September 27, 2015

Performance Tuning in Answer Set Programming

Matt Buddenhagen, *University of Nebraska at Omaha*
Yuliya Lierler, *University of Nebraska at Omaha*

Performance Tuning in Answer Set Programming*

Matt Buddenhagen[1] and Yuliya Lierler[2]

[1] University of Nebraska at Omaha mbuddenhagen@unomaha.edu
[2] University of Nebraska at Omaha ylierler@unomaha.edu

Abstract. Performance analysis and tuning are well established software engineering processes in the realm of imperative programming. This work is a step towards establishing the standards of performance analysis in the realm of answer set programming – a prominent constraint programming paradigm. We present and study the roles of human tuning and automatic configuration tools in this process. The case study takes place in the realm of a real-world answer set programming application that required several hundred lines of code. Experimental results suggest that human-tuning of the logic programming encoding and automatic tuning of the answer set solver are orthogonal (complementary) issues.

1 Introduction

Performance analysis, profiling, and tuning are well established software engineering processes in the realm of imperative programming. Performance analysis tools – profilers – collect and analyze memory usage, utilization of particular instructions, or frequency and duration of function calls. This information aids programmers in the performance optimization of code. Profilers for imperative programming languages have existed since the early 1970s, and the methodology of their design as well as their usage is well understood. The situation changes when we face constraint programming paradigms.

Answer set programming (ASP) [12,13] is a prominent representative of constraint programming. In ASP, the tools for processing problem specifications, or *encodings*, are called (answer set) *solvers*. The crucial difference between the imperative and constraint programming paradigms exemplified by ASP, is that, in the latter, the connection between the encoding and solver's execution is very subtle. Consequently, performance analysis methods that matured within imperative programming are not applicable to constraint programming. In addition, the following observations apply: (i) specified problems in constraint programming paradigms are often NP complete and commonly result in significant computational effort by solvers, (ii) there are typically a variety of ways to encode the same problem, (iii) solvers offer different heuristics, expose numerous parameters, and their running time is sensitive to the configuration used.

* We would like to thank Joshua Irvin, Marius Lindauer, Peter Schüller, Benjamin Susman, Miroslaw Truszczynski, and Victor Winter for valuable discussions related to this paper as well as anonymous reviewers for their comments.

In this work, we undertake a case study towards outlining methodology of performance analysis in constraint programming. The case study takes place in the realm of a real-world answer set programming application that required several hundred lines of code. To the best of our knowledge, this is the first effort of its kind. Earlier efforts include the work by Gebser et al. [5] and [3], who present a careful analysis of performance tuning for the *n-queens* and *ricochet robots* problems, respectively. These problems are typically modeled within a page in ASP. Parsing is one of the important tasks in natural language processing. Lierler and Schüller [11] developed an ASP-based natural language parser called ASPCCGTK. The focus of this work is the performance tuning process during the development of ASPCCGTK. The original design of the parser was based on the observation that the construction of a parse tree for a given English sentence can be seen as an instance of a planning problem. System ASPCCGTK version 0.1 (ASPCCGTK-0.1) and ASPCCGTK version 0.2 (ASPCCGTK-0.2) vary only in how specifications of the planning problem are stated, while the constraints of the problem remain the same. Yet, the performance of ASPCCGTK-0.1 and ASPCCGTK-0.2 differs significantly for longer sentences. The way from ASPCCGTK-0.1 to ASPCCGTK-0.2 comprised 20 encodings, and along that way, grounding size and solving time were the primary measures directing the changes in the encodings. Rewriting suggestions by Gebser et al. [5] guided the ASPCCGTK encodings tuning.

The goal of present paper is threefold. First, this is an effort to reconstruct and document the "20-encodings" way from ASPCCGTK-0.1 to ASPCCGTK-0.2. Second, by undertaking this effort we will make a solid step toward outlining a performance analysis methodology for constraint programming. Third, we study the question of how tuning solver parameters by means of automatic configuration tools [10] effects the performance of the studied encodings. The last question helps us understand the placement of such tools on the performance analysis map in constraint programming. Despite the fact that changing a solver's settings may substantially influence its performance, it is common to only consider the performance of a solver's default configuration. Yet, it is unclear whether the best performing encoding when using a solver's default configuration would remain the best with respect to a tuned solver configuration. Silverthorn et al. [14] performed a case study that estimated the effect of parameter tuning as well as portfolio solving approach exemplified by CLASPFOLIO [6] on performance of solvers in context of three applications. A part of the current study is a logical continuation of that effort. In summary, this paper provides experimental evidence to support the validity of a performance tuning approach that first relies on the default solver settings while browsing the encodings and second tunes the solver's parameters on the best encoding to gain a better performing solution.

The outline of the paper follows: We start with a review of basic answer set programming and modeling concepts. We then present the process of performance tuning undertaken in ASPCCGTK. We review automatic configuration and present the details of the experimental analysis performed. Last, we provide the conclusions based on the experimental and analytic findings of this work.

2 Answer Set Programming and Modeling Guidelines

Answer set programming [12,13] is a declarative programming formalism based on the answer set semantics of logic programs [8]. The concept of ASP is to first represent a given problem by a program whose answer sets correspond to solutions. Second, a solver is used to generate answer sets for this program. Unlike imperative programming, where programs specify how to find a solution from given inputs, an ASP program encodes a specification of the problem itself. The ASP system comprises two tools: grounder and solver. In this work we use solver CLASP[3] [7] and its front-end grounder GRINGO [4].

Atoms and rules are basic elements of the ASP language, and a typical logic programming rule has the form of a Prolog rule. For instance, the program

$$p.$$
$$q \leftarrow p,\ not\ r.$$

is composed of such rules. This program has one answer set $\{p, q\}$. In a rule, the right hand side of an arrow is called the *body* of a rule, the left hand side is called the *head*. A rule whose body is empty is called a *fact*. The first rule of the program above is a fact. Intuitively, facts are always part of any program's answer set. In addition to Prolog rules, GRINGO also accepts rules of other kinds – "choices", "constraints" and "aggregates". For example, rule

$$\{p, q, r\}.$$

is a choice rule. Answer sets of this one-rule program are arbitrary subsets of the atoms p, q, r. A *constraint* is a rule with an empty head that encodes a condition on answer sets. For instance, the constraint $\leftarrow p,\ not\ q.$ eliminates answer sets that include p and do not include q.

The grounder GRINGO allows the user to specify large programs in a compact way, using rules with schematic variables and other abbreviations. GRINGO takes a program "with abbreviations" as an input and produces its propositional (ground) counterpart by using an "intelligent instantiation" procedure to produce propositional program that preserves the answer sets of original program. The program is then processed by the solver CLASP, which finds its answer sets. The inference mechanism of CLASP is related to propositional satisfiability (SAT) solvers [7].

We do not expect the reader to be familiar with the concept of an answer set. For the purpose of this paper, it is sufficient to know that answer sets are special ground atom subsets of the given logic program.

A common ASP practice is to devise a generic problem encoding that can be coupled with a specific problem instance to produce a solution. A problem instance typically consists of facts built from atoms of a particular predicate signature that we call an *input* signature. Dedicated predicate symbols in a generic encoding are meant to encode the solution, and we call the set composed

[3] `http://potassco.sourceforge.net/`.

of these predicate symbols an *output* signature. Sometimes it is important to distinguish between logic programs that encode problem specifications and those that encode a problem instance. In these cases, we refer to the former as *e-programs* and the latter as *i-programs*. To illustrate these ASP concepts, consider sample *graph coloring* problem:

> *A* 3*-coloring of a graph is a labeling of its vertexes with at most* 3 *colors such that no two vertexes sharing the same edge have the same color.*

An ASP e-program

$$
\begin{array}{r|l}
\Pi_{color} & color(1). \quad color(2). \quad color(3). \\
& \{c(V,I)\} \leftarrow vtx(V),\ color(I). \\
& \leftarrow c(V,I),\ c(V,J),\ I < J,\ vtx(V),\ color(I),\ color(J). \\
& \leftarrow c(V,I),\ c(W,I),\ vtx(V),\ vtx(W),\ color(I),\ edge(V,W). \\
& \leftarrow not\ c(V,1),\ not\ c(V,2),\ not\ c(V,3), vtx(V).
\end{array}
$$

encodes a generic solution to this problem. The first three facts of the encoding specify that there are three distinct colors: 1, 2 and 3. A choice rule in line two states that each vertex V may be assigned some colors. The third line says it is impossible for a vertex to be assigned two colors. The fourth line says that two adjacent vertexes may not be assigned the same color. The last line states that every vertex must be assigned a color. Predicate signature $\{c\}$ is an output signature of program Π_{color}. Predicate signature $\{edge, vtx\}$ is an input signature so that an i-program has the following form for a given graph (V, E)

$$
\begin{array}{ll}
vtx(v). & (v \in V) \\
edge(v,w). & (\{v,w\} \in E)
\end{array}
$$

The union of any problem instance and program Π_{color} will result in a program whose answer sets encode 3-coloring of a graph.

Gebser et al. [5] outline the "hints on modeling" in ASP that follow:

1. Keep the grounding compact:
   (i) If possible, use aggregates; (ii) Try to avoid combinatorial blow-up; (iii) Project out unused variables; (iv) But don't remove too many inferences!
2. Add additional constraints to prune the search space:
   (i) Consider special cases; (ii) Break symmetries; (iii) Test whether the additional constraints really help
3. Try different approaches to model the problem
4. It (still) helps to know the systems:
   (i) GRINGO offers options to trace the grounding process; (ii) CLASP offers many options to configure the search

To the best of our knowledge, this is the prime account of guidelines for performance tuning in ASP. We call this list *Performance Guidelines*.

3 ASPCCGTK and Human-driven ASP Performance Tuning

Lierler and Schüller [11] describe parts of the ASP-based natural language parser ASPCCGTK encoding. The ASPCCGTK website – `http://www.kr.tuwien.ac.at/`

`staff/former_staff/ps/aspccgtk/` – contains the complete application code. Versions ASPCCGTK-0.1 and ASPCCGTK-0.2 differ only in how specifications of the parsing task are stated, but the difference in performance of these encodings is significant. The way from ASPCCGTK-0.1 to ASPCCGTK-0.2 is comprised of 20 manually generated versions. The Performance Guidelines items 1 and 2 guided the way in considering the various encodings.

We now enumerate the program rewriting techniques that were used to tune ASPCCGTK. We start by introducing a concept of "output-equivalent" programs, which provides an important semantic property to capture a broad class of useful rewriting techniques. We conjecture that most of the ASPCCGTK encodings are output-equivalent. We believe that a future study of output-equivalent rewriting techniques will allow the rewriting-based tuning process (stemming from items 1 and 2 of Performance Guidelines) to be automated to a large extent. We conclude this section by presenting the historical ASPCCGTK encoding tree and the details of the tuning methodology used in the process. The encoding tree presents the details on the evolution of the ASPCCGTK.

Programs Π_1 and Π_2 are called *strongly equivalent* if for any program Π, answer sets of $\Pi \cup \Pi_1$ and $\Pi \cup \Pi_2$ coincide [2]. Strong equivalence was introduced to formalize the semantic properties of techniques that could be used in optimizing ASP code. In practical settings, the concept of strong equivalence is rather restrictive. For example, transformations on programs often involve changing the predicate signature, and strong equivalence is inadequate to capture such transformations.

We introduce the notion of "output-equivalent" programs to cope with the shortcomings of strong equivalence. Given a logic program Π, by $i(\Pi)$ and $o(\Pi)$ we denote their input and output signatures respectively. For a set X of atoms and a set of predicate symbols P, by $X_{|P}$ we denote the subset of X that contains all atoms in X whose predicate symbol is in P. For instances, $\{q(a,b), p(a), p(b), r(X)\}_{\{r\}} = \{r(X)\}$. We say that e-programs Π and Π' are output-equivalent if (i) their input and output signatures coincide and (ii) for any i-program I in their input signature, any answer set X of $I \cup \Pi$ is such that there is an answer set X' of $I \cup \Pi'$ and $X_{|o(\Pi)} = X'_{|o(\Pi)}$, and vice versa. In other words, both e-programs "agree" on the atoms in the output signature with respect to the same input. Output-equivalence relates to *uniform* equivalence [2].

We now present the ASP "code-change" classification that is then used to construct the ASPCCGTK encoding tree. In ASPCCGTK tree, each transition is marked by the kind of rewrite applied to the parent encoding. We conjecture that all rewriting techniques but one, called "output signature change", result in output-equivalent programs. It is a direction of future work to generally describe the presented rewriting techniques and formally claim that such rewritings are output-equivalence preserving.

Concretion ($\mathcal{C}$) replaces overly general rules by their effectively used, partial instantiations. For example, consider e-program

$$\begin{array}{l} q(X,Y) \leftarrow p(X),\ p(Y) \\ u(X) \leftarrow q(X,X), \end{array} \tag{1}$$

whose input signature is $\{p\}$ and output signature is $\{u\}$. Using concretion on (1) will result in program

$$q(X,X) \leftarrow p(X),\ p(X).$$
$$u(X) \leftarrow q(X,X).$$

The latter program will normally result in a smaller grounding.
Projection[4] ($\mathcal{P}$) reduces the number of schematic variables in a rule so that a fewer number of ground instances is produced. Consider e-program

$$u(X) \leftarrow p(X,V),\ q(X,Y,Z,0), r(Z,W), \tag{2}$$

whose input signature is $\{p,q,r\}$ and output signature is $\{u\}$. One way to apply projection to this program results in

$$\begin{array}{l} u(X) \leftarrow p(X,W),\ q\_new(X,Z), r(Z,W). \\ q\_new(X,Z) \leftarrow q(X,Y,Z,0). \end{array} \tag{3}$$

Simplification ($\mathcal{S}$) The idea of this technique is to reduce the number of rules, particularly constraints, by eliminating the rules that are "entailed" by the rest of a program. For instance, consider e-program

$$\begin{array}{l} \{u(X)\} \leftarrow p(X). \\ \{v(X)\} \leftarrow q(X). \\ \leftarrow p(X),\ q(X). \\ \leftarrow u(X),\ v(X), \end{array}$$

whose input signature is $\{p,q\}$ and output signature is $\{u,v\}$. By simplification we may eliminate the last rule of this program.
Equivalence ($\mathcal{E}$) replaces some rules of the program by strongly equivalent rules. For instance, a program

$$\begin{array}{l} \{u(X,Y)\} \leftarrow p(X),\ q(Y) \\ \leftarrow u(X,Y),\ u(X,Y'),\ Y \neq Y' \end{array}$$

is strongly equivalent to program $\{u(X,Y) : q(Y)\}1 \leftarrow p(X)$.
Auxiliary Signature Reduction ($\mathcal{A}$) reduces the program's signature by reformulating problem specifications by means of fewer predicates. For instance, reformulating program (3) as (2) will give us such effect.
Output Signature Change ($\mathcal{O}$) changes the output signature of a program to allow different sets of predicates to encode the solution.

Figure 1 presents the relations between the 20 encodings considered on the way from ASPCCGTK-0.1 to ASPCCGTK-0.2. Each node in this tree represents an ASPCCGTK encoding and is annotated by five numbers. The first number is the encoding id and the others are discussed later in the section along with the tuning methodology used to transition from one encoding to another. An arrow in the tree suggests that an encoding of a "child" node is a modification

[4] Terms Concretion and Projection were coined by Gebser et al. [5].

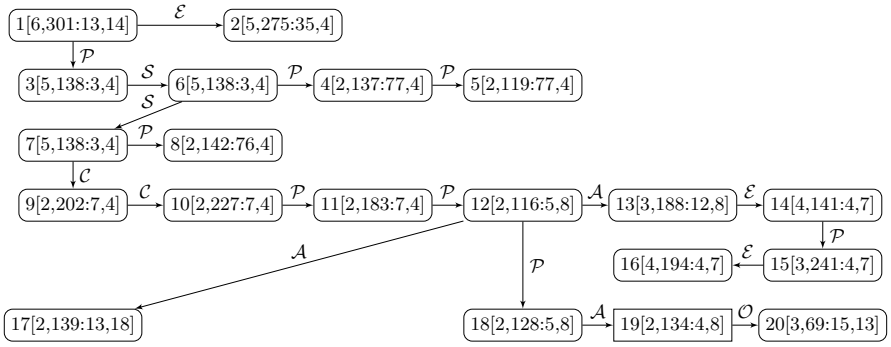

Fig. 1. ASPCCGTK Encodings Tree.

of its "parent" node encoding. For instance, encodings 2 and 3 are both modifications of encoding 1. Each arrow is annotated by a tag corresponding to the technique used to obtain the new encoding. We followed the practice of making the smallest possible change per revision. For example, when technique $\mathcal{A}$ was used then no more than one auxiliary predicate was eliminated from the encoding. ASPCCGTK-0.1 comprises encoding 1. Encoding 19 was identified as the "winner" and is the designated encoding ASPCCGTK-0.2.

A set of 30 problem instances, randomly selected from the Penn Treebank[5], was used to benchmark each ASPCCGTK encoding. Following parameters were used to evaluate the quality of each encoding: (i) number of time or memory outs (3000 sec. timeout), (ii) average ground size, (iii) average solving time (default configuration of CLASP v 2.0.2), (iv) average grounding time (default configuration of GRINGO). In Figure 1, each encoding id is annotated by four numbers [o,s:g,z], where o is the total number of timeouts/memory outs, s is the average solving time (in seconds; on instances that did not timeout/memoryout), g is the average grounding time (in seconds; on instances that did not timeout/memoryout)), and z and 10^5 are factors relating to the average number of ground rules reported by CLASP. The last number provides the relative size of ground instances produced by GRINGO. These numbers were obtained in experiments using a Xeon X5355 @ 2.66GHz CPU.

The rules of thumb used in evaluating which encoding is better follow:

1. if number of time or memory outs of encoding E exceeds these of encoding E' then E' is a better encoding, otherwise
2. if cumulative average grounding and solving time of E exceeds that of E' then E' is a better encoding, otherwise
3. if grounding size of E exceeds that of E' then E' is a better encoding.

These rules were followed "softly" during the tuning process. For instance, encoding 19 is deemed to be the best, based on solver performance, even though the rules above suggest that 12 is the better encoding.

[5] `http://www.cis.upenn.edu/~treebank/`

4 Automatic Algorithm Configuration and Tuning

Performance of answer set solvers greatly depends on their parameters-settings. In automatic algorithm configuration, the tuner evaluates the various parameter settings of the system in question and suggests an optimized configuration. Formally, the algorithm configuration problem can be formulated as follows: given a parametrized (target) algorithm $\mathcal{A}$, a set of problem instances (inputs) I, and a cost metric c, find parameter settings of $\mathcal{A}$ that minimize c on I. A *parameter-setting* is a name-value pair (p, v), where p is a parameter name and v is a value. A *configuration* is a set of parameter-settings. By $\mathcal{A}(\mathcal{P}, I)$, we denote an execution of algorithm $\mathcal{A}$ on instance I given parameter-settings $\mathcal{P}$. The cost metric c is often the runtime required to solve a problem instance, yet other factors such as solution quality maybe included. Various tools for solving the algorithm configuration problem have been proposed in the literature. System SMAC[6] [9] is a representative of such tools, and is based on the sequential model-based algorithm configuration method. Other such systems include PARAMILS [10] (precursor of SMAC) and GGA [1].

The rules of thumb listed in the end of Section 3 intuitively make sense, but given the disjointness of problem specifications from solving technology, there is no reason to believe that these rules achieve the best result in practice. Lierler and Schüller [11] and Silverthorn et al. [14] report that after applying automatic configuration tool PARAMILS to CLASP on the best encoding, the tuned version of CLASP outperformed the default version by a factor of 5. This observation raises the question: if CLASP were tuned on each encoding, would we still find 19 to be the best performing encoding as we described in Section 3? This is the question that we analyze in the rest of this paper.

We start by using the automatic configuration system SMAC version 2.06.01 to tune CLASP for each ASPCCGTK encoding. SMAC is susceptible to over-tuning. To account for this possibility, SMAC accepts both a training set of instances and a validation set. Upon reaching the end of the user-specified training time limit, SMAC uses the learned parameterization to execute a solver with found parameters on each instance of the validation set, and reports slower of the two execution metrics (one on the training set and another on the validation set) as its final result. To make final comparison of the performance of tuned versions of CLASP versus its default settings, we used a so-called held-out set of instances.

To create our pool of problem instances for SMAC, we classified the Penn Treebank instances (sentences) by word count into five word intervals, and restricted our selections to sentences having between 6 and 25 words. Our choice of boundaries was based on the previous analysis of ASPCCGTK. The time spent by ASPCCGTK parsing sentences with less than 6 words was negligible while there was marked increase in the number of solver timeouts for sentences with more than 25 words. To ensure an even distribution across the instance classes, we randomly selected an equal number of sentences from each class when creating

[6] `http://www.cs.ubc.ca/labs/beta/Projects/SMAC/`

our three disjoint test sets: a held-out set of 60 instances, a training set of 300 instances, and a validation set of 100 instances.

We used SMAC with its default setting for all but four parameters, whose values and snippets follow:

- *deterministic* is set to True. This parameter governs whether or not the target algorithm $\mathcal{A}$ is treated as deterministic. When set to True, SMAC will never execute $\mathcal{A}(\mathcal{P}, I)$ twice for any configuration $\mathcal{P}$ and instance I.
- *cutoffTime* is set to 300 seconds. Thus CPU time limit is 300 seconds for an individual target algorithm run $\mathcal{A}(\mathcal{P}, I)$.
- *wallclock-limit* is set to 480000 seconds (5.56 days). It instructs SMAC to terminate after using up a given amount of wall-clock time.
- *run-obj* is set to RUNTIME. It specifies to SMAC that the objective type that we are optimizing for is runtime.

Each execution of SMAC is non-deterministic. To account for this, performing several parallel runs is recommended by its developers. For each encoding, we executed ten instances of the SMAC tuning process and chose the best-performing configuration. The ten instances were run in parallel on independent CPU cores of a local resource cluster.

When using SMAC, the target algorithm is typically executed by way of a wrapper application. At a minimum, the wrapper implements the SMAC interface contract and calls the target algorithm with the specified parameter set, but may include other useful features such as coordinating parallel executions of SMAC. For our experiment we utilized PICLASP 1.0[7], a Python-based, SMAC-compatible wrapper for CLASP, developed by Marius Lindauer. PICLASP is explicitly compatible with the CLASP 2.1.x series, and for our experiment we used CLASP-2.1.3. To execute SMAC against the target algorithm, the algorithm's configurable parameters and their domains must be specified in parameter configuration space file. Lindauer provides a parameter configuration file for CLASP 2.1.x in PICLASP distribution, which we were able to use without modification. We implemented a small modification to PICLASP that allowed the use of separate training and validation sets, and we also created a benchmarking tool based on the Lindauer's CLASP wrapper class.

Our benchmarking tool, BENCHER, uses the CLASP wrapper class to conveniently invoke CLASP for each member of a benchmark set. When appended to the BENCHER command line, a CLASP parameter string is passed through to the solver, providing an easy way to test SMAC resultant parameterizations. If no additional parameters are provided, the solver operates in default mode. The CLASP result for each instance and the average performance is output as a JSON file to facilitate additional analysis if desired. Our modified version of PICLASP, BENCHER, the twenty encodings of ASPCCGTK, and our three instance test sets, can be downloaded from the University of Nebraska at Omaha web server: `http://faculty.ist.unomaha.edu/ylierler/projects/smac-aspccg.zip`.

Automatic tuning was conducted on a high-performance cluster node, powered by dual, 6-core, Intel Xeon X5660 2.8GHz HT processors. Each CPU had

[7] `http://www.cs.uni-potsdam.de/piclasp/`

6 physical, hyper-threaded cores providing a total of 24 virtual cores. The node had a total of 256 GB memory and a 500 GB SAN partition allocated to the experiment. We had dedicated access to the node during our experiment and used a local resource management queue to execute parallel SMAC instances exclusively on the experimental node. For each of the twenty ASPCCGTK encodings we tested, we initiated a parallel execution of ten SMAC instances with each instance executing on a separate core, with 2GB of allocated memory. Each execution of the CLASP solver was allowed 300 seconds (5 minutes) of CPU time to complete, and executions exceeding 300 seconds were reported as Timeouts. This cutoff value was selected based on previous analysis of ASPCCGTK that showed the solver was typically able to complete in less than 300 seconds for sentences having 25 or fewer words. We selected a value that would allow adequate time for the solver to complete, but would not diminish too greatly the time SMAC spent probing the parameter space and formulating solutions.

The SMAC automatic configuration phase timeout was configured at 5.56 days. We chose this value based on preliminary executions of SMAC over increasing lengths of time and comparing the benchmark times of the resulting parameterizations. We chose encoding with id 8 for the initial trials because its default benchmark time was adjacent to the median default benchmark time. Initially, speedup was significant but degraded to marginal improvements over time in what approximated a logarithmic rate. We chose a time that was clearly within the region of diminishing returns to allow for variability in the encodings, and yield more consistent results. In practice, we spent 22 weeks to tune all of the ASPCCGTK encodings.

5 Experimental Results

Figure 2 graphs the default and auto-tuned solver execution times of each ASPCCGTK encoding on the held-out set. The **Default** series represents average runtime using the default CLASP parameter values, and the **SMAC** series times were achieved using the optimized parameter configurations yielded by SMAC. Recall that the runtime variations in the default scenario are attributable to human-tuning efforts. Figure 2 reveals an observable relationship between human-tuned performance and auto-tuned performance. The results suggest that the performance optimization rules of thumb applied along the way from ASPCCGTK-0.1 to ASPCCGTK-0.2 remain valid, and automatic configuration of the solver compliments the human efforts as opposed to nullifying or subsuming their effects.

We note that speedup in auto-tuning ranged from 1.53 to 5.40 and averaged 3.26. Generally speedup deviates around the average but remains relatively consistent except in extreme cases. The worst performing encoding resulted in the least speedup and three of the best performing encodings had above average speedup. Encoding 18 stands out as a significant outlier, having only the sixth best Default benchmark but the greatest speedup of 5.4.

Figures 3 and 4 present the results on the following inquiry. We reconsidered 30 problem instances that played the key role in human-tuning described in

Encoding ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Default	62.4	42.6	39.7	24.2	24.9	39.6	39.7	27.7	26.1	25.5	25.4	19.0	35.0	35.5	31.6	32.0	21.3	24.4	15.8	22.3
SMAC	40.9	12.2	11.1	8.6	6.9	13.9	12.6	11.2	7.7	7.2	7.1	4.8	10.9	14.3	11.6	13.4	6.7	4.5	4.1	5.6

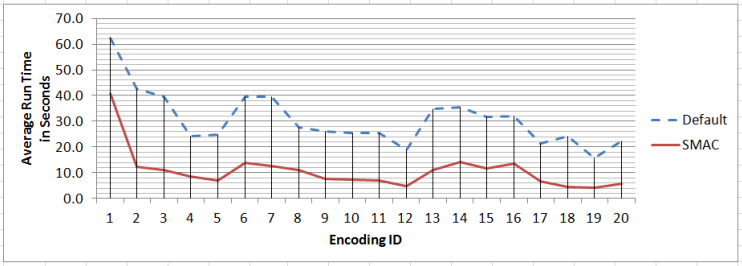

Fig. 2. Default and SMAC Benchmarks

Section 3. Recall that they were randomly selected without regard to the complexity of these instances, and substantially differ from the instances in held-out set. This set of instances includes two sentences of length 42 and 52 words; six and eleven sentences comprised of 30 and 20 words respectively; and eleven sentences that range between 9 and 19 words. We collected the following statistics on Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz using the 30 afore mentioned instances: (i) runtime with a default parameterization of CLASP-2.1.3, (ii) runtime with the parameterization of CLASP-2.1.3 reported best for the encoding in question, (iii) runtime of the parameterization of CLASP-2.1.3 reported best for encoding 1. The timeout was set at 3600 seconds.

Figure 3 presents average run times (that also include time spent on grounding) for instances that did not time or memory out on any of the encodings given any CLASP configuration. Figure 4 presents the cumulative number of time and memory outs. Row **Original** presents the data stemming from the original human-tuning process, repeating some of the information presented in Figure 1. Row **Rerun** presents the newly obtained numbers for the default parameterization of CLASP (note that the machine and CLASP version differ from **Original**). Row **SMAC** presents the data for the version of CLASP deemed to be best by SMAC for the respective encoding. Row **SMAC (Enc** 1**)** presents the data for the version of CLASP deemed to be best by SMAC for encoding 1. Presented data supports two major observations: (i) 30 random instances versus the instances of held-out set do not seem to change the outlook on which encoding is the "winner"; (ii) the parameter settings suggested by SMAC for the encoding 1 perform nearly as well as the encoding-specific SMAC parameterizations. The latter observation suggests that it is meaningful to use automatic configuration tuning early in the human-tuning process as a means to speed up the tuning process. It also makes sense to perform automatic configuration of parameters on the "winner", since the resulting solver optimization is presumably unique to the encoding in question.

Summary of Runtimes (Average for instances that never time out.)

Encoding ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Original	313.4	286.6	137.5	65.3	55.0	137.4	137.5	62.6	46.8	53.7	45.9	37.4	75.3	101.3	80.7	81.1	52.3	43.0	35.1	38.3
Rerun	251.8	111.2	64.6	32.7	32.2	63.3	63.4	36.4	32.9	28.8	35.8	26.0	41.5	112.1	64.3	41.4	42.9	28.0	18.6	35.6
SMAC	76.0	23.4	16.7	18.4	17.7	17.9	13.2	17.4	9.4	10.5	9.8	6.9	13.5	15.5	12.8	12.6	11.4	6.6	6.4	10.5
SMAC (Enc. 1)	74.4	31.6	17.7	19.8	18.9	17.7	17.7	20.4	10.3	10.1	10.1	8.0	32.4	29.0	35.0	28.1	12.1	8.5	7.6	12.0

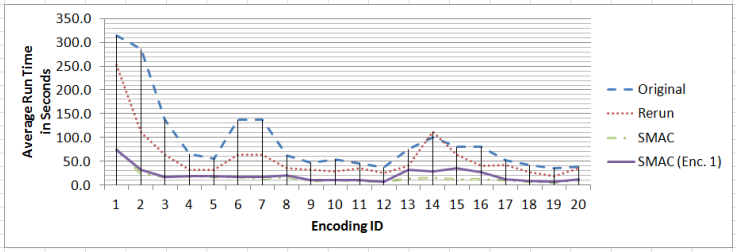

Fig. 3. Original Test Set Runtimes

Summary of Timeouts

Encoding ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Original	6	5	5	2	2	5	5	2	2	2	2	2	3	4	3	4	2	2	2	3
Rerun	5	5	5	5	2	5	5	3	2	2	2	3	2	2	3	2	3	2	2	2
SMAC	3	2	2	2	1	2	2	2	2	1	2	1	2	2	2	3	2	1	1	2
SMAC (Enc. 1)	3	2	2	2	2	2	2	2	2	2	1	2	2	2	2	2	2	2	3	1

Fig. 4. Original Test Set Timeouts

6 Conclusions

Returning to our three stated objectives, we satisfied the first one by reconstructing and documenting the human effort to optimize the ASPCCGTK parser described in Section 3. The benchmark results clearly illustrate the effects due to the progressive application of output-equivalent rewriting techniques along the way from ASPCCGTK 0.1 to ASPCCGTK 0.2. Secondly, by achieving our first objective, we have validated the principles of ASP performance tuning as suggested by Gebser et al. [5], and established such a methodology within the context of a real world application. We believe that this provides a concrete basis for future work and the development of generally applicable automated ASP code rewriting-optimization tools. Finally, our efforts help clarify the role of automatic configuration tools within the context of constraint programming and performance optimization. Our results lead us to conclude that human-tuning of the ASP implementation and automatic tuning of the solver appear to be orthogonal issues, with auto-tuning having a linear affect on performance. Further, code-based optimization principles seem to take precedence over automatic configuration.

References

1. Ansótegui, C., Sellmann, M., Tierney, K.: A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In: Proceedings of the CP'09. pp. 142–157 (2009)

2. Eiter, T., Fink, M.: Uniform equivalence of logic programs under the stable model semantics. In: Logic Programming, 19th International Conference, (ICLP). pp. 224–238 (2003), http://dx.doi.org/10.1007/978-3-540-24599-5_16
3. Gebser, M., Jost, H., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T., Schneider, M.: Ricochet robots: A transverse ASP benchmark. In: Logic Programming and Nonmonotonic Reasoning, 12th International Conference, (LPNMR). pp. 348–360 (2013), http://dx.doi.org/10.1007/978-3-642-40564-8_35
4. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to gringo, clasp, clingo, and iclingo. (2010), available at http://potassco.sourceforge.net
5. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Challenges in answer set solving. In: Balduccini, M., Son, T. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond, vol. 6565, pp. 74–90. Springer (2011)
6. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M., Ziller, S.: A portfolio solver for answer set programming: Preliminary report. In: Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11). pp. 352–357. Springer-Verlag (2011)
7. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Proceedings of 20th International Joint Conference on Artificial Intelligence (IJCAI'07). pp. 386–392. MIT Press (2007)
8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of International Logic Programming Conference and Symposium. pp. 1070–1080 (1988)
9. Hutter, F., Hoos, H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Proceedings of LION'11. pp. 507–523 (2011)
10. Hutter, F., Hoos, H., Leyton-Brown, K., Stützle, T.: ParamILS: An automatic algorithm configuration framework. Journal of Artificial Inteligence Research (JAIR) 36, 267–306 (2009)
11. Lierler, Y., Schüller, P.: Parsing combinatory categorial grammar with answer set programming: Preliminary report. In: Correct Reasoning: Essays on Logic-Based AI in Honor of Vladimir Lifschitz. Springer (2012)
12. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer Verlag (1999)
13. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)
14. Silverthorn, B., Lierler, Y., Schneider, M.: Surviving solver sensitivity: An asp practitioner's guide. In: International Conference on Logic Programming (ICLP) (2012), http://www.cs.utexas.edu/users/ai-lab/pub-view.php?PubID=127153