Analysis of Grounders and Solvers for ASP Encodings of Argumentation Frameworks

Bachelor Thesis

Michael Petritsch*

September 2010

We present an experimental evaluation of popular answer-set programming (ASP) grounders and solvers that were tested with the ASP encodings for argumentation frameworks provided in the technical report "Answer-Set Programming Encodings for Argumentation Frameworks" by Uwe Egly, Sarah Alice Gaggl and Stefan Woltran. We made 21303 tests of randomly generated argumentation frameworks with 90-200 nodes and an edge density of 10-30 percent. The tested candidates were dlv, lparse, gringo, smodels, cmodels, clasp, claspd and gnt.

Keywords: Argumentation Frameworks, Answer Set Programming, Grounder, Solver, Performance Tests

*E-mail: michael.petritsch@gmail.com, Matrikelnr.: 0126861, Kennzahl: 534

Contents

1 Introduction

Argumentation is the study of how humans reach conclusions based on premises via logical reasoning. The roots of argumentation go far back to antiquity to the first days of logic when the ancient Greek, Chinese and Indian philosophers, rhetoricians and mathematicians laid the very foundations. Today's applications include legal reasoning, argument interchange formats, argument-based recommendation technologies, semantic grids, probabilistic argumentation and argument-based machine learning [1].
Embedded in this background, argumentation frameworks (AFs) themselves are an interesting research topic that became more and more popular in artificial intelligence during the last ten years. Basically an AF is a directed cyclic graph with nodes that represent arguments and edges that represent relations between these arguments. We can derive conclusions by applying various mathematically defined extensions. Some of these extensions will be introduced in section 2. Manually working with AFs becomes more and more difficult as AFs grow larger and more complex.
Although properties and complexity are nowadays well understood, the field was lacking a uniform implementation that allowed automatic computation of AFs. Recently Egly et al. [2] introduced ASP encodings for argumentation frameworks allowing the computation of AFs on any kind of answer-set programming (ASP) based solvers. ASP is a declarative programming style that is perfect for defining and computing logical problems.
In the past there have been two large competitions comparing all competing solvers performance-wise: the first answer set competition at the University of Potsdam [3] and the second answer set competition at the Catholic University of Leuven [4]. Both competitions tested solvers computing different kinds of problems. However, there were not any tests made using AFs and the recently introduced encodings. The motivation of this work was to fill out this gap and thoroughly test common ASP grounders and solvers out-of-the-box (using the least necessary optional arguments) specifically with random basic argumentation frameworks ranging from 90 nodes with and edge density of ten percent up to 200 nodes with an edge density of thirty percent.

This paper is structured as follows: In section 2 we give a short introduction to answer-set programming and introduce the definitions of argumentation frameworks and the respective ASP encodings. Next, in section 3 we give an overview over the tested ASP grounders and solvers. We will first have a look at the detailed results of each ASP system and then will compare these systems. Finally we have a look at the test set-up and present the test results in detail in section 4.

2 Answer-Set Programming and Argumentation Frameworks

In this section we will recall answer-set programming as well as the definition of basic argumentation frameworks and the extensions thereof as well as the respective ASP encodings which were used for testing.

2.1 Answer-Set Programming

Answer-set programming (ASP) is a declarative form of programming. There are two languages: lparse and DLV. Since our space is limited we will only introduce the DLV variant:

An ASP program consists of rules which have a head followed by the symbol ":-" with a body at the end. The ":-" means the left side of the rule is true if the right side is. The head and the body of these rules both consist of atoms, with the difference that the atoms in the head can be separated by disjunctions and the atoms in the body are separated by conjunctions. The atoms of the body can also be default negated by adding "not" or true negated by adding "-" before them. Basically the rules are of the form:
$a_1 \vee ... \vee a_n$:- $b_1, ..., b_k$, not $b_{k+1}, ...,$ not b_m
Informally, such a rule is to be read as follows: If $b_1, ..., b_k$ hold and there is no evidence for $b_{k+1}, ..., b_m$ then at least one out of $a_1, ..., a_n$ is true.
Furthermore a program consists of facts which are also referred to as the input database. These facts for example can be something like "male(jack).", "female(jane)." and "married(jack,jane).". This means that jack is male and jane is female and both are married. Facts basically are rules with empty bodies and no variables meaning that they are always true.

Finally a rule with an empty head is called a (strong) constraint.
To finally get some results, all the facts, rules and constraints are taken by the ASP grounder/solver and the answer sets are computed.

2.2 Basic Argumentation Frameworks

The ASP encodings are based on those of [2] (which also includes excellent examples as well as an outlook on ASP encodings for different types of AFs) except that the defeated(a,b) relation was changed to att(a,b).

An argumentation framework is a pair $F = (A, R)$. $A \subseteq U$ is a set of arguments and $R \subseteq A \times A$ is a set of pairs (a, b). A pair $(a, b) \in R$ means that a attacks b. An argumentation framework can be represented as a graph where the elements of A are nodes and the Elements of R are directed edges between these nodes.
The ASP encodings of an AF, more specifically of arguments and attack relations are:

```
arg(a).
att(a,b).
```

Conflict-free Sets

Let $F = (A, R)$ be an AF. A set $S \subseteq A$ is conflict-free if there are no $a, b \in S$, such that $(a, b) \in R$.
The ASP encodings for the conflict-free set (π_{cf}) are defined as:

π_{cf} = {
```
in(X) :- not out(X), arg(X);
out(X) :- not in(X), arg(X);
:- in(X), in(Y), att(X, Y)
```
}

Initially, the first two rules guess any subset of A while the constraint eliminates all such guessed candidates which contain a conflict.

2.3 Extensions of Argumentation Frameworks

For all of the following extensions we assume that $F = (A, R)$ is an AF.

Stable Extension

A set S is a stable extension of F, if $S \in cf(F)$ and each $a \in A \setminus S$ is defeated by S in F.
The ASP encodings for the stable extension (π_{stable}) are:

π_{stable} = π_{cf} $\cup$ {
```
defeated(X) :- in(Y), att(Y,X);
:- out(X), not defeated(X)
```
}

Admissible Extension

A set S is an admissible extension of F, if $S \in cf(F)$ and each $a \in S$ is defended by S in F. An argument a is defended by a set S, if for each b with $(b, a) \in R$ there is a $c \in S$ with $(c, b) \in R$.
The ASP encodings for the admissible extension (π_{adm}) are:

π_{adm} = π_{cf} $\cup$ {
```
defeated(X) :- in(Y), att(Y,X);
:- in(X), att(Y,X), not defeated(Y)
```
}

Complete Extension

A set S is a complete extension of F, if $S \in adm(F)$ and, for each $a \in A$ defended by S (in F), $a \in S$ holds.
The ASP encodings for the complete extension (π_{comp}) are:

π_{comp} = π_{adm} $\cup$ {
```
not_defended(X) :- att(Y,X), not defeated(Y);
:- out(X), not not_defended(X)
```
}

Grounded Extension
The grounded extension of F is the least (wrt set inclusion) complete extension of F.

The ASP encodings for the grounded extension (π_{ground}) are more complex. We have to define additional sets (that are also used for the definition of preferred and semi-stable extensions later). The first one is $\pi_<$:

```
π< = {
lt(X,Y) :- arg(X), arg(Y),X < Y;
nsucc(X,Z) :- lt(X,Y), lt(Y,Z);
succ(X,Y) :- lt(X,Y), not nsucc(X,Y);
ninf(Y) :- lt(X,Y);
inf(X) :- arg(X), not ninf(X);
nsup(X) :- lt(X,Y);
sup(X) :- arg(X), not nsup(X)
}
```

The second one is $\pi_{defended}$:

```
πdefended = {
defended_upto(X,Y) :- inf(Y), arg(X), not att(Y,X);
defended_upto(X,Y) :- inf(Y), in(Z), att(Z,Y), att(Y,X);
defended_upto(X,Y) :- succ(Z,Y), defended_upto(X,Z),
not att(Y,X);
defended_upto(X,Y) :- succ(Z,Y), defended_upto(X,Z), in(V),
att(V,Y), att(Y,X);
defended(X) :- sup(Y), defended_upto(X,Y)
}
```

And finally the grounded extension is:

```
πground = π< ∪ πdefended ∪ { in(X) :- defended(X) }
```

Preferred Extension
A set S is a preferred extension of F, if $S \in adm(F)$ and for each $T \in adm(F)$, $S \not\subset T$. For the ASP encodings of the preferred extension(π_{pref}) we also have to define additional sets of rules:

```
πeq = {
eq_upto(X) :- inf(X), in(X), inN(X);
eq_upto(X) :- inf(X), out(X), outN(X);
eq_upto(X) :- succ(Y,X), in(X), inN(X), eq_upto(Y);
eq_upto(X) :- succ(Y,X), out(X), outN(X), eq_upto(Y);
eq :- sup(X), eq_upto(X)
};
```

$\pi_{undefeated}$ = {
```
undefeated_upto(X,Y) :- inf(Y), outN(X), outN(Y);
undefeated_upto(X,Y) :- inf(Y ), outN(X), not att(Y,X);
undefeated_upto(X,Y) :- succ(Z,Y), undefeated_upto(X,Z),
outN(Y);
undefeated_upto(X,Y) :- succ(Z,Y), undefeated_upto(X,Z),
not att(Y,X);
undefeated(X) :- sup(Y), undefeated_upto(X,Y)
};
```

$\pi_{satpref}$ = {
```
inN(X) ∨ outN(X) :- out(X);
inN(X) :- in(X);
sat :- eq;
sat :- inN(X), inN(Y), att(X,Y);
sat :- inN(X), outN(Y), att(Y,X), undefeated(Y);
inN(X) :- sat, arg(X);
outN(X) :- sat, arg(X);
:- not sat
}
```

Now finally we can define:

π_{pref} = π_{adm} ∪ $\pi_{<}$ ∪ π_{eq} ∪ $\pi_{undefeated}$ ∪ $\pi_{satpref}$

Semi-stable Extension

For a set $S \subseteq A$, let S_R^+ be defined as $S \cup \{b | \exists a \in S, suchthat(a,b) \in R\}$. A set S is a semi-stable extension of F, if $S \in adm(F)$ and for each $T \in adm(F)$, $S_R^+ \not\subset T_R^+$.
For the ASP encodings of the semi-stable extension(π_{semi}) we also have to define additional sets of rules:

π_{eq}^+ = {
```
eqplus_upto(X) :- inf(X), in(X), inN(X);
eqplus_upto(X) :- inf(X), in(X), inN(Y), att(Y,X);
eqplus_upto(X) :- inf(X), in(Y), inN(X), att(Y,X);
eqplus_upto(X) :- inf(X), in(Y), inN(Z), att(Y,X), att(Z,X);
eqplus_upto(X) :- inf(X), out(X), outN(X), not defeated(X),
undefeated(X);
eqplus_upto(X) :- succ(Z,X), in(X), inN(X), eqplus_upto(Z);
eqplus_upto(X) :- succ(Z,X), in(X), inN(Y), att(Y,X),
eqplus_upto(Z);
eqplus_upto(X) :- succ(Z,X), in(Y), inN(X), att(Y,X),
eqplus_upto(Z);
eqplus_upto(X) :- succ(Z,X), in(Y), inN(U), att(Y,X), att(U,X),
eqplus_upto(Z);
eqplus_upto(X) :- succ(Y,X), out(X), outN(X), not defeated(X),
```

```
undefeated(X), eqplus_upto(Y);
eqplus :- sup(X), eqplus_upto(X)
};
```

$\pi_{satsemi}$ = {
```
inN(X) ∨ outN(X) :- arg(X);
sat :- eqplus;
sat :- inN(X), inN(Y), att(X,Y);
sat :- inN(X), outN(Y), att(Y,X), undefeated(Y);
sat :- in(X), outN(X), undefeated(X);
sat :- in(Y), att(Y,X), outN(X), undefeated(X);
inN(X) :- sat, arg(X);
outN(X) :- sat, arg(X);
:- not sat
}
```

Now finally we can define:

$\pi_{semi} = \pi_{adm} \cup \pi_{<} \cup \pi_{eq}^{+} \cup \pi_{undefeated} \cup \pi_{satsemi}$

3 Tested ASP Systems

In this section we will have a look at the tested ASP grounders and solvers.

3.1 Grounders

Grounders are the first stage of computing an ASP program. The program code is used as input for a grounder and the grounder removes all variables and produces ground terms. The input language basically consists of four kinds of things: Constants, Variables, Atoms and Rules [5].
All grounders are capable of processing disjunctive rules. A disjunctive rule is a rule with a disjunction in its head. The syntax for disjunctive rules however differs.

Dlv

Dlv[1] is a deductive database system, based on disjunctive logic programming, which offers front-ends to several advanced KR formalisms. It is a collaboration between various departments from the Vienna University of Technology and the University of Calabria [6] and is both a grounder and a solver combined into one program [7].
It uses a slightly different input syntax than the other grounders. Since it is both a grounder and a solver the output is a solved program. As a result the grounding times were not tested separately. dlv automatically detects whether an input program contains disjunctive rules or not. Disjunctive rule heads in dlv have the following syntax:

```
a v b
```

[1] http://www.dbai.tuwien.ac.at/proj/dlv/

The version used for testing was 2007-10-11.

Lparse

Lparse[2] is the front-end part of the smodels system that consists of smodels, an efficient implementation of the stable model semantics for normal logic programs, and lparse itself, that transforms user programs into a form that smodels understands [5]. It is used as front-end for any other solver (except for dlv which has it's own built in grounding module) tested in this paper.
Lparse's grounding times were tested separately and the results will be presented in section 4. For disjunctive programs lparse has to be started with the option –dlp. Disjunctive rule heads in lparse have the following syntax:

```
a | b
```

The version used for testing was 1.0.3.

GrinGo

Gringo[3] is a part of the "Potsdam Answer Set Solving Collection" [8].
It is a grounder that combines and extends techniques from dlv [9]. It uses an extended version of the lparse language as input and produces a compatible output format [10]. As a result all programs that understand lparse output also understand gringo output.

gringo's grounding times were tested separately and the results will be presented in 4. Gringo automatically detects whether an input program contains disjunctive rules or not.
The version used for testing was 2.0.4.

Grounders round up

Grounder	Version	disjunctions	notes
dlv	2007-10-11	yes, automatically	output is a solved program
lparse	1.0.3	yes, with option –dlp	output compatible with all solvers except dlv
gringo	2.0.4	yes, automatically	output compatible with all solvers except dlv

3.2 Solvers

Next we have a look at the tested solvers. All of the tested solvers except for dlv were tested using both lparse and gringo.

Dlv

See 3.1 in the previous subsection.

[2] http://www.tcs.hut.fi/Software/smodels/
[3] http://potassco.sourceforge.net/

Smodels
Smodels[4] is the solving part of the smodels system. It is not capable of solving disjunctive rules.
The version used for testing was 2.34. [11]

Cmodels
Cmodels[5] was written by Yulia Lierler with the help of other members of Texas Action Group at Austin and Marco Maratea. The program includes some parts of the code written by Patrik Simons for system smodels. It is a solver that is capable of solving disjunctive rules. Support for disjunctive rules is activated with the option -dlp [12]. However, in some cases it failed during testing.
The version used for testing was 3.79.

Clasp
Clasp[6] is a part of the "Potsdam Answer Set Solving Collection" [8]. It is not capable of solving disjunctive rules [13, 14]. The version used for testing was 1.3.2.

ClaspD
ClaspD[7] was written by Members of the University of Potsdam. It is an extended version of clasp that is capable of solving disjunctive rules [15].
The version used for testing was 1.1.

Gnt
Gnt[8] (Generate'n'Test) was written by members of the Helsinki University of Technology. It is based on an architecture consisting of two interacting smodels solvers for non-disjunctive programs [16].
Gnt was difficult to set up (requiring a lot of extra libraries) and in the end it failed on all disjunctive programs. The version used for testing was 2.1.

Solvers round up

Solver	Version	disjunctions	default output
dlv	2007-10-11	yes	all models
smodels	2.34	no	one model
cmodels	3.79	yes, with option -dlp	one model
clasp	1.3.2	no	one model
claspd	1.1	yes, automatically	one model
gnt	2.1	yes, automatically	one model

[4] http://www.tcs.hut.fi/Software/smodels/
[5] http://userweb.cs.utexas.edu/~tag/cmodels/
[6] http://potassco.sourceforge.net/
[7] http://www.cs.uni-potsdam.de/claspD/
[8] http://www.tcs.hut.fi/Software/gnt/

4 Experiments

In this section we will have a look at the experiments, starting with an overview how the tests were set up, how they were performed, how the results were obtained and finally we present these results.

4.1 Test set-up

Test System

The machine on which testing took place was a Pentium 4 with 2 GHz, 2 GB Ram running openSUSE 11.1 (Kernel 2.6.27).

Basic AF generator

Prior to testing we had to write a generator for random basic argumentation frameworks. With this generator we created ten test sets (folders), each containing random argumentation frameworks ranging from 90 nodes with 10 percent edge density to 200 nodes with 30 percent edge density with a total of 36 instances (files) per set.

Preparation of Encodings

The ASP encodings from [2] which are in dlv syntax were put into separate files. Since the lparse/gringo input syntax slightly differs from the dlv input syntax adjusted encodings were put into different files. Both the original dlv encodings and the adjusted ones were tested to verify their correctness.

Installation of grounders and solvers

Next we installed the grounders and solvers and wrote bash scripts for automatic test execution. The scripts were designed to execute the grounders and solvers in the most basic way possible, with the least necessary options.

Testing and Test output

The computing times of each program were measured with Linux' time command [17] and for each test performed we created two files, one containing the output of the grounder/solver and the other containing the time taken by the test.
A total of 21303 tests were performed which occupied the processor for 626637 seconds. After performing the tests a program was written to collect the computing times and the number of models of each test from the files that were created during testing to put them into a database.

Creating statistics

A third program was written to create statistics from the data in the database which were then put into Gnuplot to generate diagrams.

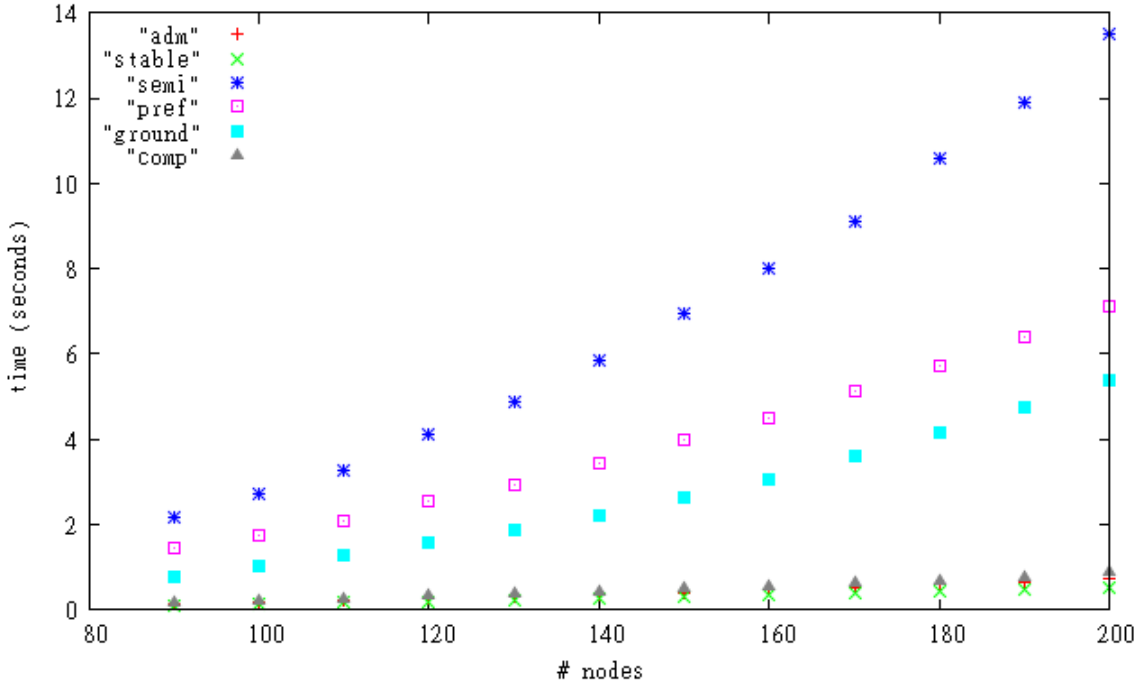
4.2 Detailed test results for each ASP system

In this section we will have a look at the actual test results. For every tested solver and grounder we will present diagrams which show the number of nodes on the x-axis and the amount of time in seconds on the y axis. Every diagram represents a different edge density ranging from ten to thirty percent. An edge density of 100 percent means that every possible edge in a directed cyclic graph is existent (n^2 edges where n is the number of nodes). Every diagram contains all the tested extensions unless the solver failed to solve them or the computation times were way too high.

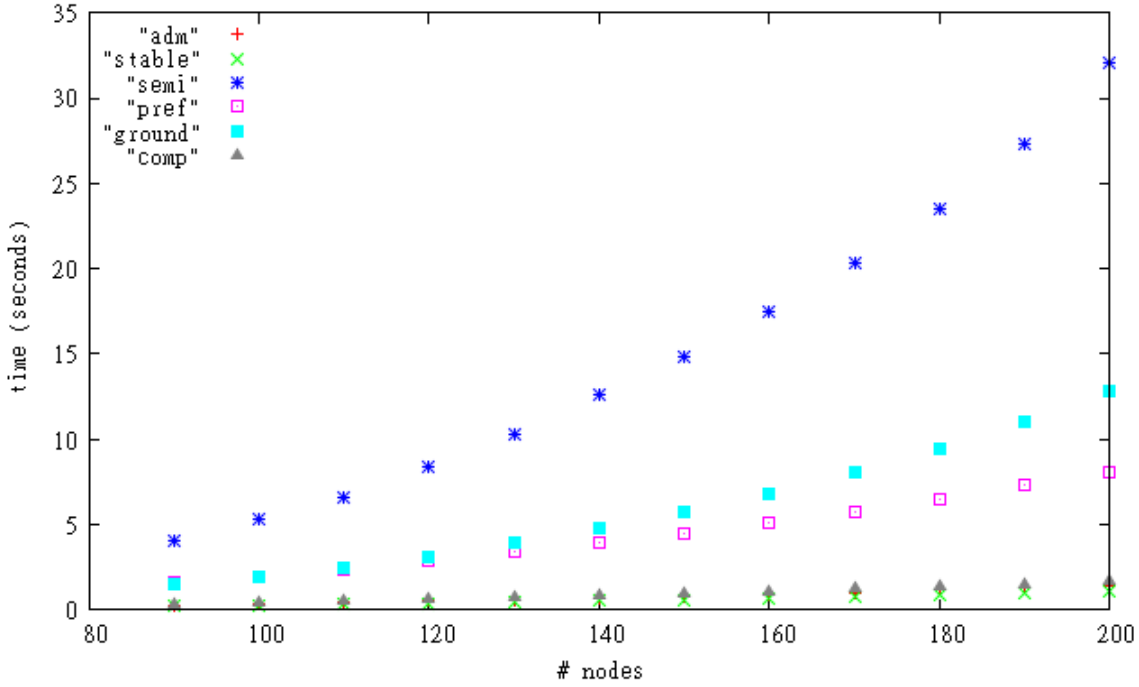
4.2.1 Lparse

As we can see lparse grounds all the samples quite fast. Admissible, stable and complete are quite equal followed by preferred and grounded. Interestingly preferred is slower than grounded on ten percent edge density but faster on twenty and thirty percent edge density. Semi-stable is the extension that takes the longest.

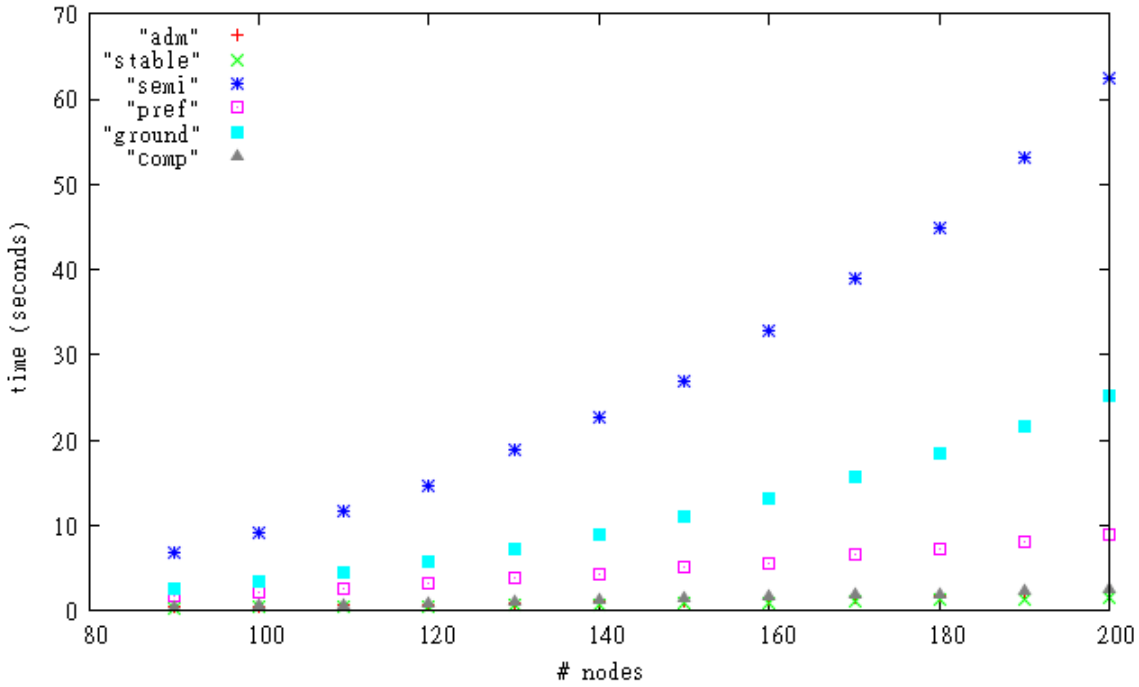
Edge density 10 %



Edge density 20 %



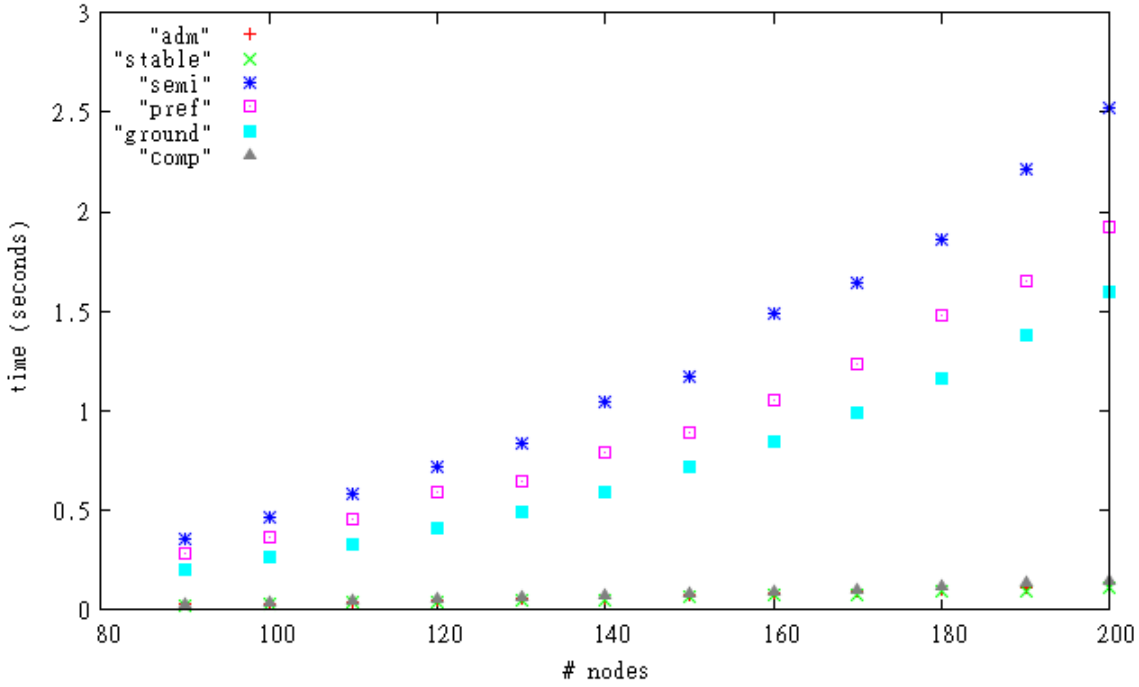
Edge density 30 %



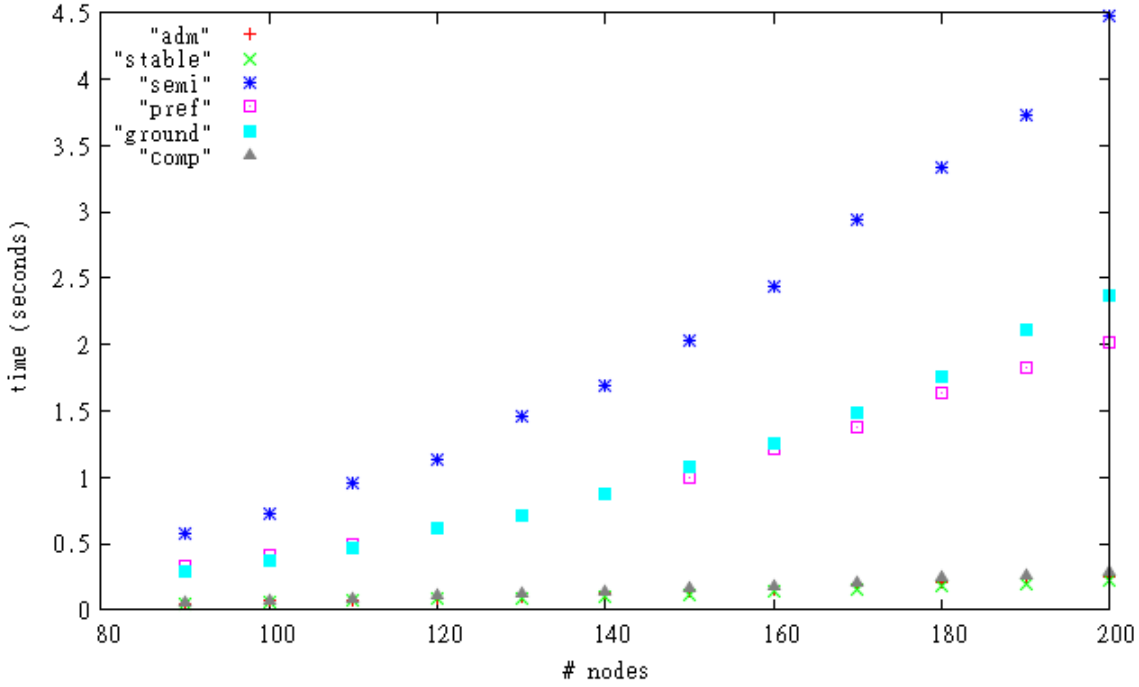
4.2.2 GrinGo

With gringo we have an extension behaviour that is quite similar to lparse, except that everything is computed faster.

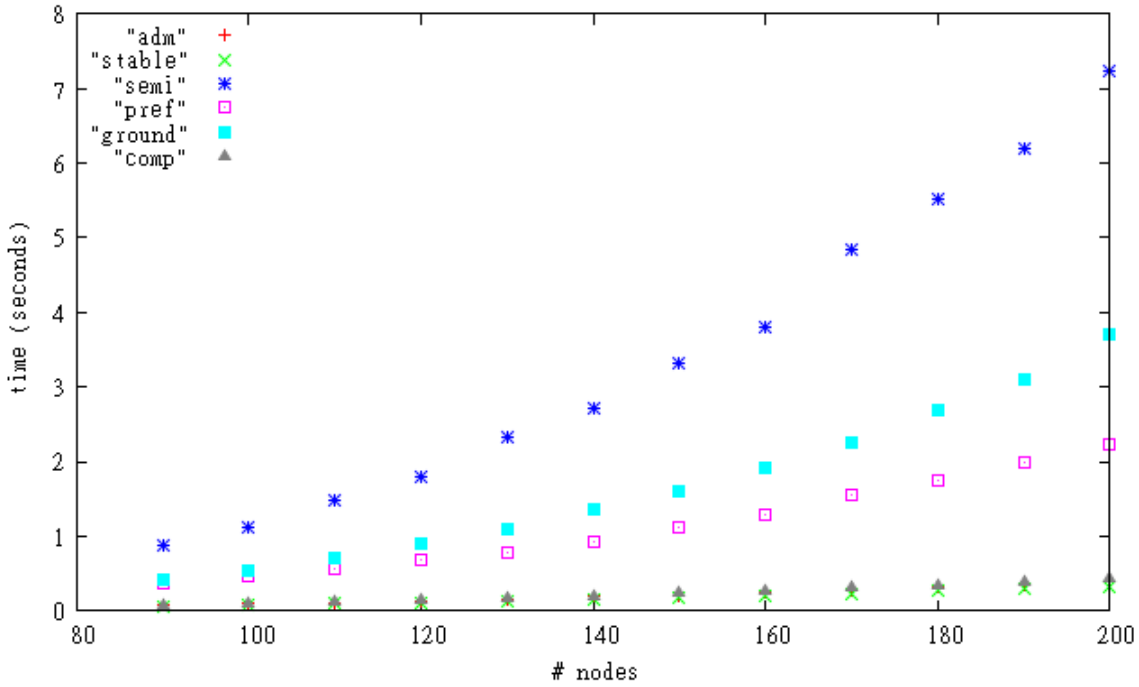
Edge density 10 %



Edge density 20 %



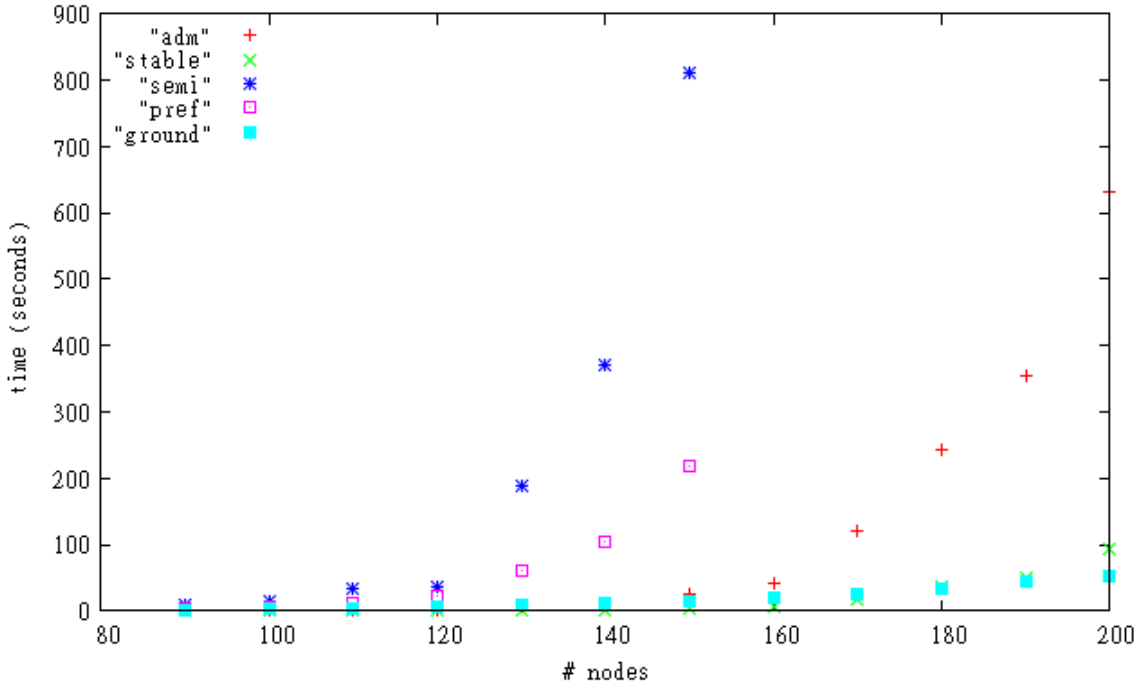
Edge density 30 %



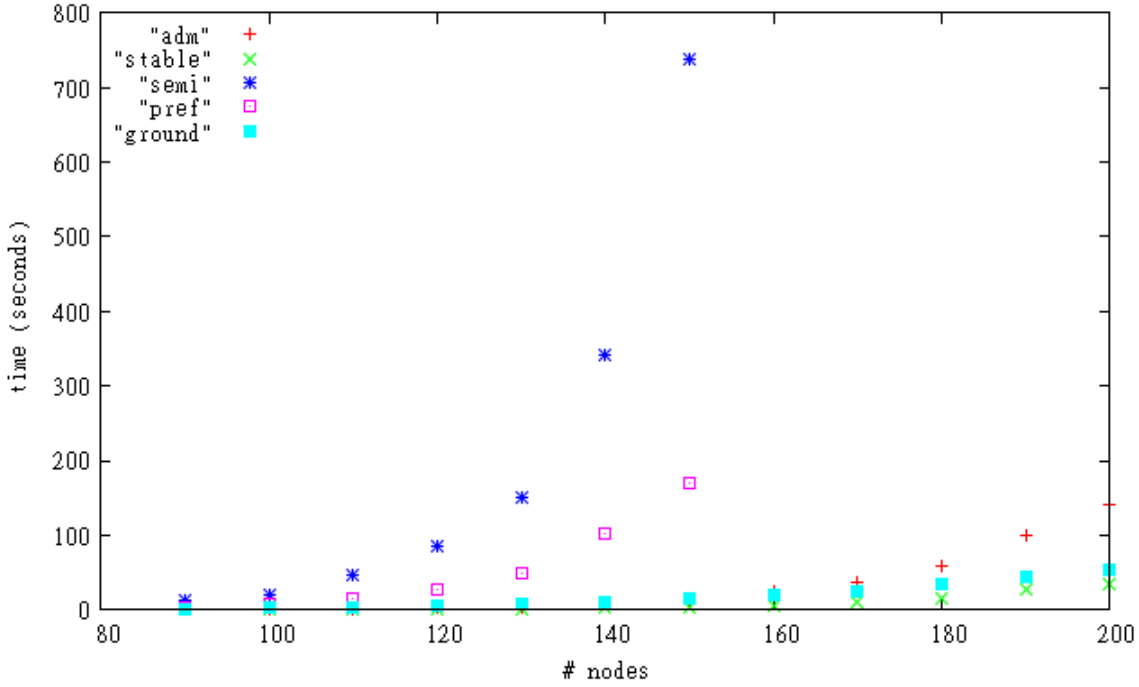
4.2.3 Dlv

Because of high computation times we decided not to include the results for the complete extensions in the diagrams for dlv. Here we can observe an interesting behaviour of most extensions for the first time: their computation times decrease with increasing edge density. Here we can see again that semi-stable is the slowest, followed by preferred. Stable, admissible and grounded are quite similar, however, admissible shows a noteworthy increase at ten percent edge density when reaching higher amounts of nodes.

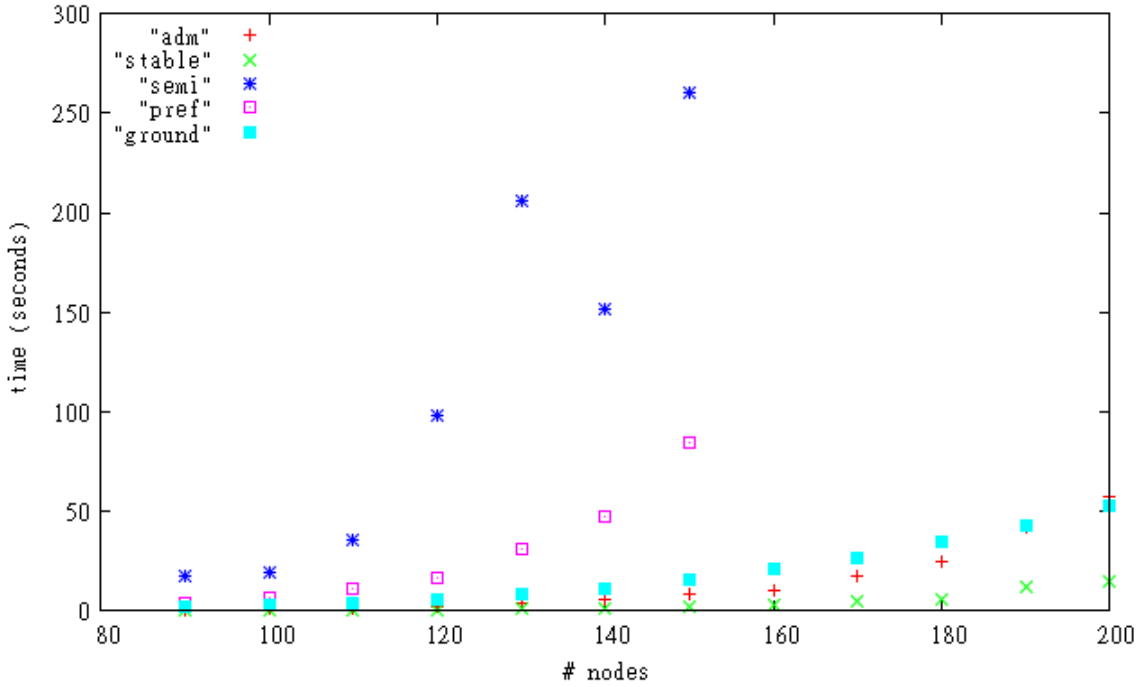
Edge density 10 %



Edge density 20 %



Edge density 30 %

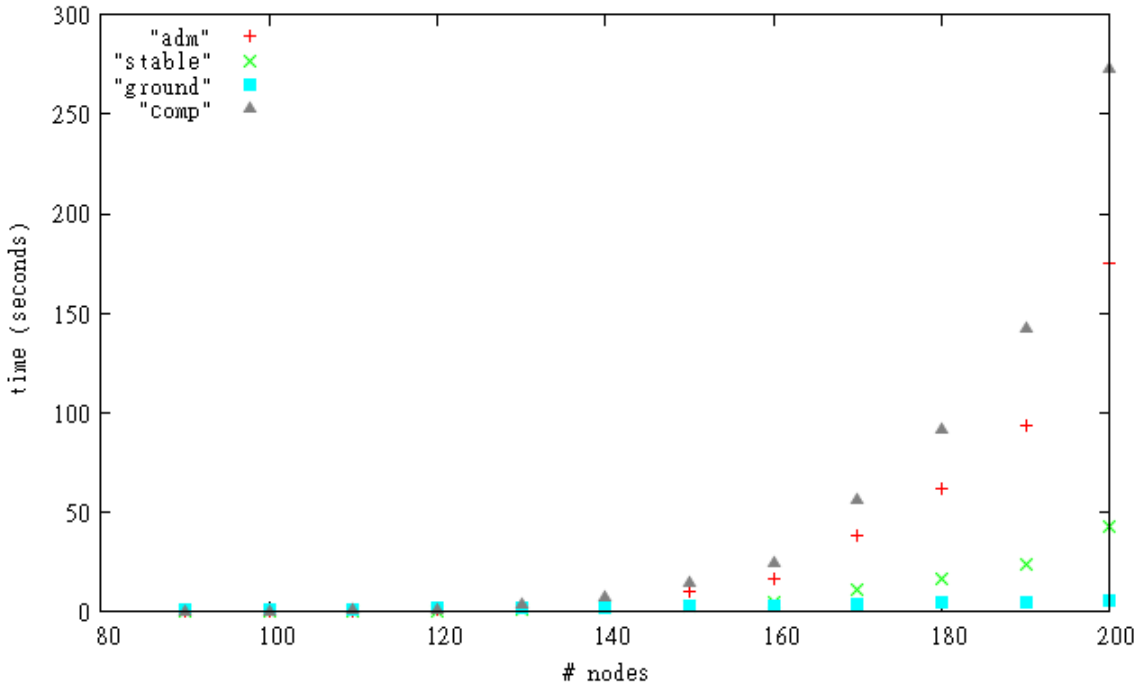


4.2.4 Smodels

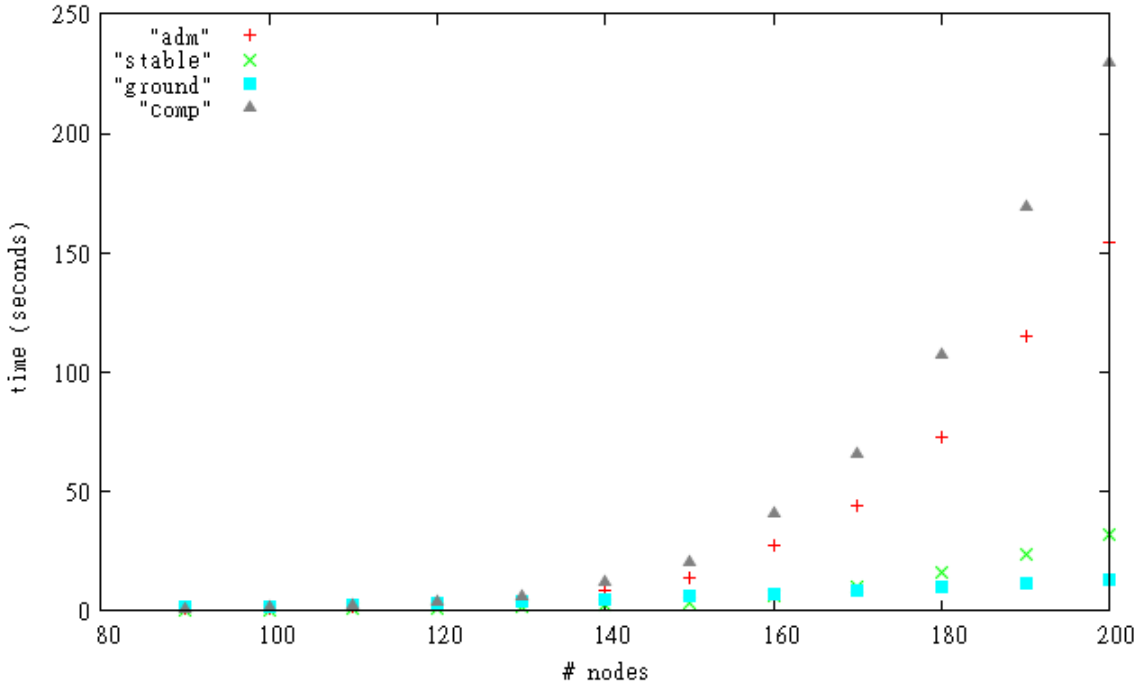
With smodels we notice ground and stable taking the least amount of time and admissible and complete being slower. Also smodels is faster with lparse grounding than with gringo.

Lparse results

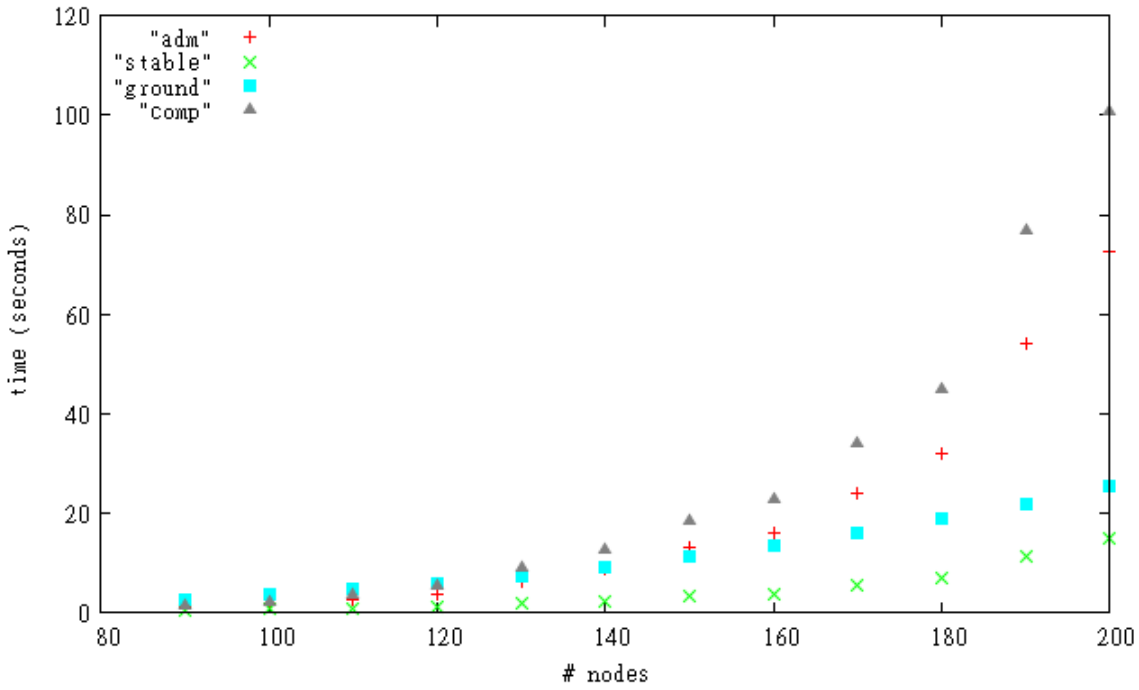
Edge density 10 %



Edge density 20 %

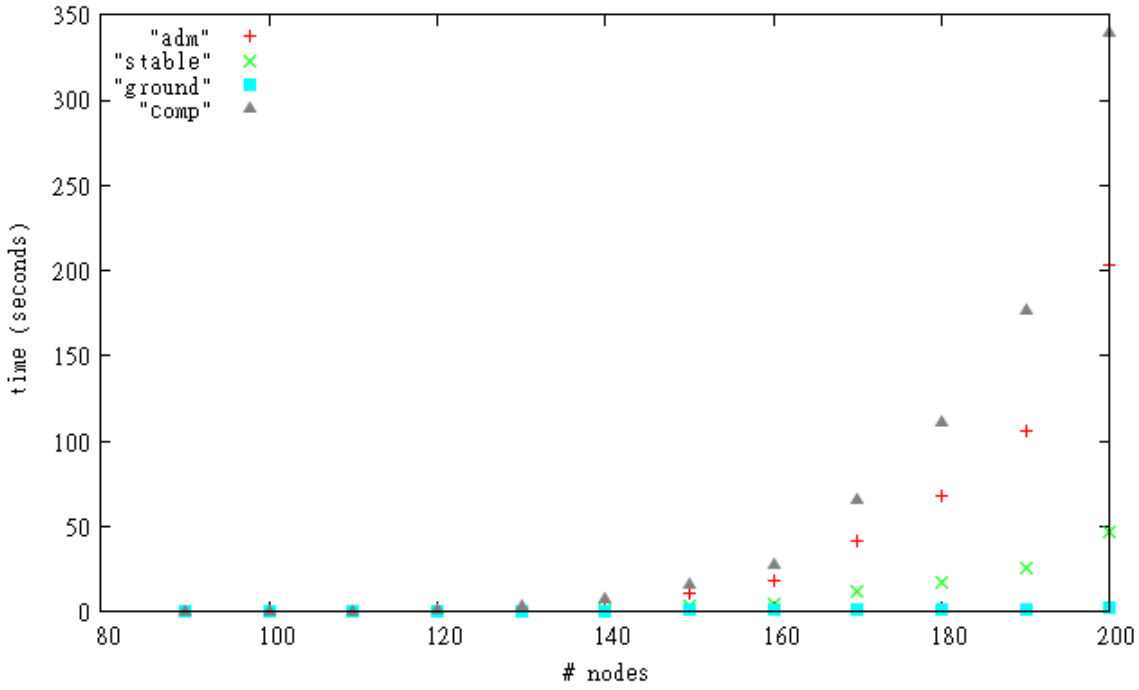


Edge density 30 %

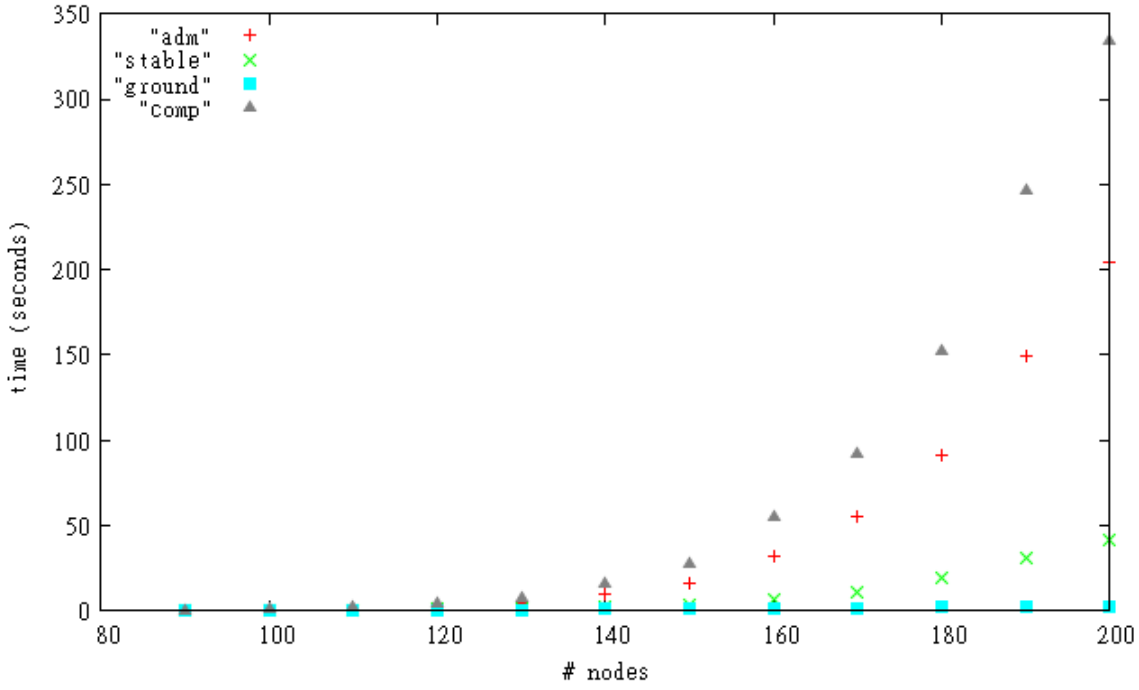


GrinGo results

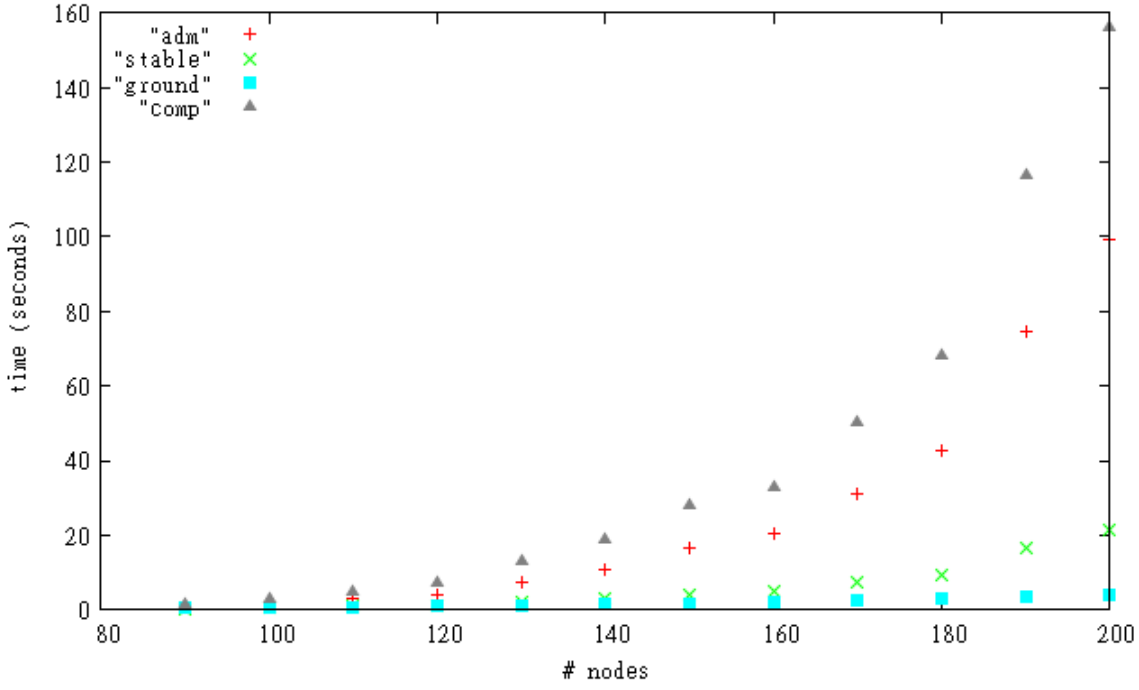
Edge density 10 %



Edge density 20 %



Edge density 30 %

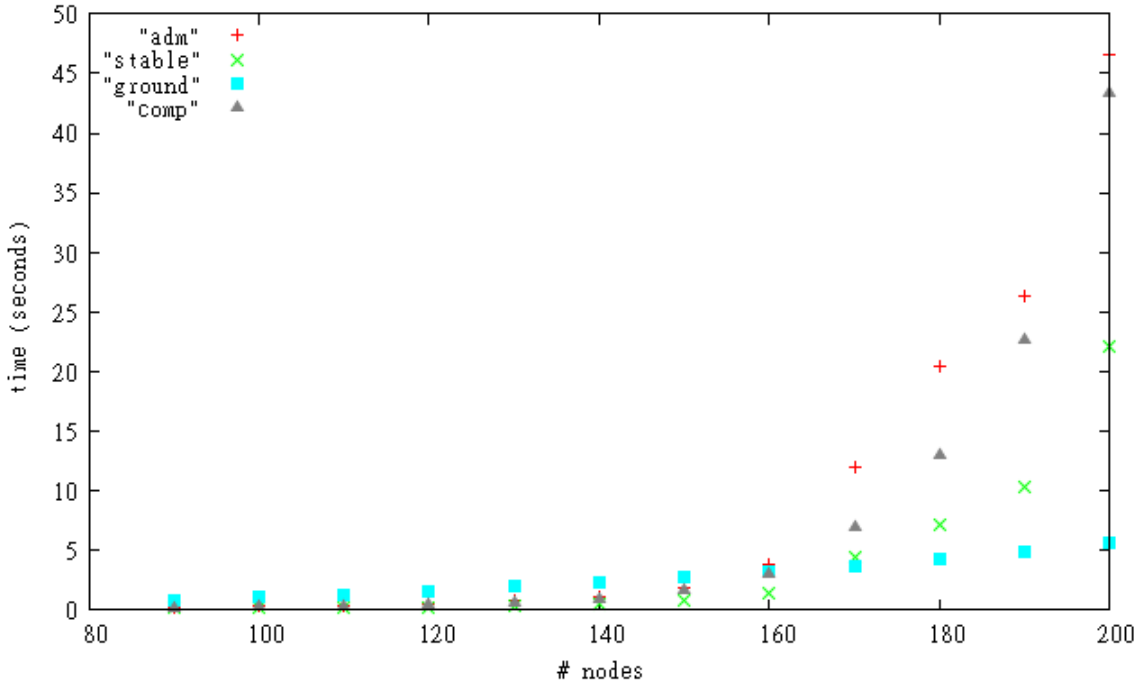


4.2.5 Cmodels

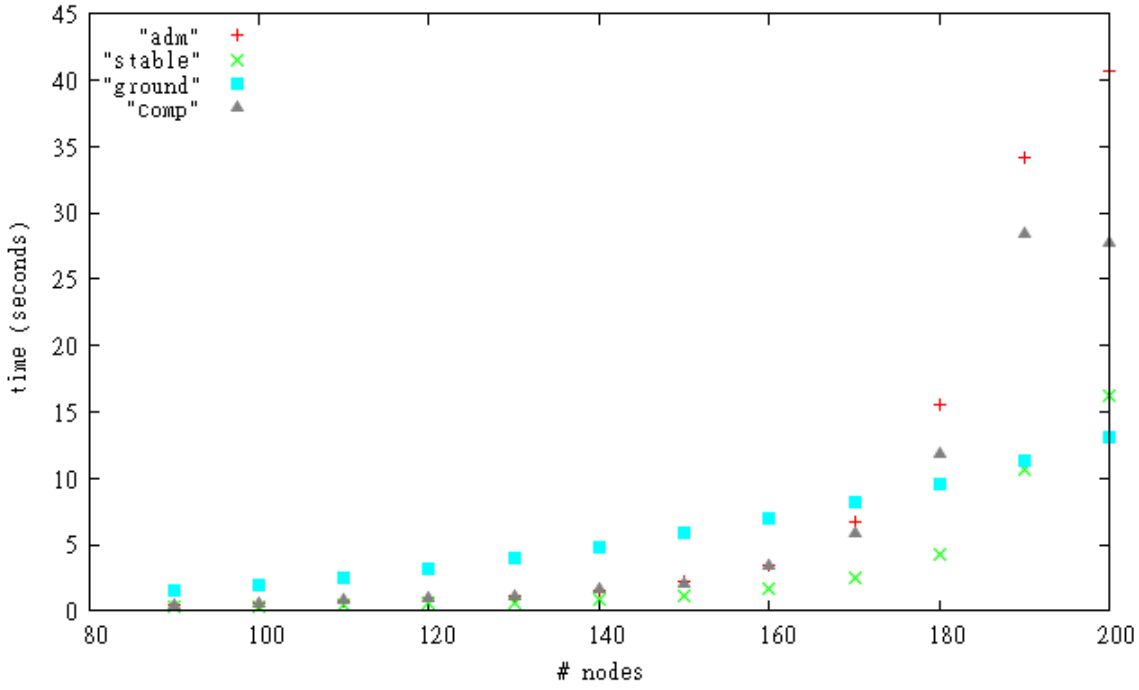
Lparse results

The results for semi-stable and preferred might as well be ignored since cmodels crashed on most tests with gringo and on all tests with lparse. Here we notice grounded rising with increasing edge density, whereas admissible and stable are falling. Cmodels performed better with gringo.

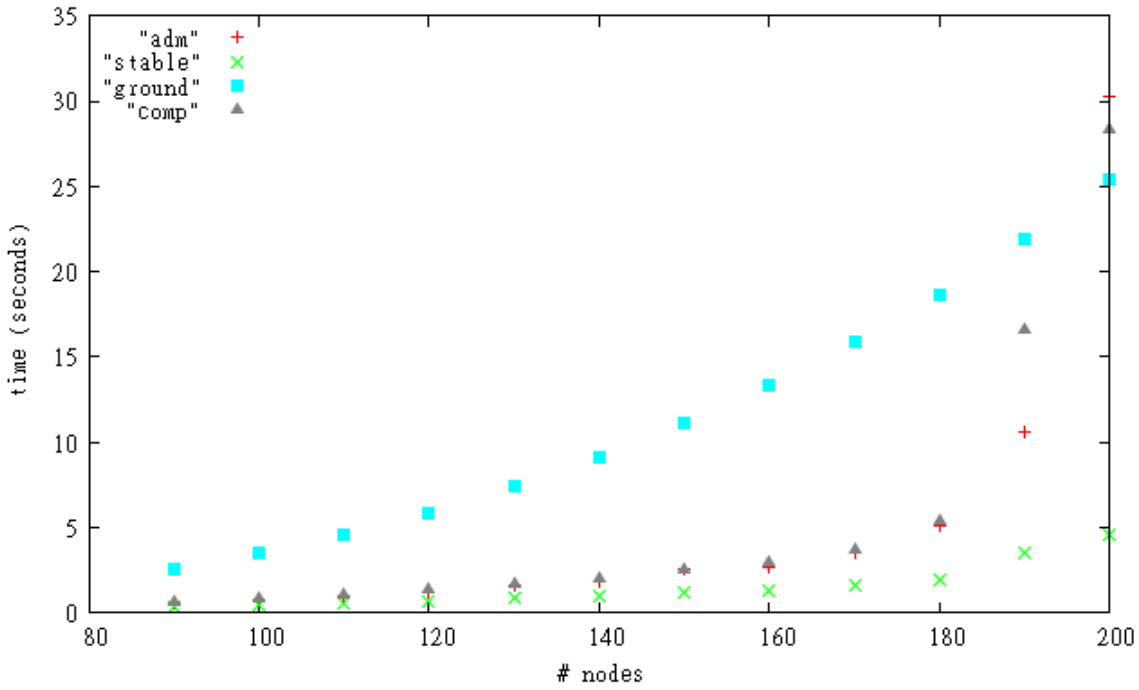
Edge density 10 %



Edge density 20 %

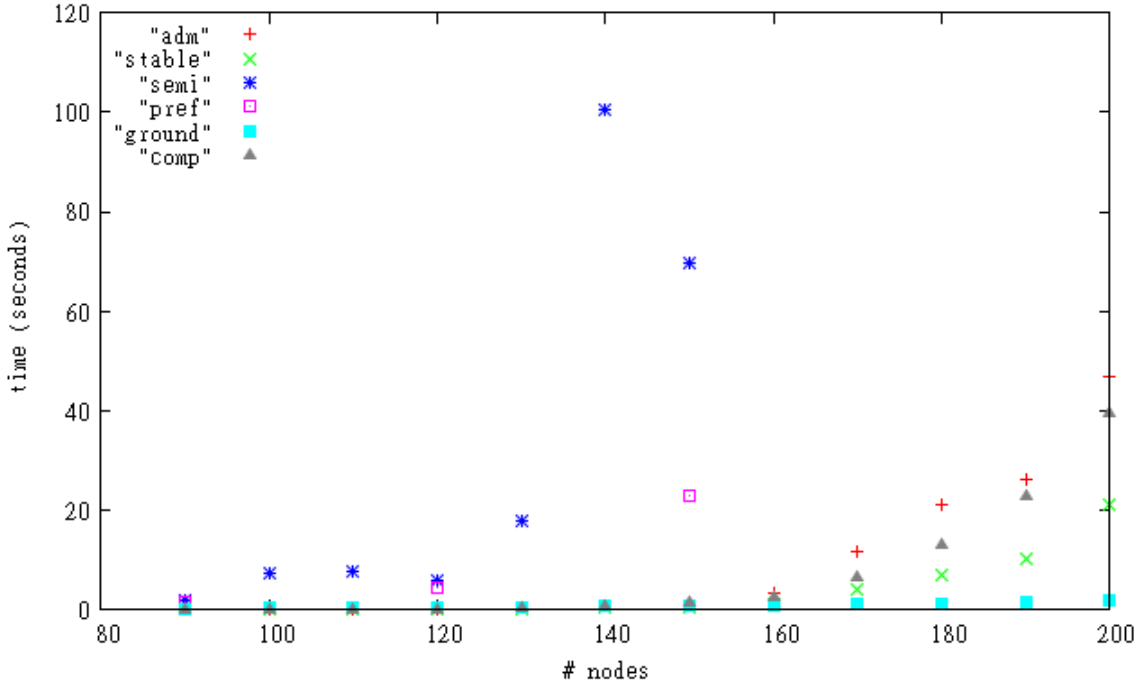


Edge density 30 %

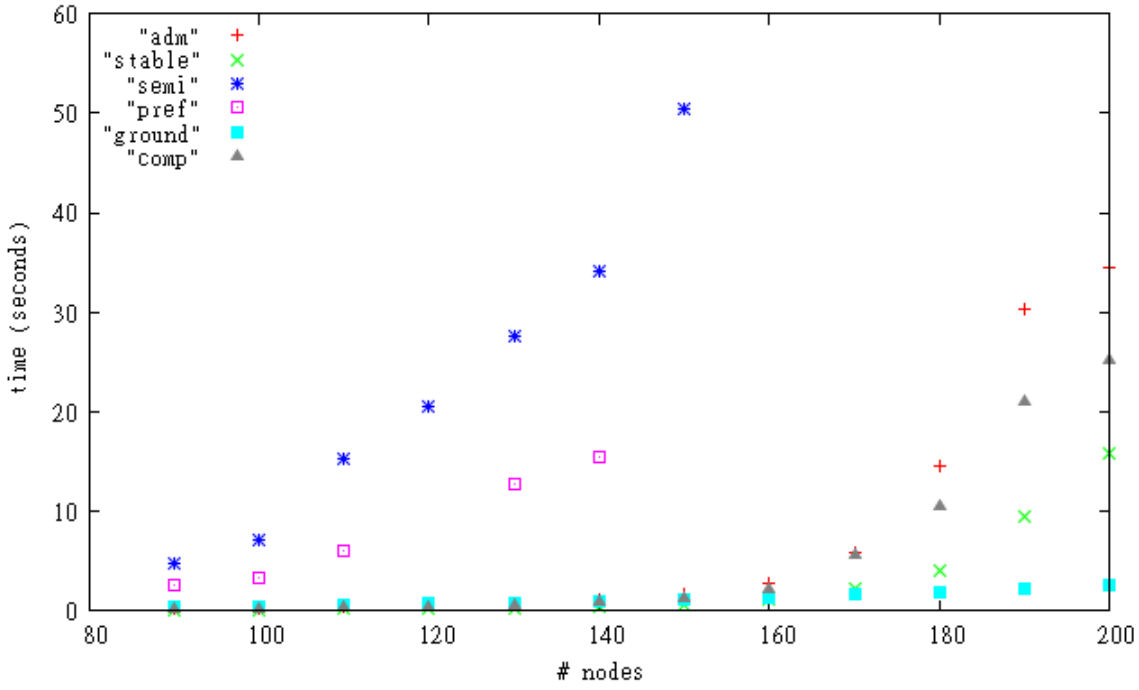


GrinGo results

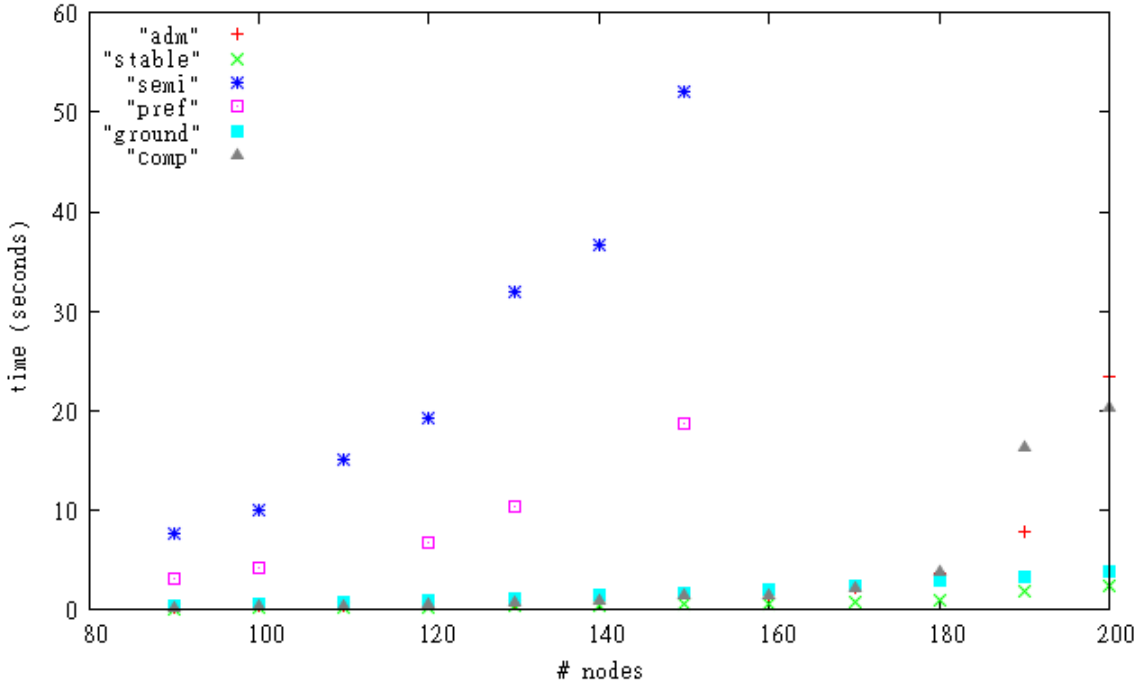
Edge density 10 %



Edge density 20 %



Edge density 30 %

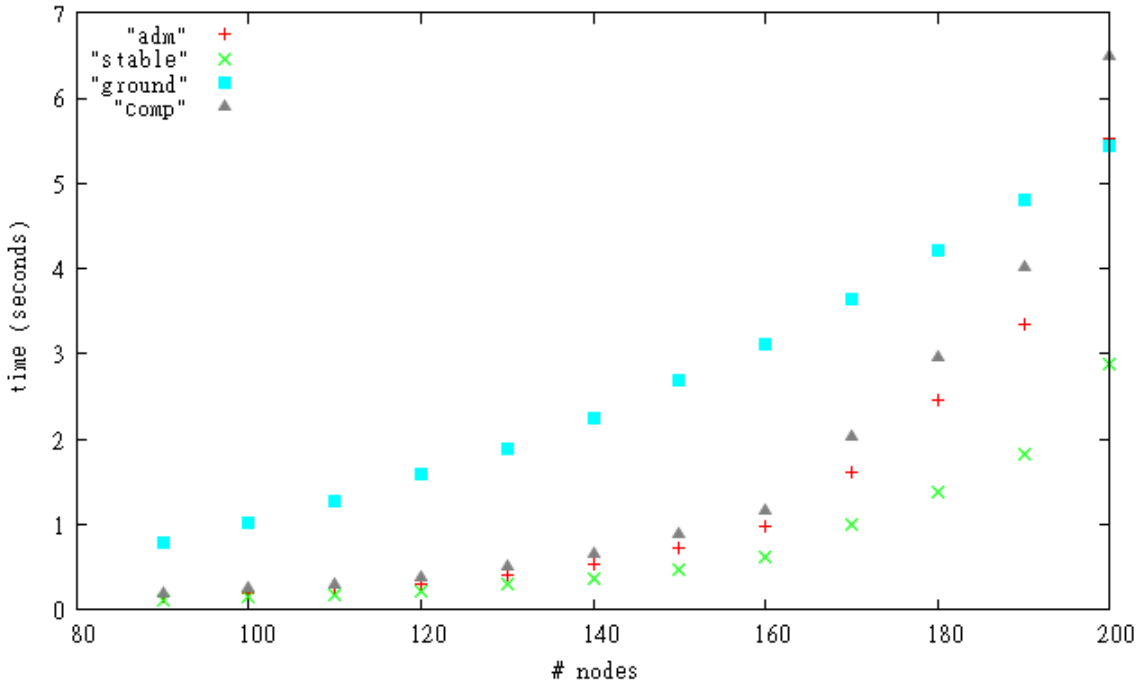


4.2.6 Clasp

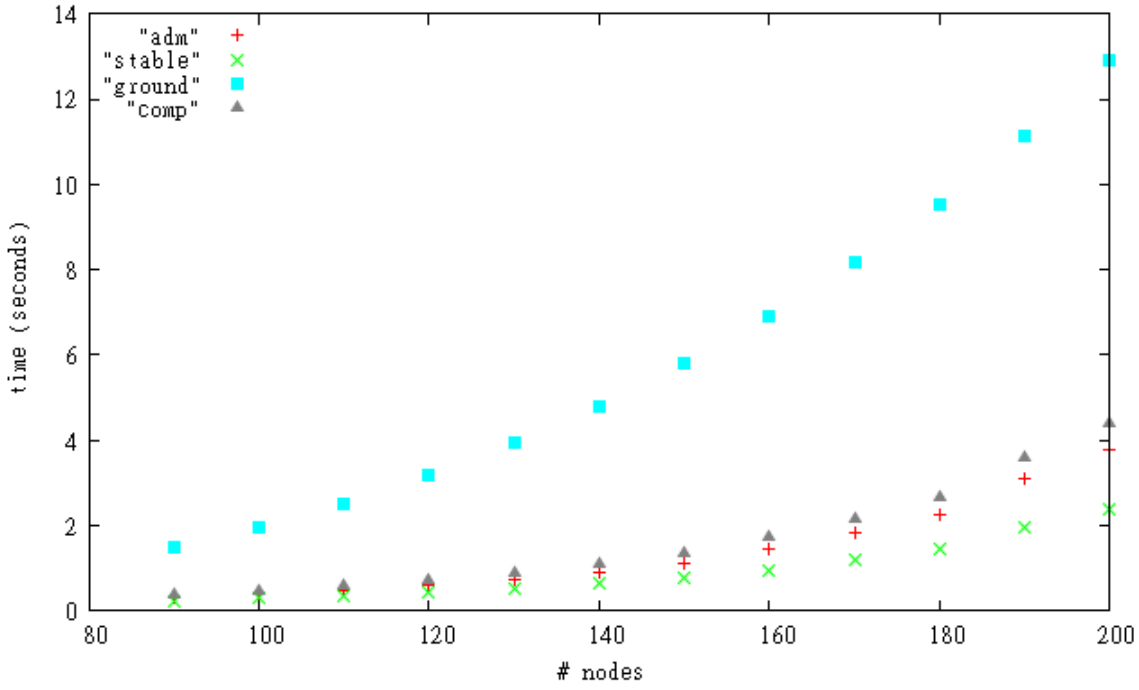
Clasp was the fastest solver for non-disjunctive encodings in the field. It performed better with gringo than with lparse.

Lparse results

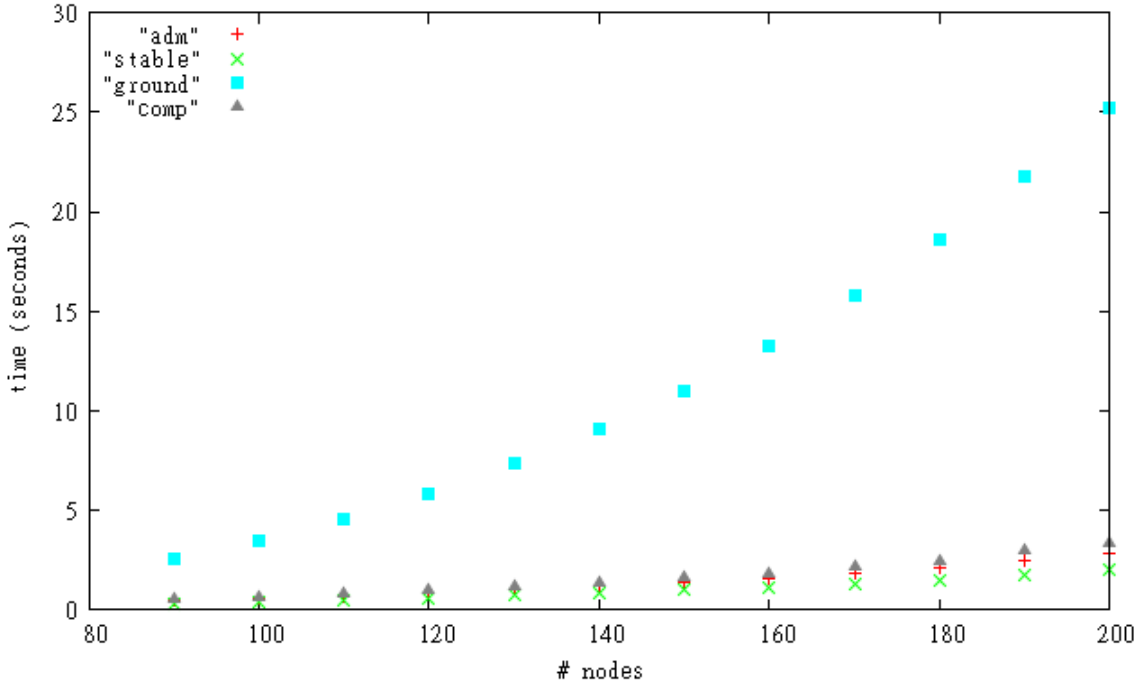
Edge density 10 %



Edge density 20 %

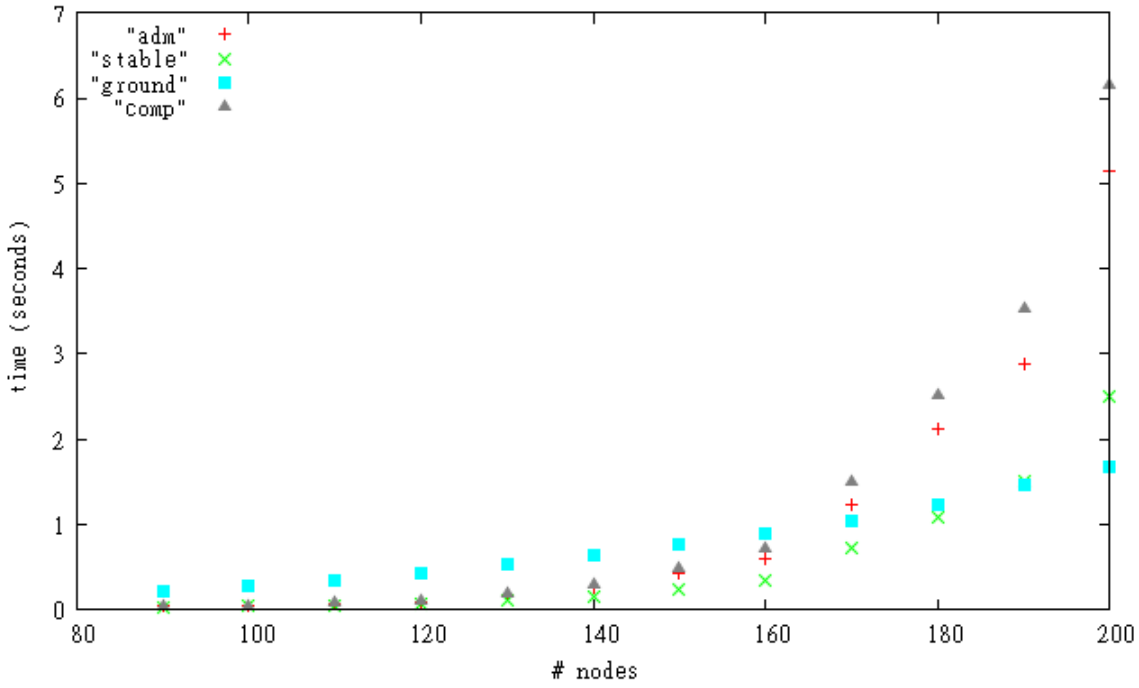


Edge density 30 %

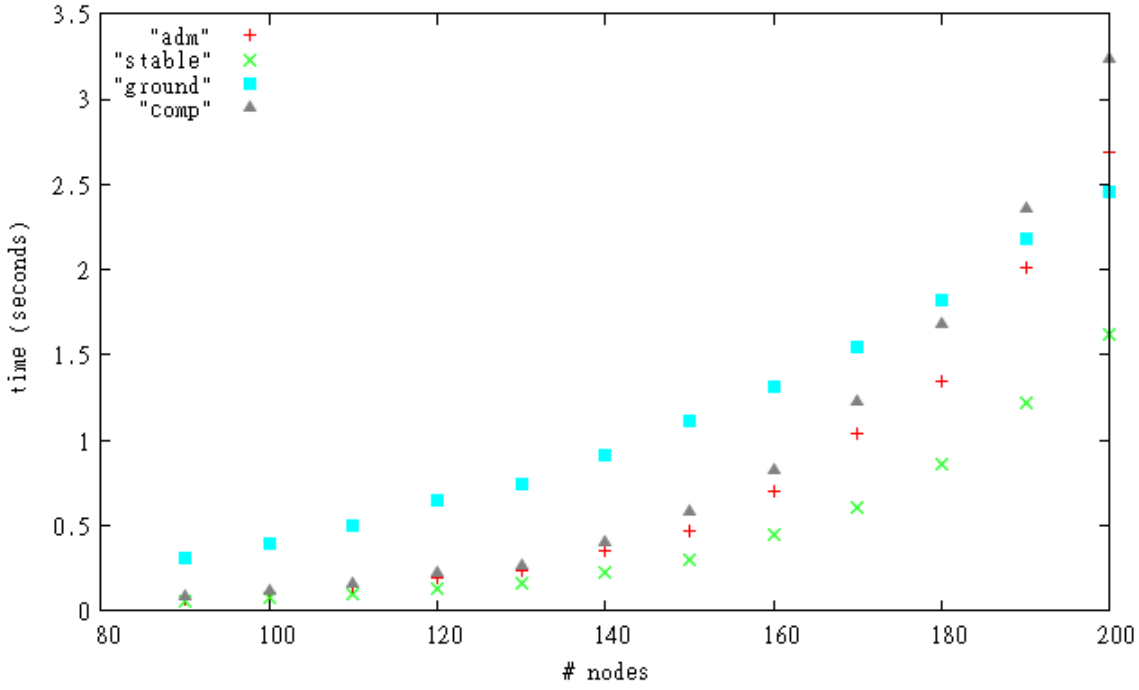


GrinGo results

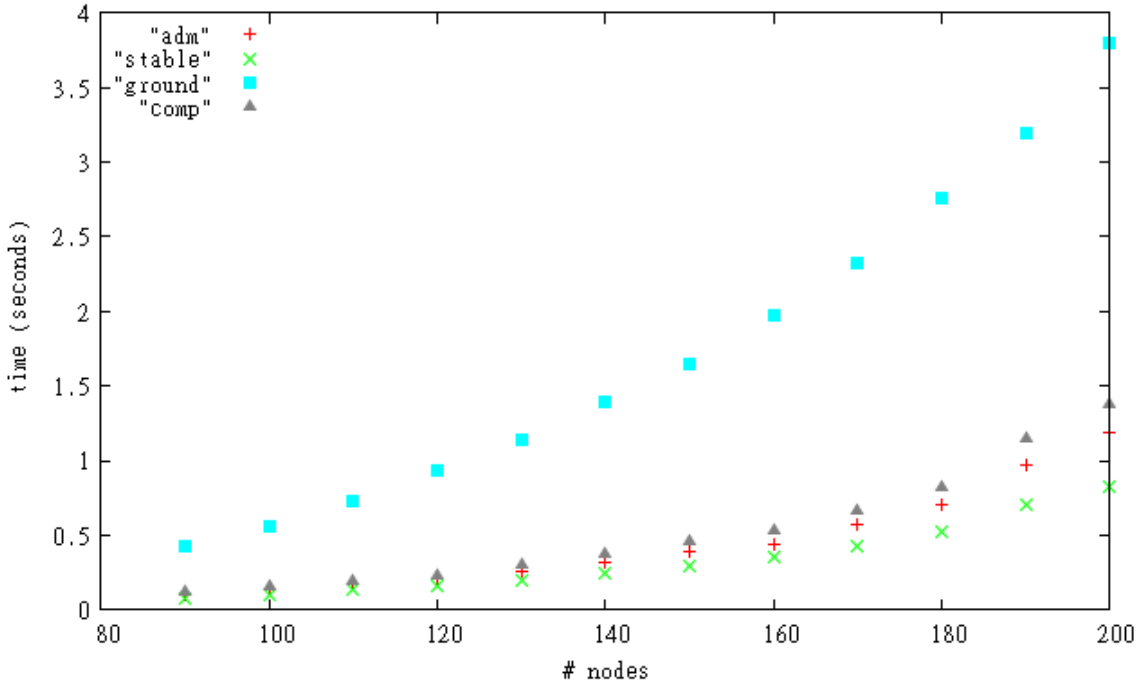
Edge density 10 %



Edge density 20 %



Edge density 30 %

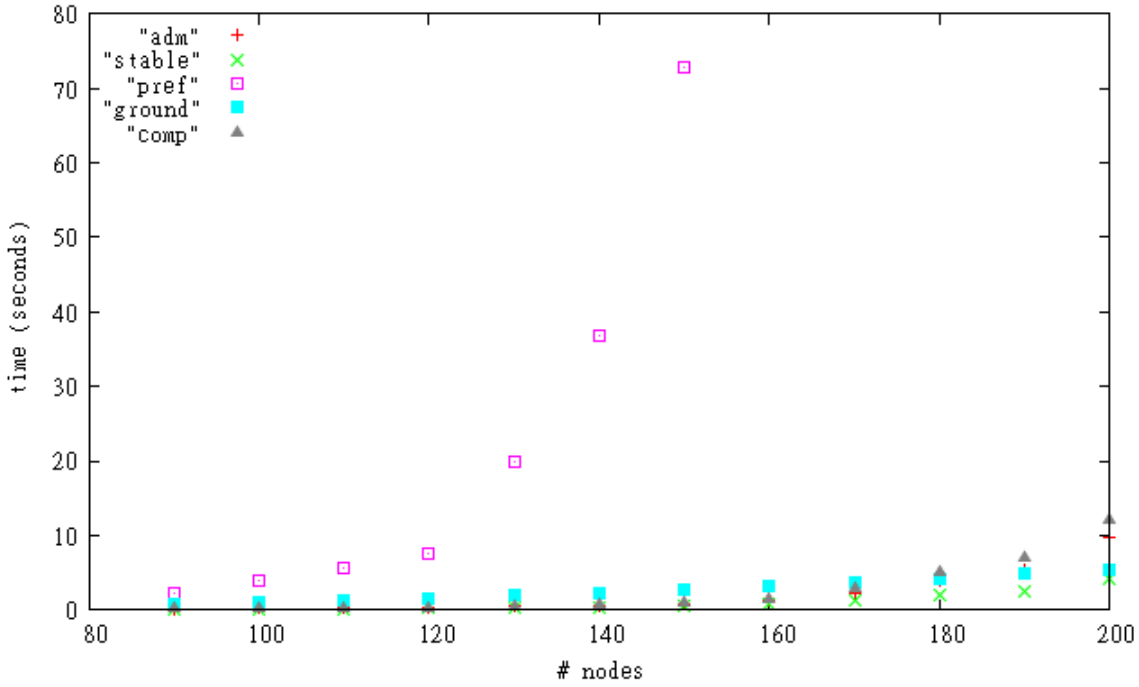


4.2.7 ClaspD

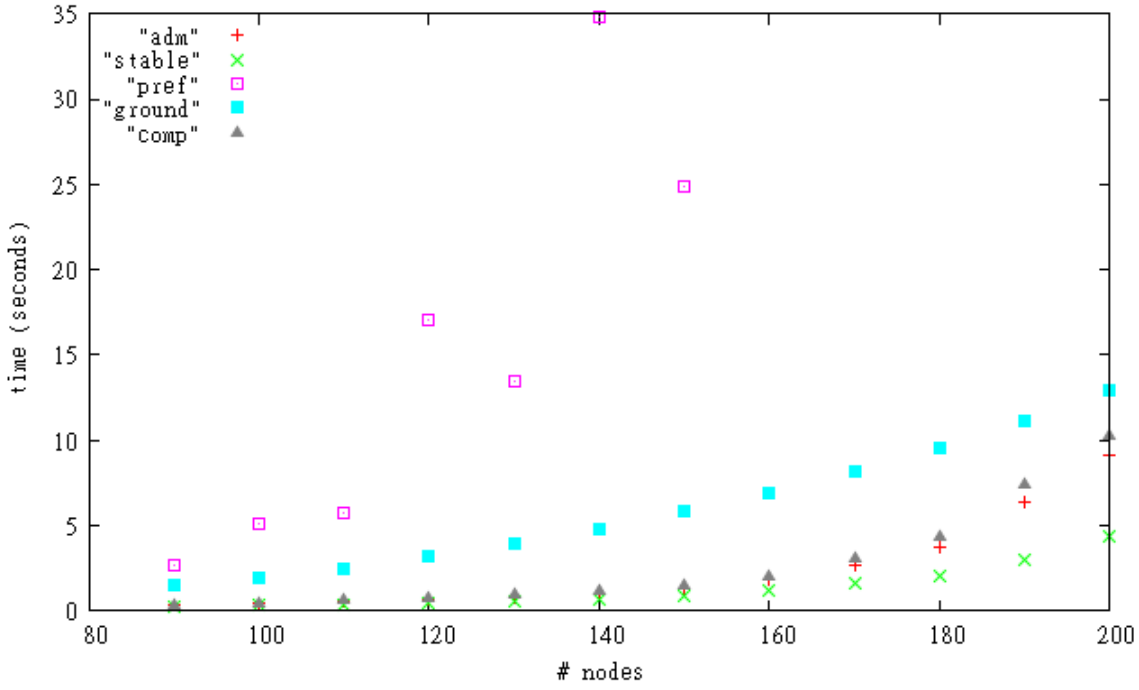
Lparse results

Claspd was the fastest solver for disjunctive encodings. For claspd we did not include the results for semi-stable due to discordant values which would make the plots in this section unreadable. However, the results for semi-stable are included in subsection 4.3. Like clasp and cmodels, claspd also performed better when using data grounded by gringo.

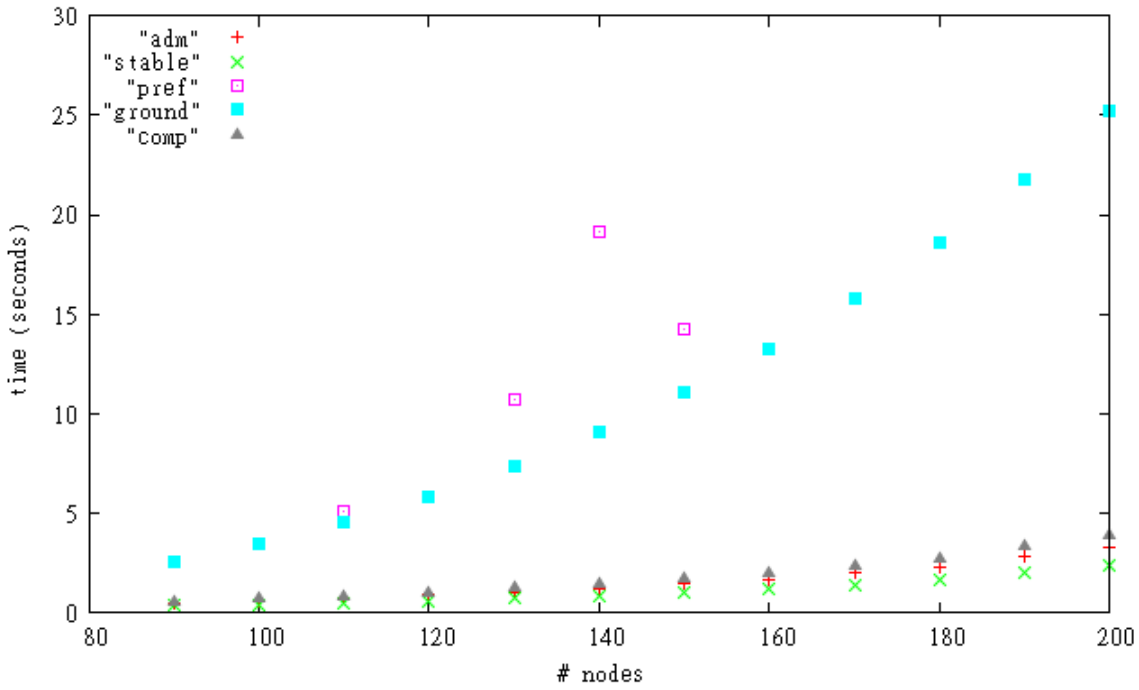
Edge density 10 %



Edge density 20 %

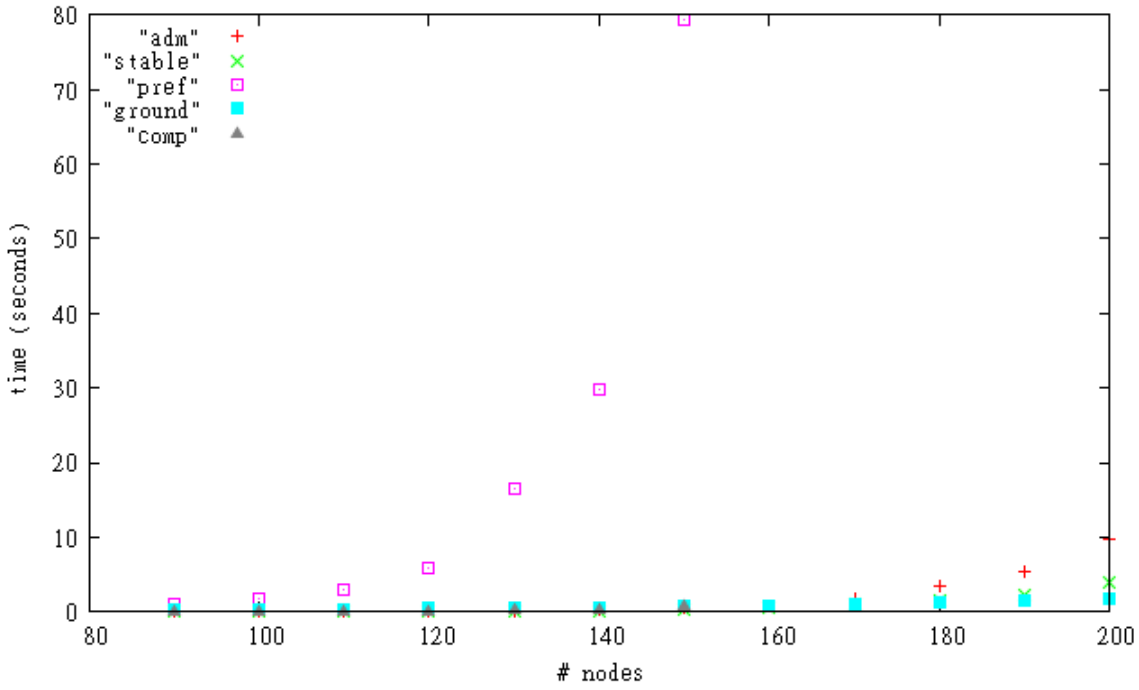


Edge density 30 %

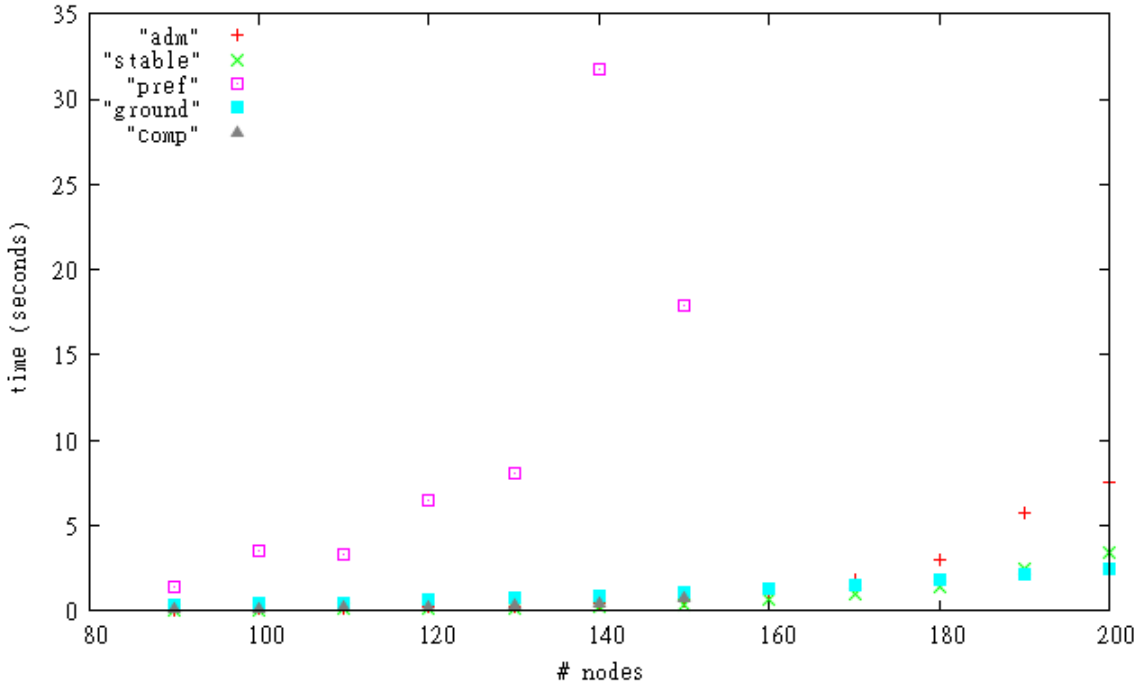


GrinGo results

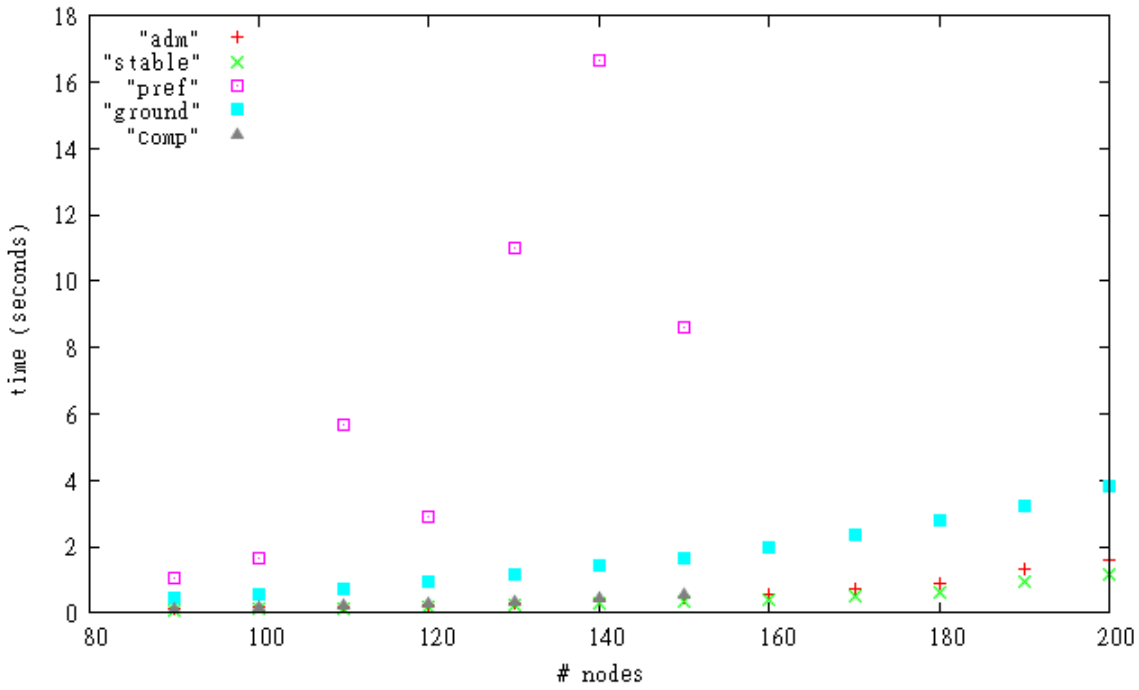
Edge density 10 %



Edge density 20 %



Edge density 30 %

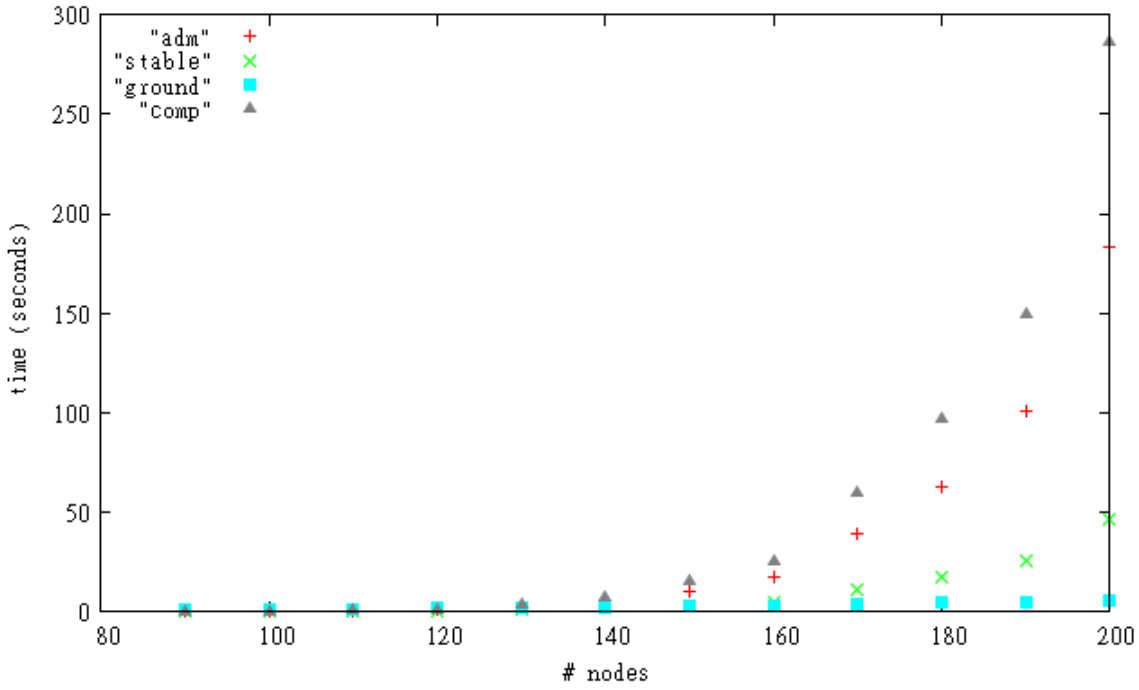


4.2.8 Gnt

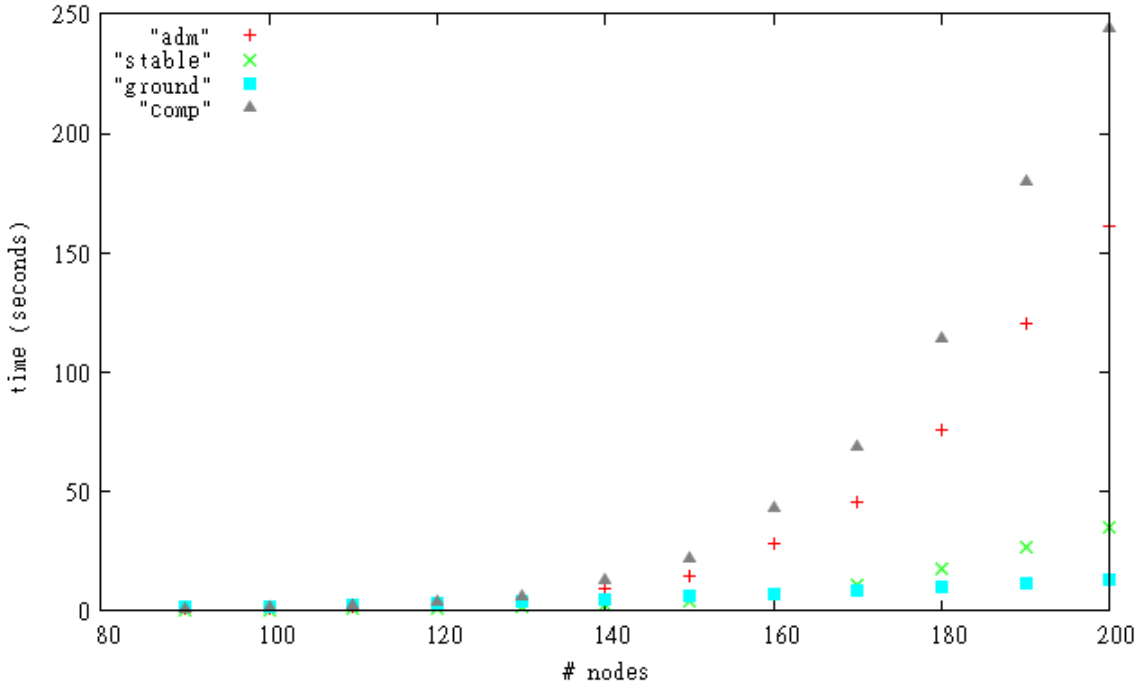
For some reason gnt2 failed on all extensions containing disjunctions (semi-stable and preferred). Gnt2 performed better with lparse.

Lparse results

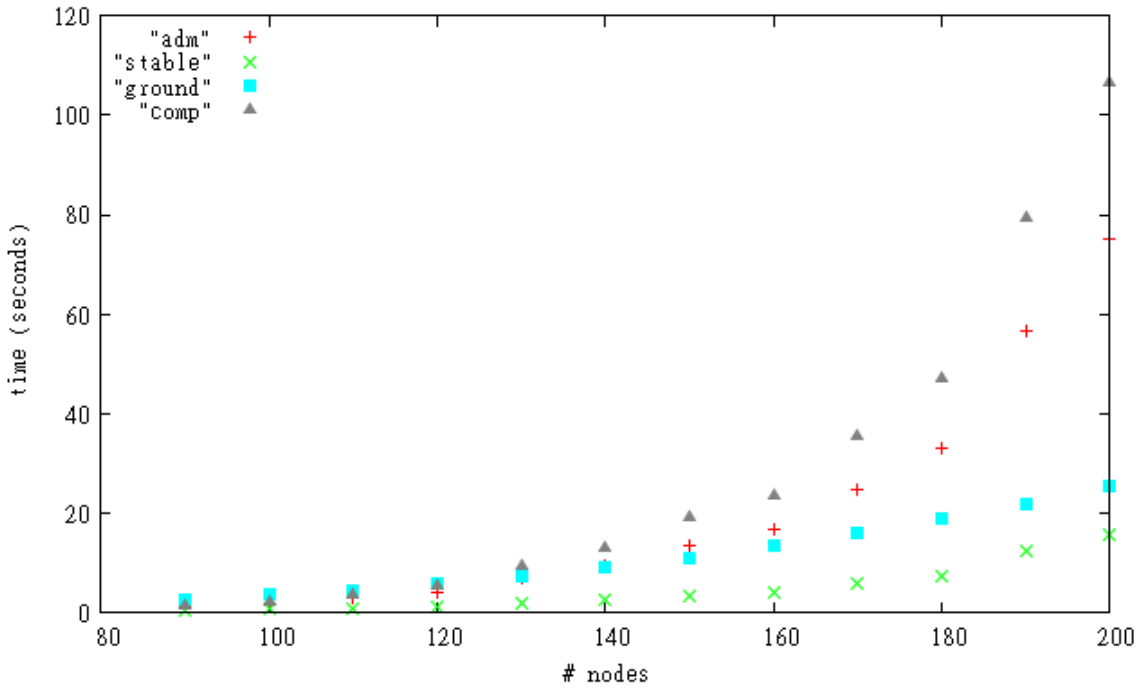
Edge density 10 %



Edge density 20 %

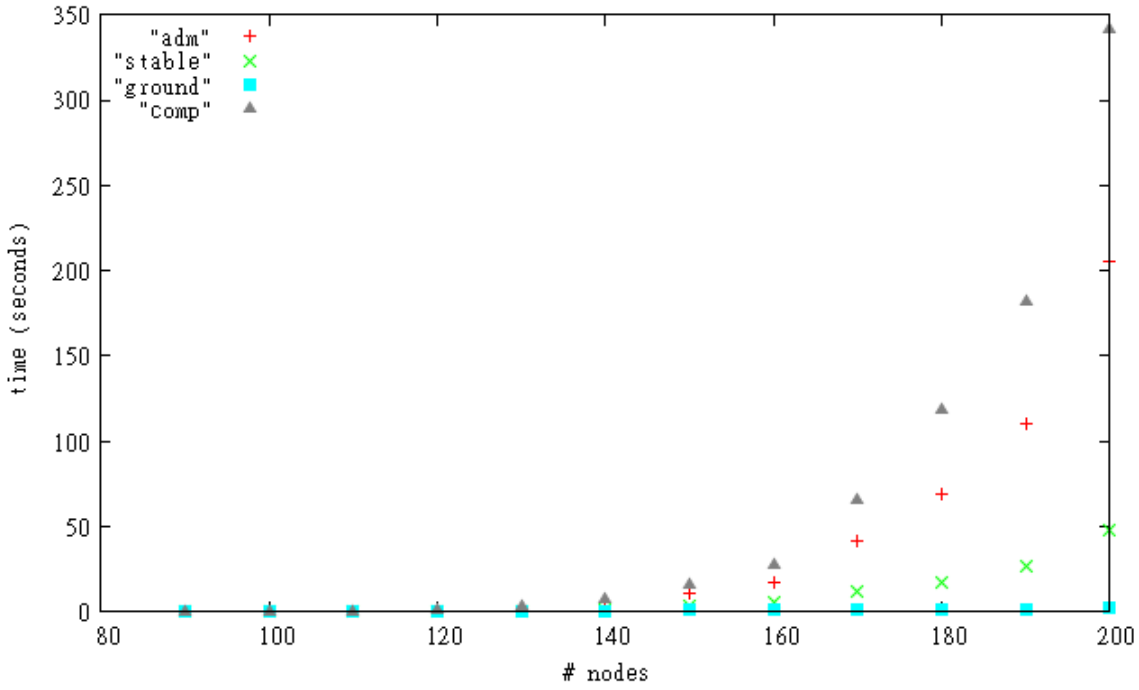


Edge density 30 %

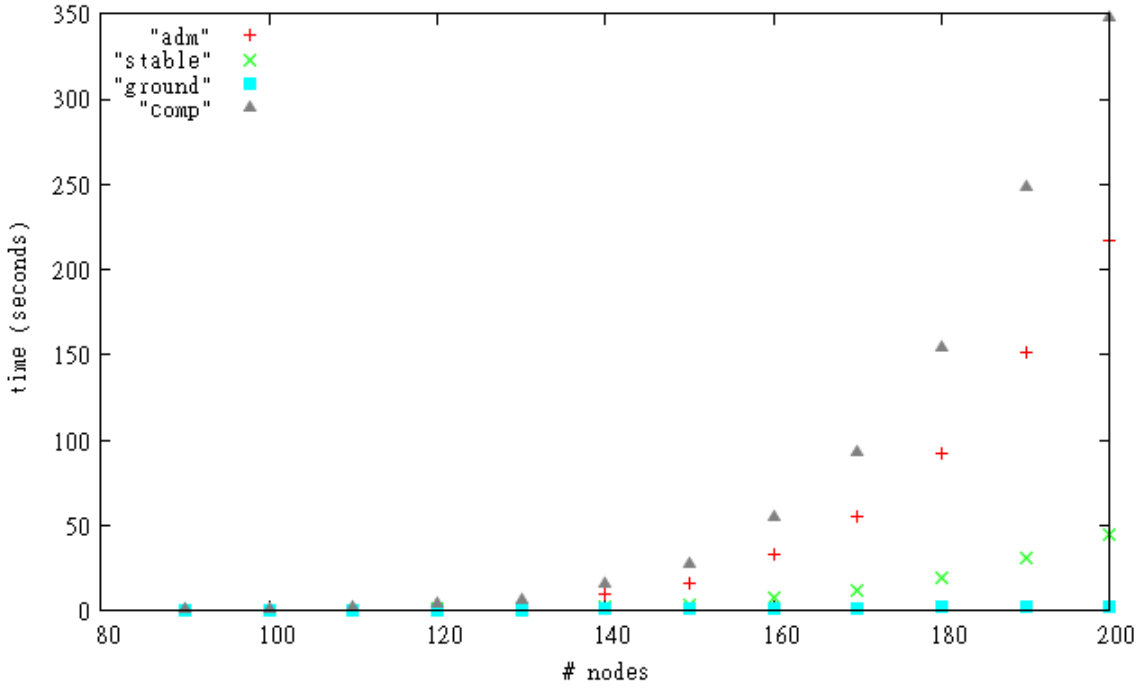


GrinGo results

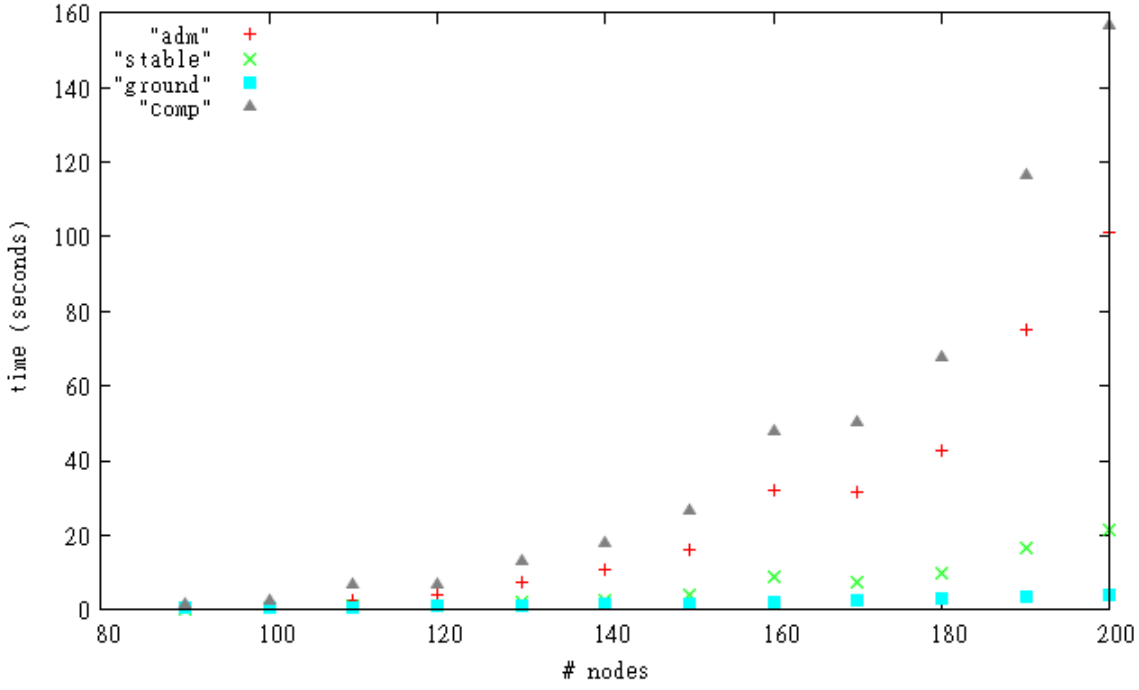
Edge density 10 %



Edge density 20 %



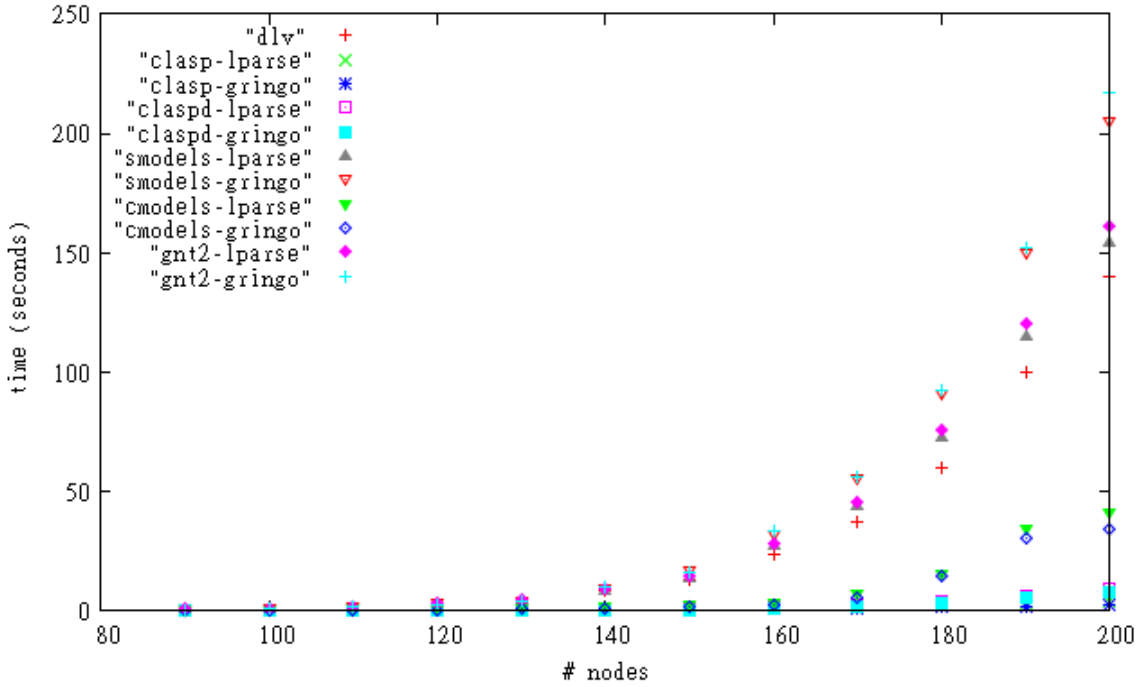
Edge density 30 %



4.3 Test results grouped by semantics

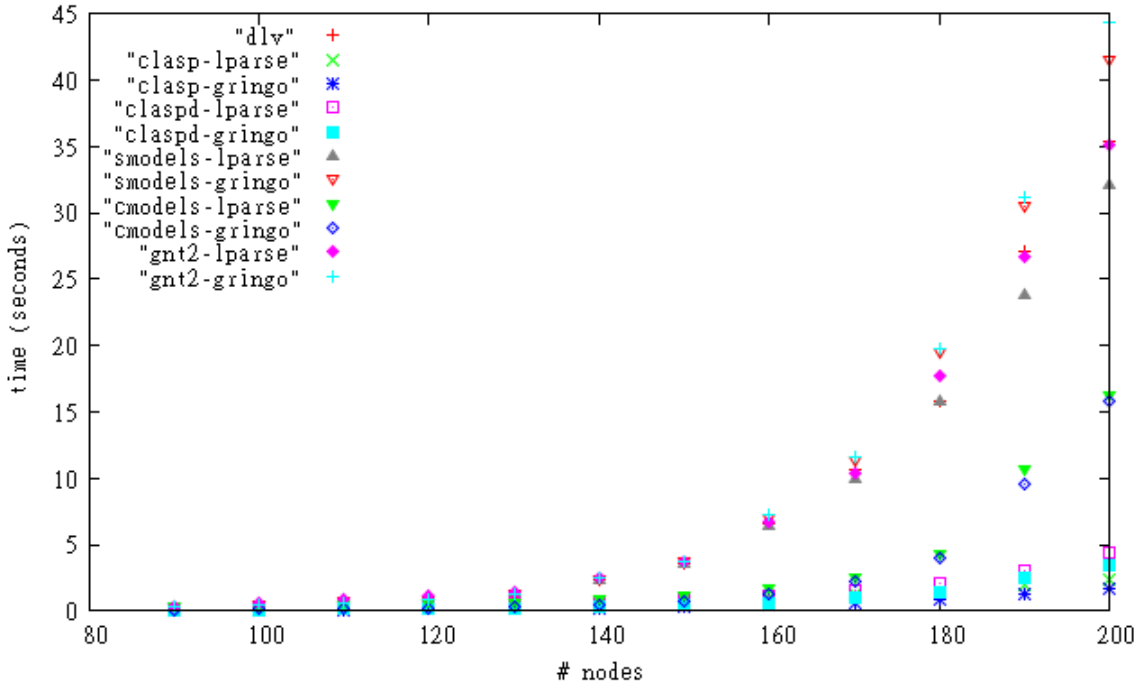
To get a better overview over the differences between the solvers and grounders we directly compare them with respect to every extension with an edge density of twenty percent next.

admissible



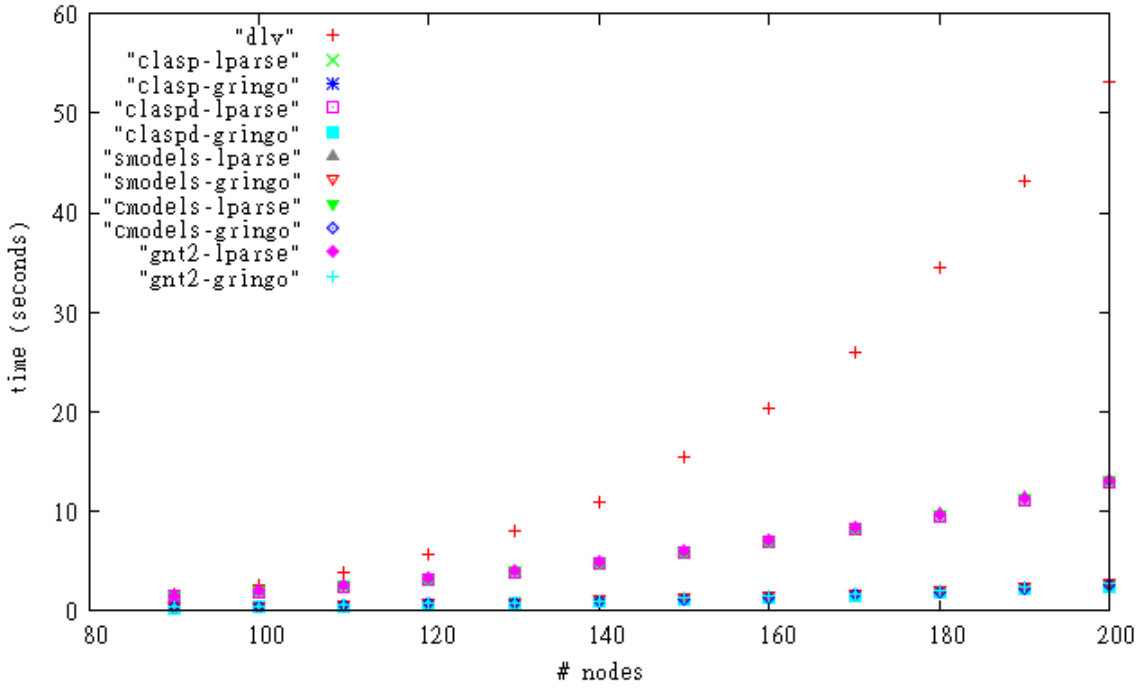
Here we can clearly see clasp coming out first, closely followed by claspd. Third is cmodels with a small gap. Each of them is faster with the grounder gringo than with lparse. Dlv is fourth closely followed by smodels and gnt2 (both with lparse). Last are smodels and gnt2 with gringo.

stable



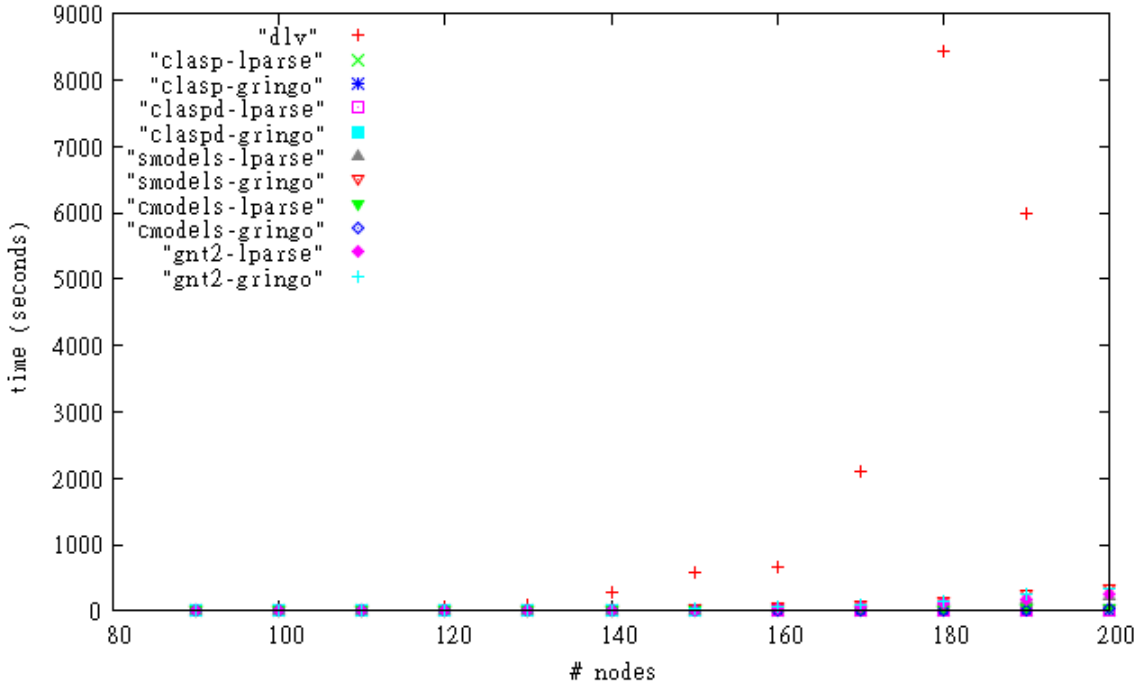
Here we see the same tendencies as in the admissible comparison. Except that dlv now is on par with smodels and gnt2.

grounded

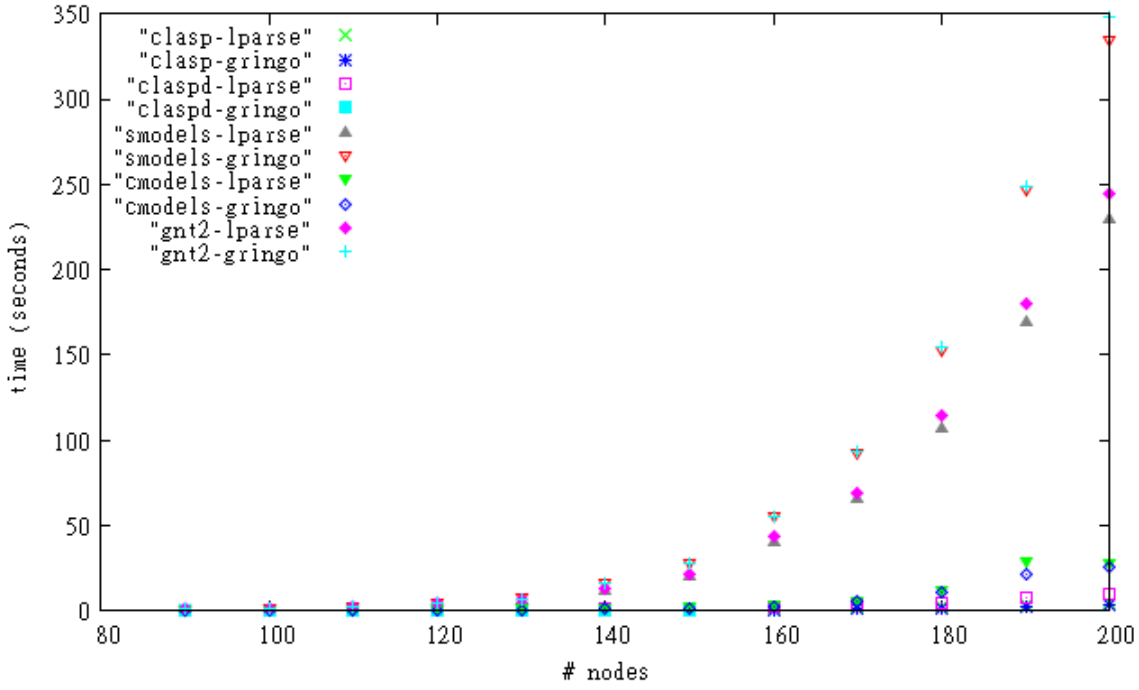


This image shows nicely how the performance of computing grounded extensions relies on grounder performance. Solver times are almost neglectable. Thus all solvers using gringo come out first, followed by the same solvers with lparse. Last is dlv which might lead to the conclusion that dlv's grounding could need a bit of tweaking.

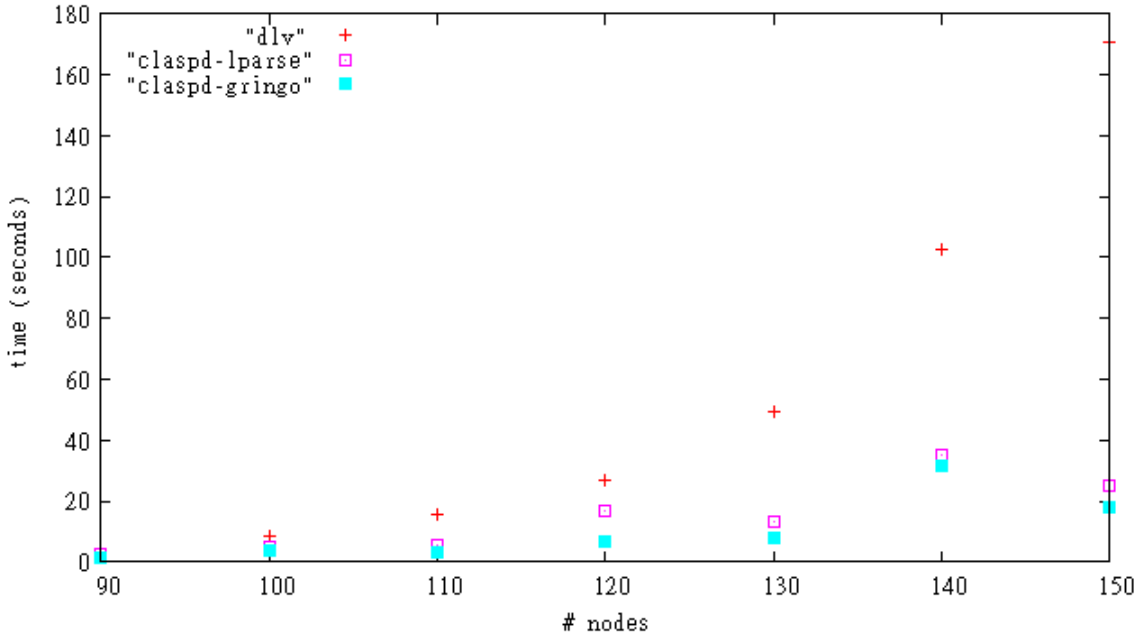
complete



This image shows the results for the complete extensions. Clearly we can see dlv having troubles with these. For that reason we also present an image with dlv excluded, and again we see similar results as with admissible and stable extensions:



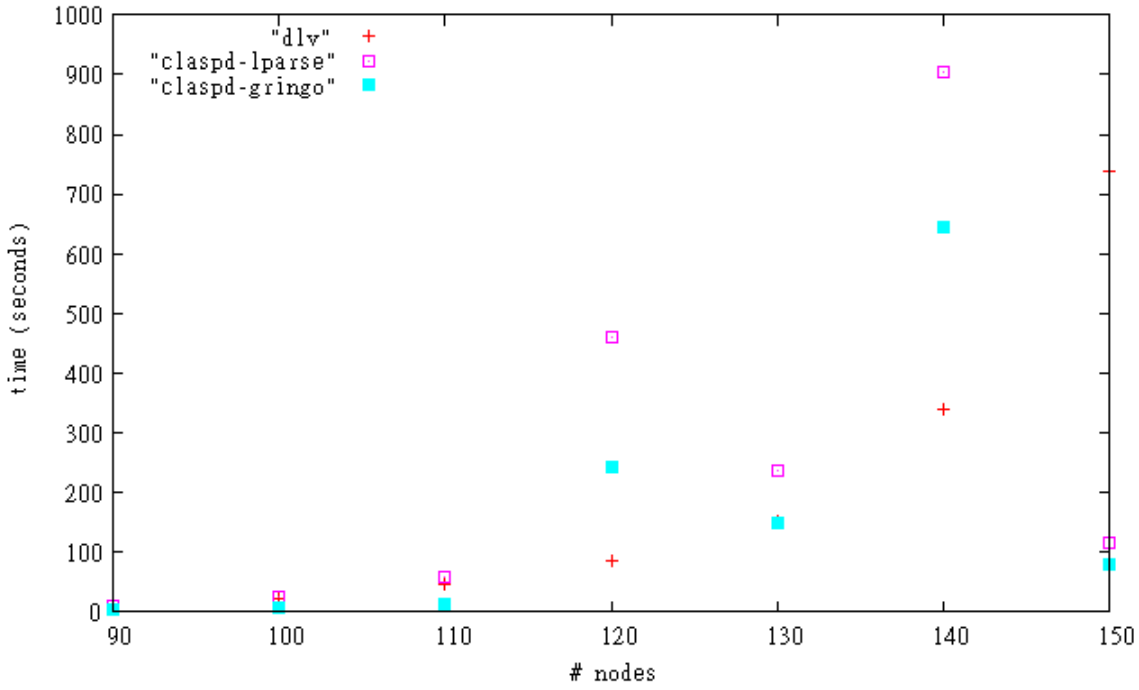
preferred



Since only claspd and dlv were able to solve preferred extensions reliably we exclude the results for cmodels (which failed on most with gringo and all with lparse) and gnt2

(which failed on all). For preferred extensions it is quite obvious, that claspd is the best choice:

semi-stable



With semi-stable we have a very similar picture as in preferred concerning the reliability of solvers. Only claspd and dlv were able to solve them again. This time, however, dlv has sometimes a slight edge over claspd since the computation times are far more stable than those of claspd. As always, claspd performs better with gringo than with lparse.

5 Conclusion

All grounders performed perfect on all samples. When it comes to isolated grounding performance, gringo has clearly an edge over lparse. However, when looking at how solvers performed with the output of these grounders there is a notable difference. Clasp, claspd and cmodels performed better with gringo, while smodels and gnt2 performed better with lparse.

All tested solvers had no problems solving disjunction-free encodings (admissible, stable, grounded and complete) and it would be possible to even go further and test some of them with even higher amounts of nodes and edges. Overall clasp clearly outperformed all other contestants followed by claspd. Cmodels was somewhere in the mid of all contestants. Smodels, gnt2 and dlv finished last with the latter one having some remarkably high computation times for complete and grounded extensions.

When it comes to disjunctive encodings (semi-stable and preferred) only dlv and claspd were able to solve all (unless aborted due to long computation times) with the gringo/-claspd combination performing best. In a few semi-stable test samples claspd had some very long computation times resulting in very high average times (see image below). In those cases it might be useful to switch to dlv which performed very solid in all cases. Cmodels was not able to solve most of them and it always failed with lparse, however, when it did not fail, it performed well. Unfortunately, gnt2 was not able to solve any disjunctive extensions at all because it did not accept the disjunctive output by both lparse and gringo. This could be the result of the compilation problems mentioned earlier.

Another interesting observation was, that computation times in many cases decrease with increasing edge density.

References

[1] Iyad Rahwan and Guillermo R. Simari. *Argumentation in Artificial Intelligence*. Springer, 2009.

[2] Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. Answer-set programming encodings for argumentation frameworks. Technical report, Vienna University of Technology, 2010. `http://benner.dbai.tuwien.ac.at/research/project/argumentation/EglyGW08c.pdf` (old version).

[3] Martin Gebser, Lengning Liu, Gayathri Namasivayam, André Neumann, Torsten Schaub, and Miroslaw Truszczyński. The first answer set programming system competition. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of LNCS, pages 3–17. University of Potsdam, Springer, 2007. `http://asparagus.cs.uni-potsdam.de/contest/downloads/report.pdf`.

[4] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Miroslaw Truszczyński. The second answer set programming competition. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of LNCS, pages 637 – 654. University of Leuven, Springer, 2009. `http://dtai.cs.kuleuven.be/events/ASP-competition/paper.pdf`.

[5] Tommi Syrjänen. *Lparse 1.0 User's Manual*. Helsinki University of Technology, 2000. `http://www.tcs.hut.fi/Software/smodels/lparse.ps`.

[6] Robert Bihlmeyer, Wolfgang Faber, Giuseppe Ielpa, Vincenzino Lio, and Gerald Pfeifer. *DLV - User Manual*. Vienna University of Technology. `http://www.dbai.tuwien.ac.at/proj/dlv/man/`.

[7] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. In *ACM Transactions on Computational Logic (TOCL)*, volume 7, pages 499–562. ACM, 2006. http://www.mat.unical.it/leone/ai/materiale/files/TOCL%20-%20dlv.pdf.

[8] University of Potsdam. *Potsdam answer set solving collection.* http://potassco.sourceforge.net/.

[9] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo: A new grounder for answer set programming. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of LNCS, pages 266–271. University of Potsdam, Springer, 2007. http://www.cs.uni-potsdam.de/wv/pdfformat/gescth07a.pdf.

[10] Martin Gebser, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Sven Thiele. On the input language of asp grounder gringo. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of LNCS, pages 502–508. University of Potsdam, Springer, 2009. http://www.springerlink.com/content/2042223543142887/.

[11] Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: A system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning.* Helsinki University of Technology, arXiv.org, 2000. http://arxiv.org/abs/cs/0003033v1.

[12] Yuliya Lierler. Cmodels for tight disjunctive logic programs. In *Proceedings of 19th Workshop on (Constraint) Logic Programming W(C)LP*, pages 163–166. Ulmer Informatik, 2005. http://www.cs.utexas.edu/users/yuliya/papers/sysCmodelsWCLP.ps.

[13] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. *A User's Guide to gringo and clasp and clingo and iclingo.* University of Potsdam, 2008. http://potassco.sourceforge.net/.

[14] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of LNCS, pages 260–265. University of Potsdam, Springer, 2007. http://www.cs.uni-potsdam.de/wv/pdfformat/gekanesc07b.pdf.

[15] Christian Drescher, Martin Gebser, Torsten Grote, Benjamin Kaufmann, Arne König, Max Ostrowski, and Torsten Schaub. Conflict-driven disjunctive answer set solving. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (2008)*, pages 422–432, 2008. http://www.aaai.org/Papers/KR/2008/KR08-041.pdf.

[16] Tomi Janhunen and Ilkka Niemelä. Gnt - a solver for disjunctive logic programs. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, volume 2923/2003 of LNCS, pages 331–335. Helsinky University of Technology, Springer, 2004.

[17] *Linux User's Manual.* `http://www.kernel.org/doc/man-pages/online/pages/man1/time.1.html`.