Naive
Ipopt
Time [s]
0.000
0.002
0.004
0.006
0.008
0.010
Instance

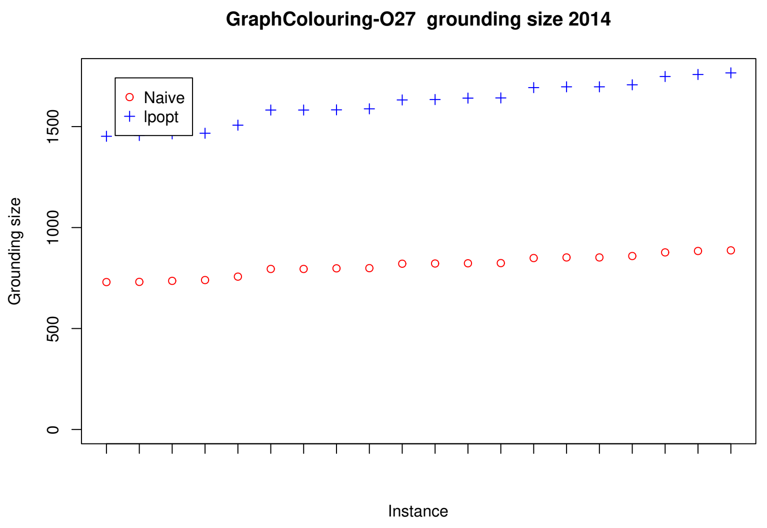

CrossingMinimization solving time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

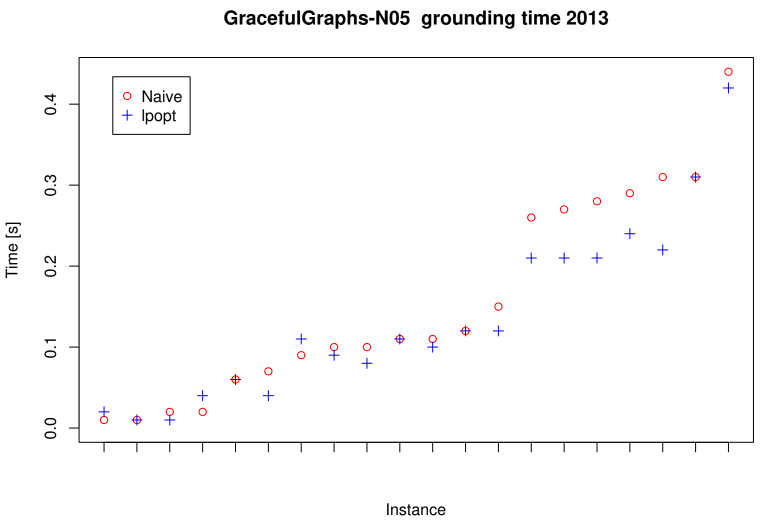

CrossingMinimization solving time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

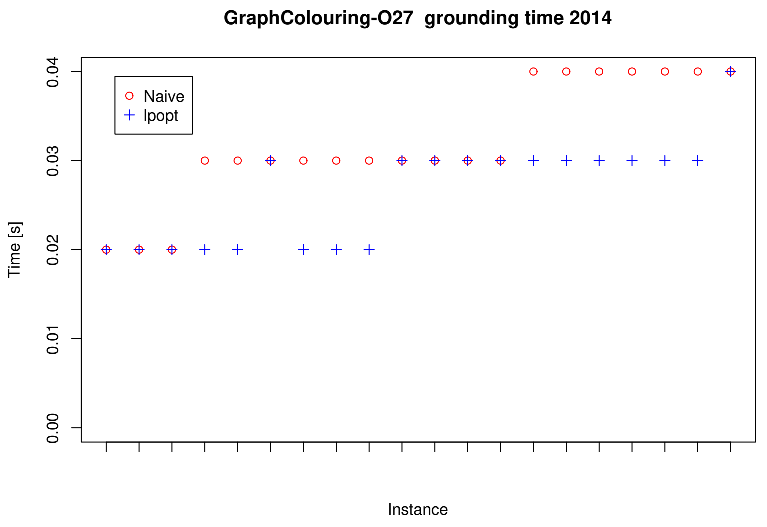

CrossingMinimization sat-prepro time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

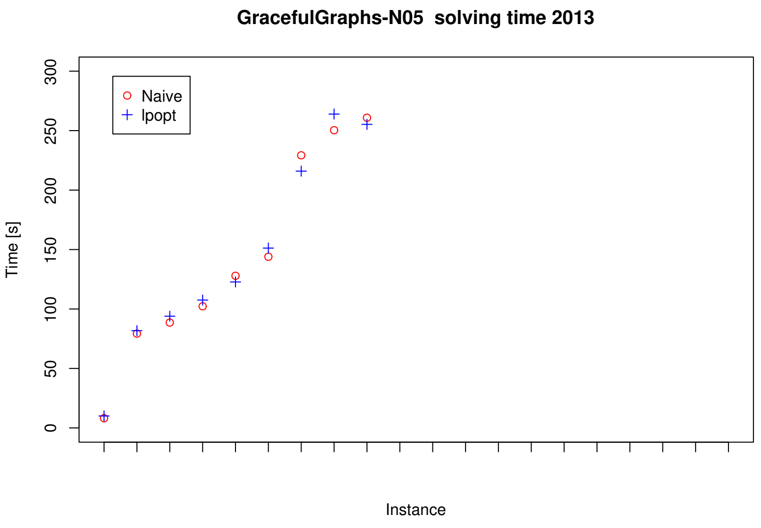

CrossingMinimization sat-prepro time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

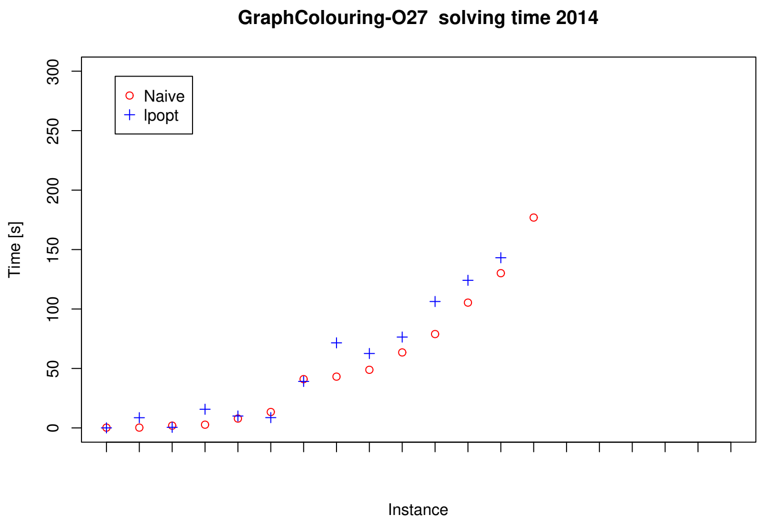

GracefulGraphs-N05 grounding size 2013
Naive
Ipopt
Grounding size
Instance
0
1000
2000
3000
4000
5000

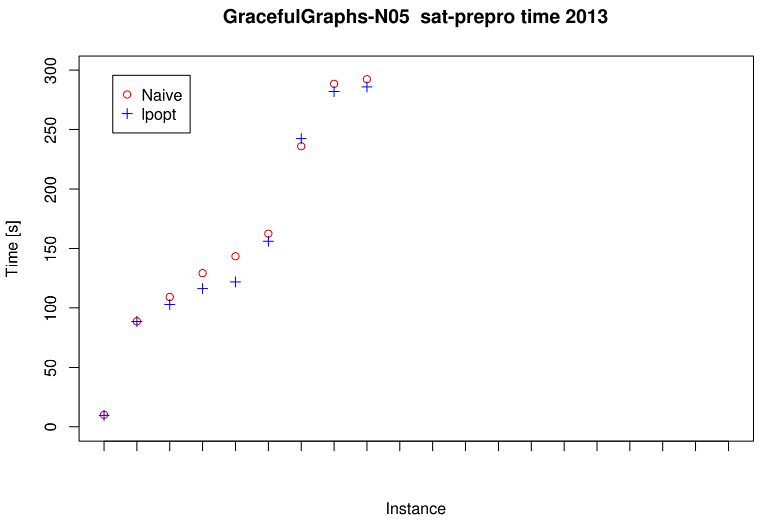

GraphColouring-O27 grounding size 2014
Naive
Ipopt
Grounding size
Instance
0
500
1000
1500

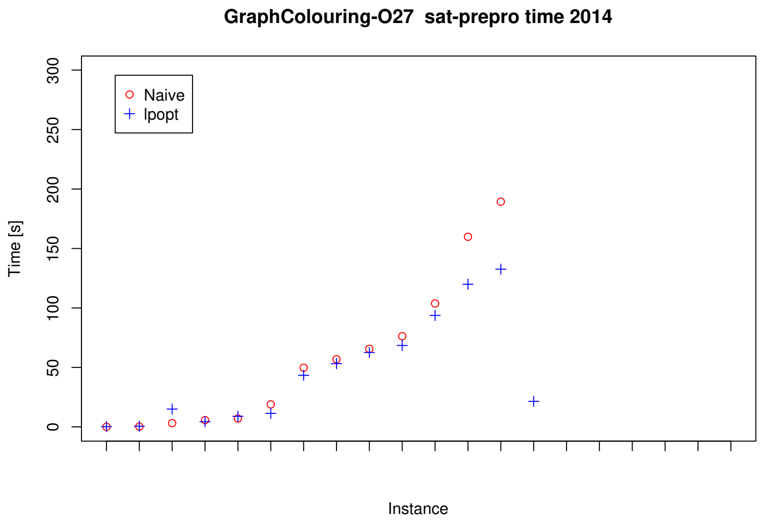

GracefulGraphs-N05 grounding time 2013
Naive
Ipopt
Time [s]
Instance
0.0
0.1
0.2
0.3
0.4

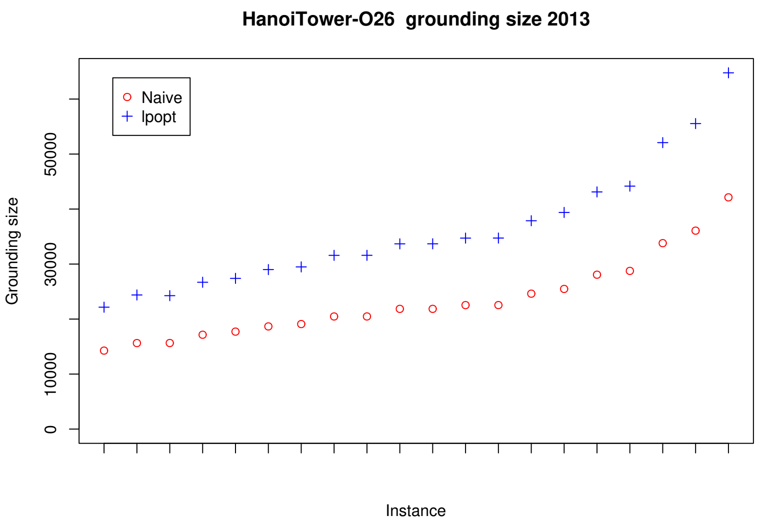

GraphColouring-O27 grounding time 2014
Naive
Ipopt
Time [s]
Instance
0.00
0.01
0.02
0.03
0.04

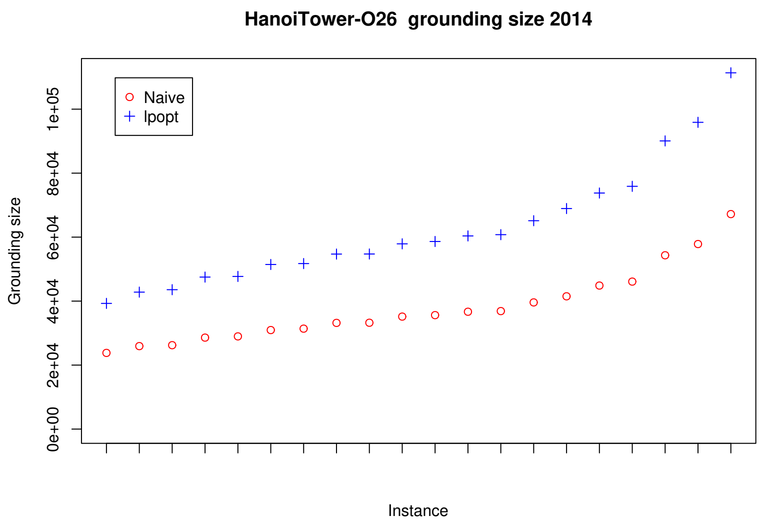

GracefulGraphs-N05 solving time 2013
Naive
Ipopt
Time [s]
Instance
0
50
100
150
200
250
300

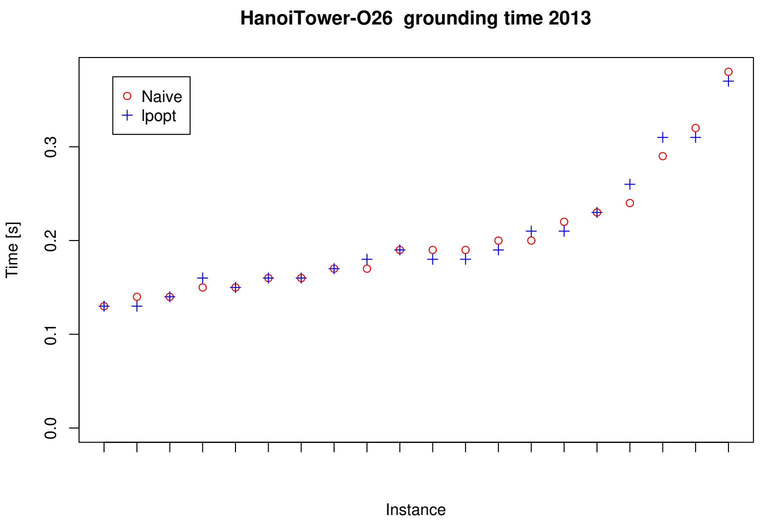

GraphColouring-O27 solving time 2014
Naive
Ipopt
Time [s]
Instance
0
50
100
150
200
250
300

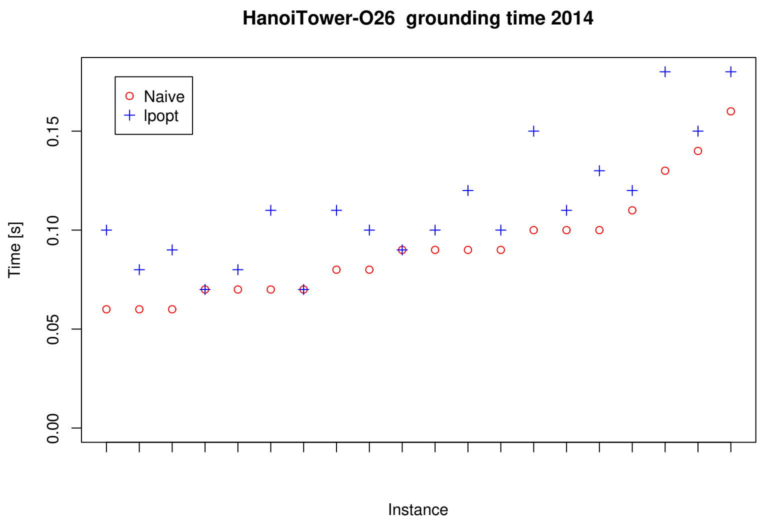

GracefulGraphs-N05 sat-prepro time 2013
Naive
Ipopt
Time [s]
Instance
0
50
100
150
200
250
300

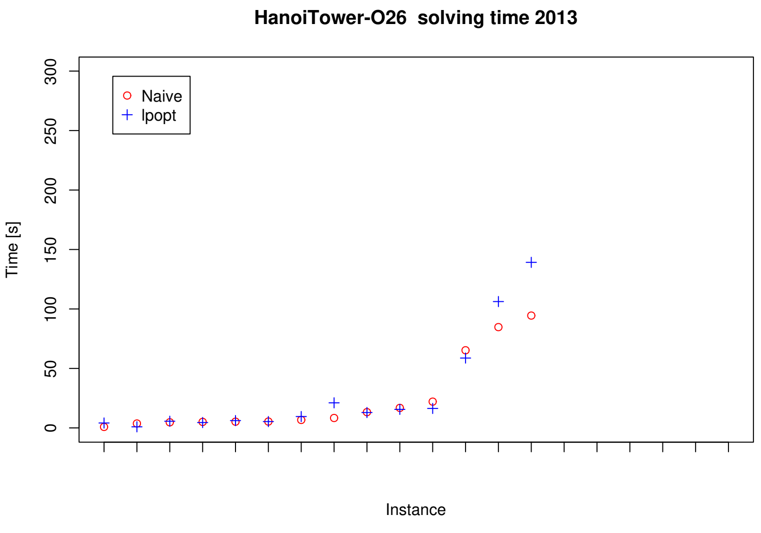

GraphColouring-O27 sat-prepro time 2014
Naive
Ipopt
Time [s]
Instance
0
50
100
150
200
250
300

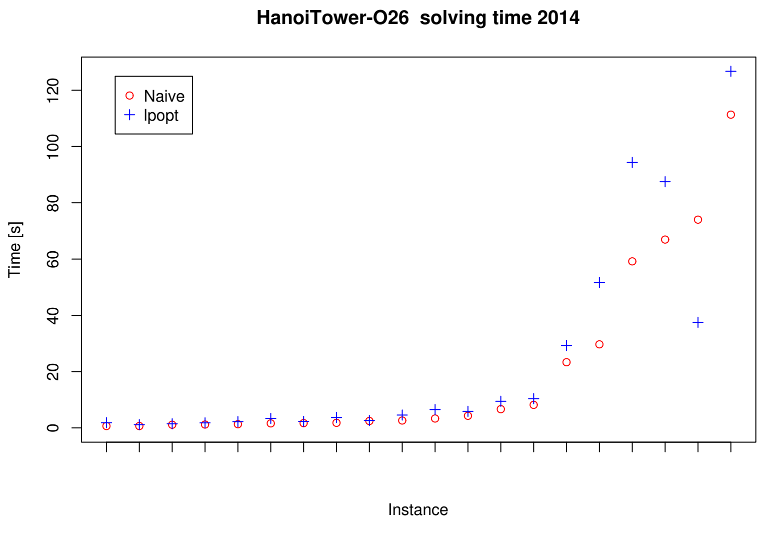

HanoiTower-O26 grounding size 2013
Grounding size
Instance
Naive
Ipopt
0
10000
30000
50000

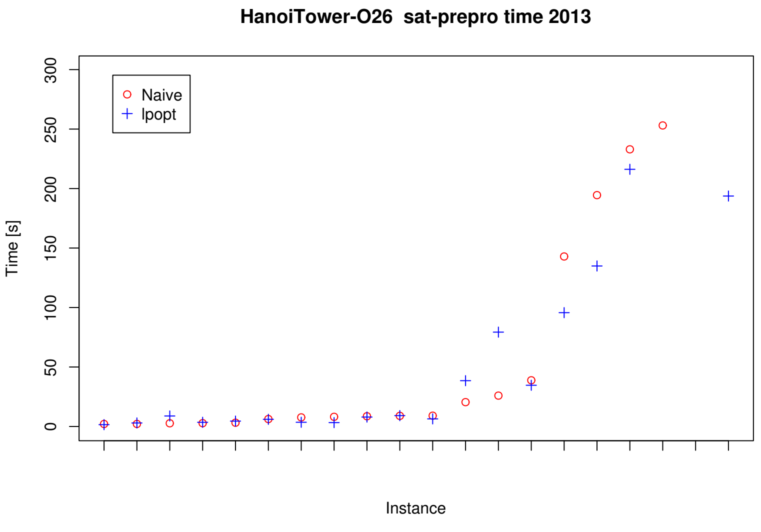

HanoiTower-O26 grounding size 2014
Grounding size
Instance
Naive
Ipopt
0e+00
2e+04
4e+04
6e+04
8e+04
1e+05

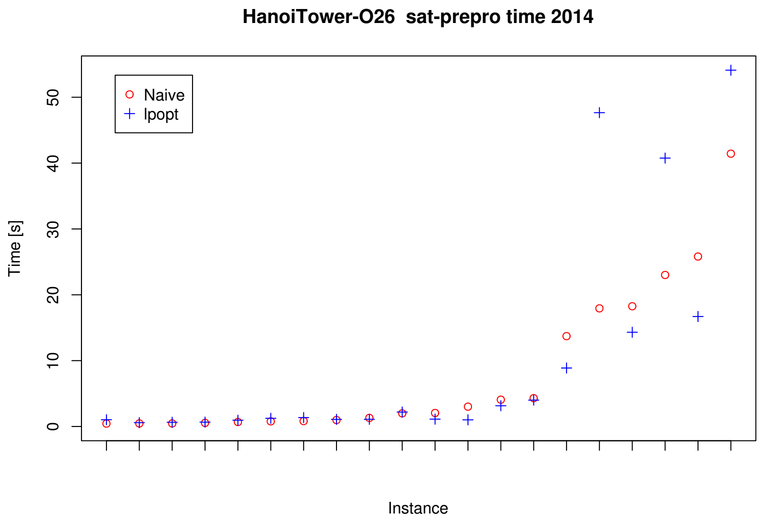

HanoiTower-O26 grounding time 2013
Time [s]
Instance
Naive
Ipopt
0.0
0.1
0.2
0.3

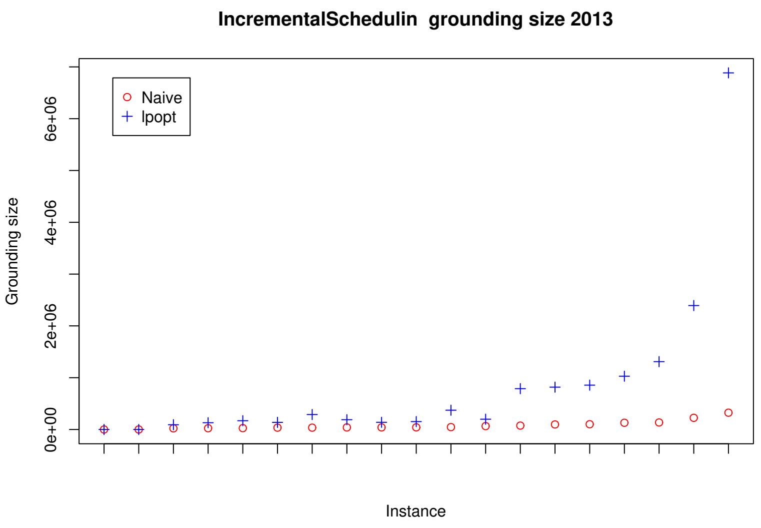
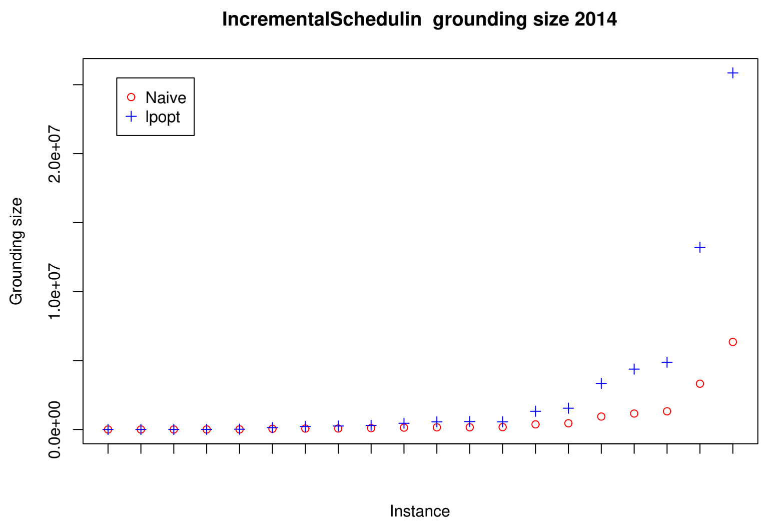
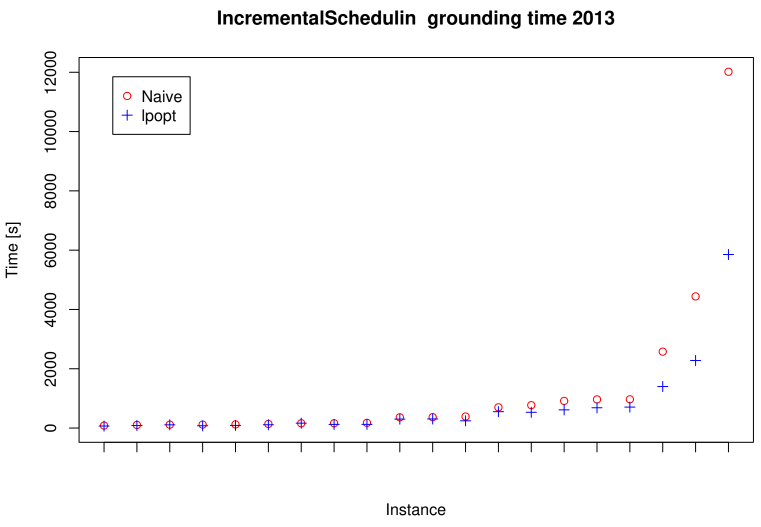
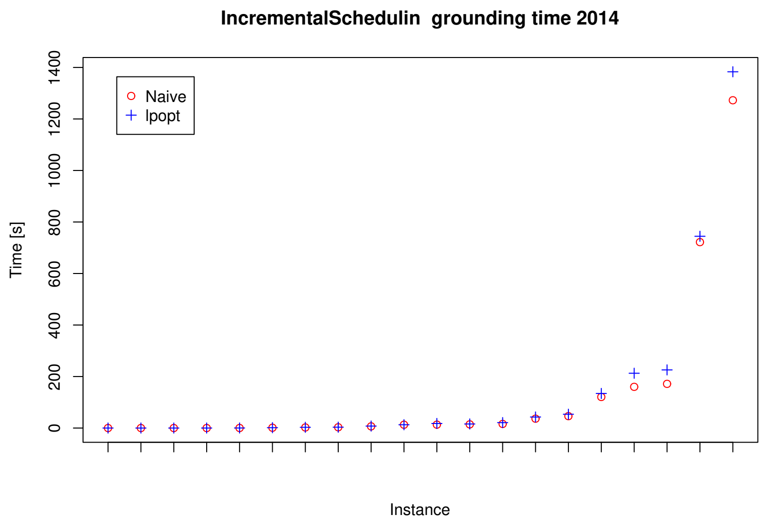
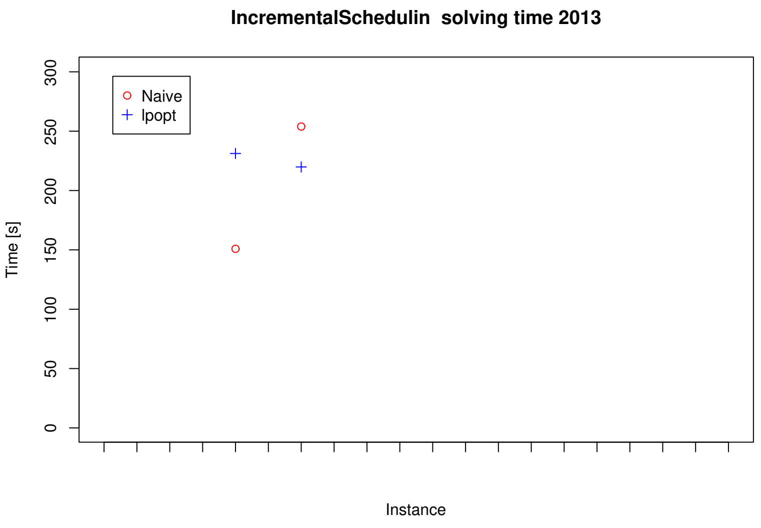
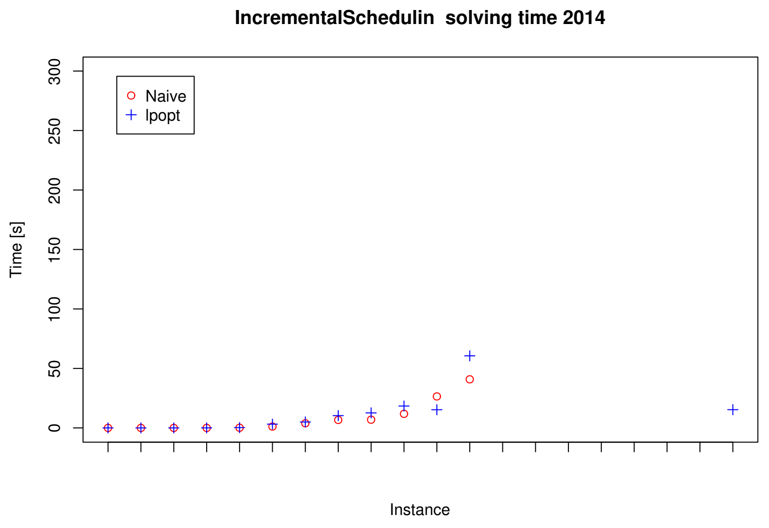
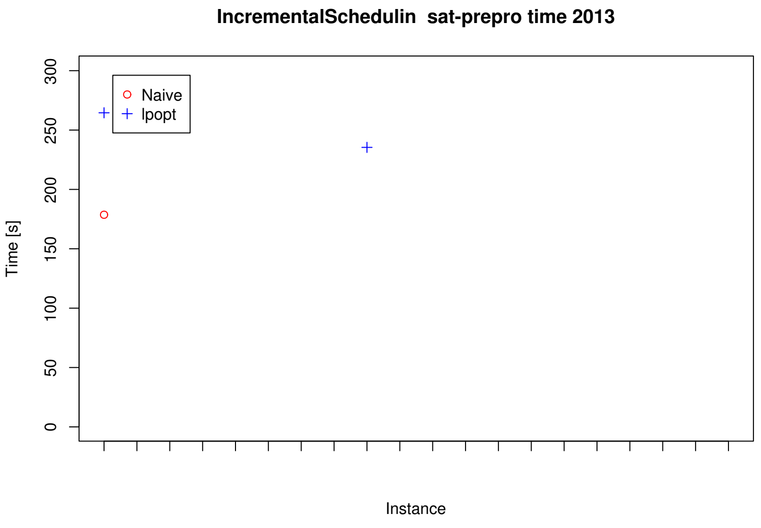
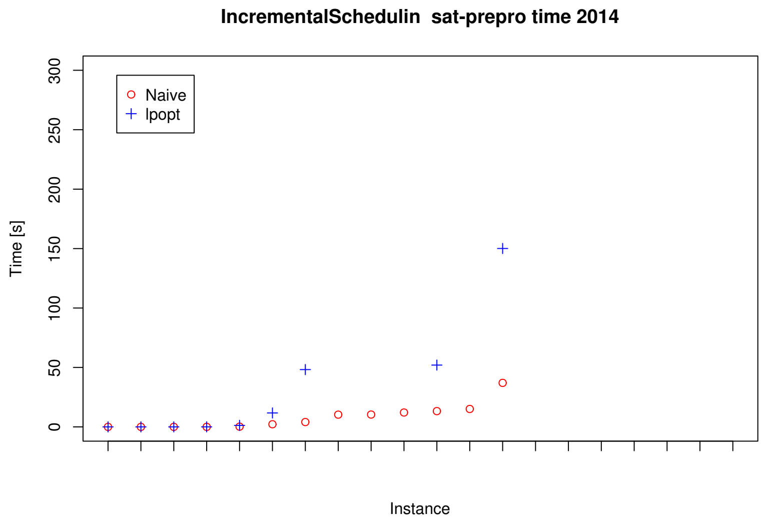

HanoiTower-O26 grounding time 2014
Time [s]
Instance
Naive
Ipopt
0.00
0.05
0.10
0.15

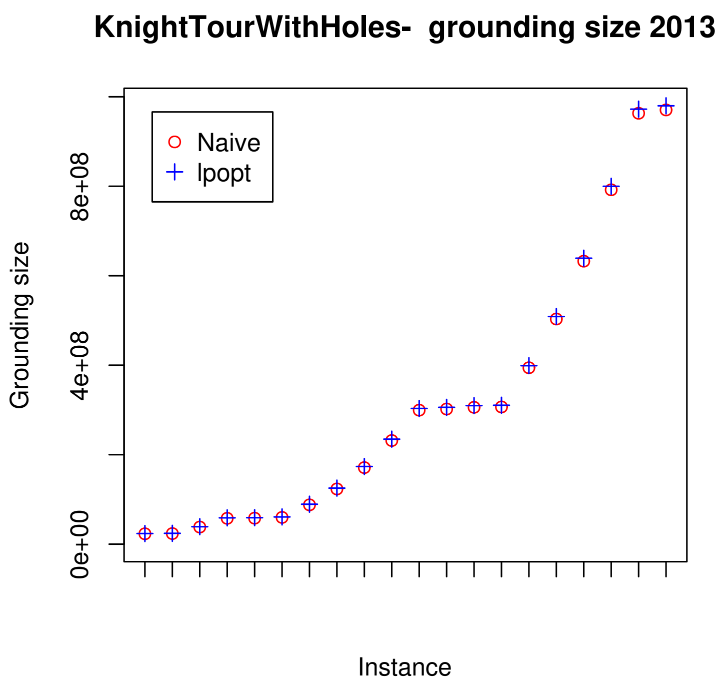

HanoiTower-O26 solving time 2013
Time [s]
Instance
Naive
Ipopt
0
50
100
150
200
250
300

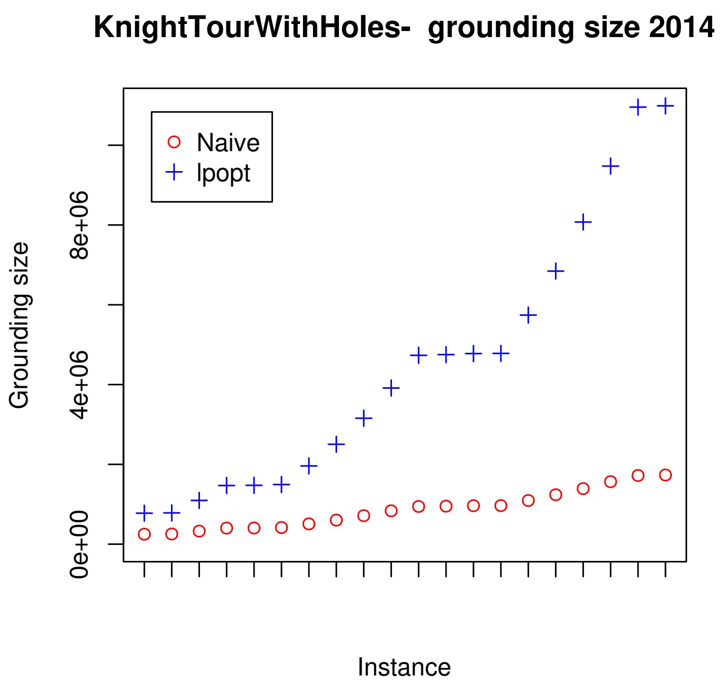

HanoiTower-O26 solving time 2014
Time [s]
Instance
Naive
Ipopt
0
20
40
60
80
100
120

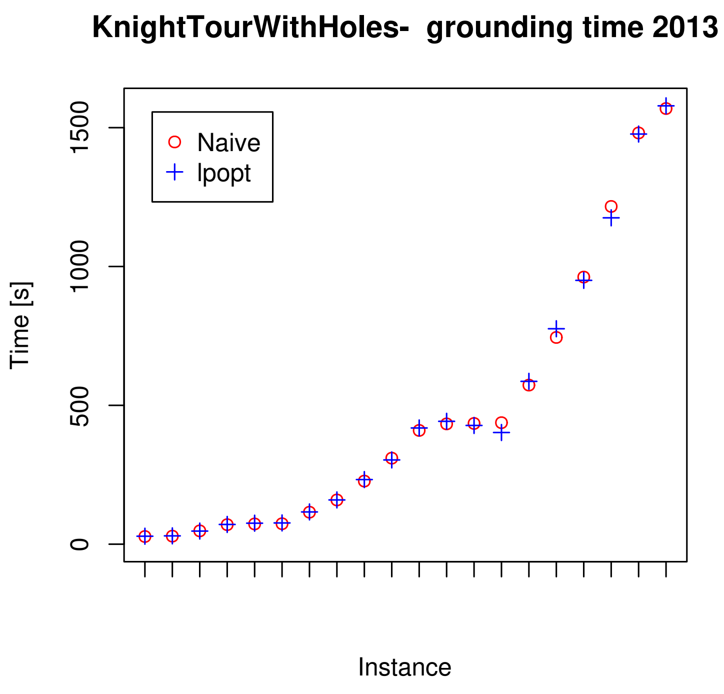

HanoiTower-O26 sat-prepro time 2013
Time [s]
Instance
Naive
Ipopt
0
50
100
150
200
250
300

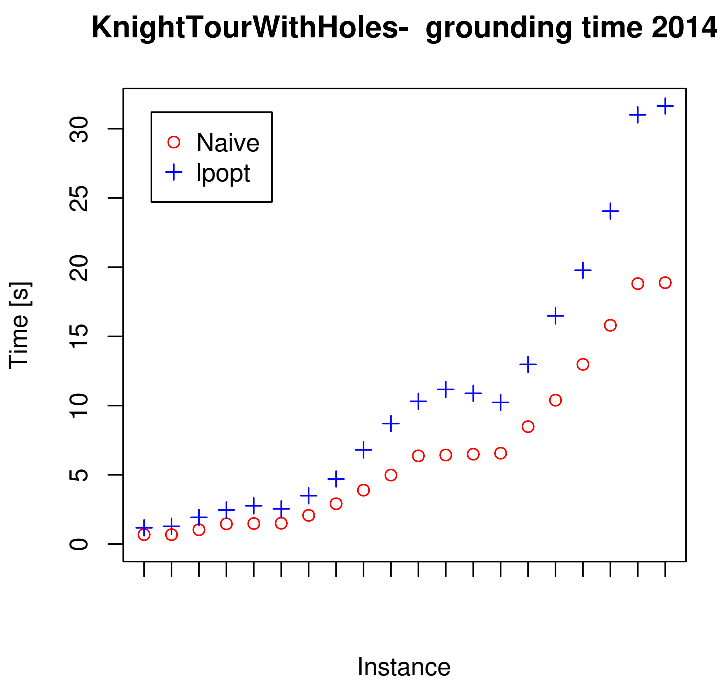

HanoiTower-O26 sat-prepro time 2014
Time [s]
Instance
Naive
Ipopt
0
10
20
30
40
50

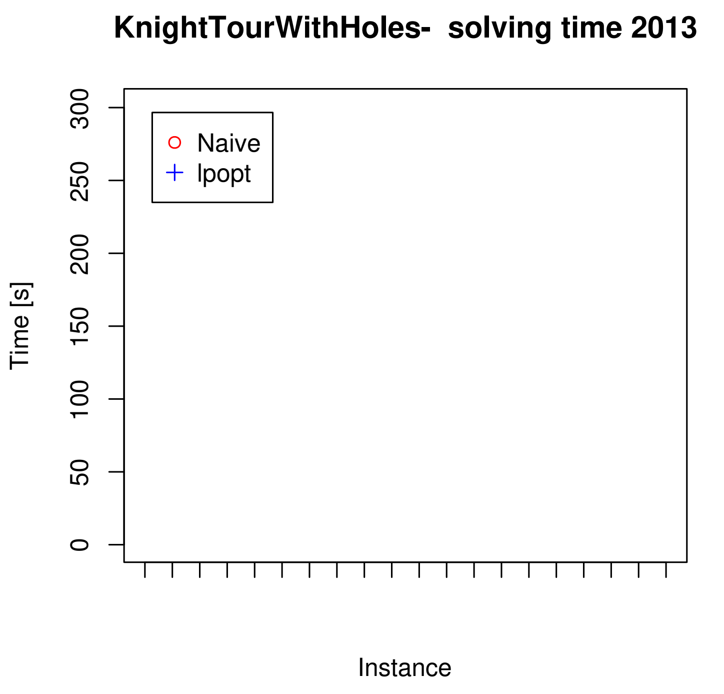

IncrementalSchedulin grounding size 2013
Naive
Ipopt
Grounding size
0e+00
2e+06
4e+06
6e+06
Instance

IncrementalSchedulin grounding size 2014
Naive
Ipopt
Grounding size
0.0e+00
1.0e+07
2.0e+07
Instance

IncrementalSchedulin grounding time 2013
Naive
Ipopt
Time [s]
0
2000
4000
6000
8000
10000
12000
Instance

IncrementalSchedulin grounding time 2014
Naive
Ipopt
Time [s]
0
200
400
600
800
1000
1200
1400
Instance

IncrementalSchedulin solving time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

IncrementalSchedulin solving time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

IncrementalSchedulin sat-prepro time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

IncrementalSchedulin sat-prepro time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

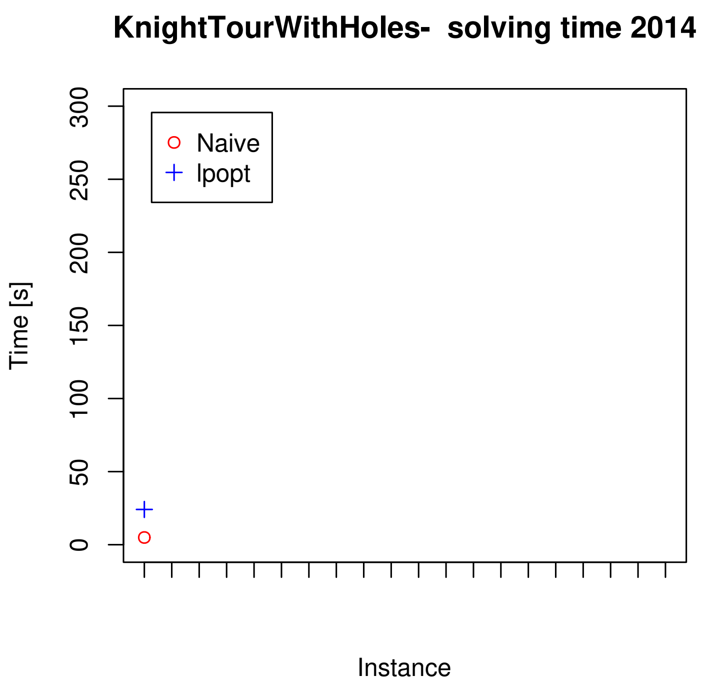

KnightTourWithHoles- grounding size 2013
Naive
Ipopt
Grounding size
0e+00
4e+08
8e+08
Instance

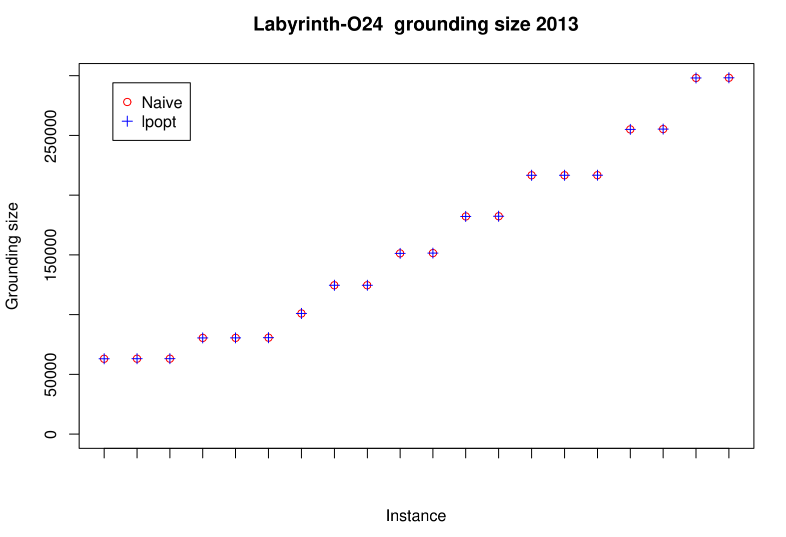

KnightTourWithHoles- grounding size 2014
Naive
Ipopt
Grounding size
0e+00
4e+06
8e+06
Instance

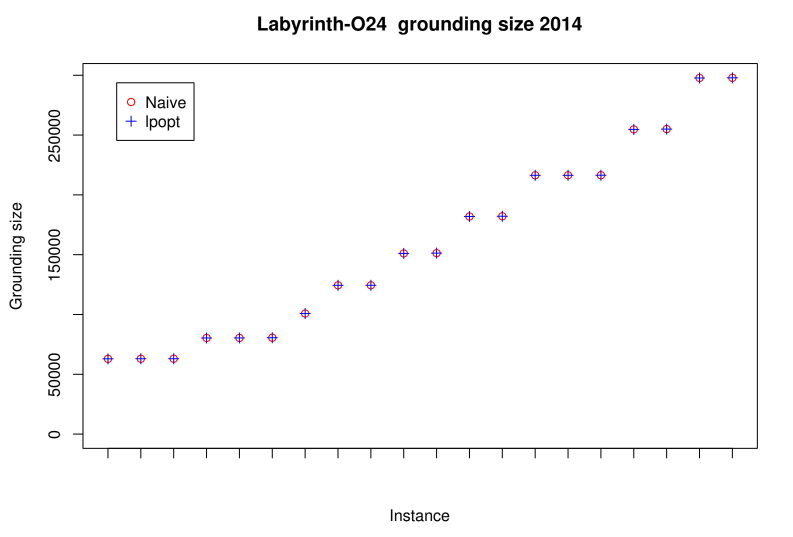

KnightTourWithHoles- grounding time 2013
Naive
Ipopt
Time [s]
0
500
1000
1500
Instance

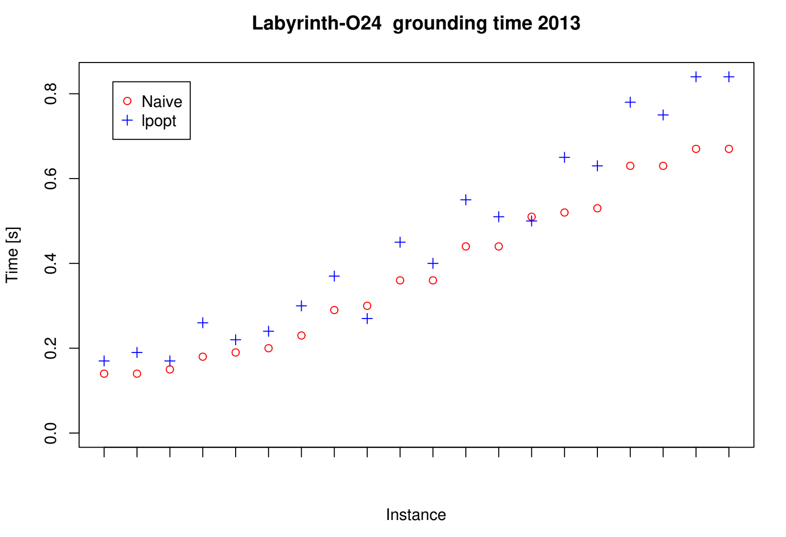

KnightTourWithHoles- grounding time 2014
Naive
Ipopt
Time [s]
0
5
10
15
20
25
30
Instance

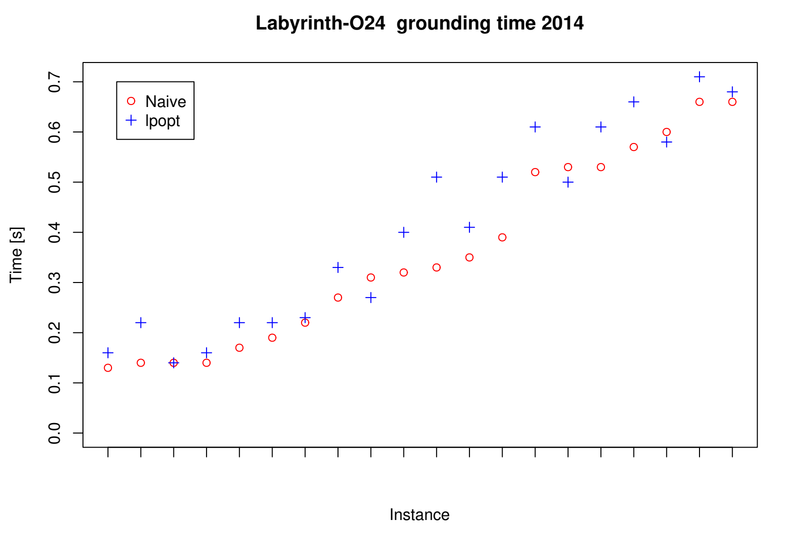

KnightTourWithHoles- solving time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

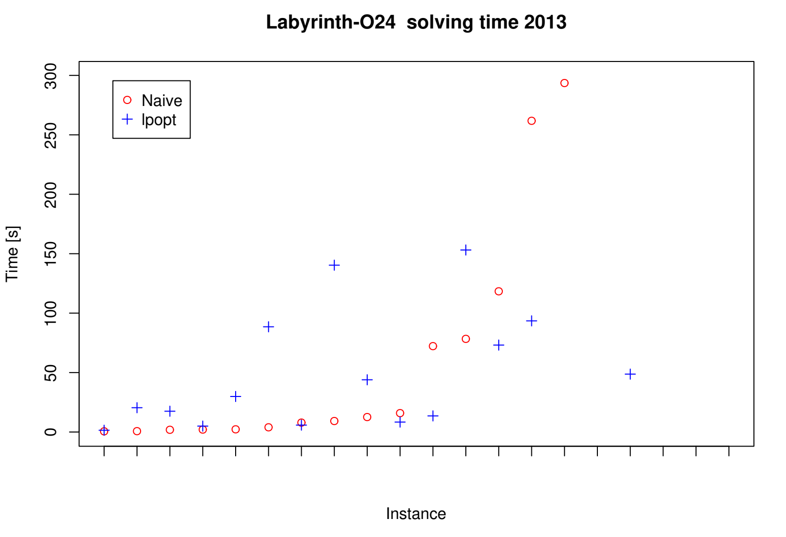

KnightTourWithHoles- solving time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

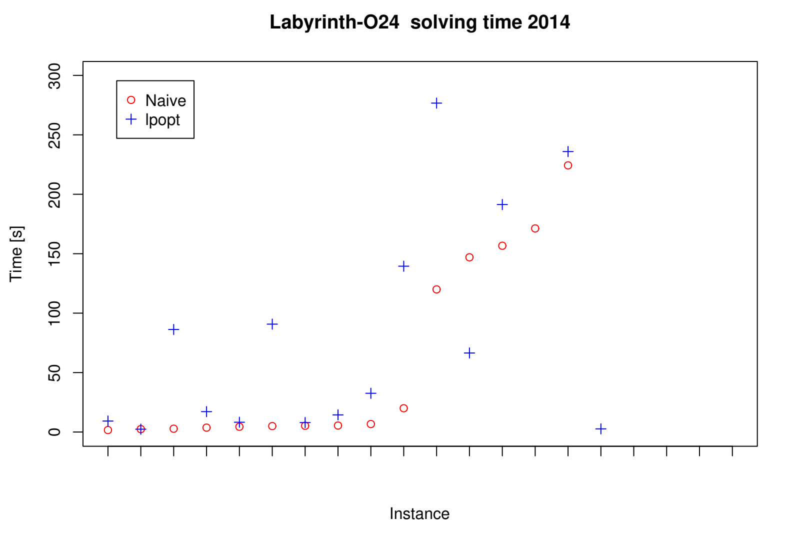

Labyrinth-O24 grounding size 2013
Naive
Ipopt
Grounding size
0
50000
150000
250000
Instance

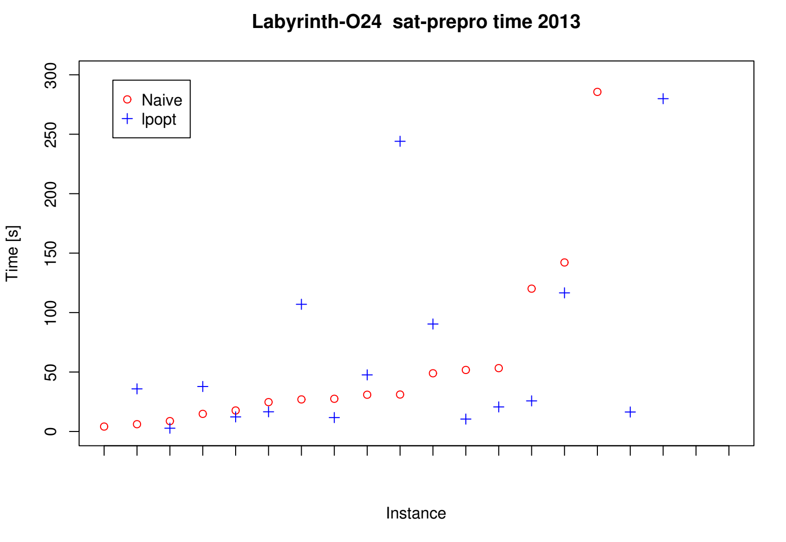

Labyrinth-O24 grounding size 2014
Naive
Ipopt
Grounding size
0
50000
150000
250000
Instance

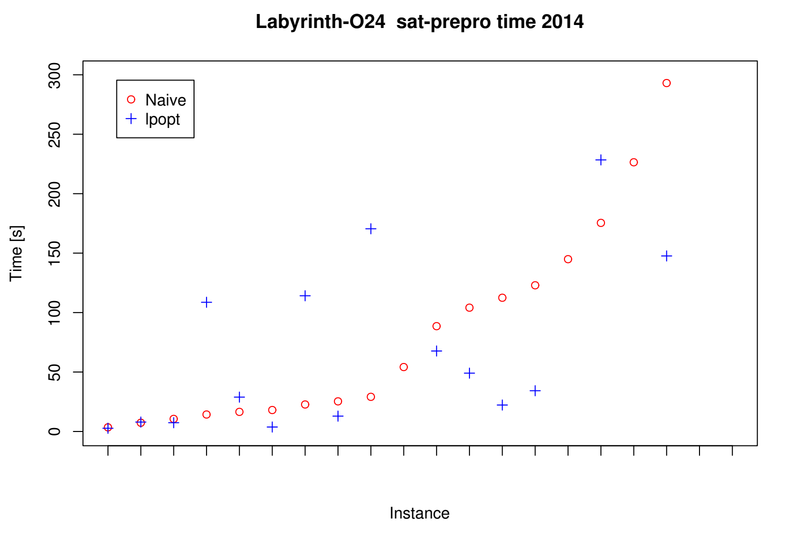

Labyrinth-O24 grounding time 2013
Naive
Ipopt
Time [s]
0.0
0.2
0.4
0.6
0.8
Instance

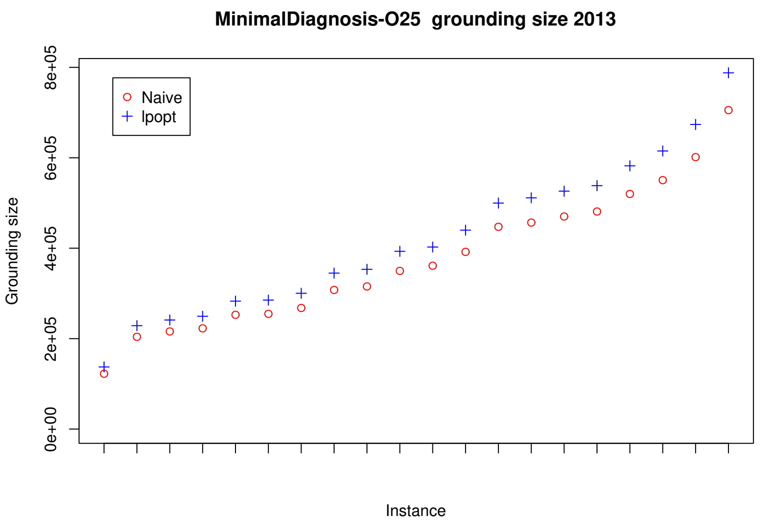

Labyrinth-O24 grounding time 2014
Naive
Ipopt
Time [s]
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
Instance

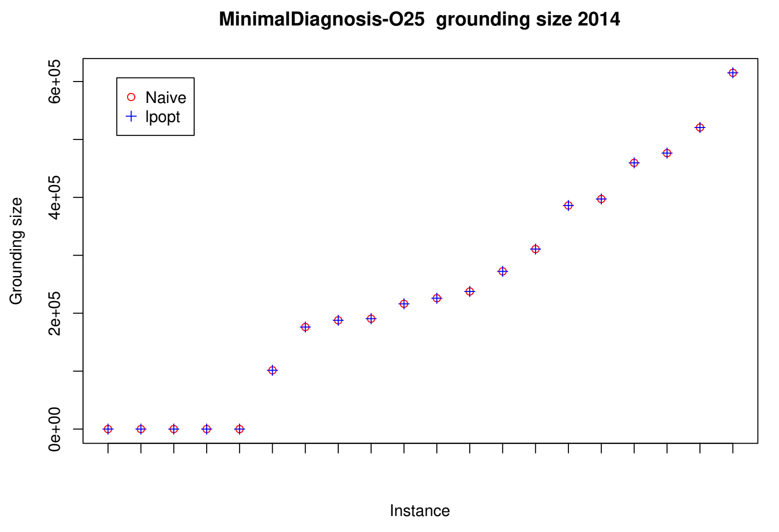

Labyrinth-O24 solving time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

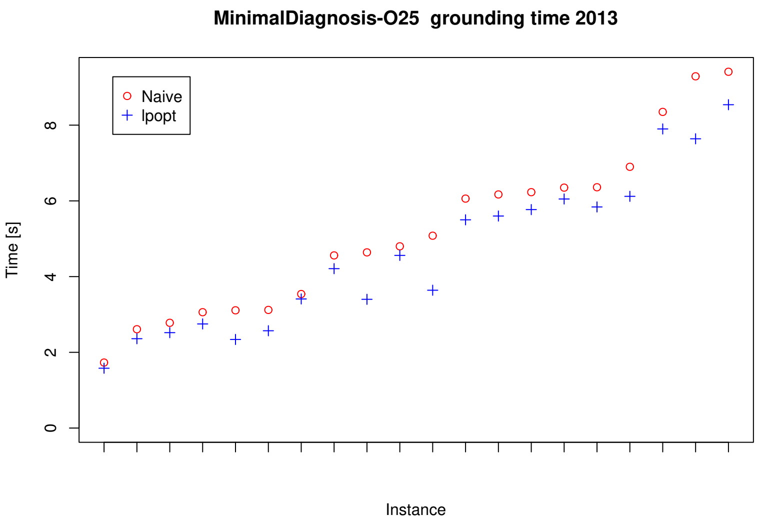

Labyrinth-O24 solving time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

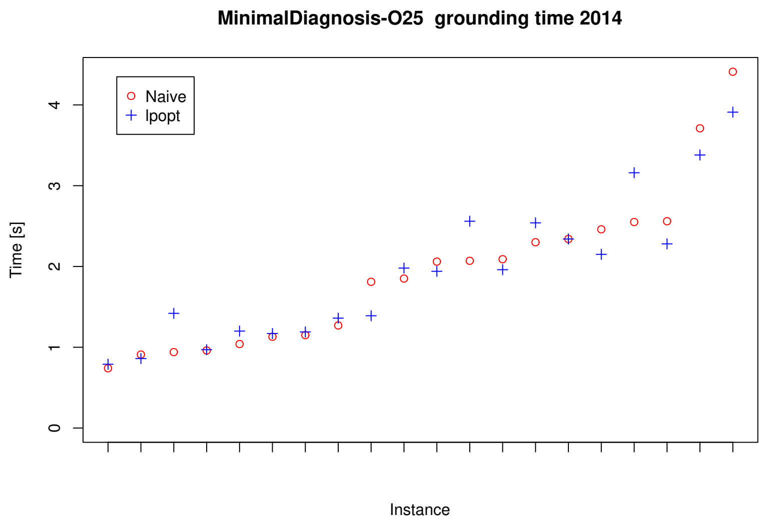

Labyrinth-O24 sat-prepro time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

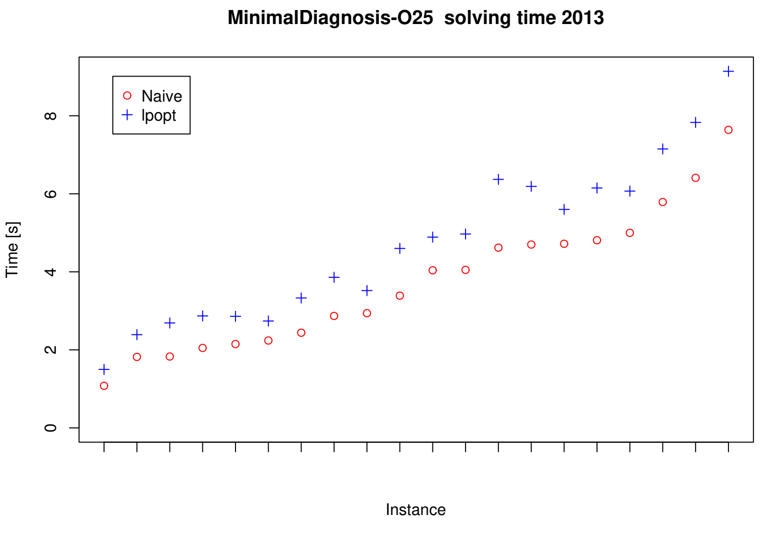

Labyrinth-O24 sat-prepro time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

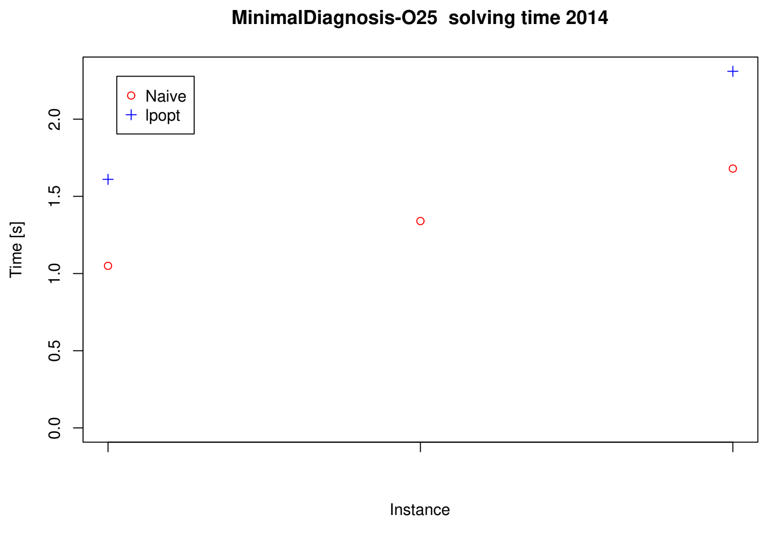

MinimalDiagnosis-O25 grounding size 2013
○ Naive
+ Ipopt
Grounding size
0e+00
2e+05
4e+05
6e+05
8e+05
Instance

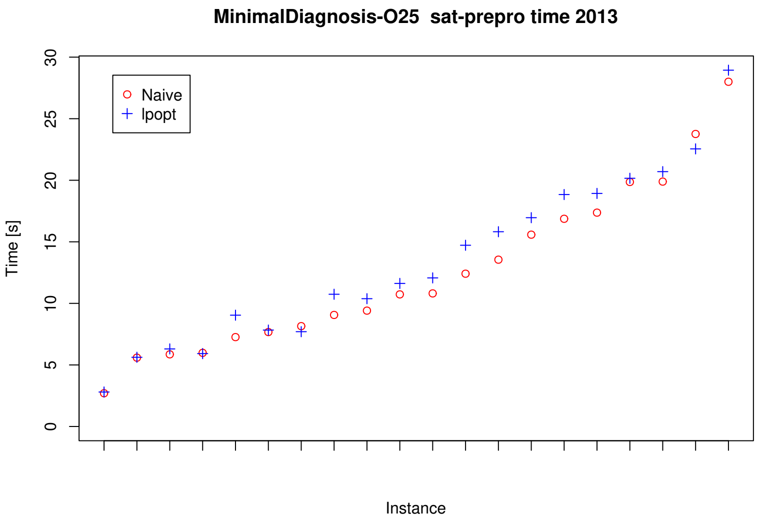

MinimalDiagnosis-O25 grounding size 2014
○ Naive
+ Ipopt
Grounding size
0e+00
2e+05
4e+05
6e+05
Instance

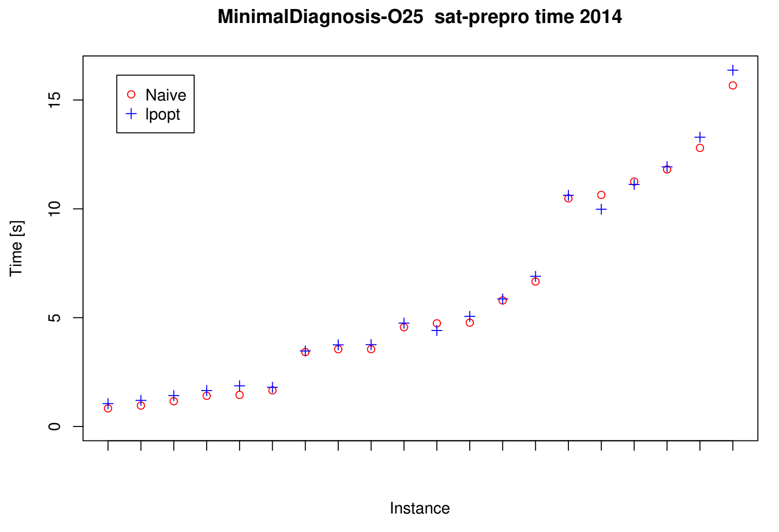

MinimalDiagnosis-O25 grounding time 2013
○ Naive
+ Ipopt
Time [s]
0
2
4
6
8
Instance

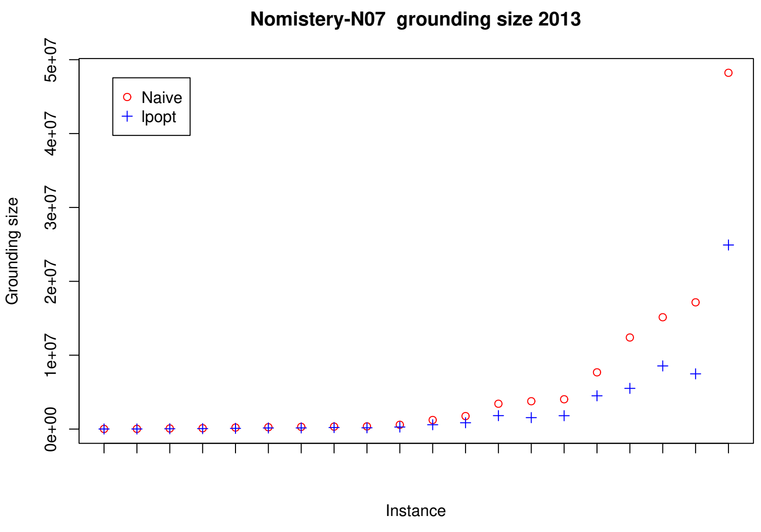

MinimalDiagnosis-O25 grounding time 2014
○ Naive
+ Ipopt
Time [s]
0
1
2
3
4
Instance

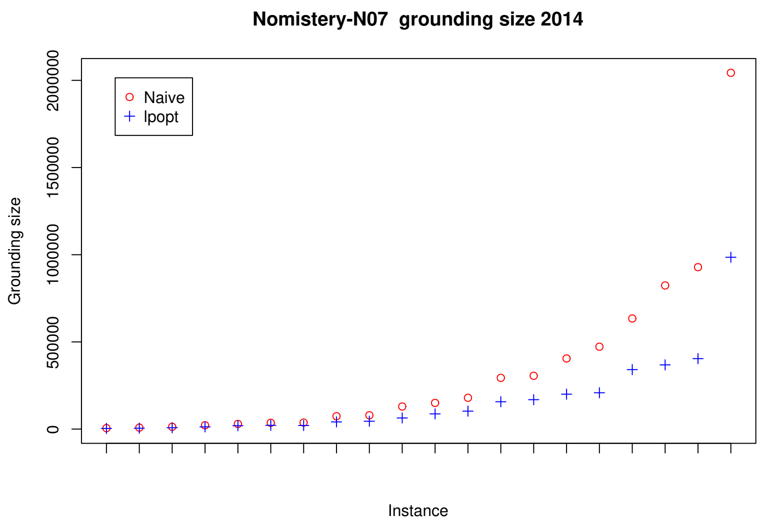

MinimalDiagnosis-O25 solving time 2013
○ Naive
+ Ipopt
Time [s]
0
2
4
6
8
Instance

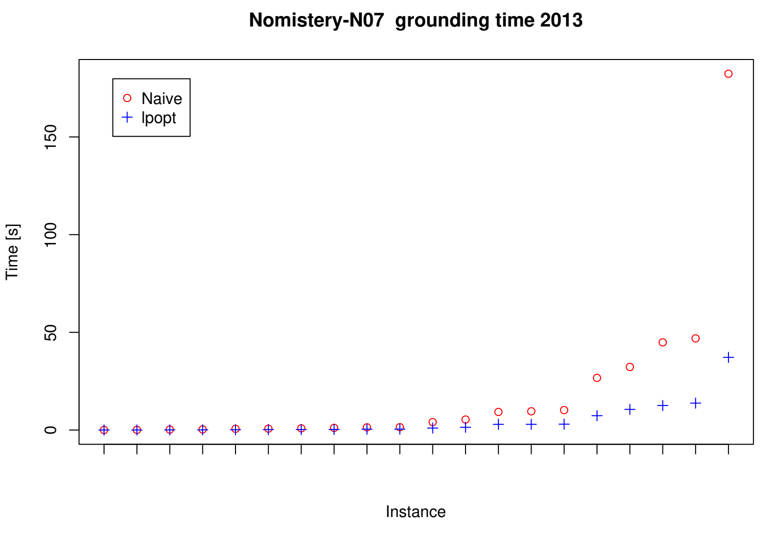

MinimalDiagnosis-O25 solving time 2014
○ Naive
+ Ipopt
Time [s]
0.0
0.5
1.0
1.5
2.0
Instance

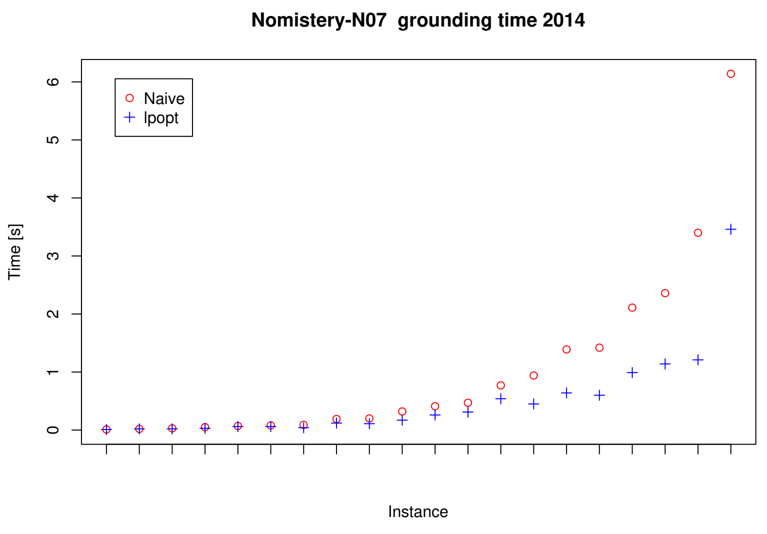

MinimalDiagnosis-O25 sat-prepro time 2013
○ Naive
+ Ipopt
Time [s]
0
5
10
15
20
25
30
Instance

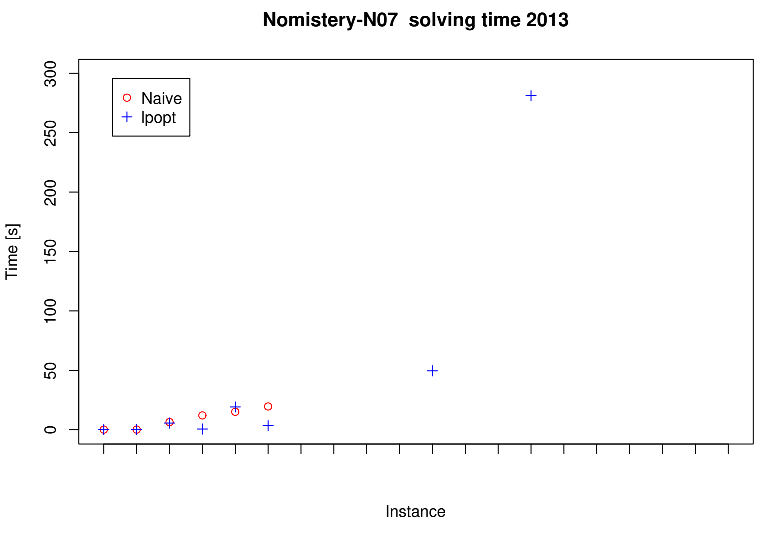

MinimalDiagnosis-O25 sat-prepro time 2014
○ Naive
+ Ipopt
Time [s]
0
5
10
15
Instance

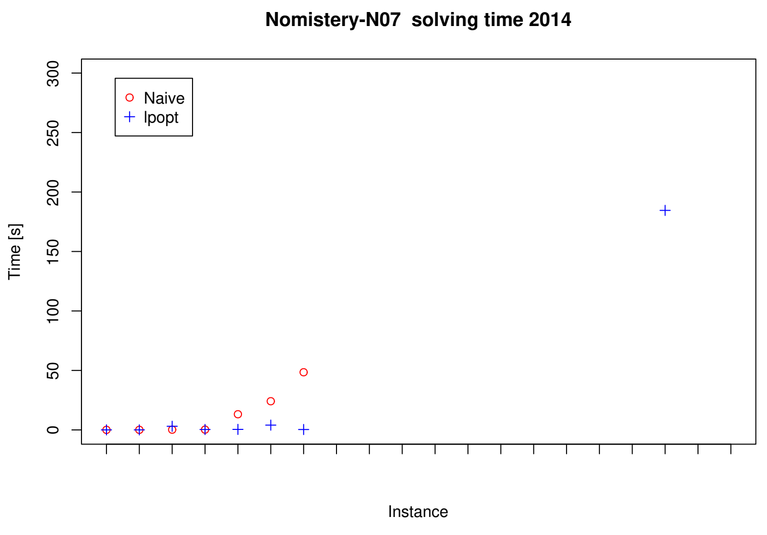

Nomistery-N07 grounding size 2013
Grounding size
0e+00
1e+07
2e+07
3e+07
4e+07
5e+07
Instance
Naive
Ipopt

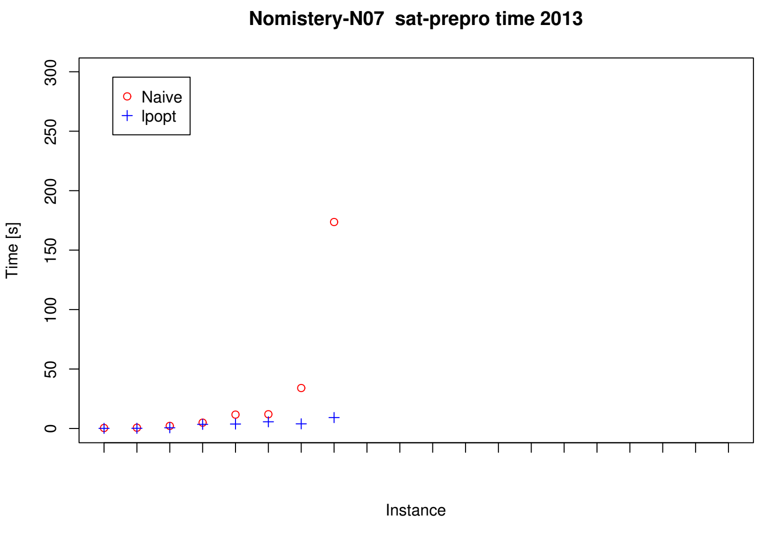

Nomistery-N07 grounding size 2014
Grounding size
0
500000
1000000
1500000
2000000
Instance
Naive
Ipopt

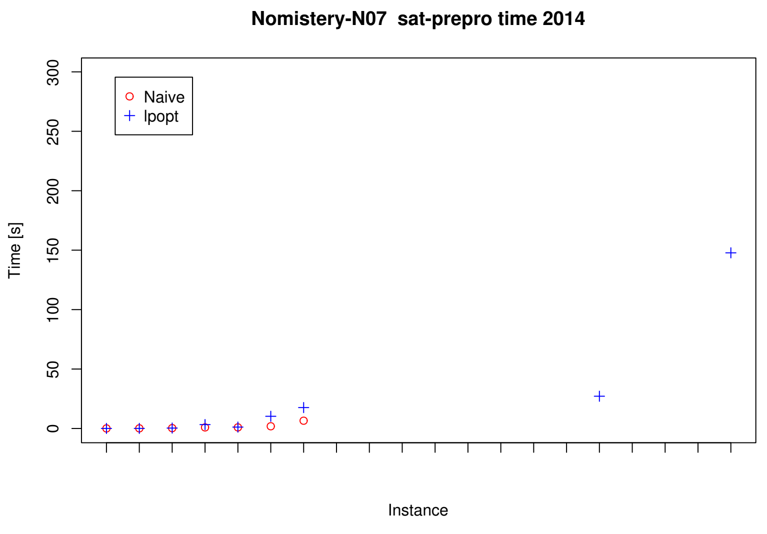

Nomistery-N07 grounding time 2013
Time [s]
0
50
100
150
Instance
Naive
Ipopt

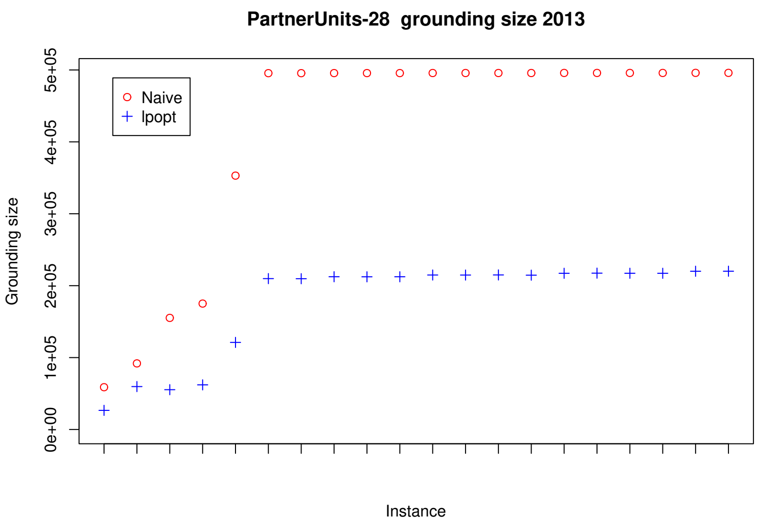

Nomistery-N07 grounding time 2014
Time [s]
0
1
2
3
4
5
6
Instance
Naive
Ipopt

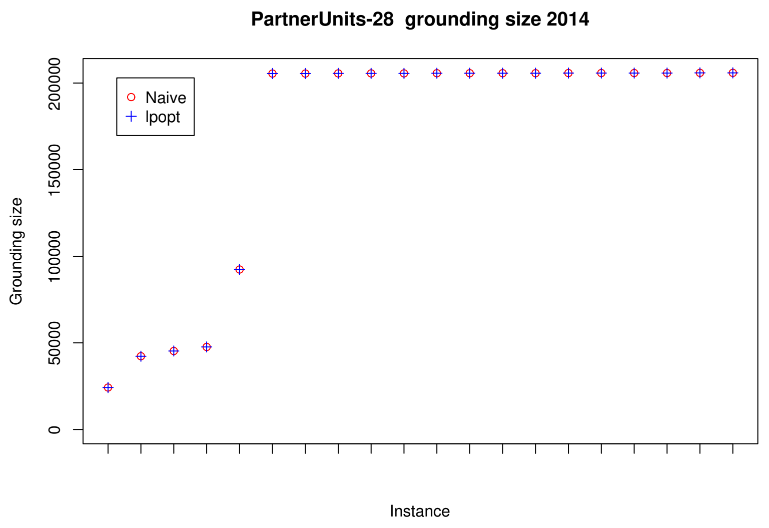

Nomistery-N07 solving time 2013
Time [s]
0
50
100
150
200
250
300
Instance
Naive
Ipopt

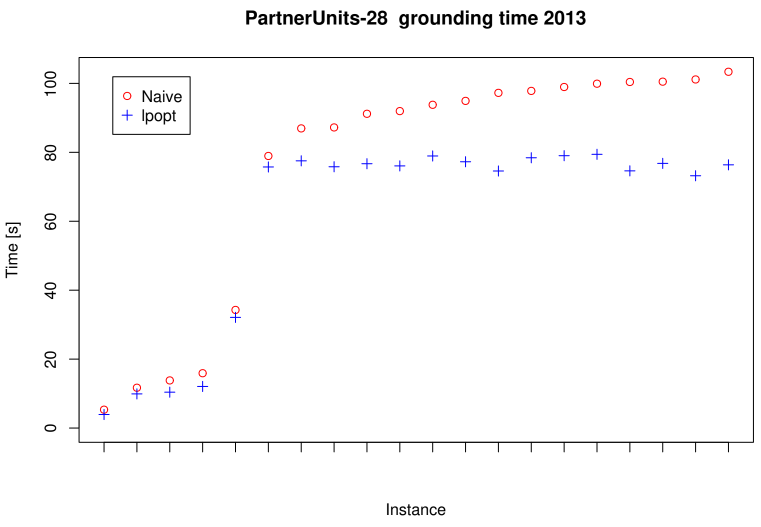

Nomistery-N07 solving time 2014
Time [s]
0
50
100
150
200
250
300
Instance
Naive
Ipopt

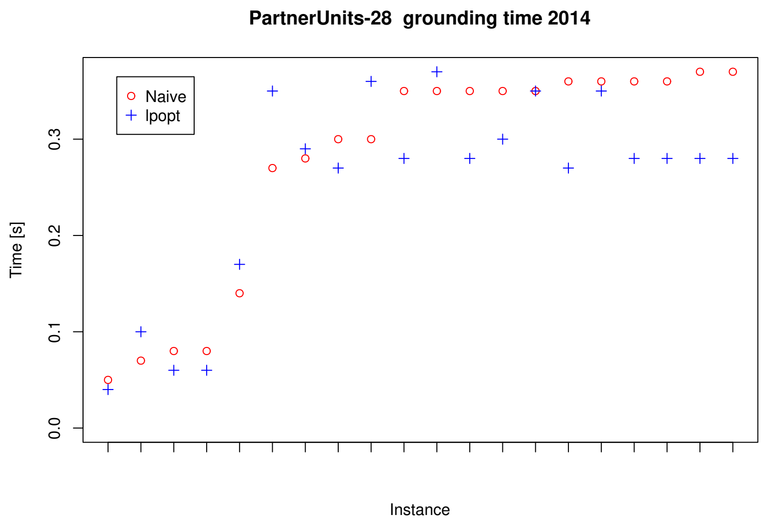

Nomistery-N07 sat-prepro time 2013
Time [s]
0
50
100
150
200
250
300
Instance
Naive
Ipopt

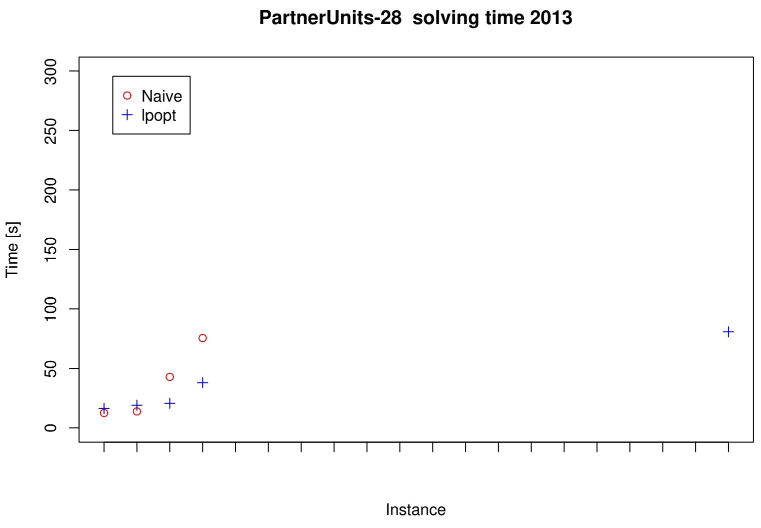

Nomistery-N07 sat-prepro time 2014
Time [s]
0
50
100
150
200
250
300
Instance
Naive
Ipopt

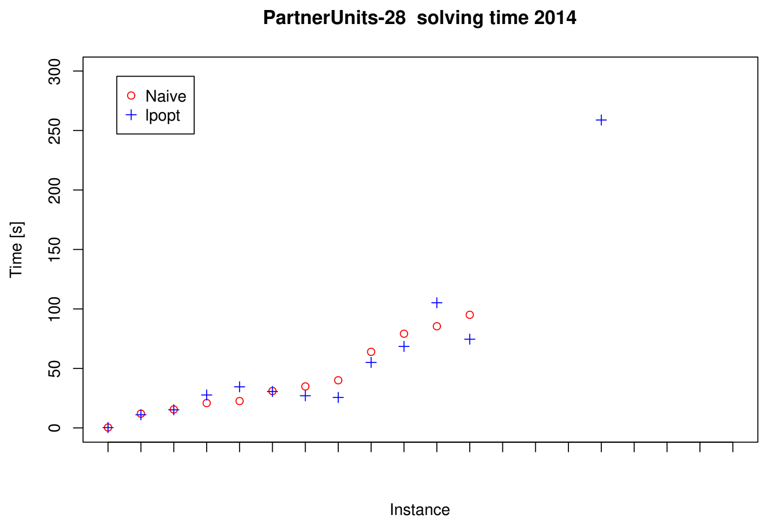

PartnerUnits-28 grounding size 2013
Grounding size
Instance
Naive
Ipopt
0e+00
1e+05
2e+05
3e+05
4e+05
5e+05

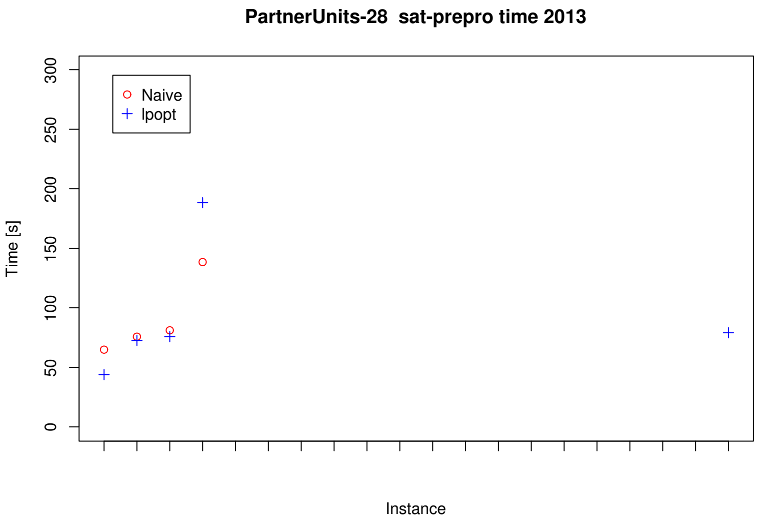

PartnerUnits-28 grounding size 2014
Grounding size
Instance
Naive
Ipopt
0
50000
100000
150000
200000

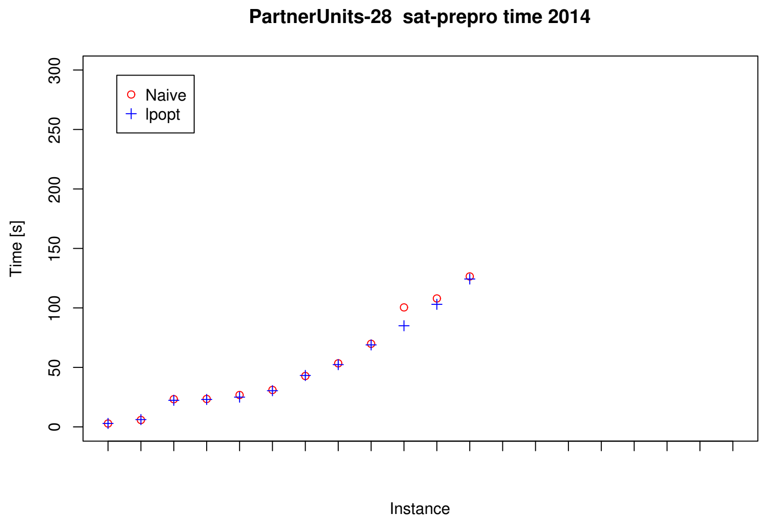

PartnerUnits-28 grounding time 2013
Time [s]
Instance
Naive
Ipopt
0
20
40
60
80
100

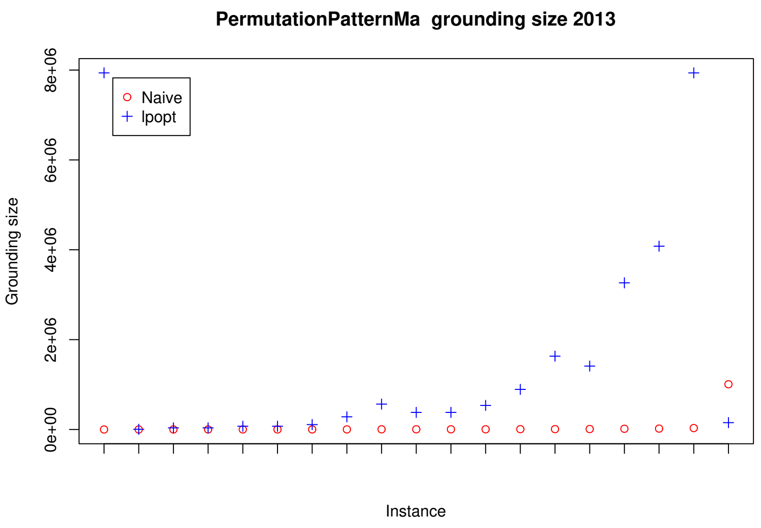

PartnerUnits-28 grounding time 2014
Time [s]
Instance
Naive
Ipopt
0.0
0.1
0.2
0.3

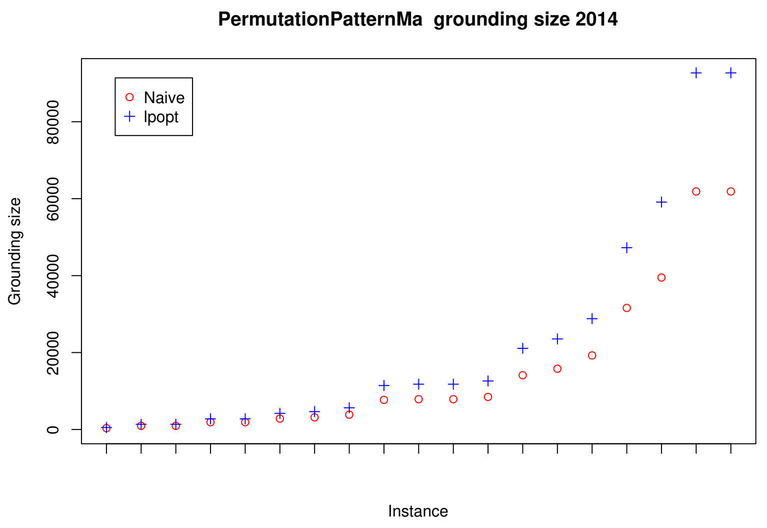

PartnerUnits-28 solving time 2013
Time [s]
Instance
Naive
Ipopt
0
50
100
150
200
250
300

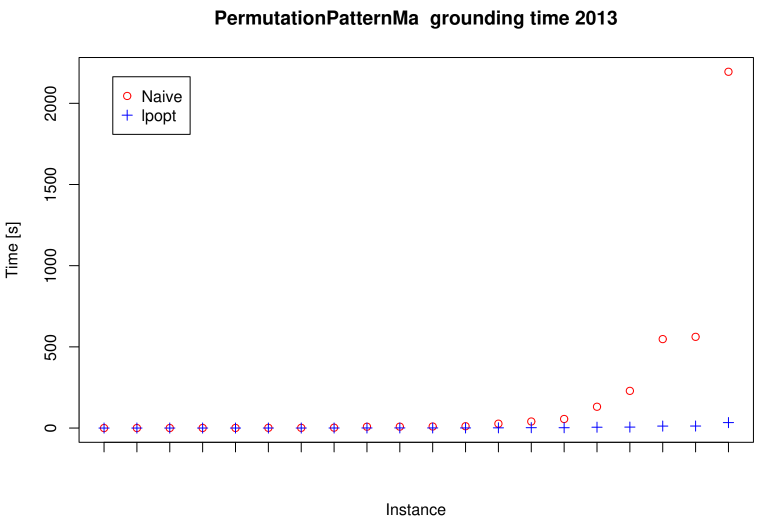

PartnerUnits-28 solving time 2014
Time [s]
Instance
Naive
Ipopt
0
50
100
150
200
250
300

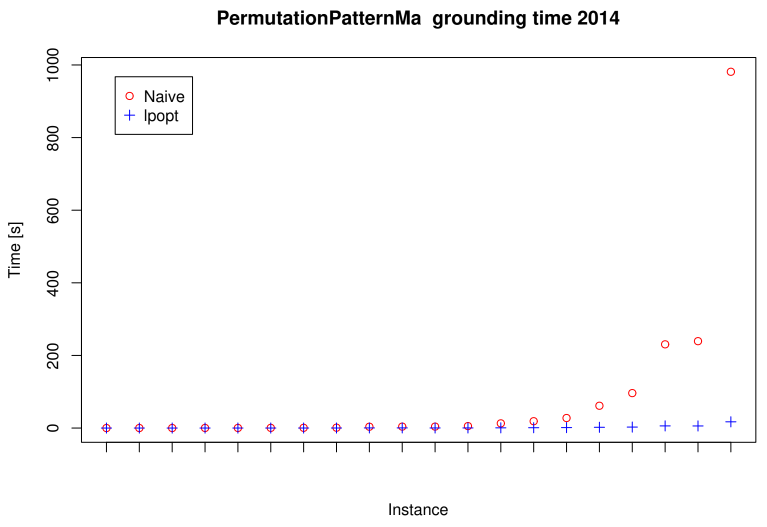

PartnerUnits-28 sat-prepro time 2013
Time [s]
Instance
Naive
Ipopt
0
50
100
150
200
250
300

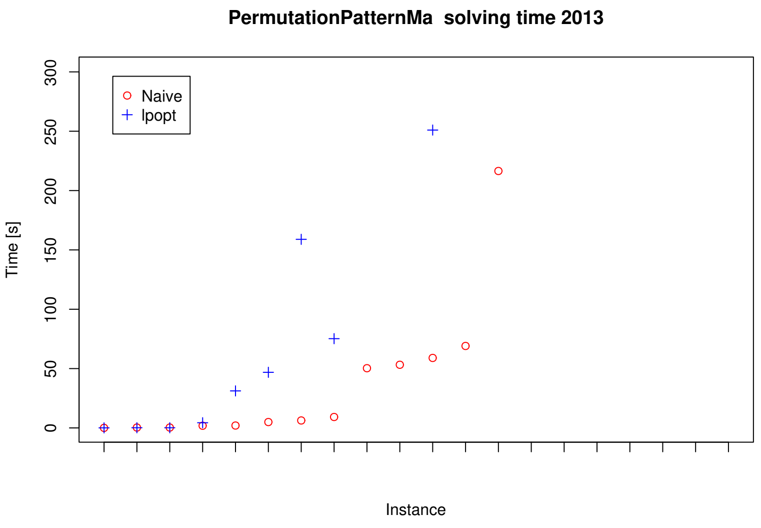

PartnerUnits-28 sat-prepro time 2014
Time [s]
Instance
Naive
Ipopt
0
50
100
150
200
250
300

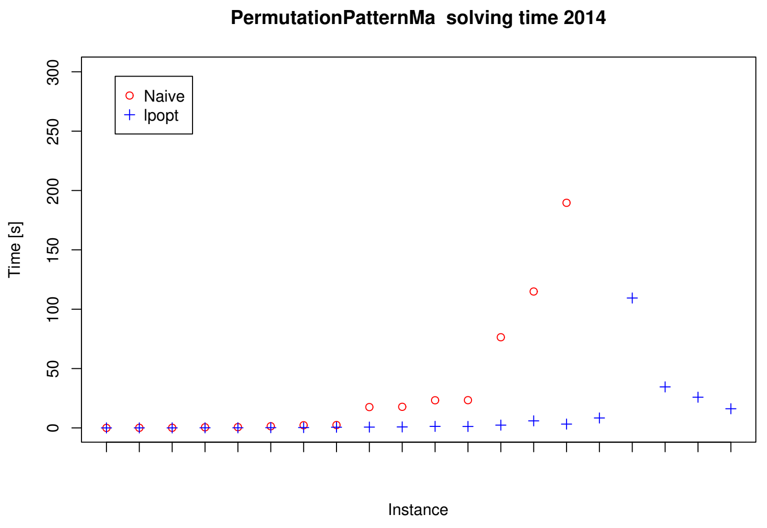

PermutationPatternMa grounding size 2013
Grounding size
0e+00
2e+06
4e+06
6e+06
8e+06
Instance
Naive
Ipopt

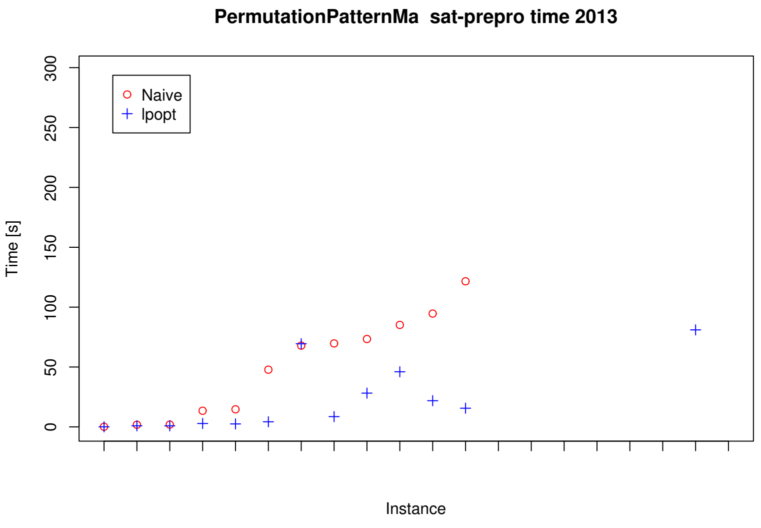

PermutationPatternMa grounding size 2014
Grounding size
0
20000
40000
60000
80000
Instance
Naive
Ipopt

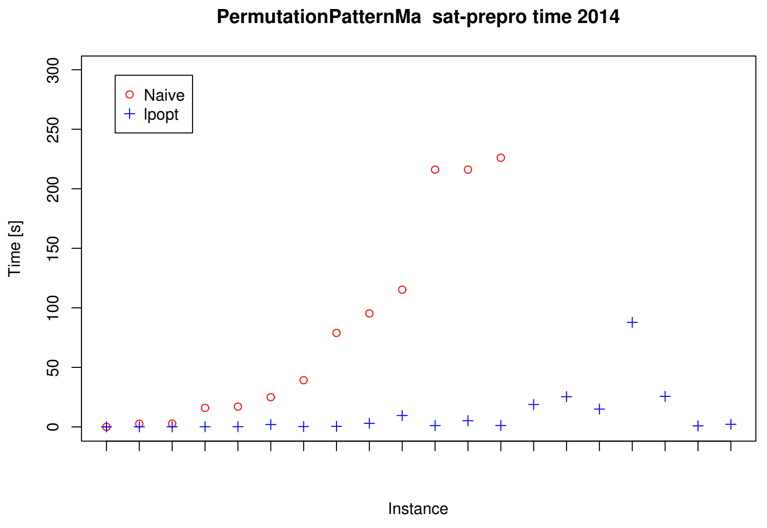

PermutationPatternMa grounding time 2013
Time [s]
0
500
1000
1500
2000
Instance
Naive
Ipopt

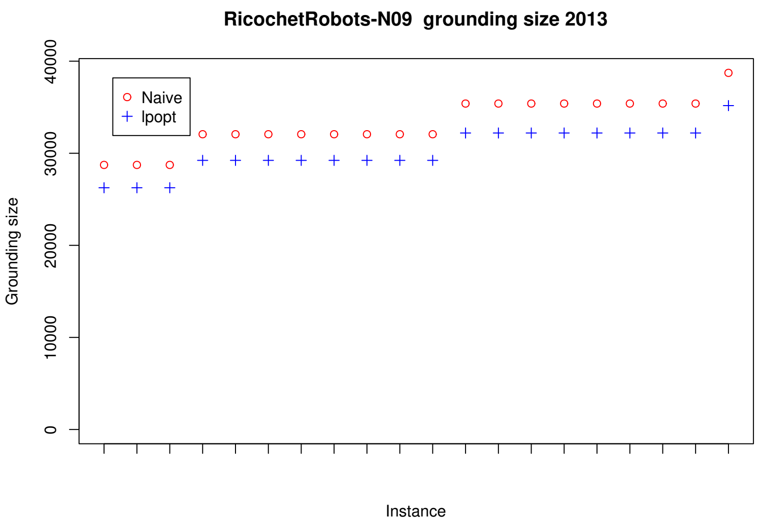

PermutationPatternMa grounding time 2014
Time [s]
0
200
400
600
800
1000
Instance
Naive
Ipopt

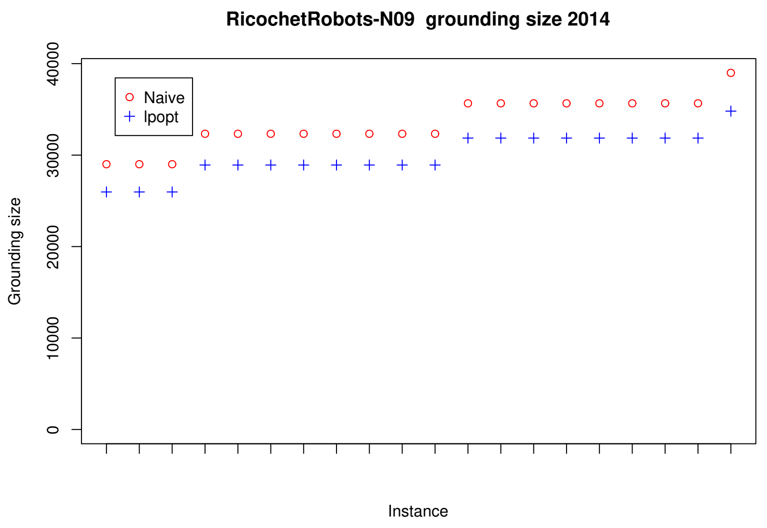

PermutationPatternMa solving time 2013
Time [s]
0
50
100
150
200
250
300
Instance
Naive
Ipopt

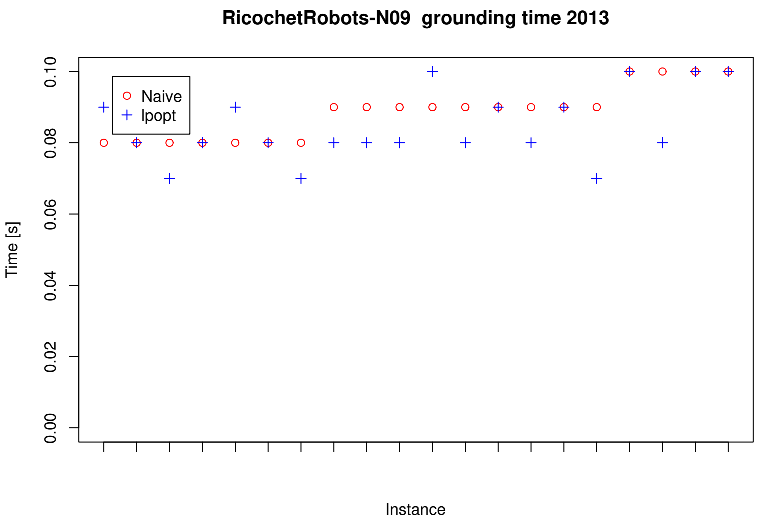

PermutationPatternMa solving time 2014
Time [s]
0
50
100
150
200
250
300
Instance
Naive
Ipopt

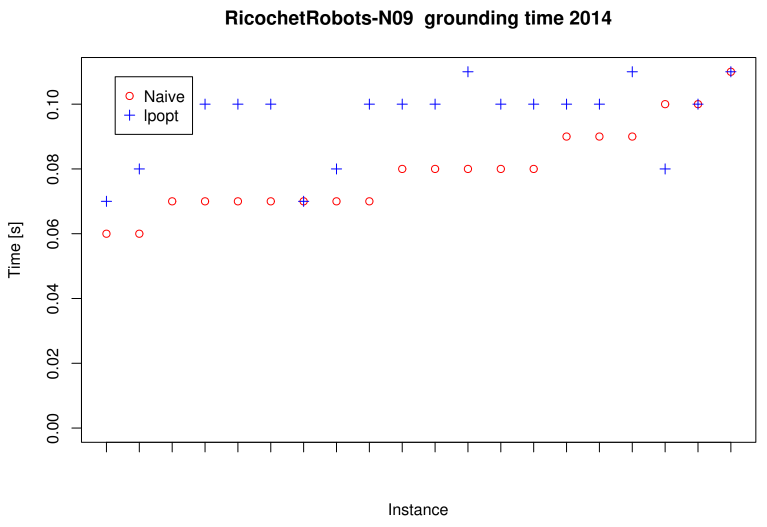

PermutationPatternMa sat-prepro time 2013
Time [s]
0
50
100
150
200
250
300
Instance
Naive
Ipopt

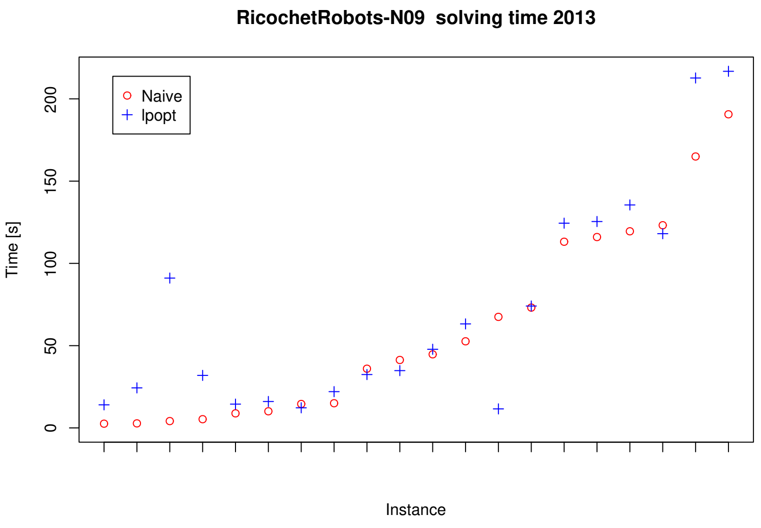

PermutationPatternMa sat-prepro time 2014
Time [s]
0
50
100
150
200
250
300
Instance
Naive
Ipopt

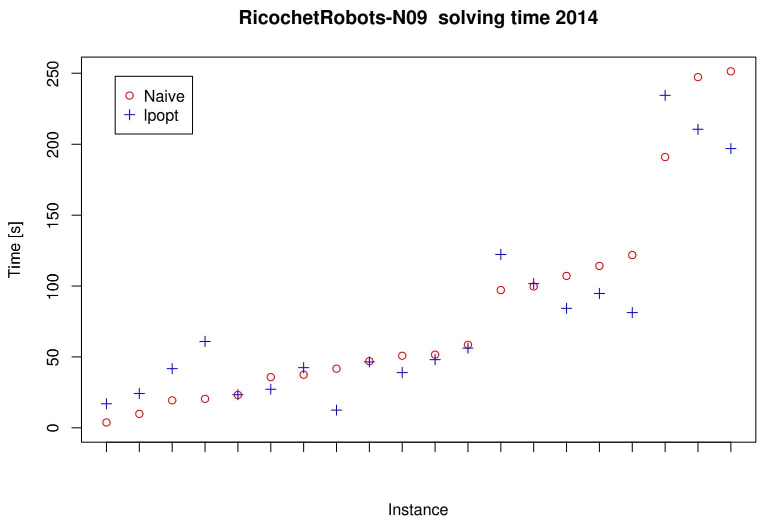

RicochetRobots-N09 grounding size 2013
Grounding size
0
10000
20000
30000
40000
Naive
Ipopt
Instance

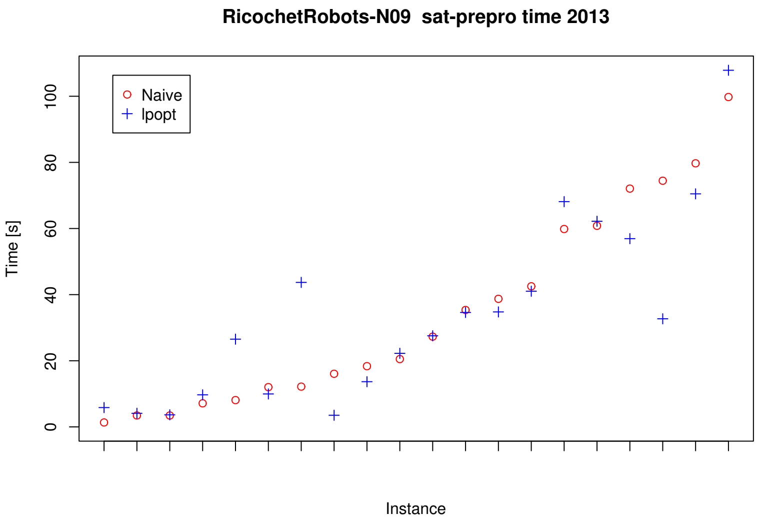

RicochetRobots-N09 grounding size 2014
Grounding size
0
10000
20000
30000
40000
Naive
Ipopt
Instance

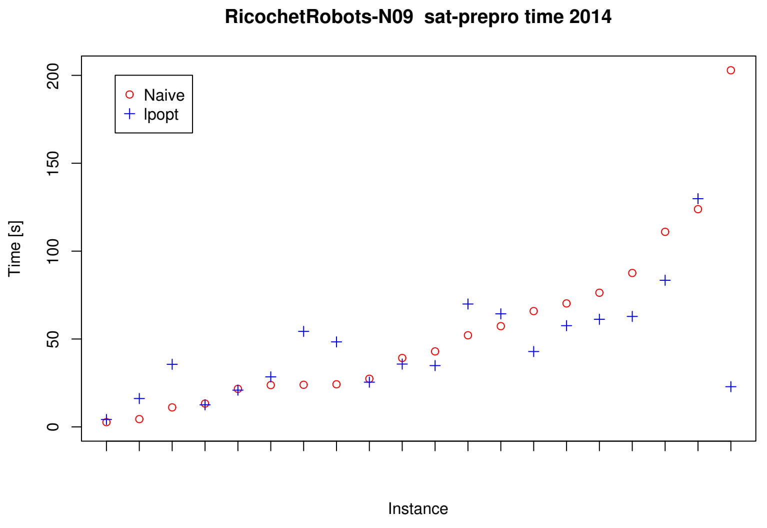

RicochetRobots-N09 grounding time 2013
Time [s]
0.00
0.02
0.04
0.06
0.08
0.10
Naive
Ipopt
Instance

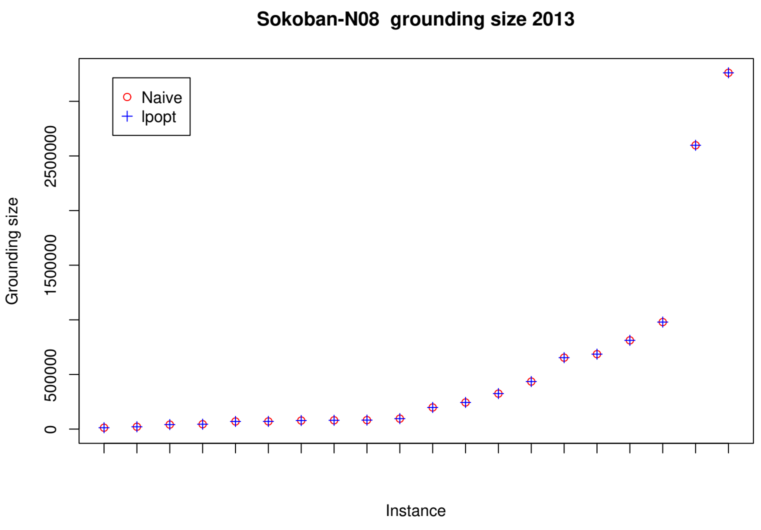

RicochetRobots-N09 grounding time 2014
Time [s]
0.00
0.02
0.04
0.06
0.08
0.10
Naive
Ipopt
Instance

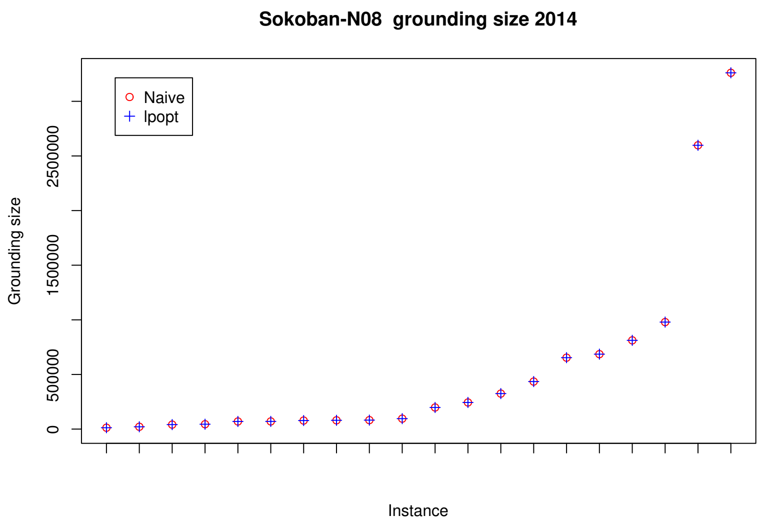

RicochetRobots-N09 solving time 2013
Time [s]
0
50
100
150
200
Naive
Ipopt
Instance

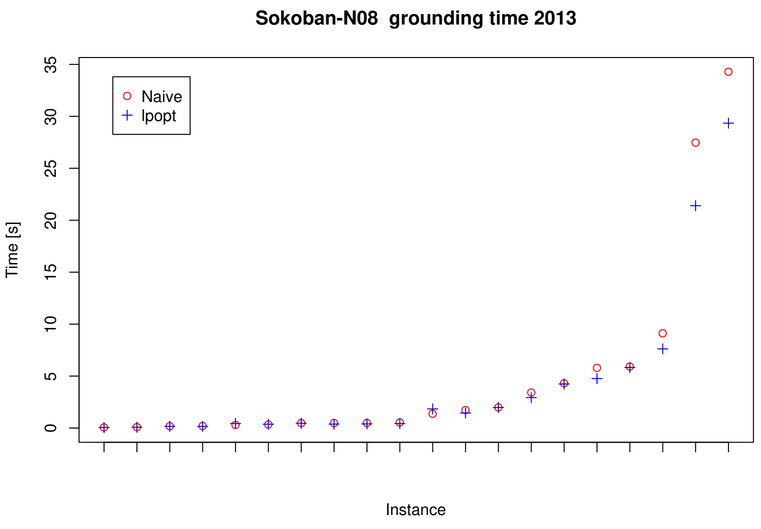

RicochetRobots-N09 solving time 2014
Time [s]
0
50
100
150
200
250
Naive
Ipopt
Instance

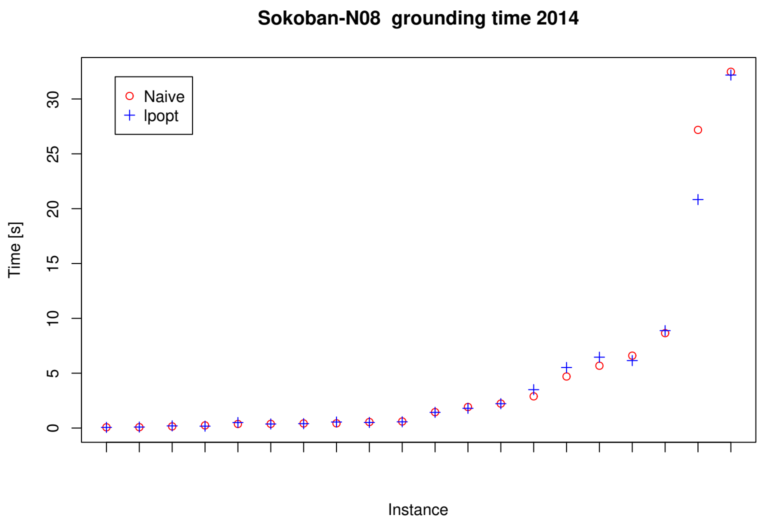

RicochetRobots-N09 sat-prepro time 2013
Time [s]
0
20
40
60
80
100
Naive
Ipopt
Instance

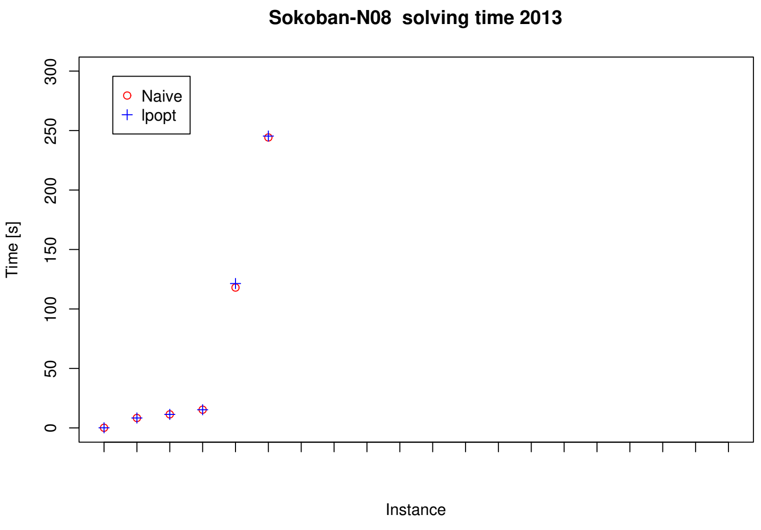

RicochetRobots-N09 sat-prepro time 2014
Time [s]
0
50
100
150
200
Naive
Ipopt
Instance

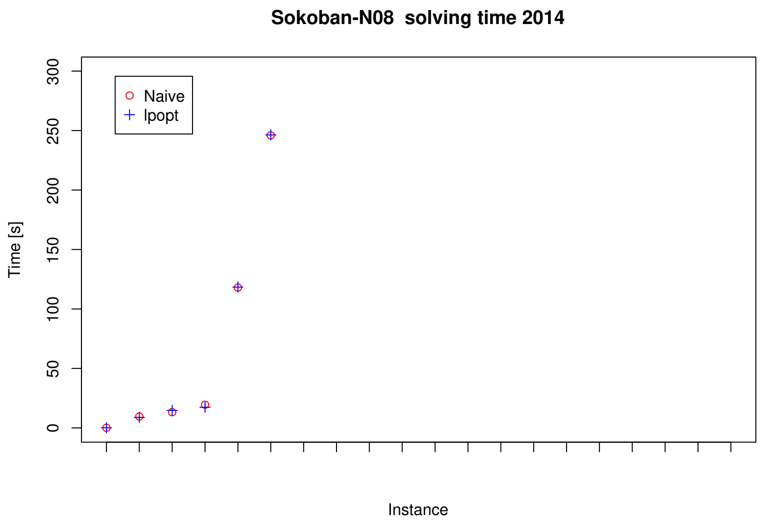

Sokoban-N08 grounding size 2013
Naive
Ipopt
Grounding size
0
500000
1500000
2500000
Instance

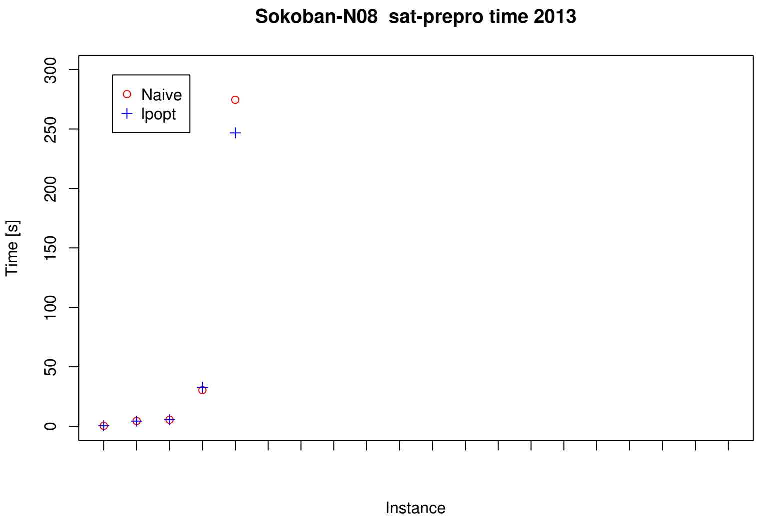

Sokoban-N08 grounding size 2014
Naive
Ipopt
Grounding size
0
500000
1500000
2500000
Instance

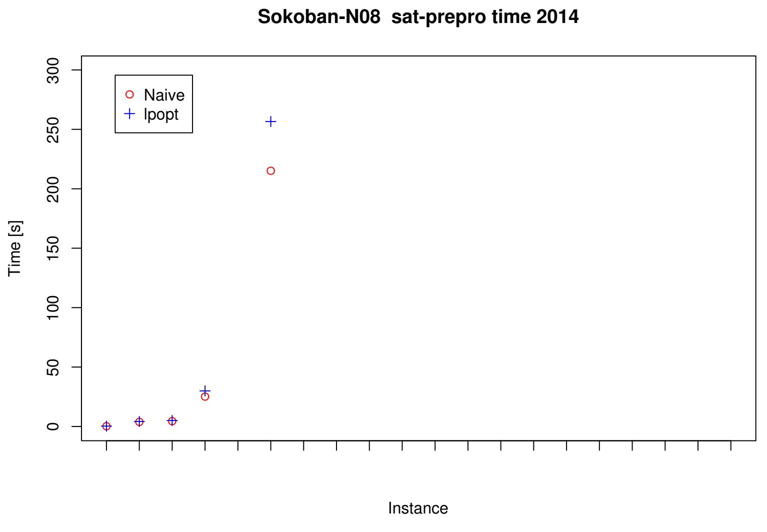

Sokoban-N08 grounding time 2013
Naive
Ipopt
Time [s]
0
5
10
15
20
25
30
35
Instance

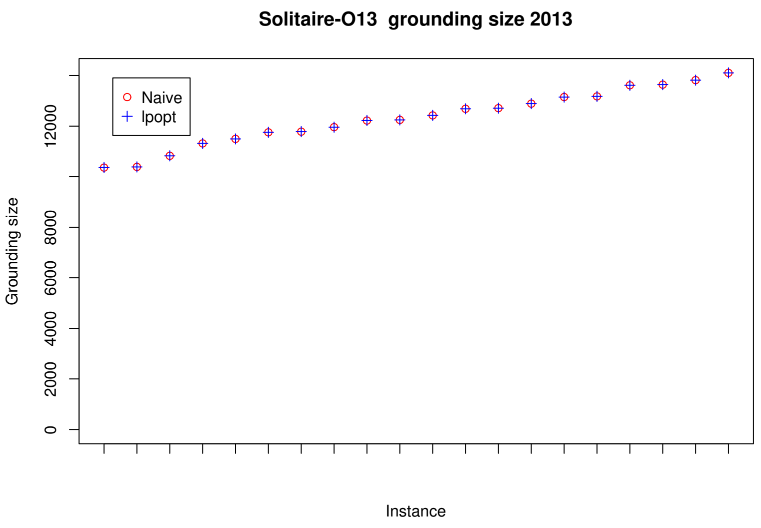

Sokoban-N08 grounding time 2014
Naive
Ipopt
Time [s]
0
5
10
15
20
25
30
Instance


Sokoban-N08 solving time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

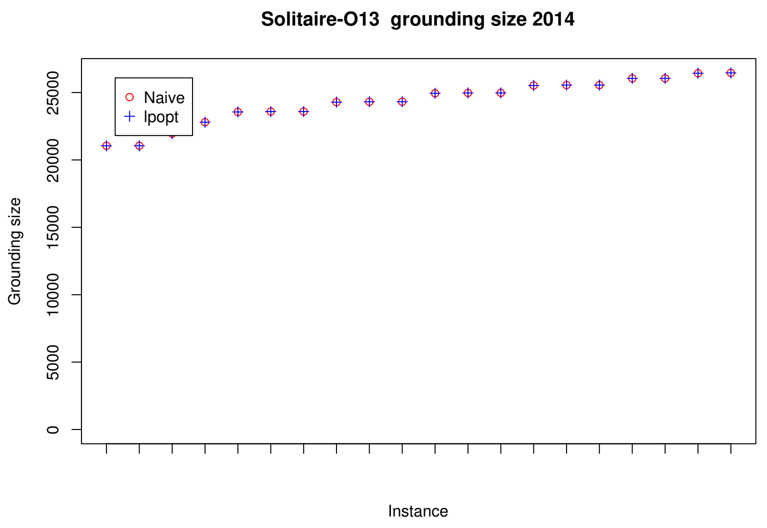

Sokoban-N08 solving time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance


Sokoban-N08 sat-prepro time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

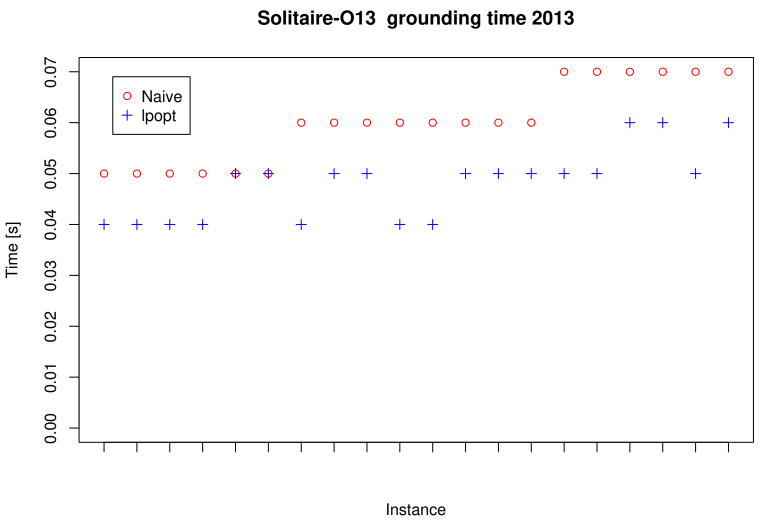

Sokoban-N08 sat-prepro time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance



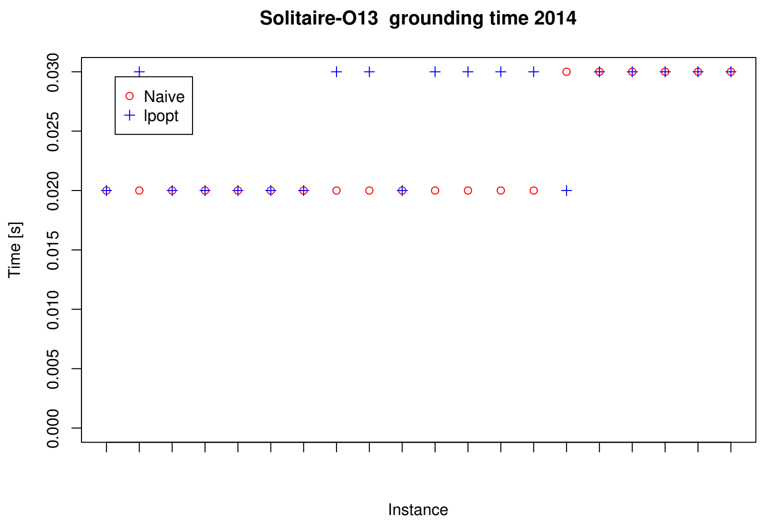
Solitaire-O13 grounding size 2013
Naive
Ipopt
Grounding size
Instance



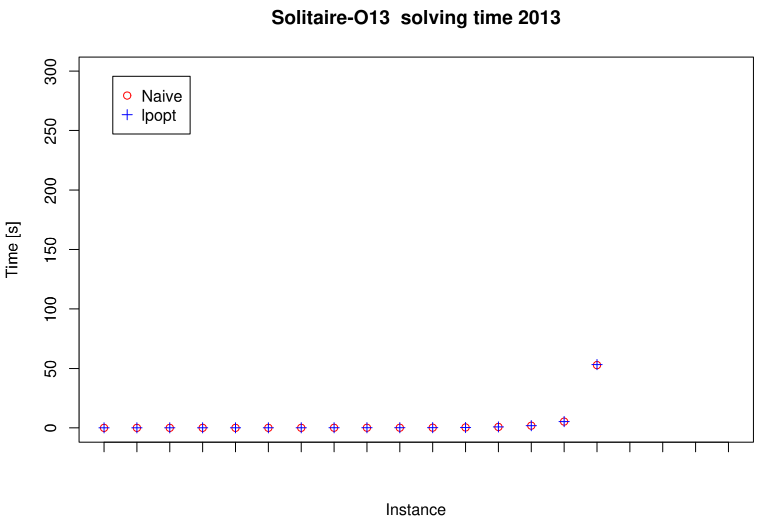
Solitaire-O13 grounding size 2014
Naive
Ipopt
Grounding size
Instance



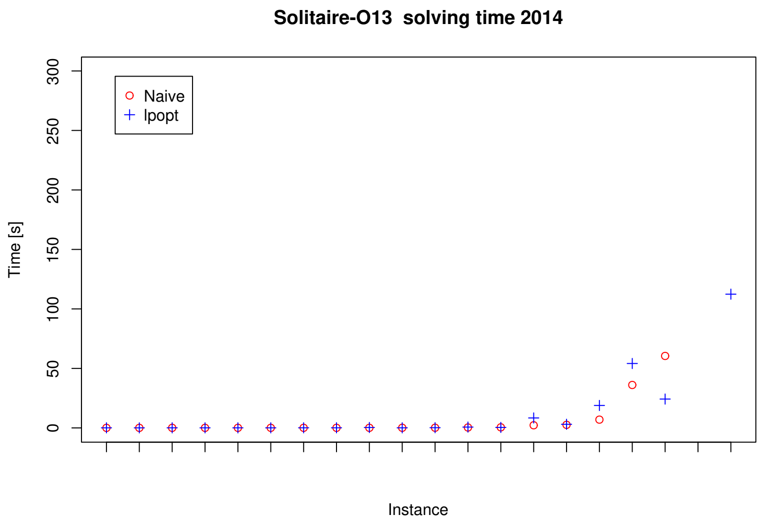
Solitaire-O13 grounding time 2013
Naive
Ipopt
Time [s]
Instance



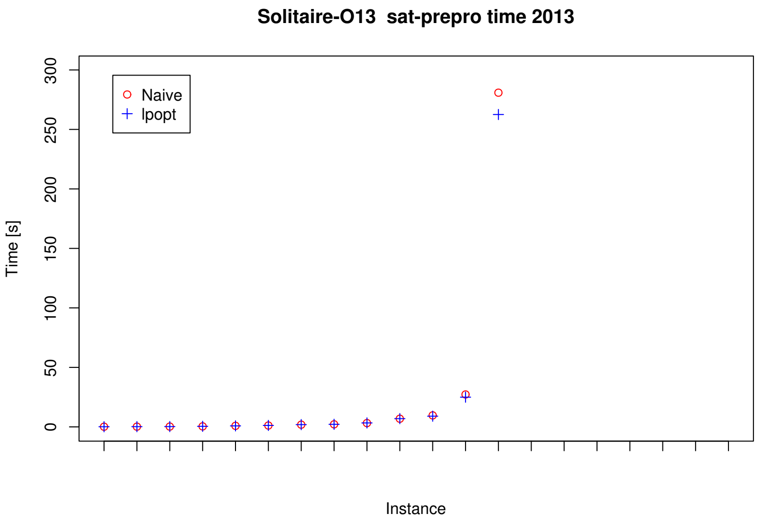
Solitaire-O13 grounding time 2014
Naive
Ipopt
Time [s]
Instance



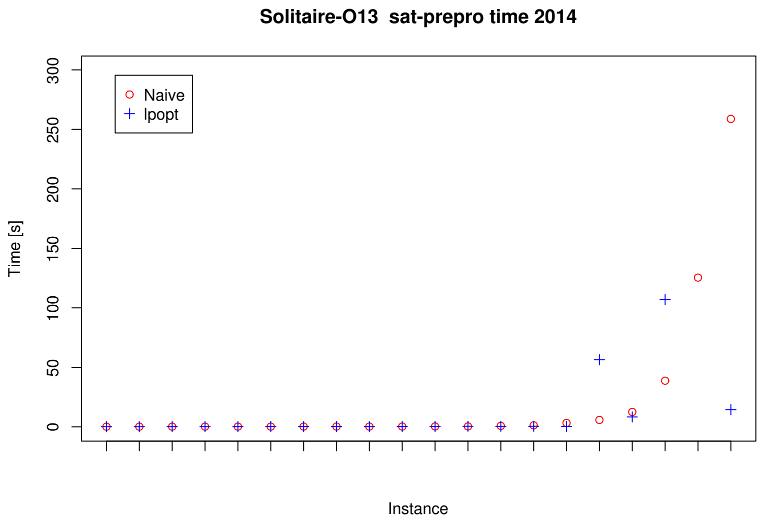
Solitaire-O13 solving time 2013
Naive
Ipopt
Time [s]
Instance



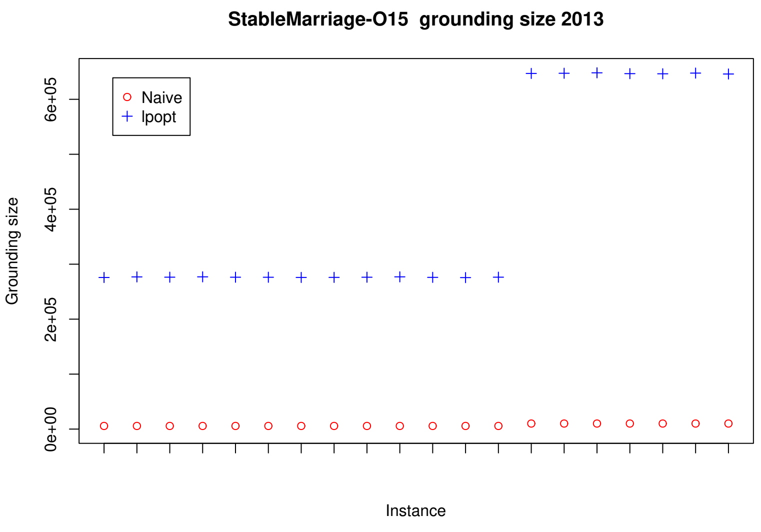
Solitaire-O13 solving time 2014
Naive
Ipopt
Time [s]
Instance

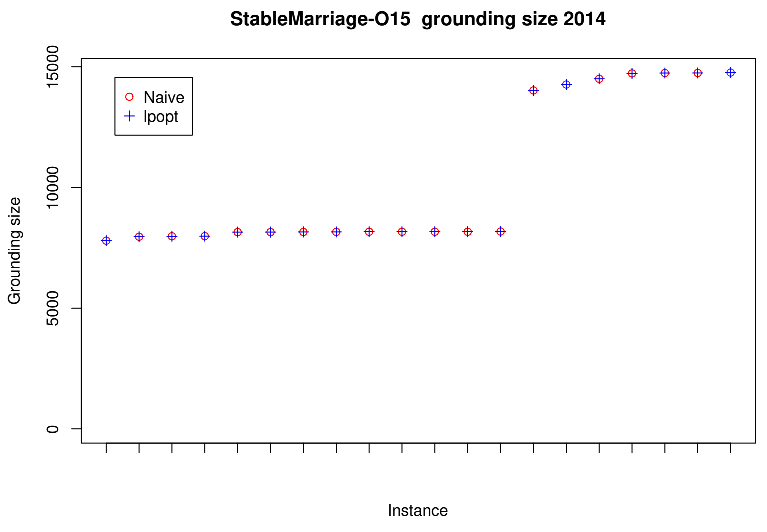


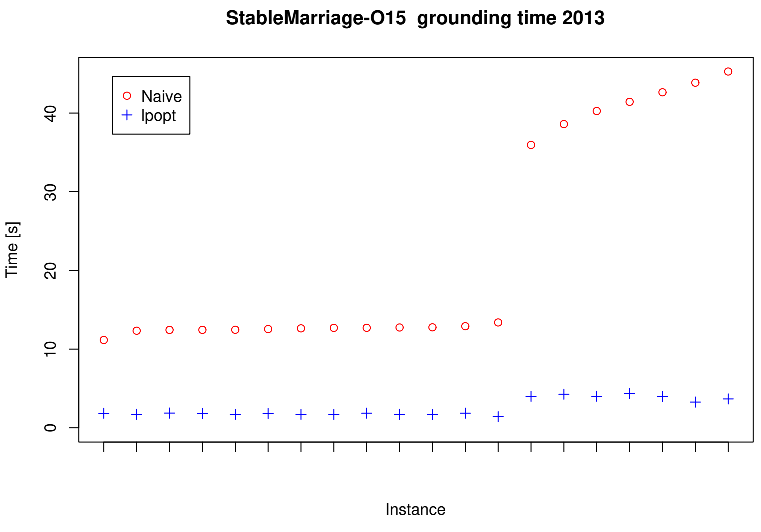
Solitaire-O13 sat-prepro time 2013
Naive
Ipopt
Time [s]
Instance

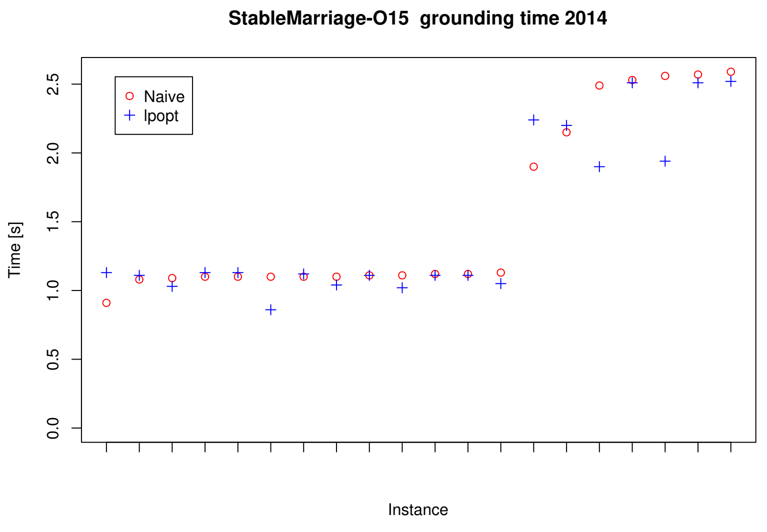


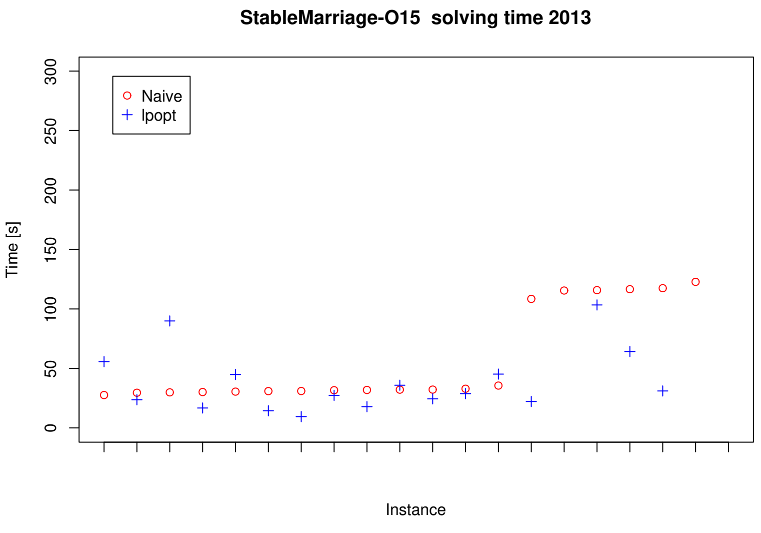
Solitaire-O13 sat-prepro time 2014
Naive
Ipopt
Time [s]
Instance

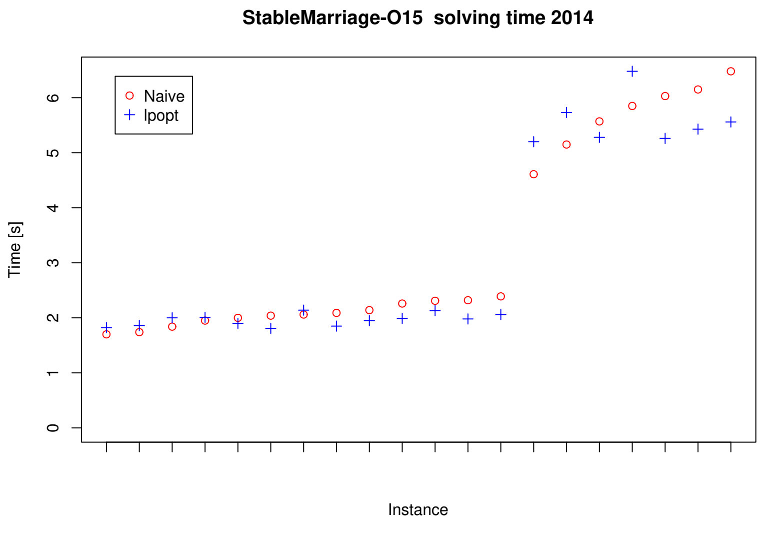

StableMarriage-O15 grounding size 2013
Naive
Ipopt
Grounding size
0e+00
2e+05
4e+05
6e+05
Instance

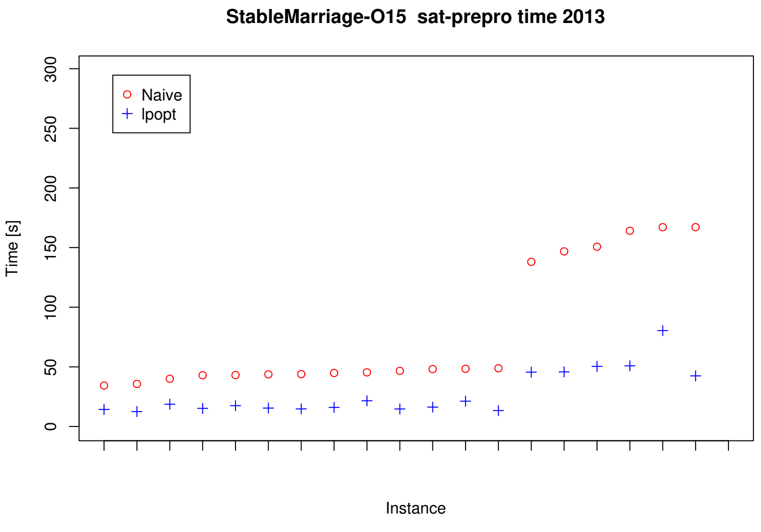

StableMarriage-O15 grounding size 2014
Naive
Ipopt
Grounding size
0
5000
10000
15000
Instance

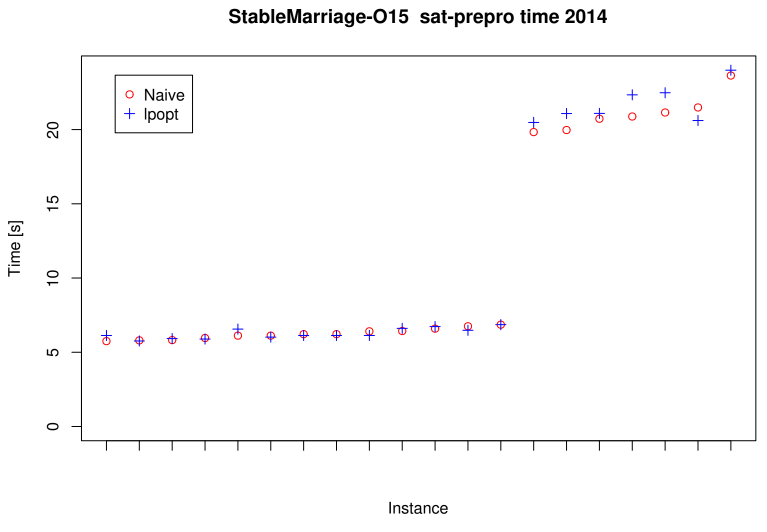

StableMarriage-O15 grounding time 2013
Naive
Ipopt
Time [s]
0
10
20
30
40
Instance

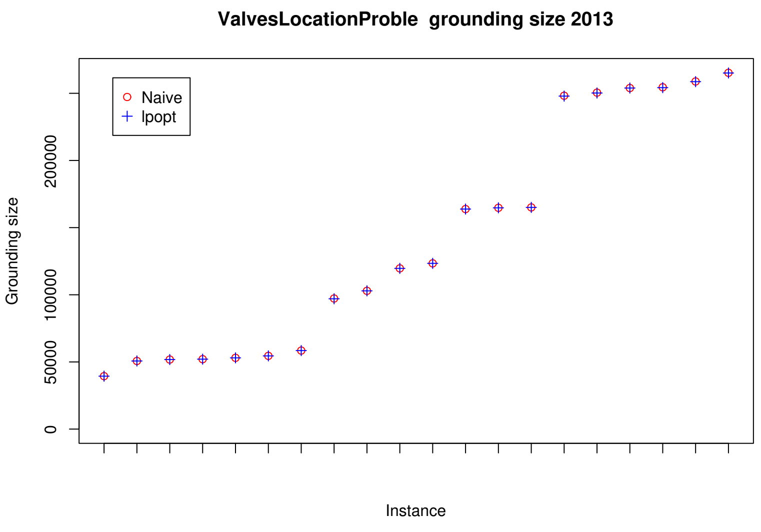

StableMarriage-O15 grounding time 2014
Naive
Ipopt
Time [s]
0.0
0.5
1.0
1.5
2.0
2.5
Instance

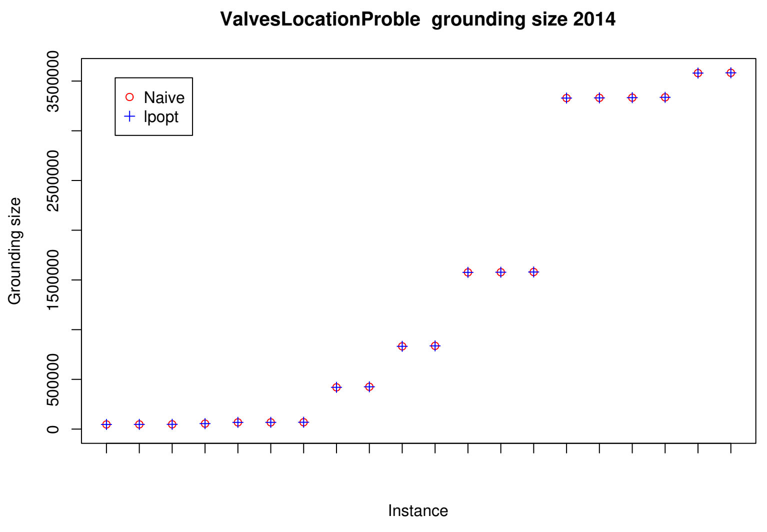

StableMarriage-O15 solving time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

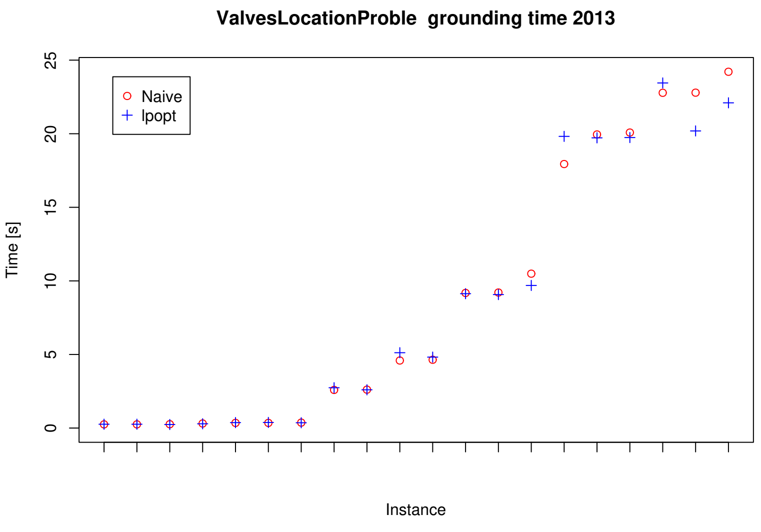

StableMarriage-O15 solving time 2014
Naive
Ipopt
Time [s]
0
1
2
3
4
5
6
Instance

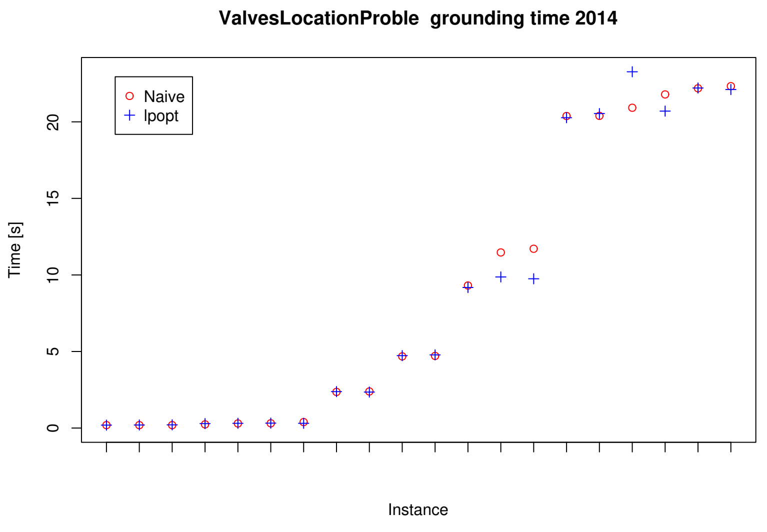

StableMarriage-O15 sat-prepro time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

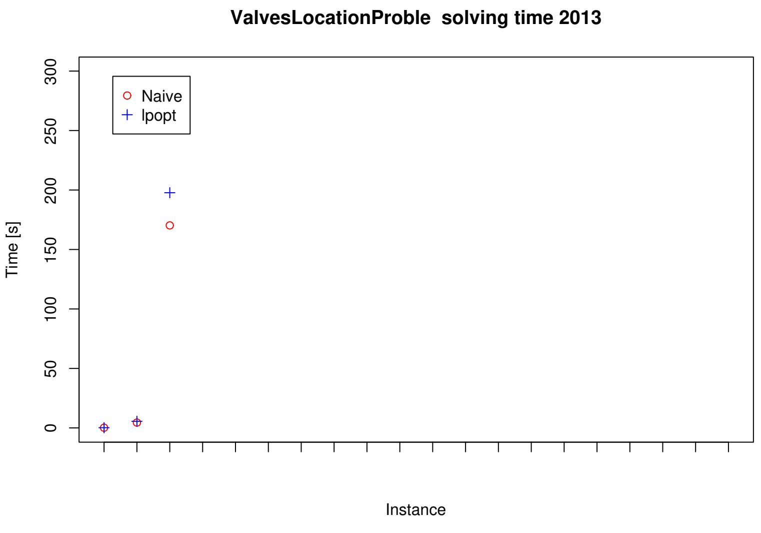

StableMarriage-O15 sat-prepro time 2014
Naive
Ipopt
Time [s]
0
5
10
15
20
Instance

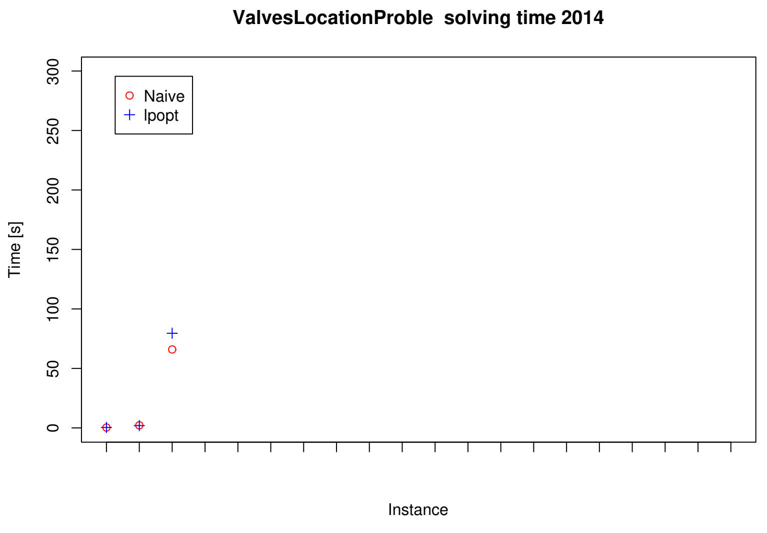

ValvesLocationProble grounding size 2013
Naive
Ipopt
Grounding size
0
50000
100000
200000
Instance

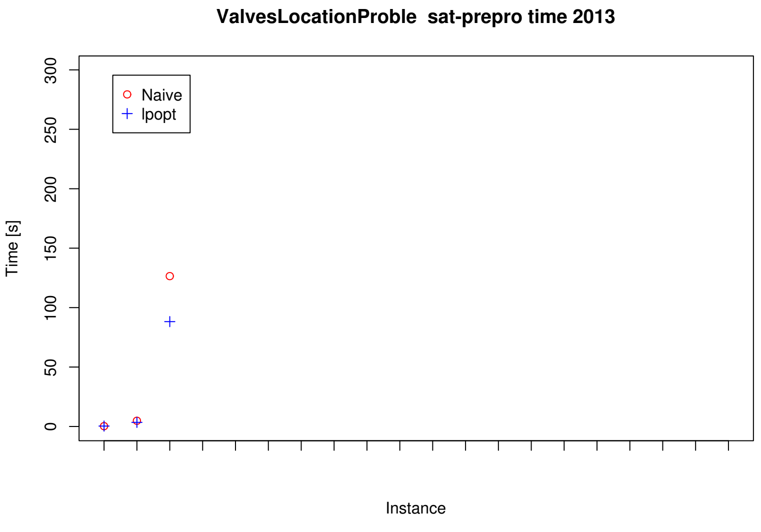

ValvesLocationProble grounding size 2014
Naive
Ipopt
Grounding size
0
500000
1500000
2500000
3500000
Instance

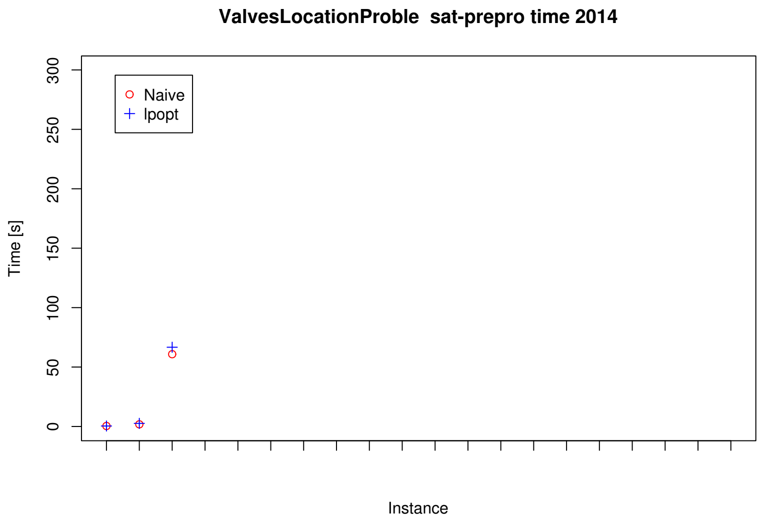

ValvesLocationProble grounding time 2013
Naive
Ipopt
Time [s]
0
5
10
15
20
25
Instance

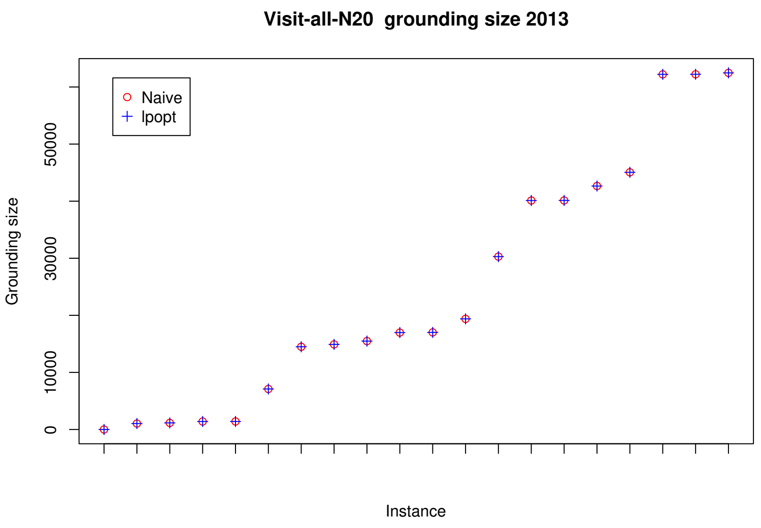

ValvesLocationProble grounding time 2014
Naive
Ipopt
Time [s]
0
5
10
15
20
Instance

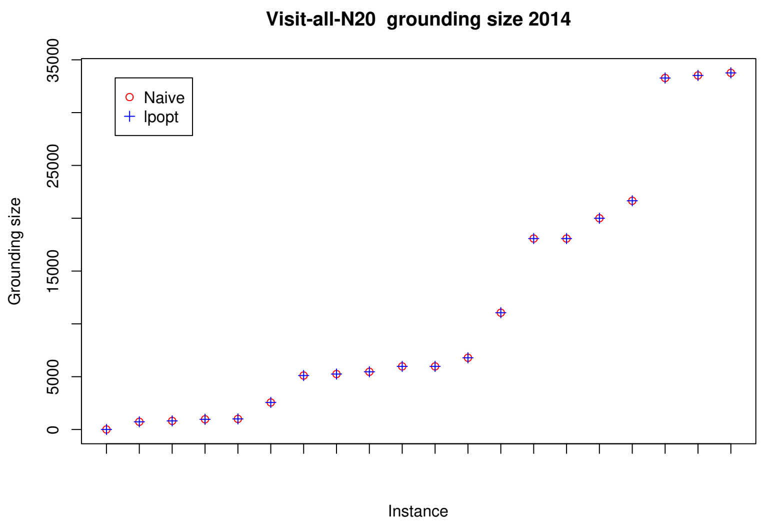

ValvesLocationProble solving time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

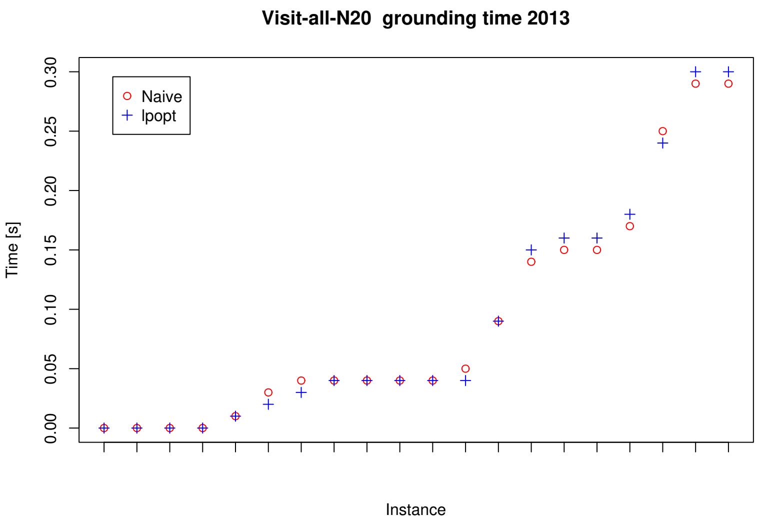

ValvesLocationProble solving time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

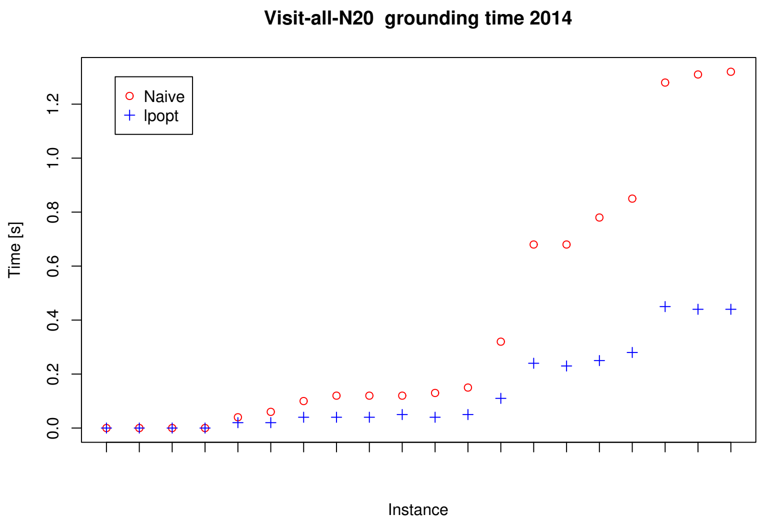

ValvesLocationProble sat-prepro time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

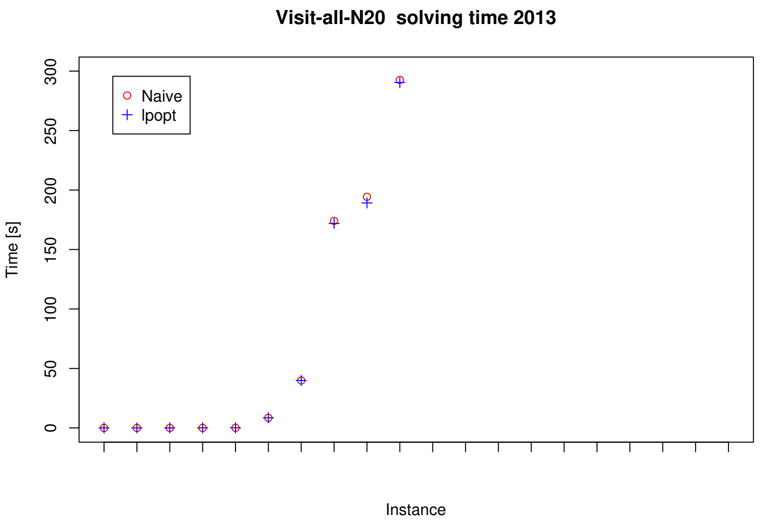

ValvesLocationProble sat-prepro time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

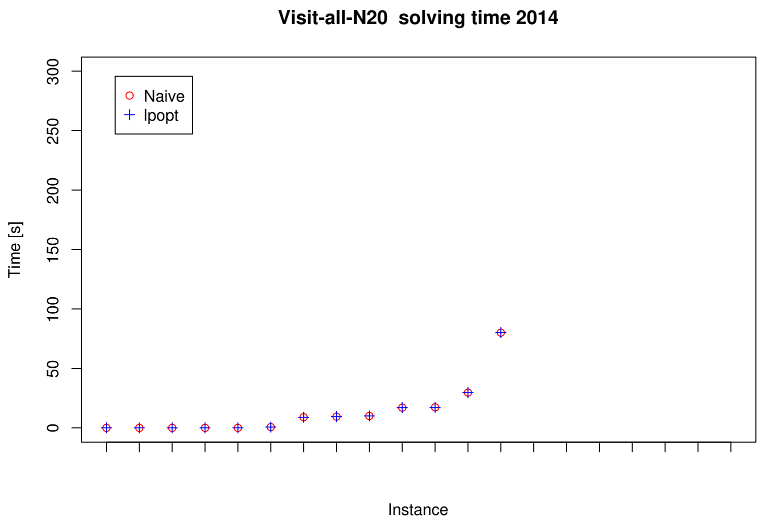

Visit-all-N20 grounding size 2013
Naive
Ipopt
Grounding size
0
10000
30000
50000
Instance

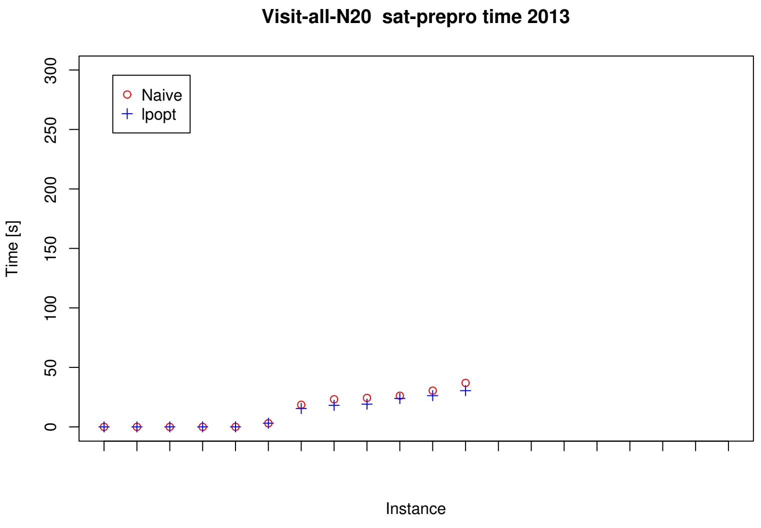

Visit-all-N20 grounding size 2014
Naive
Ipopt
Grounding size
0
5000
15000
25000
35000
Instance

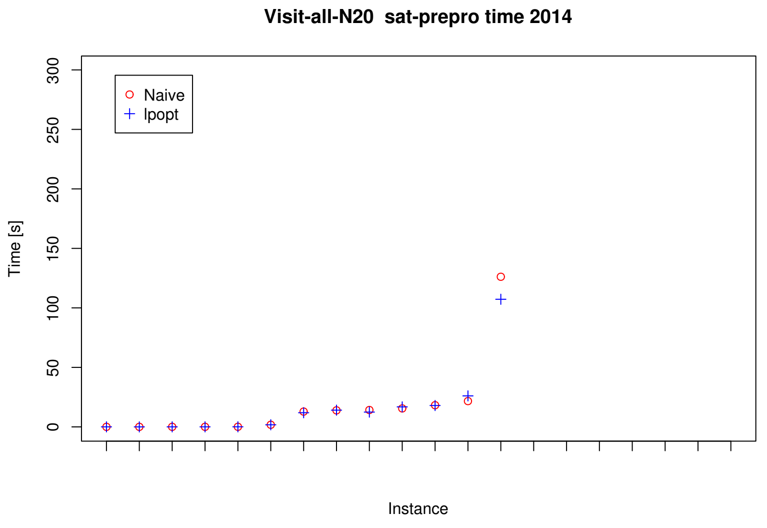

Visit-all-N20 grounding time 2013
Naive
Ipopt
Time [s]
0.00
0.05
0.10
0.15
0.20
0.25
0.30
Instance

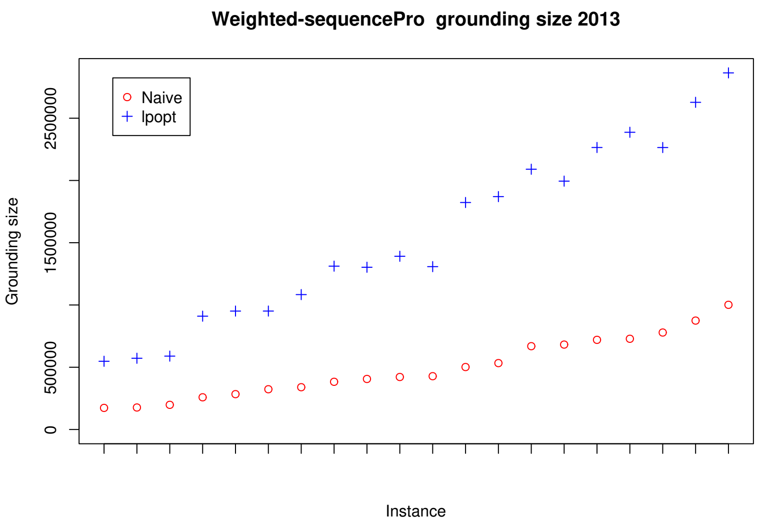
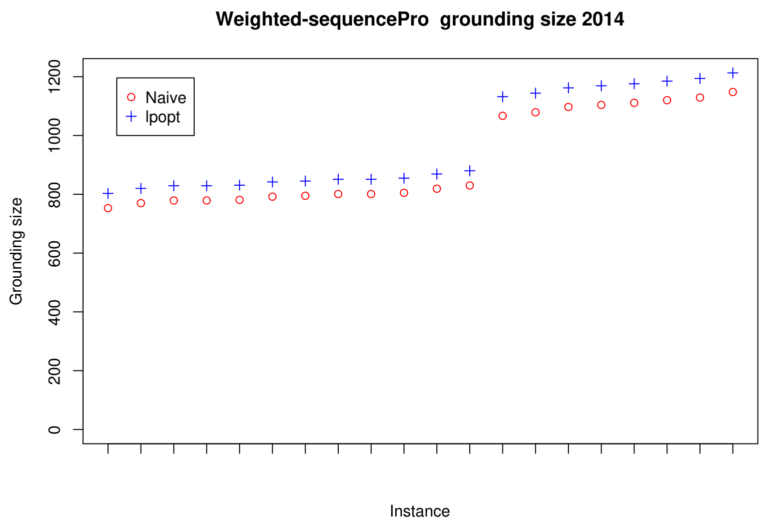
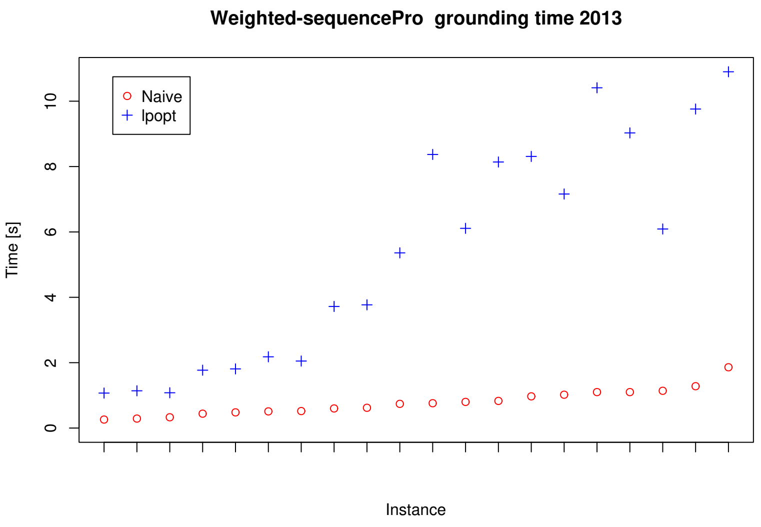
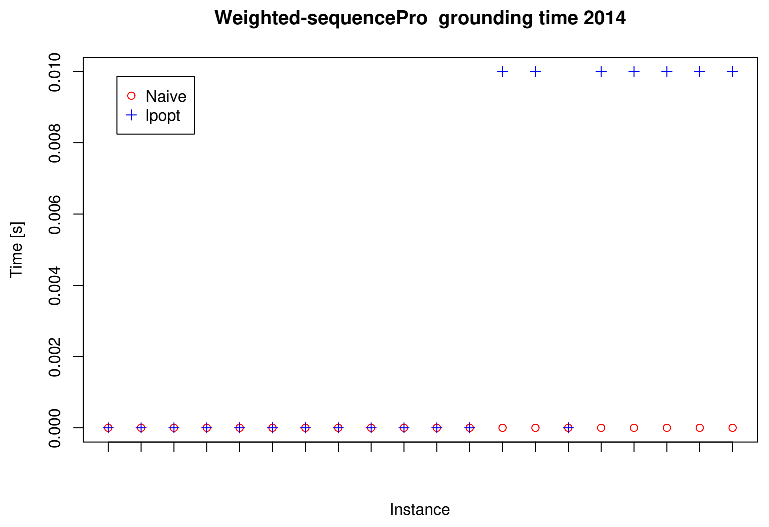
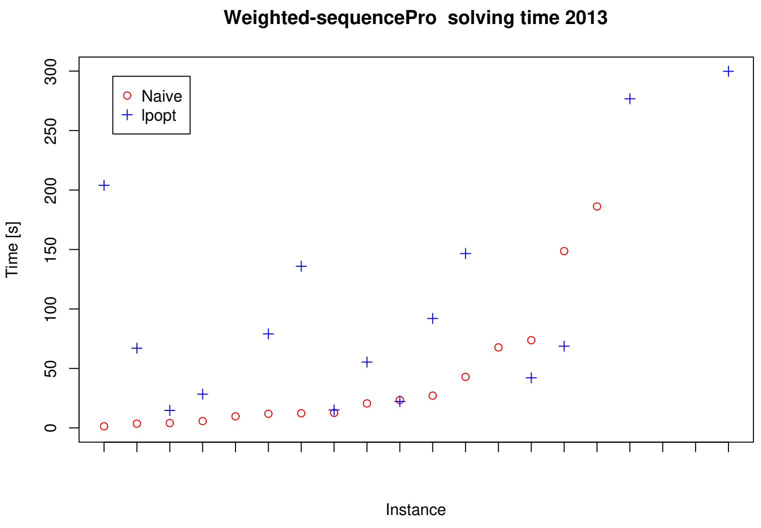
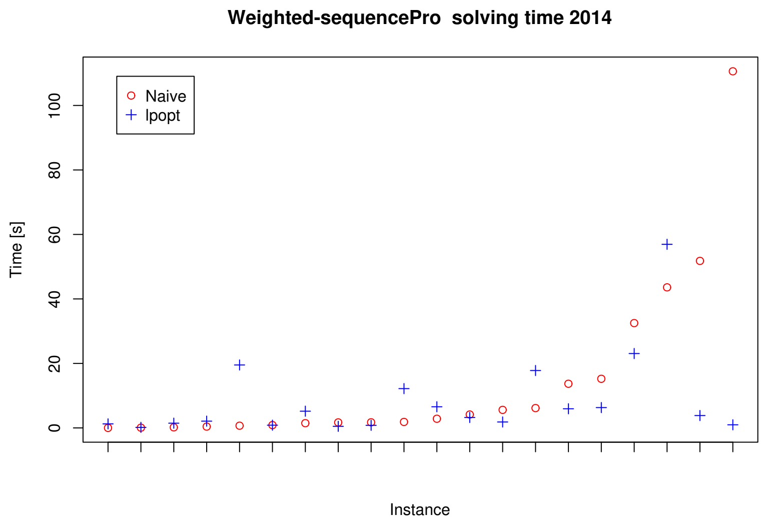
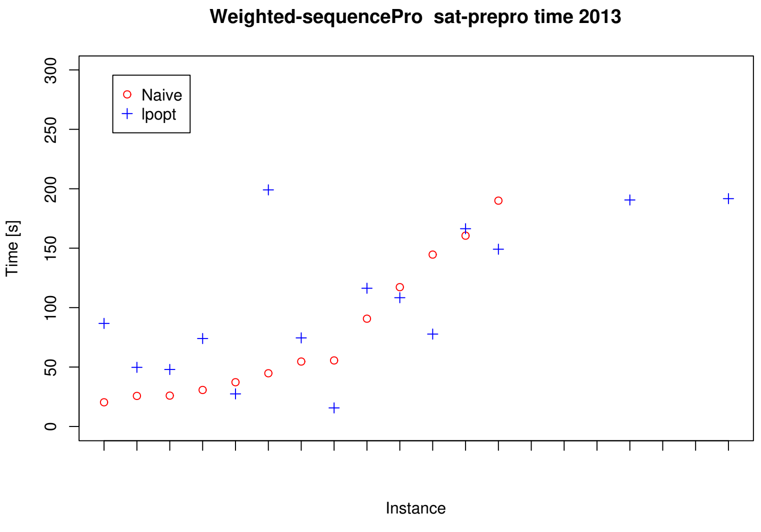
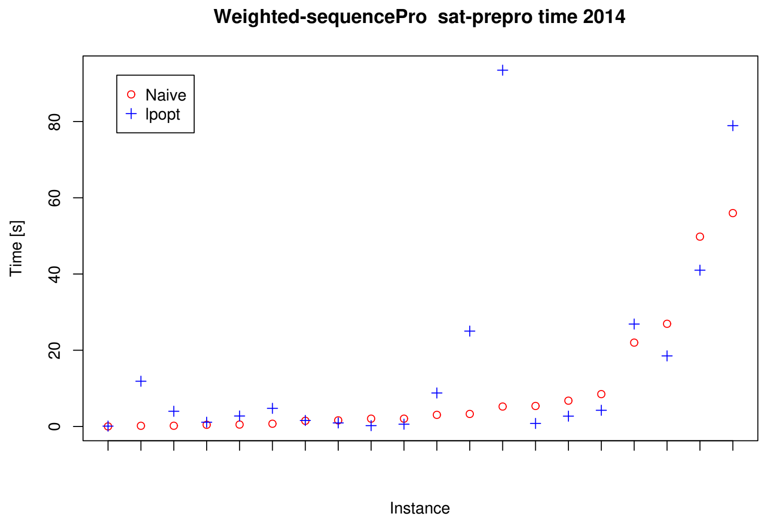

Visit-all-N20 grounding time 2014
Naive
Ipopt
Time [s]
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Instance


Visit-all-N20 solving time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance


Visit-all-N20 solving time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance


Visit-all-N20 sat-prepro time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance


Visit-all-N20 sat-prepro time 2014
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance


Weighted-sequencePro grounding size 2013
Naive
Ipopt
Grounding size
0
500000
1500000
2500000
Instance

Weighted-sequencePro grounding size 2014
Naive
Ipopt
Grounding size
0
200
400
600
800
1000
1200
Instance

Weighted-sequencePro grounding time 2013
Naive
Ipopt
Time [s]
0
2
4
6
8
10
Instance

Weighted-sequencePro grounding time 2014
Naive
Ipopt
Time [s]
0.000
0.002
0.004
0.006
0.008
0.010
Instance

Weighted-sequencePro solving time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

Weighted-sequencePro solving time 2014
Naive
Ipopt
Time [s]
0
20
40
60
80
100
Instance

Weighted-sequencePro sat-prepro time 2013
Naive
Ipopt
Time [s]
0
50
100
150
200
250
300
Instance

Weighted-sequencePro sat-prepro time 2014
Naive
Ipopt
Time [s]
0
20
40
60
80
Instance

Bibliography

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

[Calimeri et al., 2012] Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., and Schaub, T. (2012). Asp-core-2: Input language format.

[Dermaku et al., 2008] Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B. J., Musliu, N., and Samer, M. (2008). Heuristic methods for hypertree decomposition. In *Proc. MICAI*, volume 5317 of *LNCS*, pages 1–11. Springer.

[Gebser et al., 2015] Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., and Thiele, S. (2015). Potassco user guide. *Institute for Informatics, University of Potsdam*, pages 330–331.

[Gebser et al., 2012] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2012). *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

[Gebser et al., 2007] Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. (2007). *clasp* : A conflict-driven answer set solver. In Baral, C., Brewka, G., and Schlipf, J. S., editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer.

[Leone et al., 2006] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562.

[Morak and Woltran, 2012] Morak, M. and Woltran, S. (2012). Preprocessing of complex non-ground rules in answer set programming. In *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, pages 247–258.

[Robertson and Seymour, 1984] Robertson, N. and Seymour, P. D. (1984). Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64.