The Performance Optimization of ASP Solving Based on Encoding Rewriting and Encoding Selection

DISSERTATION

A dissertation submitted in partial
fulfillment of the requirements for
the degree of Doctor of Philosophy
in the College of Engineering at the
University of Kentucky

By
Liu Liu
Lexington, Kentucky

Director: Dr. Miroslaw Truszczynski, Professor of Computer Science
Lexington, Kentucky
2022

ABSTRACT OF DISSERTATION

The Performance Optimization of ASP Solving Based on Encoding Rewriting and Encoding Selection

Answer set programming (ASP) has long been used for modeling and solving hard search problems. These problems are modeled in ASP as encodings, a collection of rules that declaratively describe the logic of the problem without explicitly listing how to solve it. It is common that the same problem has several different but equivalent encodings in ASP. Experience shows that the performance of these ASP encodings may vary greatly from instance to instance when processed by current state-of-the-art ASP grounder/solver systems. In particular, it is rarely the case that one encoding outperforms all others. Moreover, running an ASP system on one encoding for a specific instance may "take forever," while running it on another encoding for this instance may yield a solution in a fraction of a second. The selection of a "good" encoding for each instance is crucial to the performance of ASP solving. In this thesis, I propose methods to improve the performance of ASP solving that exploit these observations. First, I designed and implemented methods that, given an encoding for a problem, rewrite it in several ways into new different but equivalent encodings. Second, I designed and implemented a system that given a set of input encodings of a problem, a set of problem instances, and an ASP grounder/solver system, automatically generates equivalent encodings and builds for each selected encoding its performance model. The model predicts for any instance the execution time that the grounder/solver system takes to process the instance under the corresponding encoding. These performance models are then used to improve solving efficiency: whenever a new instance arrives, the system selects the encoding predicted to perform the best on the instance and invokes the grounder/solver. The system also supports a scheduled execution and an interleaved execution of encodings, which are complementary to machine learning techniques. Third, I implemented algorithms that generate hard structured instances for several combinatorial problems I selected for our experimental study of the efficacy of the methods I developed. Hard instances can serve as the benchmark for evaluating the hardness of specific problems and contribute as training data to the platform I created to help build encoding selection models. The process can also provide meaningful insights into finding hard instances of other combinatorial problems.

KEYWORDS: ASP Solving, Encoding Rewriting, Encoding Selection, Machine Learning, Encoding Schedule, Interleaving Schedule, Hard Instances Generation

Liu Liu

September 7, 2022

The Performance Optimization of ASP Solving Based on Encoding Rewriting and Encoding Selection

By
Liu Liu

Dr. Miroslaw Truszczynski
Director of Dissertation

Dr. Simone Silvestri
Director of Graduate Studies

September 7, 2022
Date

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1 Introduction

The main goal of my research is to automate Answer Set Programming (ASP) solving performance optimization through encoding rewriting techniques, an encoding portfolio-based encoding selection platform, and algorithms for generating hard instances for selected combinatorial problems.

ASP [46, 49] is a declarative formalism for solving difficult search and optimization problems. ASP comes with a modeling language and program processing tools. The language of all common versions of ASP is loosely based on the syntax of Prolog [8]. A common core is specified by the ASP-2-Core standard [6, 9]. The language allows one to express constraints as *rules*. *Programs* in ASP, often called *encodings*, are sets of rules. In ASP, problems are modeled as *answer set encodings*(AS encodings, for short), with rules of these encodings representing constraints of the problem. Specific instances of the problem are modeled as collections of special rules called *facts*. To solve a problem for a particular instance, the encoding of the problem is expanded with the facts representing the instance and then processed by a special program called a *grounder*. The grounder simply produces another ASP encoding that has the same solutions, referred to in ASP as *answer sets*, but is easier to process. That encoding is then passed on to another program, called a *solver* that computes answer sets, or informs the user that non exists. These answer sets represent solutions to the problem (the absence of answer sets indicates the absence of solutions).

A search problem consists of finding elements in the search space of that problem that satisfy all constraints (requirements, conditions) of the problem. To take an example, the *Hamiltonian cycle* problem is to find a cycle in a given directed graph that visits each node exactly once. The search space consists of all subsets of the set of edges; solutions are those sets of edges that form a cycle visiting all vertices

exactly once. Formally, given a graph G, the set of edges E, the search space consists of subsets of E. To be a solution, a subset H of E must satisfy the following requirements: 1) For each node x of G, there is exactly one edge in H that starts in x; 2) For each node x of G, there is exactly one edge in H that ends in x; 3) All nodes x of G are reachable from each other through edges in H. The first two conditions describe collections of disjoint cycles covering all nodes of the input graph; the last condition guarantees that the collection consists of exactly one cycle.

Such search problems have direct and straightforward solutions in ASP. ASP takes as the first part of the program a graph G, called an instance in ASP, which represents the problem instance to be solved. Here, we want to find if a Hamiltonian cycle exists in a graph G. An encoding modeling the constraints related to the requirements is the second part of the program. For the Hamiltonian cycle problem above, each requirement can be accomplished by one or two rules describing constraints (see Appendix 9.A). These two parts of the program are passed to ASP tools to compute answer sets. The answer sets represent solutions to the search problem. If there is an answer set, the resulting answer set above contains related edge information explaining how a Hamiltonian cycle is formed. On the other hand, if there is no answer set, it means the graph contains no Hamiltonian cycle.

The language of ASP supports default negation, recursive definitions, and aggregate operators. This rich functionality of the ASP language allows programmers to build intuitive and elegant representations of many classes of constraints appearing in natural language statements of search and optimization problems. Further, current ASP grounder/solver systems are highly optimized. They proved effective in solving several problems of practical importance. For example, ASP was applied to make diagnostic tasks for NASA shuttle [50], extract traveling information from text files to help find promising offers for customers in e-Tourism systems [44], cooperate multiple robots to clean a house [13], detect and correct syntactic and semantic

medical errors in Italian National Healthcare System [53], answer complex biomedical queries related to drug discovery over several biomedical knowledge ontologies and databases [14]. All this makes ASP a promising paradigm for modeling and solving hard computational problems.

However, some issues arise when one wants to use ASP efficiently. On the one hand, ASP tools come with tens or hundreds of parameters that affect the solving performance. Such tools always perform much better when they are fine-tuned with respect to a given instance set. Meanwhile, for most problems of interest, every tool, even if a fine-tuned one, at best performs better than other tools only on a fraction of all instances, its "area of excellence," outside of which other tools are more efficient. In other words, sets of tools typically show *complementary performance* or *performance diversity*. Different tools, or one tool with different configurations, show performance diversity where one excels in one area while another in the other area. A technique that always selects the best tool among tool candidates for each problem can be used to boost the performance of AS solving. On the other hand, an AS problem always admits several logically equivalent encodings. Similar to AS solving tools, performance diversity is always observed when these equivalent encodings are used to solve a set of instances with the same AS solving tool. Choosing the best encoding for each instance, which we call encoding selection, is also meaningful in terms of performance improvement in ASP. Different from AS solving tools, encoding selection is not discussed before in the literature and is the main focus of my research.

In this thesis, I will discuss some contributions in the area of equivalent encoding generation, encoding selection, and hard instance generation. In particular, I present my methods to generate logically equivalent encodings through encoding rewritings. The availability of several encodings offers a chance of finding groups showing complementary performance. I present methods to construct groups of encodings with complementary performance with respect to a given set of problem instances. I show

how to apply machine learning techniques to build for each encoding in such a group a model predicting for a given instance the running time of an ASP tool in hand when run on this encoding for this instance. These models form a foundation for encoding selection-based ASP solving in one of several forms: select the encoding predicted to perform the best, execute several encodings expected to perform well according to some fixed schedule, or execute several encodings expected to perform well in an interleaved fashion. I discuss a software system I built that automates this process. The system I developed requires the availability of sets of instances for problems. In the thesis, I also discuss a methodology I developed to generate hard instances to search problems. Finally, I discuss two case studies that demonstrate the potential of encoding selection by showing that my tools indeed result in performance improvements.

Chapter 2 Introduction to ASP

Before we continue, I briefly introduce the basic syntax of AS encodings. Later on, I will introduce some of its important extensions. AS encodings are sets of rules of the form

$$a \leftarrow b_1, \ \ldots, \ b_m, \ not\ c_1, \ \ldots, \ not\ c_n. \tag{2.1}$$

where a is called the *head* of the rule, $b_1, \ \ldots, \ b_m, \ not\ c_1, \ \ldots, \ not\ c_n$ is the *body* of the rule, and a, b_i, and $not\ c_i$ are called *literal*s. An informal reading of the rule is: if all b_i's as above are established to be true and none of c_i's is eventually true, then a must be true. The $\leftarrow$ is replaced by `:-` when we present program listings.

To explain the definition of literals, I start with the *signature* Σ of the language of ASP. A signature Σ consists of two sets O and P, the set of constants and predicate symbols, respectively. In addition to elements of the signature, ASP language uses variable symbols to represent constants in the set O. We denote the set of variable symbols by V. Constants and variables are *terms*. We write T for the set of terms of the language.

Atoms are expressions $p(t_1, \ldots, t_n)$, where p is a predicate symbol from P and t_i's are terms from T. If all the terms in an atom $p(t_1, ...t_n)$ are ground terms, the atom is a *ground atom*. For example, the atom $reach(2, 4)$ (node 2 is reachable from node 4) is a ground atom.

Expressions a and $not\ a$, where a is an atom, are *literals*. If a is ground, the corresponding literal is a ground literal, meaning a ground atom and its negation are both ground literals.

The *not* in the rule (2.1) is called *default negation* and it is different from classical negation. Informally, the expression $not\ A$ means that the program does not justify A. This does not mean that A is false. Thus, *not* and the classical negation $\neg$ are

different.

To illustrate the difference informally, consider a program consisting of just one rule

$$a :- \mathit{not}\ c.$$

The program does not justify c (there is no way to derive c from the program, as no rule has c in its head). Thus, *not* c holds and we can derive a (a is true in this program). But if we replaced *not* with classical negation $\neg$, we would not be able to derive a as the program provides no information that c is actually false.

An AS encoding (or program) is a collection of rules of three types: facts, constraints, and normal rules. Specifically, the rule with an empty body is called a *fact*.

$$a.$$

The rule with an empty head is called a *constraint*.

$$\leftarrow\ b_1,\ \ldots,\ b_m,\ \mathit{not}\ c_1,\ \ldots,\ \mathit{not}\ c_n.$$

The rule with a non-empty head is called a *normal rule*.

An ASP program specifies a collection of *answer sets*. To define an answer set of a normal program, I start by recalling the definition of satisfiability of a propositional program. An interpretation is a subset of the set of atoms in the language. Given an interpretation S and a program Π,

1. S satisfies a positive (non-negated) literal b in the body of a rule, if $b \in S$.

2. S satisfies a negative literal *not* c in the body of a rule, if $c \notin S$.

3. S satisfies the body of a rule, if it satisfies every literal in the body. In particular, the empty body is satisfied by every set S.

4. S satisfies the head of a rule a, if $a \in S$. The empty head is always satisfied.

5. S satisfies a rule, if whenever it satisfies the body, it satisfies the head.

6. S satisfies the program Π, if it satisfies all the rules.

Interpretations that satisfy all rules of a program are *models* of the program. For example, consider the following program Π,

```
b1.
a :- b1, b2.
```

and the satisfiability of the following two interpretations, S_1

$$\{b1,\ b2\}$$

and S_2

$$\{a,\ b1\}.$$

The first rule in the Π is a fact, which contains nothing in the body, so the body is satisfied by any set. To satisfy this rule, the set must contain the head, $b1$. Here both S_1 and S_2 satisfy the first rule. For the second rule, the body consists of two literals $b1$ and $b2$. Since $b1 \in S_1$ and $b2 \in S_1$, S_1 satisfies the body of the rule. However, since $a \notin S_1$, the head is not satisfied by S_1, and thus the rule is not satisfied by S_1. Since $b2 \notin S_2$, S_2 does not satisfy the body of the second rule, so the satisfiability of the head is not important. Therefore, S_2 satisfies the program Π. We can easily check there are other sets satisfying the program, such as $\{b1\}$, $\{b1, b2, a\}$.

The next key concept needed to define answer sets is that of a *reduct* of a program with respect to an interpretation (set of atoms) S [21]. Given a program Π and a set of atoms S, the reduct Π^S is obtained by:

1. removing all rules containing *not* l where $l \in S$, and

2. removing all literals containing *not* from other rules.

To illustrate the process of the reduct, let Π be the program

```
a :- not b.
b :- not a.
```

and S be $\{b\}$. To get the reduct Π^S, we first remove the first rule, as this rule contains *not* b and $b \in S$. The remaining program contains

```
b :- not a.
```

Then we remove the literal *not* a from this rule. The Π^S is

```
b.
```

Let us consider a normal logic program Π. It follows from the definition of the reduct that for every set S of atoms, Π^S is a normal Horn program. A Horn program is a program that consists only of normal rules or constraints with only positive atoms in the body (no negation operator). A normal Horn program is a program that consists only of normal rules with positive atoms in the body. Normal Horn programs have the following property [54].

Theorem 1. *Let* Π *be a normal Horn program. Then* Π *has a least model (which is necessarily unique).*

The least model is a model that is the subset of every other model. Now I define an answer set as follows:

Definition 1. *Let* Π *be a normal program. An interpretation (set of atoms)* S *is an* answer set *of* Π *if* S *is the least model of the reduct* Π^S*.*

This is a well-structured definition because the reduct Π^S is a Horn program and, consequently, it has a least model. To illustrate the definition, consider a normal program Π

```
b1.
a :- b1, b2, not c.
```

and two interpretations, S_1

$$\{b1,\ b2\}$$

and S_2

$$\{b1,\ b2,\ a\}$$

To test if S_1 is an answer set, I need to obtain Π^{S_1}. I first remove the second rule in Π, as this rule contain *not* c, and $c \notin S_1$. Since there is nothing left to remove, the Π^{S_1} is

```
b1.
```

The Π^{S_1} is a Horn program and the least model is $\{b1\}$, not $\{b1,\ b2\}$, so S_1 is not an answer set of Π.

Let us consider the set S_2, $\{b1,\ b2,\ a\}$. There is no rule to remove this time, as $c \notin S_2$. Then I remove the literal contain *not*, which is *not* c. As a result, the Π^{S_2} is

```
b1.
a :- b1, b2.
```

Since $\{b1\}$ is the least model of Π^{S_2}, $\{b1,\ b2,\ a\}$ is not an answer set of Π. Using the same method, I can check $\{b1\}$ is the answer set of Π.

In general, a normal program may have no answer set, a single answer set, or many answer sets. For example, the program

```
a :- not a.
```

has no answer set. The program

```
a :- not b.
```

has one answer set {a}. The program

```
a :- not b.
b :- not a.
```

has two answer sets {a} and {b}.

I still have to extend the definition of answer sets to programs that contain constraints.

Definition 2. *Let* Π *be a program whose normal rules form a program* Π' *and constraints form a program* Π'' *(hence,* $\Pi = \Pi' \cup \Pi''$*). An interpretation* S *is an answer set of* Π *if it is an answer set of* Π' *and a model of* Π''*.*

To explain the concept, I add a constraint to the last program. Let Π be the program

```
a :- not b.
b :- not a.
:- a.
```

which consists of the normal rules from program Π'

```
a :- not b.
b :- not a.
```

and the constraints form program Π''

```
:- a.
```

We first compute answer sets of Π', and then the answer sets of Π are those answer sets of Π' that satisfy the constraint that forms Π''. As explained above, the normal program Π' has two answer set {a} and {b}. With these two candidate answer sets, we then check which one is a model of the constraint program Π''. We find {b} is a model of Π''. So, the answer set of Π is {b}. The other candidate, {a}, is not.

ASP is extended to support programs that contain variables. The process of *grounding* converts a program Π with variables into a *ground* program $ground(\Pi)$. This consists of replacing in each rule variables with constants in all possible ways (always replacing the same variable with the same constant). The result, $ground(\Pi)$, is a propositional program. Hence, the concept of an answer set for $ground(\Pi)$ is well defined.

Definition 3. *An interpretation S is an answer set of Π if it is an answer set of ground $ground(\Pi)$.*

For example, the program

```
a(1). a(2). a(3). a(4). b(1). b(2).
c(X) :- a(X), not b(X).
```

is grounded into

```
a(1). a(2). a(3). a(4). b(1). b(2).
c(1) :- a(1), not b(1).
c(2) :- a(2), not b(2).
c(3) :- a(3), not b(3).
c(4) :- a(4), not b(4).
```

This program has one answer set

```
a(1). a(2). a(3). a(4). b(1). b(2). c(3). c(4).
```

Thus, it is also an answer set of the original program. It follows that the process of computing answer sets can be viewed as a two-step process consisting of grounding and solving. The first step is performed by a tool called a *grounder*. Given a program Π, it produces the ground version, $ground(\Pi)$. In the grounding phase, variables in the program are replaced by variable-free terms. In some grounders, simplifications

are applied to the rules obtained. These simplifications include eliminating from the bodies the literals that hold in the instance, and eliminating the rules whose body contains literals that do not hold in the instance. In the example above, the grounded rules

```
c(1) :- a(1), not b(1).
c(2) :- a(2), not b(2).
```

will be removed as the bodies contain b(1) and b(2) that do not hold. The following two rules

```
c(3) :- a(3), not b(3).
c(4) :- a(4), not b(4).
```

will be simplified into

```
c(3).
c(4).
```

because the literals that hold (in this case, a(3), not b(3), a(4), and not b(4)) are eliminated from the bodies. This phase results in a ground program that contains no variables but has the same answer sets as the original one. The second step is performed by a tool called a *solver*. Solvers compute answer sets of propositional programs. By computing answer sets of $ground(\Pi)$, a solver computes answer sets of the original program Π. The whole workflow of ASP, including modeling, grounding, and solving, can be summarized in Figure 2.1.

With the help of variables, we can separate facts from the encodings. The part only containing facts is called an instance. In the example above, the instance

```
a(1). a(2). a(3). a(4). b(1). b(2).
```

contains information of two sets a and b. The remaining that encodes the property of a problem to be solved is called an encoding. In this case, the encoding

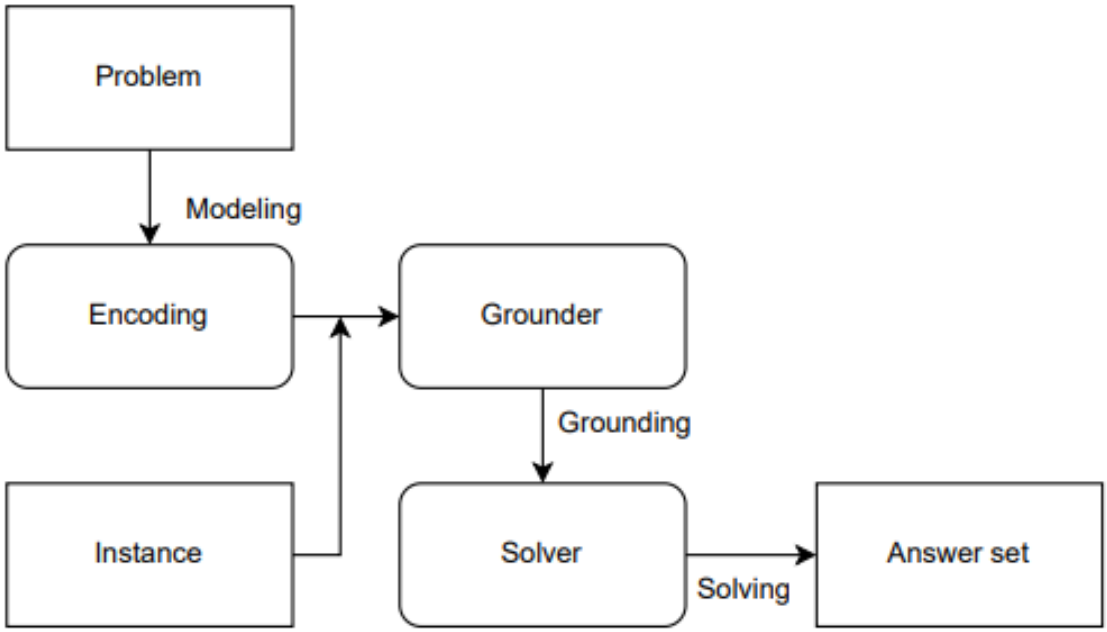


Figure 2.1: The workflow of Answer Set Programming

```
c(X) :- a(X), not b(X).
```

encodes the subtraction of sets $a - b$, that is, a set of elements in a that do not belong to set b.

To solve a problem in ASP, we first model its logic as an encoding. Then we can use the encoding to solve the problem for specific inputs (instances), represented as facts. In this way, we separate the problem description from specific instances for which the problem is to be solved.

Later on, when considering encoding rewriting, we will discuss programs built of modules satisfying some precedence property. I will now introduce basic concepts and results concerning such programs, focusing on the case, when programs can be decomposed (or split) into two modules.

Definition 4. *Let* Π *be a program whose normal rules form a program* Π' *and a program* Π''*, where* Π'' *is a collection of rules such that no predicate symbol in the head of a rule in* Π'' *appears in* Π' *(hence,* $\Pi = \Pi' \cup \Pi''$*). An interpretation* S *of* Π'

is an answer set of Π *if and only if* S *is extended by all ground atoms in the heads of ground instances of rules in* Π'' *whose body is satisfied in* S*.*

To explain the concept, I extend the last program with an additional rule. Let Π be the program

```
a(1). a(2). a(3). a(4). b(1). b(2).
c(X) :- a(X), not b(X).
d(X) :- a(X), b(X).
```

consisting of two programs Π'

```
a(1). a(2). a(3). a(4). b(1). b(2).
c(X) :- a(X), not b(X).
```

and Π''

```
d(X) :- a(X), b(X).
```

There is no predicate symbol in the head of a rule in Π'' that appears in Π'. Since we know the answer set of the program Π' is

```
a(1). a(2). a(3). a(4). b(1). b(2). c(3). c(4).
```

An interpretation S must extend all ground atoms in the heads of ground instances of rules in Π'' whose body is satisfied in S. In this case, we must include $d(1)$ and $d(2)$, as the bodies of the following rules are satisfied.

```
d(1) :- a(1), b(1).
d(2) :- a(2), b(2).
```

The answer set for program Π ($\Pi = \Pi' \cup \Pi''$) is

```
a(1). a(2). a(3). a(4). b(1). b(2). c(3). c(4). d(1). d(2).
```

The syntax of ASP has been extended to support concise modeling. I introduce two important extensions that I used in my work, choice rules and aggregates.

A *choice rule* is a rule of the form

$$m\{a_1, ..., a_i\}n \ \leftarrow \ b_1, ..., b_m, not\ c_1, ..., not\ c_n.$$

This rule states that if its body is satisfied, any arbitrary number from m to n of elements a_i may be selected into a solution. If all b_i's are established to be true and none of c_j's eventually turns out to be true, at least m and no more than n of a_k's must be true. If m and n are omitted, any element a_i can be selected into a solution.

For example, the encoding

$$\{a, b\}.$$

has four answer sets,

$$\{\emptyset\}.\ \{a\}.\ \{b\}.\ \{a, b\}.$$

Many problems can be solved by combing choice rules with constraints. We first generate candidate answer sets using choice rules and then eliminate some candidates that violate the constraints. For example, I now discuss the problem where given sets $S1 = \{1, 2, 3, 4\}$ and $S2 = \{1, 2\}$, one needs to find one number from each set so that the sum is greater than a constant value c.

I first encode the problem facts

```
s1(1;2;3;4). s2(1;2).
```

Then I choose one number from each set using choice rules

```
1{num1(X):s1(X)}1.
1{num2(X):s2(X)}1.
```

The choice rules state that the sizes of generated sets $num1$ and $num2$ are both exactly 1 so that only 1 number is selected from the original sets $S1$ and $S2$. The

results of these steps will generate 4*2 answer set candidates (select 1 out of 4 from $S1$ and 1 out of 2 from $S2$). Then I introduce a constraint to eliminate the candidates where the sum of the chosen numbers is less than or equal to the constant c, say 4.

```
:- num1(X),num2(Y),X+Y<=4.
```

The resulting program has three answer set candidates to the problem.

```
{s1(1;2;3;4). s2(1;2). num1(3). num2(2).}
{s1(1;2;3;4). s2(1;2). num1(4). num2(1).}
{s1(1;2;3;4). s2(1;2). num1(4). num2(2).}
```

If we change the constant c to 6, the constraint will now have the form

```
:- num1(X),num2(Y),X+Y<=6.
```

With this change, the program has no answer sets, consistently with the fact that the problem has no solutions. Now we take another look at the process above. To solve the problem, we first generate 8 answer set candidates, each containing a value selected from set $S1$ and a value from $S2$. Then we eliminate some of the answer set candidates that violate the sum constraints. The process above uses a generate-and-test structure that is commonly used in ASP programs. The generate step is to generate different candidate answer sets from the search space by using the knowledge base of facts or ground atoms. The test step is to narrow down the space of these to those that represent solutions to the problem by imposing constraints. Some of these constraints may involve auxiliary concepts (I will discuss it in the Hamiltonian encoding later, see Listing 2.4).

A *counting aggregate* is an expression of the form

$$number1 \prec \#count\{t_1 : L_1; ...; t_n : L_n\} \prec number2.$$

The aggregate calculates the number of unique elements of a set. Specifically, it counts a set of non-empty terms t_i's subject to literals L_i's. The literals are evaluated and

the corresponding tuples are obtained for which the literals are true in an answer set. When literals and colons are omitted, all the non-empty term tuples are obtained. The number of unique terms obtained above is returned by the count aggregate. Such number is then compared with the comparison predicates $\prec$ to number1 and number2 to decide if the expression is satisfied or not. Here, the $\prec$ can be one of the comparison symbols from set $\{<, \leq, =, \neq\}$ and either side of the comparison can be omitted.

There are two main ways to use counting aggregates in a rule. First, a counting aggregate can be used to collect the number of valid information we are interested in using the comparison symbol $=$. The rule

```
outdegree_1(N) :- N = #count{ Y: edge(1,Y)}
```

counts the number of vertices Y such that $(1, Y)$ is an edge. In other words, the out-degree (the number of edges coming out from a vertex in a directed graph) of vertex 1. Formally, it establishes the truth of the atom $outdegree\_1(N)$ for the integer N that is the out-degree of vertex 1 in a given graph G (given by the set of facts $edge(x, y)$).

A counting aggregate can also be used in a constraint to eliminate candidate answer sets. The rule

```
:- 2<= #count{ Y: edge(1,Y)}.
```

states there is a contradiction if there are two or more different Y's in the form of $edge(1, Y)$ in an answer set. This restricts the out-degree of vertex 1 must be 1 or 0.

The same logic can be expressed by using default negation *not* in front of counting aggregate. The rule above is equivalent to the following rule

```
:- not #count{ Y: edge(1,Y)} <2.
```

stating there is a contradiction if the out-degree of vertex 1 is not less than 2.

With these concepts, I now apply answer set programming to solve *Hamiltonian cycle* problems. A Hamiltonian cycle instance is modeled as collections of ground atoms of the form *node(X)* and *edge(X,Y)*, which specify nodes and edges of the graph (an example is shown in Figure 2.2). The search space of a Hamiltonian cycle problem is defined by means of a rule of the form:

Listing 2.1: Search space

```
{ hcedge(X,Y)} :- edge(X,Y).
```

In the rule Listing 2.1, *edge(X,Y)* is information given in the input instance. It stands for "the *edge(X,Y)* can be selected for a candidate solution." The rule above states that only the edges of the graph can be selected for a Hamiltonian cycle in that graph and each edge may be selected or not. Rules of the above form are called *choice* rules. Different selections result in different sets of selected edges, which are called candidate answer sets. The rule alone plus the facts have as answer sets all subsets of the edge set of the graph. To represent solutions, subsets have to satisfy constraints defining Hamiltonian cycles.

The requirement that for each node, there is exactly one selected edge that starts in it can be modeled by the following rule, called an *integrity constraint* or just a *constraint*:

Listing 2.2: One starting edge

```
:-  hcedge(X,Y1), hcedge(X,Y2), Y1!=Y2.
```

The above requirement in Listing 2.2 says that "if (X,Y1) is a selected edge and (X,Y2) is a selected edge, and if Y1 and Y2 are different, then contradiction follows; or: it is impossible that (X,Y1) is a selected edge, (X,Y2) is a selected edge and Y! and Y2 are different." Thus, a set of selected edges satisfies the constraint, precisely when it does not contain two different edges starting in the same node.

The requirement that exactly one selected edge ends in each node can be modeled in a similar way by the rule Listing 2.3:

Listing 2.3: One ending edge

```
:-  hcedge(X1,Y), hcedge(X2,Y),X1!=X2.
```

The requirement says "it is impossible that two selected edges end in the same node." Thus, a set of selected edges satisfies the constraint precisely when it does not contain two different edges ending in the same node.

Moreover, a set of selected edges satisfies both these constraints if and only if no two selected edges start in the same node and no two selected edges end in the same node. It is clear that such sets of edges define a collection of vertex disjoint paths and cycles in the graph. The set of answer sets to the program consisting of the choice rule (Listing 2.1), and the two constraints (Listing 2.2 and 2.3) are precisely the subsets of the set of edges of the input graph that span its collections of disjoint cycles.

Therefore, a set of edges satisfying these two constraints is a cycle visiting each vertex in the graph (is a Hamiltonian cycle, a solution to the Hamiltonian cycle problem) if and only if every node is reachable from every other node by a path consisting of selecting edges only. To model this requirement in ASP, I need to use auxiliary predicates and a recursive definition. To model this constraint, I first recursively define what is reachable, and use an integrity constraint to force every two nodes to be reachable from each other. This requires several rules as shown in Listing 2.4:

Listing 2.4: Reachable

```
reach(X,Y) :- hcedge(X,Y).
reach(X,Z) :- reach(X,Y), hcedge(Y,Z).
:- not reach(X,Y), node(X), node(Y).
```

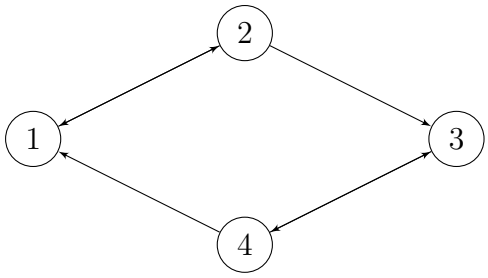


Figure 2.2: A directed graph with four nodes and six edges

The first two rules form a recursive definition of the concept of reachability. The first rule establishes the base case: Y is reachable from X (via selected edges) if (X, Y) is a selected edge. The second rule provides a recursive rule for establishing reachability via selected edges: Z is reachable from X if (Y, Z) is a selected edge and I already know that Y is reachable from X via selected edges. With the concept of reachability in hand, the third rule enforces the requirement that every node must be reachable from every other. The negation *not* is used in the rule. The constraint states that it is impossible to have two nodes X and Y such that Y is not reachable from X. A set of selected edges that satisfies the first two constraints satisfies the third constraint if and only if it forms a Hamiltonian cycle.

In this way, we have the whole ASP encoding to solve the *Hamiltonian cycle* problems. The encoding also uses a generate-and-test structure. we generate edge subsets by using edge information from an instance (see Listing 2.1). Then we narrow down the space of answer set candidates by imposing constraints (see Listing 2.2, 2.3, and 2.4). The last constraint (see Listing 2.4) encodes complex concepts by using auxiliary predicate reachability.

The *Hamiltonian cycle* encoding describes the logic of the problem and requires instances that provide information about the edges and nodes. A directed graph is an input instance for the *Hamiltonian cycle* problem. A graph with four nodes and six edges (see Figure 2.2) can be modeled as follows:

```
node(1..4).
edge(1,2). edge(2,1). edge(2,3). edge(3,4). edge(4,1). edge(4,3)
```

Combined with the description of a graph, the *Hamiltonian cycle* encoding can be used to search for the *Hamiltonian cycle* in that graph.

Processing a combined problem-data description consists of two steps: grounding and solving. For example, the instance and the first rule of the Hamiltonian cycle encoding

```
{ hcedge(X,Y)} :- edge(X,Y).
```

is grounded into

```
node(1..4).
edge(1,2). edge(2,1). edge(2,3). edge(3,4). edge(4,1). edge(4,3).
{hcedge(1,2)}. {hcedge(2,1)}. {hcedge(2,3)}.
{hcedge(3,4)}. {hcedge(4,1)}. {hcedge(4,3)}
```

The variables *X,Y* in *hcedge* are instantiated with values specified by extitedge so that programs with variables become propositional programs. The tool used for the instantiation is called a *grounder*. As mentioned above, the choice rules will generate 2^e different answer set candidates. In this case, there will be 64 answer sets. For example, consider three possible answer sets (answer sets all contain atoms defining the input, such as all *edge*s and *node*s, but I only show *hcedge*s here for convenience):

```
1st: { hcedge(2,1). hcedge(2,3).}
2nd: { hcedge(1,2). hcedge(2,1). hcedge(3,4). hcedge(4,3).}
3rd: { hcedge(1,2). hcedge(2,3). hcedge(3,4). hcedge(4,1).}
```

We can see the first candidate answer set has two edges coming out of node 2, so it violates the constraint of the Hamiltonian cycle encoding that for each node there is exactly one edge coming out from it. Specifically, it violates the grounded constraint

```
:-  hcedge(2,1), hcedge(2,3), 1!=3.
```

The second candidate answer set has exactly one edge coming out of each node and one edge coming into each node, but it does not satisfy the reachability constraint. Now I show part of the grounded program related to reachability

```
reach(1,2) :- hcedge(1,2).
reach(2,1) :- hcedge(2,1).
reach(3,4) :- hcedge(3,4).
reach(4,3) :- hcedge(4,3).
reach(1,1) :- reach(1,2), hcedge(2,1).
reach(1,2) :- reach(1,1), hcedge(1,2).
reach(2,2) :- reach(2,1), hcedge(1,2).
reach(2,1) :- reach(2,2), hcedge(2,1).
reach(3,3) :- reach(3,4), hcedge(4,3).
reach(3,4) :- reach(3,3), hcedge(3,4).
reach(4,4) :- reach(4,3), hcedge(3,4).
reach(4,3) :- reach(4,4), hcedge(4,3).
:- not reach(1,1), node(1), node(1).
:- not reach(1,2), node(1), node(2).
:- not reach(1,3), node(1), node(3).
:- not reach(1,4), node(1), node(4).
...
```

We observe that the last two constraints are not satisfied (while there are more not shown in the grounded program above).

The third satisfies all three constraints, and hence, together with the input instance atoms, is the answer set of the program. It is easy to check if it has exactly one edge coming out of each node and one edge coming into each node. Now I verify

why it satisfies the reachability constraints by showing part of the grounded program related to reachability

```
reach(1,2) :- hcedge(1,2).
reach(2,3) :- hcedge(2,3).
reach(3,4) :- hcedge(3,4).
reach(4,1) :- hcedge(4,1).
reach(1,3) :- reach(1,2), hcedge(2,3).
reach(1,4) :- reach(1,3), hcedge(3,4).
reach(1,1) :- reach(1,4), hcedge(4,1).
reach(2,4) :- reach(2,3), hcedge(3,4).
reach(2,1) :- reach(2,4), hcedge(4,1).
reach(2,2) :- reach(2,1), hcedge(1,2).
...
:- not reach(1,1), node(1), node(1).
:- not reach(1,2), node(1), node(2).
:- not reach(1,3), node(1), node(3).
:- not reach(1,4), node(1), node(4).
:- not reach(2,1), node(2), node(1).
:- not reach(2,2), node(2), node(2).
:- not reach(2,3), node(2), node(3).
:- not reach(2,4), node(2), node(4).
...
```

We observe that all the nodes are reachable from each other in the grounded program, so it satisfies all the reachability constraints. Since it also satisfies the out-degree and in-degree constraints, it is one of the answer sets of the Hamiltonian cycle problem above.

Moreover, the atoms hcedge(a,b) in this (and any) answer set define a Hamiltonian cycle. The process to calculate these answer sets is called solving. In the solving phase, we need to compute answer sets of the program resulting from grounding. Tools developed for that task are commonly referred to as *solvers*. Several such solvers have been proposed. Most notable solvers are *dlv* [37],[1] *gringo/clasp* [17][2] and *wasp* [1].[3] These tools have been shown to be especially effective on search and optimization problems whose decision versions are in the class NP. The grounded rules and programs without variables can be treated as propositional. The solvers take the grounded propositional rules and look for the assignment of 0's and 1's to the propositional variables so that they obey the definitions of an answer set (in particular, are consistent with the constraints). Algorithms used in many of these answer set solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [10] and conflict-driven clause learning (CDCL) procedure [47], developed for testing propositional satisfiability. The key difference between ASP and satisfiability comes from a different treatment of the rule connective $\leftarrow$ (written as :- in programs) and the negation connective *not*.

[1] `http://www.dlvsystem.com`
[2] `https://potassco.org`
[3] `http://alviano.github.io/wasp/`

Chapter 3 Research Challenges — the Scope of the Thesis

Due to the rich high-level modeling capabilities of ASP and the continuous improvement of solvers, ASP has been applied in many areas of theoretical or practical importance. However, despite the ease of modeling and the demonstrated potential of ASP, using it is not without challenges. There are major bottlenecks in both the grounding and solving phases. For some problems, grounding may take a huge amount of time, especially when there are multiple variables in the rules. The step of instantiating these variables is computationally expensive, which slows down the overall performance of the ASP system. For example, a program may contain a rule with n variables:

```
:- n1(N1),n2(N2),n3(N3),...,nn(Nn).
```

If each variable has k possible instantiations, then the grounding size of this rule is k^n. Consider now a situation, not uncommon in practice, when each variable has, say, 1000 instantiations. If a program contains a rule with just 4 variables, that rule potentially contributes 10^{12} rules to the ground program. Generating a grounding of a program may be infeasible in such cases. Even if some grounders employ smart grounding procedures, which can generate a ground program of a smaller size, grounding such rules is still a challenge. One way to reduce the grounding time of inefficient encodings is to decrease the number of variables in the rules. For example, a program for the *Hamiltonian cycle* problem may use these rules to impose the constraint of reachability (see the previous chapter):

```
...
reach(X,Y) :- hcedge(X,Y).
reach(X,Z) :- reach(X,Y), hcedge(Y,Z).
```

```
:- not reach(X,Y), node(X), node(Y).
```

These rules use three variables X, Y, Z to define the auxiliary predicate *reach*. Grounding the program and a random graph instance with 1000 nodes and 10000 edges [1] generates around 11 million lines in the grounding output. The grounding time is 31.30s and the total solving time (including grounding time) is 318.63s. [2]

To overcome the "grounding" problem, one can use encoding rewriting algorithms to reduce the number of variables in rules by applying projection [16]. Another approach is to find a different way to model the constraint so that the corresponding encoding uses fewer variables in the rules [3]. For instance, the reachability constraint can be modeled by first selecting a node (either hardwiring the choice or using a choice rule and constraints to select one). Suppose a selected node is s. Clearly, the reachability constraint that requires that each vertex be reachable from s by a path of selected edges enforces that candidates are collections of disjoint cycles covering all vertices and cycles form actually a single cycle. This constraint can be stated as follows (note that I use the same name for the reachability predicate as before, and I chose to hardwire the choice of the "start" node to 1; the program will work only for those graphs that contain 1 as one of their vertices). The program can be rewritten as

```
...
reach(X) :- hcedge(1,X).
reach(Y) :- reach(X), hcedge(X,Y).
:- not reach(X), node(X).
```

The new program (see ham_1x in Appendix 9.A) uses rules with at most two variables.

[1] https://drive.google.com/file/d/1Qz_jG8CyrGdpXEy4F8iI9RlhLb0wrInD/view?usp=sharing

[2] The program (see ham_xy in Appendix 9.A) was grounded with *gringo* (versions 5.2.2) and solved by *clasp* (versions 3.3.3) with default configuration, on a computer with Intel(R) Core(TM) i7-7700 CPU and 16 GB Memory, running on Linux 5.4.0-91-generic x86_64

Grounding the program and the same instance as above only generates 241,000 lines of grounding output. The grounding time is 0.60s and the total solving time (including grounding time) is 0.783s.

The second challenge concerns solving. The bottleneck for solving derives from the inherent hardness of the problem it handles. Namely, the problem to decide whether a propositional program has an answer set is NP-complete (for some versions of the language, even Σ_2^P-complete [4]). As is widely conjectured, those problems simply do not admit solutions that would guarantee good performance. As a result, any solver, including the best state-of-the-art solvers, can only solve a portion of problems when tested on a broad class of instances. Interestingly, vast experimental evidence collected over the years shows that good solvers have an area, a class of instances, where they excel but, unavoidably, will fail to perform well on many instances of other types. Typically, areas of good performance of different solvers do not coincide. So, it is unlikely a single solver will emerge that would uniformly outperform all others. That opens a possibility of improving solving with ASP solvers by selecting the right solver on a per instance basis.

Another challenge, and opportunity, for the solving phase, comes from the fact that current state-of-the-art solvers are designed with tens or even hundreds of control parameters (settings), which affect the decisions made during the searching process. Examples include the number of conflicts to trigger a new restart, the conflict counting policy (local vs global), the number of learned causes to store, the number of learnt causes to delete after a restart, and the number of restarts before shuffling internal data. Previous works [32, 30] show that a well-chosen parameter configuration can boost the performance of solving by several orders of magnitude compared with a random configuration. The state-of-the-art solvers come with a default parameter configuration that in many cases works well but, for some specific types of instances, the customized parameters are more efficient. Alternatively, instead of searching for

Table 3.1: Performance of individual encodings and the oracle.

Encoding	Solved Percentage%	Average Solved Runtime	Number of Wins
Encoding 1	82.3	84.1	102
Encoding 2	71.8	46.6	126
Encoding 3	55.3	29.7	110
Encoding 4	76.2	42.9	155
Encoding 5	55.4	31.9	120
Encoding 6	77.4	47.7	151
Oracle	98.0	22.8	

the best parameter configuration, one can construct a collection of solvers from a single one by selecting different parameter settings and then apply solver selection techniques to this ensemble. The technique is referred to as *portfolio solving*.

Not surprisingly, solver selection, portfolio solving, and automated parameter configuration have all been extensively studied in ASP [45, 29], as well as in other fields [51, 33]. In order to take advantage of the varying performance on different problems, researchers proposed to make solver selection on the *per-instance* basis, that is, to use machine learning techniques to build performance models for individual solvers. Given a new instance, these models provide estimates for the performance of the corresponding selections (solvers, parameter configurations) on that instance. These predictions can be then used to select a solver or a parameter configuration to use on that instance.

There is yet another possibility for enhancing the effectiveness of ASP. As we observed earlier, it is also well known that search problems commonly admit many alternative equivalent encodings. These encodings, while logically equivalent, typically perform differently when run on a set of instances. Experimental results with alternative AS encodings of the same problem accumulated in the past two decades suggest that different AS programs for a given problem may differ significantly in their performance. Namely, it is rarely the case that the same encoding performs best (under a selected grounder-solver tool) across all data instances for the prob-

lem. Choosing one encoding for a specific instance may make the grounder-solver run "forever," while selecting another may yield a solution in a fraction of a second. Even though researchers tried to understand how the performance depends on particular ways the program encodes constraints contained in a specification of a considered problem, no universally valid principles emerged. These observations suggest that the availability of multiple encodings can be turned into an asset that might improve the efficiency of solving in ASP. This observation is the main "high-level" contribution of my thesis. For example, Table 3.1 summarizes the performance of six equivalent *Hamiltonian cycle* encodings (see Appendix 9.A) on a set of 784 graph instances[3]. The table reports the percentage of instances solved, average runtime for solved instances, and the number of times the encoding records the best solving time (I refer to such cases as *wins* for the encoding). The table also includes the performance of an encoding selection oracle, which always selects the best encoding on a per-instance basis. We observe that there is no uniformly best encoding for the dataset of instances, each encoding excels in different instances.

Previously researchers explored the possibility of exploiting multiple solving algorithms available for tackling problems both outside of ASP and in ASP. However, unlike solver selection and parameter configuration, encoding selection has not been extensively studied yet. Selecting the best encoding (or even any of some number of top encodings) is a challenge emerging from our work.

More precisely, two possible lines of attack emerge: (1) to establish encoding rewriting techniques to generate better performing encodings, and (2) to develop methods for encoding selection and encoding portfolio solving, similar to those used in algorithm selection and portfolio solving [23, 29]. The first idea has received some attention in recent years [5, 3, 26]. However, the approach to capitalize on the availability of collections of equivalent encodings, produced "by hand" or generated

[3] `https://drive.google.com/drive/folders/1DAiCQmsmrDmDJ8N3nsJ3C8YDNhddoRCX?usp=sharing`

automatically from a "hand-made" one, has not yet been explored.

In the remaining chapters of the thesis, I will discuss related work in the areas of algorithm selection, encoding rewriting, and hard instance generation. Then I will continue with the discussion of my work. Specifically, I discuss the related work in Chapter 4. I will introduce some techniques to rewrite encodings in both the grounding and solving phases. There will be a discussion on encoding rewriting by creating pre-calculated predicates to avoid time-consuming jobs in the ASP grounding phase in Section 5.1. While this technique rewrites encoding manually (as the process of predicate generation by using a non-ASP program cannot be automated), I introduce the next technique that allows for rewriting automation. I will explain my extension on an automatic encoding rewriting tool in Section 5.2, where I expanded the scope of the original work in the aggregate introduction and implemented aggregate elimination. I then explore the effect of rewriting encodings by the duplication of rules in Section 5.3. Next, I will introduce my solution to encoding selection in Chapter 6, a platform that performs automatic equivalent encodings generation and selection, given a set of input encodings of a problem and a set of instances. What follows is the discussion of methods to generate benchmark instance sets in Chapter 7, hard instances of different combinatorial problems I selected for testing the performance of the encoding selection platform. Next in Chapter 8, I discuss several test cases that explain the design and results of each experiment. The last Chapter 9 gives a discussion of the techniques used, limitations, and future work. There is an appendix following the last chapter, where I put all the encodings, instance sets, the performance data, and the software about hard instance generation and the encoding selection platform.

Chapter 4 Related Work

4.1 Algorithm Selection

Over the past two decades, ASP researchers have developed several high-performance ASP solvers [48, 40, 38, 20, 2]. These solvers often have different areas of strength. To choose the most efficient ASP solver for each individual problem, the idea of solver selection in ASP was proposed by Gebser et al. [18], who built on the algorithm selection work by Rice [51] and subsequent specializations and extensions of the original idea to propositional satisfiability solver selection [56]. In algorithm selection, given a set of candidate algorithms and a specific instance to be solved, one determines which algorithm is likely to perform best on the instance. A dominant approach to algorithm selection today is based on machine learning. Given a set of benchmark instances and several candidate solvers, one extracts features of these benchmark instances and obtains performance data of each solver on each instance. Then one trains machine learning models on the feature and performance pairs $\langle F, P \rangle$ to learn the mapping from an instance's features to the performance of each algorithm on that instance. Lastly, one uses the models, when given a new instance feature, to predict each algorithm's performance and select the one with the best-predicted performance from a set of given algorithm candidates.

The algorithm selection approach was successfully used in propositional satisfiability (SAT) and constraint programming (CP), where it is known as *solver selection*. Xu et al. [56] proposed the solver *SATzilla*, a portfolio-based algorithm selection approach in the area of SAT. It competed with more than 30 solvers in the 2007 SAT Competition. It won in three competition categories and was second and third in two other. Inspired by *SATzilla*, Gebser et al. [18] implemented *claspfolio*, the first ASP system applying solver selection techniques. *Claspfolio* takes a set of *clasp* solvers

with different parameter settings (configurations) as the input and selects the best *clasp* configuration for each instance. The system employs machine learning techniques to build Support Vector Regression models for each candidate *clasp* solver. Given an instance, *claspfolio* uses the models to estimate the effectiveness of each *clasp* configuration and selects one that is predicted to be most effective. *Claspfolio* was tested on the benchmark classes of the 2009 ASP competition. It was trained on 3096 instances from the Asparagus benchmark repository[1] and the 2009 ASP competition. The run time results showed *claspfolio* saved more solving time than the carefully hand-tuned *clasp*. Besides regression models, researchers also studied classification models. Maratea et al. [45] proposed a tool *ME-ASP*, which exploits different classification models (APC, J48, KNN, SVM, etc.) to choose a solver from a selection of candidate ASP solvers. The performance of *ME-ASP* was evaluated on the grounded instances at the third ASP Competition against various solvers, *clasp*, *claspd*[2], *cmodels*[41][3], *dlv*, and *idp*[11][4]. The results showed that almost any classification method resulted in *ME-ASP* outperforming all solvers that compete with it.

An algorithm portfolio provides another way of exploiting the complementary performance of different solvers. The term algorithm portfolio [23] is used to describe a method of running several algorithms in parallel or in sequence, with different amounts of time assigned to each algorithm, and with algorithms running according to a per-instance computed schedule. The portfolio may consist of different solvers or the same solver with different configurations. Algorithm portfolios are of two main types, *static* and *dynamic*. In static portfolios, the schedule of solvers is predetermined and each solver runs the same amount of time in solving any given problem instance. This idea is easier to implement than algorithm selection and in some cases

[1] `https://asparagus.cs.uni-potsdam.de/`
[2] `https://potassco.org/cemetery/claspd/`
[3] `https://www.cs.utexas.edu/users/tag/cmodels/`
[4] `https://www.idp-z3.be/`

is very effective compared with running a single solver[28, 29]. It is often used in many solver selection systems to perform pre-solving for a short amount of time. In the pre-solving phase, easy problem instances are quickly solved. For these instances, there is no need to run machine learning models to predict which solver to use. Those that are not quickly solved in the pre-solving phase are passed to feature extraction and solver selection. In dynamic portfolios, solvers and schedules are both determined dynamically by the system.

Hoos et al. [29] proposed the solver *Claspfolio2* to implement the combination of per-instance solver selection and algorithm portfolio. In *Claspfolio2*, the solver selection and algorithm portfolio are two separate parts with different time budgets. Time budgets are determined by the cross-validation result of the solver selection model. During the training process, *Claspfolio2* learns a model that maps instance features to the performance of solvers. The cross-validation is used to get the performance estimation of the learned machine learning model. Based on the estimation, the time budgets are determined. If the machine learning model performs well, solver selection gets all the time in the budget, and the algorithm portfolio will not run. If the model performs extremely poorly, the algorithm portfolio is allocated almost all the time in the budget. Based on the allocated time budget, the system then computes the best solvers and schedules to include in the algorithm portfolio. To solve a new instance, the system runs the machine model's selection first, and if it does not solve within its time budget, the system runs an algorithm portfolio.

Besides the solver selection, another approach to improve solver performance is by addressing the problem of parameter configuration of solvers. Solvers, as I mentioned above, typically come with tens and even hundreds of parameters. The problem to select a configuration that might promise a good performance is a difficult one. Hutter et al. [31] introduced *ParamILS*, an algorithm configuration framework that adopts iterated local search strategy to exploit configuration space. After random parameter

initialization, *ParamILS* employs iterated local search method to search parameter configuration space. It proceeds by changing the value of just one parameter at a time. It also uses a fixed number of random moves for solution perturbation, as well as random parameter re-initialization with a certain probability to escape from local optima. *ParamILS* provides methods for optimizing a target solver's performance on the distribution of problem instances by searching from a set of ordinal and categorical parameters. Similar to the parameter configuration of solvers, the solver selection methods have hyper-parameters that need to be carefully tuned. Lindauer et al. [42] proposed an automatically configured algorithm selection method, *AutoFolio*, which cannot only select the best algorithm but also set the hyper-parameters correctly.

4.2 Encoding Rewriting

The topic of encoding optimization has been considered before. Gebser et al. [16] conclude their paper with a section on hints on modeling that can be used as principles to guide the process of non-ground encoding optimization. The first principle is to use rules that help keep the ground program compact. The techniques to accomplish that involve introducing aggregates and limiting the number of variables in rules by applying projection or some *ad hoc* rewritings. The second direction is to introduce additional constraints to prune the search space by considering special cases and applying symmetry breaking. Experiments run by Gebser et al. show the potential of these encoding optimization techniques. In particular, Gebser et al. observe an improvement in both grounding as well as solving time in 22 out of 32 common benchmarks. The most obvious decrease is on Sudoku, where grounding time was reduced from 221.94 seconds to 3.64 seconds, grounding size was reduced from 1643.8 MB to 34.1 MB, and solving time dropped from over 20 minutes, the cut-off time, to a bit more than 4 seconds.

Inspired by the work by Gebser et al. [16], Buddenhagen and Lierler [5] proposed

an encoding rewriting tree that uses several program rewriting techniques, such as projection and simplification, to generate high-performance encodings. Projection reduces the number of variables in a rule and results in fewer ground instances of the rule in the ground program, while simplification eliminates some rules that are entailed by other part of the program. The encoding rewriting tree suggests the rewriting direction at each stage so that the encoding rewriting processes can be automated. Experiments reported by Buddenhagen and Lierler [5] showed promising results. In particular, for one of the encodings considered, the average solving time dropped from over 300 seconds to about 60 seconds.

The problem of automating the rewriting into rules using fewer variables was studied. One problem with most natural and direct encodings is that their rules often have large sizes. The size of a rule is defined by the number of literals contained in the body of that rule. A rule with many literals is called informally a large rule (there is no strict threshold for a rule to be large, but four or more literals in the body would typically qualify a rule as large). The grounding time of a large rule with many variables is ofen prohibitively large. Bichler et al. [3] propose a rule optimization tool, *lpopt*, aiming to reduce the size of large non-ground rules. The tool works by first computing the tree decomposition of a rule and then splitting the rule up into multiple, smaller chunks according to this decomposition. By decomposing large logic programming rules into smaller rules, *lpopt* helps to reduce the size of a grounded program and thus improves solving performance. Bichler et al. tested the tool by comparing the performance of grounding and solving of programs optimized with *lpopt* against non-preprocessed ones. When tested on 49 already hand-tuned encodings provided by the ASP competition, *lpopt* was still able to decompose and rewrite 41 of them. The results show that rewritten programs always have a smaller grounding time than the original programs.

Hippen and Lierler [26] proposed an automated rewriting technique for non-

ground logic programs, which they implemented in a tool called *projector*. Similar to *lpopt*, *projector* also works by implementing the principle of projection, and aims to split a large rule into several short rules with fewer variables. But Hippen and Lierler used different heuristics to select rules to apply projection to and variables to project away. By rewriting rules into their equivalent forms, *projector* produces fewer ground instances than the original program and thus improves the performance.

4.3 Hard Instance Generation

The generation of hard instances is of practical interest since it supports benchmarking for investigating the hardness of the problems and testing the effectiveness of corresponding algorithms. Cheeseman et al. [7] showed that there exist one or more parameters for some NP problems, such that hard instances always occur around some particular critical values of these parameters. They observed that the correct setting of the critical values forms a boundary that divides the problem instances into two completely different regions, an underconstrained region, and an overconstrained region. In an underconstrained region, almost all instances have solutions, and finding solutions is relatively easy. In an overconstrained region, almost all instances have no solution, and since many search algorithms will terminate early, it is also easy to find there is no solution. However, in the region where critical values are located, for many instances, it is hard to tell where a solution exists. The corresponding phenomenon where problems transition between the underconstrained region and the overconstrained region is referred to as the *phase transition*. Cheeseman et al. identified the location of the phase transition domain for a handful of NP-complete problems and verified the connection between phase transition and hardness.

Selman et al. [52] proposed the idea of generating hard satisfiability problems to test the average-case difficulty of SAT testing. Their work confirmed the previous observation that many instances of SAT testing are quite easy and also showed many

hard instances can be generated by controlling the ratio of the number of clauses to the number of variables. They tested the fixed clause length model and found that when the ratio of clauses to variables is close to 4.3 for 3CNF, the resulting instances are computationally challenging. This happens to be the ratio where randomly generated instances have the probability of about 0.5 of being SAT and UNSAT, which is exactly near the phase transition point.

Gent et al. [22] confirmed domain transition is associated with hard instances of the traveling salesman problem. As opposed to the optimization problem, where an optimal tour length is returned, they focused on the decision problem to answer if there is a solution under a given tour length boundary. They generated random graph instances by placing cities on a square of the area and set the boundary of tour length using a carefully designed function related to three arguments, the size of the area, the number of cities, and a controlling parameter k. They tested the runtime of generated instances using a branch and bound algorithm with the well-known Hungarian heuristic for branching [36]. They found when the controlling parameter k is small, the boundary of tour length is small, and problems are in the insoluble phase, which means there is no tour of the length or less; when k is large, the boundary of tour length is large, and problems are in a soluble phase, which means solutions always exist with a tour of the length or less. When the parameter k is set to 0.6, they observed that 50% of the problems have no tour of the corresponding length. They also observed sharp runtime peaks in the phase transition region.

Chapter 5 Encoding Rewriting

5.1 Encoding Rewrites by New Predicates Introduction

As explained above, any search and optimization problem in ASP admits many alternative equivalent encodings. These encodings may be syntactically different but are semantically equivalent. Encoding selection, by providing a method to select the most effective encoding on a per-instance base, takes advantage of the varying performance of encodings on different instances. However, the performance of encoding selection depends on the diversity of the solving performance of encoding candidates, so we need methods to generate enough encodings.

There are two basic ways to generate candidate encodings: manually and automatically. One can generate encodings and modify them by hand based on the knowledge of ASP. Previous works [14, 7, 4, 21] suggest many rewriting techniques that can be applied to improve the solving performance of candidate encodings. By applying different heuristic encoding rewriting techniques manually, we can generate encodings that have the potential to perform well. Here I introduce an encoding rewriting technique by introducing new predicates into the original rules. I develop a method for manual rewriting of rules that involve arithmetic atoms (atoms that compare two arithmetic expressions) with the goal of improving the grounding times. The rewritten programs can later be subjected to other rewritings and can be used in per-instance encoding selection and encoding schedule building.

Asp provides an easy and effective way to model arithmetic operations. The addition, multiplication, and exponentiation symbols used in ASP are

$$+,\ *,\ **$$

Users can combine variables with integer constants to express complex operations.

For example, a quadratic function $x^2+2*x+1$ can be easily encoded in the following program

```
n(1,2,3).
p(X**2+ 2*X + 1) :- n(X).
```

The values of n are 1,2 and 3, so the stable model of the program consists of the atoms (where 4, 6, and 9 are the values of $2x^2+2x+1$ for $x=1,2,3$):

```
p(4).p(6).p(9).
```

Despite the effectiveness of modeling arithmetic operations using these arithmetic symbols, modeling problems using arithmetic operations should be dealt with care, as grounding rules involving arithmetic operations may consume a great amount of time. Specifically, the exponentiation operation does not scale well in the grounding phase to handle large input values. In the following of the section, I list two problems that involve arithmetic operations, show the inefficiency of arithmetic operations with regard to the grounding time, and provide a solution that dramatically reduces the grounding time by means of new predicates introduction.

5.1.1 Pythagorean Triple

A Pythagorean triple consists of three positive integers a, b, and c, such that $a^2+b^2=c^2$.[1] The Pythagorean triple problem in ASP is to find the largest numbers n to partition numbers from 1 to n into two parts so that no part contains three positive integers a, b, and c satisfying $a^2+b^2=c^2$. Huelle, Kullmann, and Marek [25] proved that this problem has no model when n is greater than 7825. The Pythagorean triple problem can be easily modeled by constraints. We can define a constraint to enforce that no part contains the square sum relation and then incrementally adjust n to find

[1] `https://en.wikipedia.org/wiki/Pythagorean_triple`

the final solution, the first n to make the problem unsatisfiable. An encoding to the Pythagorean triple can be modeled in the following way

```
number(1..n).
part(1;2).
{partition(X,Y) : part(Y)}=1 :- number(X).
:- partition(X,P), partition(Y,P), partition(Z,P), X*X+Y*Y=Z*Z, part(P).
```

The first rule by using the interval operation .. defines the space of integers to partition, the range of numbers from 1 to n. The second rule states there are two parts to split the numbers. The third rule by applying the choice rule expresses each number is assigned to only one of the parts. The last rule is a constraint, stating that any part should not contain three values X, Y, and Z so that $a^2 + b^2 = c^2$.

The encoding above describes the correct split of the numbers into two parts and each does not contain a Pythagorean triple if it is satisfiable. Otherwise, there will be no result if it is unsatisfiable, and then the largest number n to split is found.

When solving the Pythagorean triple problems using the above encoding, I observed the runtime (grounding + solving) grows exponentially with the growth of the size of input n (See red line in Figure 5.1). A further investigation revealed that most of the runtime was spent in the phase of grounding. In Figure 5.1, we saw that the grounding time (blue line) is almost equal to the total runtime (red line), which means that whenever a problem is grounded, it can be easily solved. A problem arises that how I can reduce the amount of time spent on grounding in order to improve the efficiency.

In the grounding phase, the grounder instantiates X, Y, and Z with all possible values and performs the square sum operation. It also eliminates the instantiations that do not satisfy $X^2 + Y^2 = Z^2$ and only keeps those that satisfy the square sum relation. For example, when n is set to 10, the grounding output is

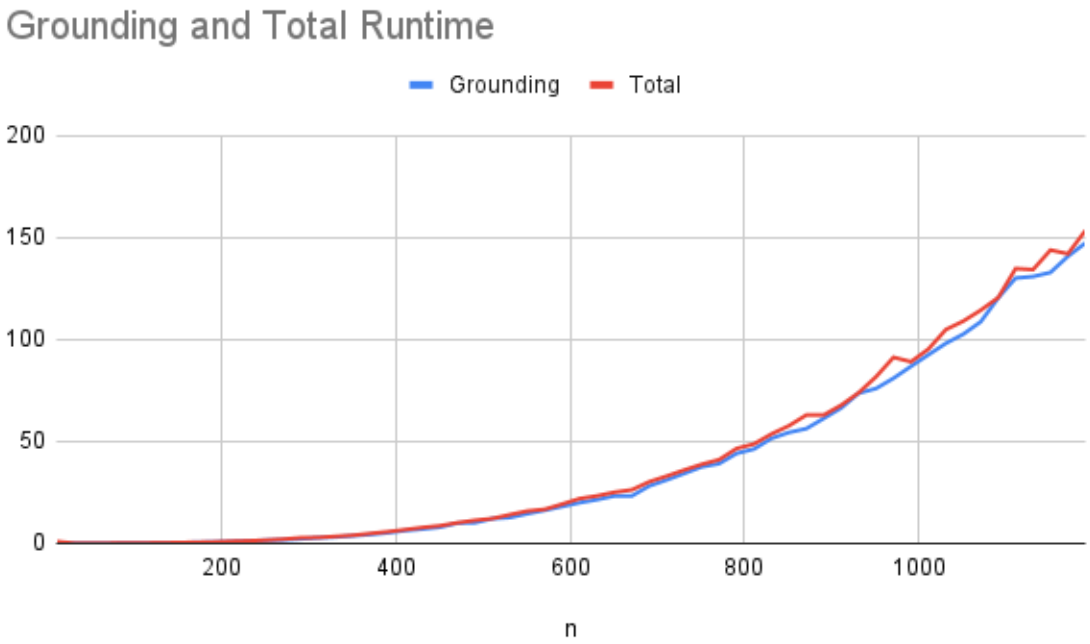


Figure 5.1: Grounding time and the total runtime of original Pythagorean encoding

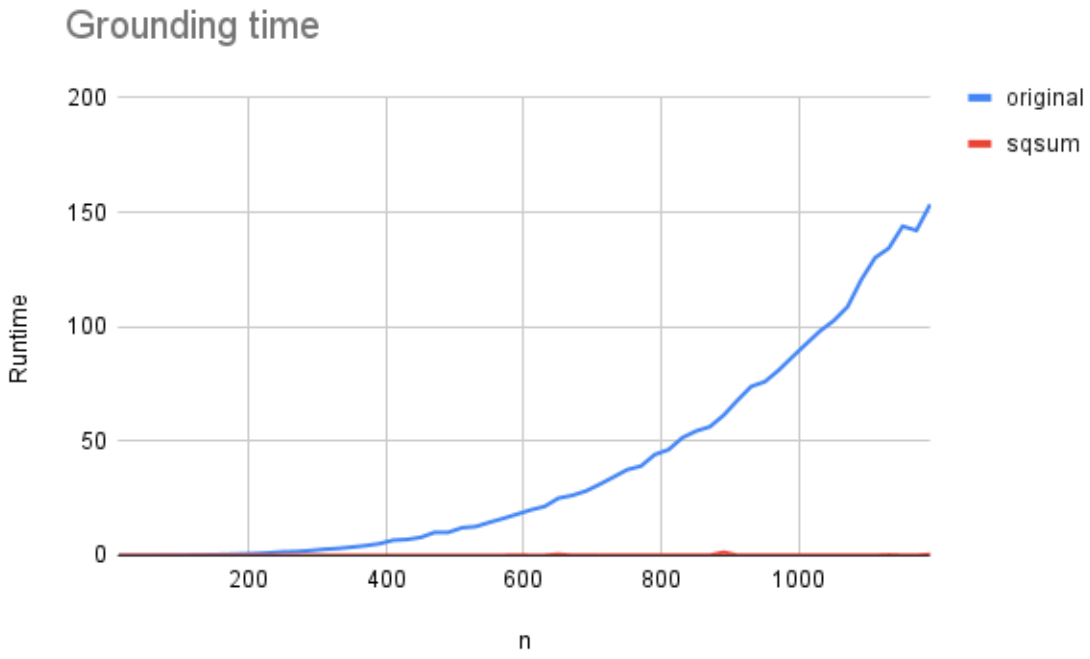


Figure 5.2: Grounding time of Pythagorean encodings (original vs sqsum)

```
number(1..10).
part(1;2).
:-partition(5,1),partition(3,1),partition(4,1).
:-partition(5,1),partition(4,1),partition(3,1).
:-partition(10,1),partition(6,1),partition(8,1).
:-partition(10,1),partition(8,1),partition(6,1).
:-partition(5,2),partition(3,2),partition(4,2).
:-partition(5,2),partition(4,2),partition(3,2).
:-partition(10,2),partition(6,2),partition(8,2).
{partition(1,1),partition(1,2)}=1.
{partition(2,1),partition(2,2)}=1.
{partition(3,1),partition(4,2)}=1.
{partition(4,1),partition(4,2)}=1.
{partition(5,1),partition(5,2)}=1.
{partition(6,1),partition(6,2)}=1.
{partition(7,1),partition(7,2)}=1.
{partition(8,1),partition(8,2)}=1.
{partition(9,1),partition(9,2)}=1.
{partition(10,1),partition(10,2)}=1.
```

We see that the grounded program only keeps instantiations with $X^2 + Y^2 = Z^2$ for each part. Thus, first, a large number of instantiations are generated (of the order of n^3), and then most of them are removed. A possible solution is to precompute the relevant combinations of x, y, and z (in our example, all Pythagorean triples over integers in $\{1, 2 \ldots, n\}$) outside of the ASP program, collect them into a list of facts over a new predicate, expand the program by these new facts, and replace the arithmetic atom in appropriate rules in the program by an atom involving the new predicate. In this way, when grounding, no spurious instantiations are generated.

Therefore, the solution to the problem is to adopt the predicate introduction, where I pre-calculate the list of triples (a, b, c) such that $c^2 = a^2 + b^2$, outside of the grounder and integrate the result with an appropriately modified AS program modeling the problem. In the example above, I can pre-calculate the extension of the predicate $sqsum(a, b, c)$, that contains all triples (a, b, c) of integers such that $1 \leq a, b, c \leq n$ and satisfying the formula $a^2 + b^2 = c^2$. Then the precomputed predicate *sqsum* is used to replace the original occurrence of the arithmetic equality atom, and all the facts $sqsum(a, b, c)$ are appended to the program.

In this case, I use the following rule

```
:- partition(X,P), partition(Y,P), partition(Z,P), sqsum(X,Y,Z),part(P).
```

to replace the square sum operation with the new predicate *sqsum*. I first pre-calculate all the $sqsum(X, Y, Z)$ predicates that satisfy $X^2 + Y^2 = Z^2$ outside the ASP system using other tools and save the results. For example, $sqsum(3, 4, 5)$ is one of the values of the new predicate. Then such values are passed as facts into the ASP system.

Since the grounder only generates instantiations for the Pythagorean triples and never generates instantiations that will later be removed, the grounding time reduces dramatically. The Figure 5.2 shows the grounding time comparison of original encoding and new encoding (I call sqsum). With pre-calculated square sum tuples as the input, the grounding time reduces dramatically. We notice that the grounding time of the original encoding grows exponentially while the grounding time sqsum does not change over time. We need to note that the grounding time of the sqsum encoding is always less than 0.2s. The image does not reflect the time required to pre-calculate the predicate *sqsum*s outside the ASP problem. I use *python* to generate such predicates and the time spent is less than 0.2s for any n in the Figure 5.2.

The total runtime comparison of these two Pythagorean encodings (original vs sqsum) is reported Figure 5.3. The blue line shows the total grounding and solving

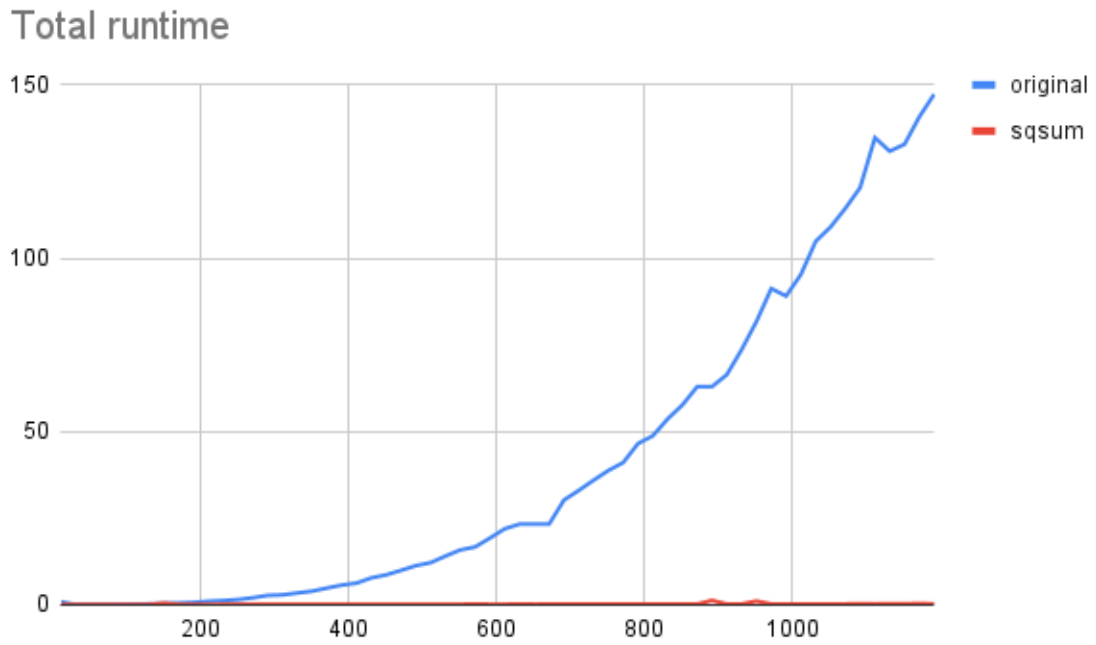


Figure 5.3: Total runtime of Pythagorean encodings (original vs sqsum)

time of the original encoding, while the red shows the total of the sqsum generation, grounding, and solving time of the sqsum encoding. The result shows that with the introduction of the new predicate *sqsum*, I significantly reduce the total running time.

To prove the correct, we consider the following programs. Let Π be a program containing a rule r of the form

$$H \leftarrow \bigwedge_{1 \leq i \leq b} F(X_i, Y),\ Q(X), G(Y). \tag{5.1}$$

where H is the head of a rule and can be empty to express a constraint, X is a tuple of variables $X_1,\ X_2, \ldots, X_b$, Y is a tuple of variables disjoint with X, F is a predicate, $Q(X)$ is a list of literals over variables $X_1,\ X_2, \ldots, X_b$ involving arithmetic operations, and G is a list of literals.

Let Π' be a program obtained from Π by replacing Q with a predicate P over variables $X_1,\ X_2, \ldots, X_b$, and extended with all facts $P(x_1, x_2, \ldots, x_b)$ such that

all literals in the list Q hold when $X_1, \ldots, X_b$ are replaced with $x_1, \ldots, x_b$, respectively. Here, we use $P(\mathbf{x1}), P(\mathbf{x2}), \ldots, P(\mathbf{xk})$ to denote all satisfying facts of the form $P(x_1, x_2, \ldots, x_b)$.

$$\begin{aligned} H \ \leftarrow \ & \bigwedge_{1 \leq i \leq b} F(X_i, Y), \ P(X), G(Y). \\ & P(\mathbf{x1}).P(\mathbf{x2}).\ldots.P(\mathbf{xk}). \end{aligned} \tag{5.2}$$

Theorem 2. *The programs* Π *and* Π' *have the same answer sets modulo ground atoms of the form* $P(\boldsymbol{x})$.

Proof. Assume in Π, we have a ground instance r corresponding to rule (5.1), and let $F(x1, y1)$, $F(x2, y1)$, ..., and $F(xb, y1)$ be the atoms in the body of r for the ground instance of $F(\mathbf{X}, y1)$. There always exist two ground rules r' and r'' in Π', obtained from the first and second line of the form (5.2)) respectively, using the same variable instantiation to produce r. An interpretation I of Π is an answer set of program Π if and only if $I \cup J$ is an answer set of Π', where J consists of all ground atoms provided by the predicate $P(\mathbf{x})$.

□

5.1.2 Schur number

To further demonstrate the potential of such rewritings, I consider next the Schur problem. Schur's theorem states that for any partition of the positive integers into a finite number of parts, one of the parts contains three integers x, y, z with $x+y = z$.[2] This theorem ensures that given any positive integer k, we can always find the smallest number $S(k)$ (Schur number) so that for any partition of the integers from 1 to $S(k)$ into k parts, at least one part contains x, y, z satisfying $x + y = z$. Yet there is another definition of the Schur number also prevalent in the literature, where Schur

[2] `https://en.wikipedia.org/wiki/Schur%27s_theorem`

number $S(k)$ is defined as the largest integer for which the integers from 1 to $S(k)$ can be partitioned into k parts, with no part containing x, y, z with $x + y = z$ [15]. In ASP, the problem of finding Schur numbers can be easily implemented by constraints. I first set a constraint requiring that no part containing x, y, z with $x + y = z$, and then gradually increase n so that I find the first n that makes the problem UNSAT. Fredricksen and Sweet [15] proved the lower bound for $S(6)(\geq 538)$ and $S(7)(\geq 1682)$. My experiment is focused on $S(8)$. The encoding of the Schur number problem can be modeled in the following way

```
number(1..n).
part(1..8).
{partition(X,Y) : part(Y)}=1 :- number(X).
:- partition(X,P), partition(Y,P), partition(X+Y,P),
    X<=Y, number(X),number(Y),part(P).
```

The first two rules state the range of numbers, which is from 1 to n, and the range of parts to partition the numbers. The third rule assigns each number to only one part. The last one is a constraint, stating that any part should not contain three values X, Y, and $X+Y$. The result is the correct partition of the numbers into eight parts and neither contains the sum relation. Otherwise, there will be no result if it is unsatisfiable, and then the largest number n to split is found.

Based on the experience of the Pythagorean triple problem, we can adopt the idea of predicate introduction to rewrite the encoding above. I pre-calculate a list of triples (x, y, x) so that $x + y = z$ outside of the grounder and integrate the result with an appropriately modified AS program rule remodeling the problem. Here, the precomputed predicate *trisum* is calculated outside the ASP program and used to replace the original occurrence of the arithmetic equality atom.

The following rule is used to replace the last rule of the Schur number encoding above

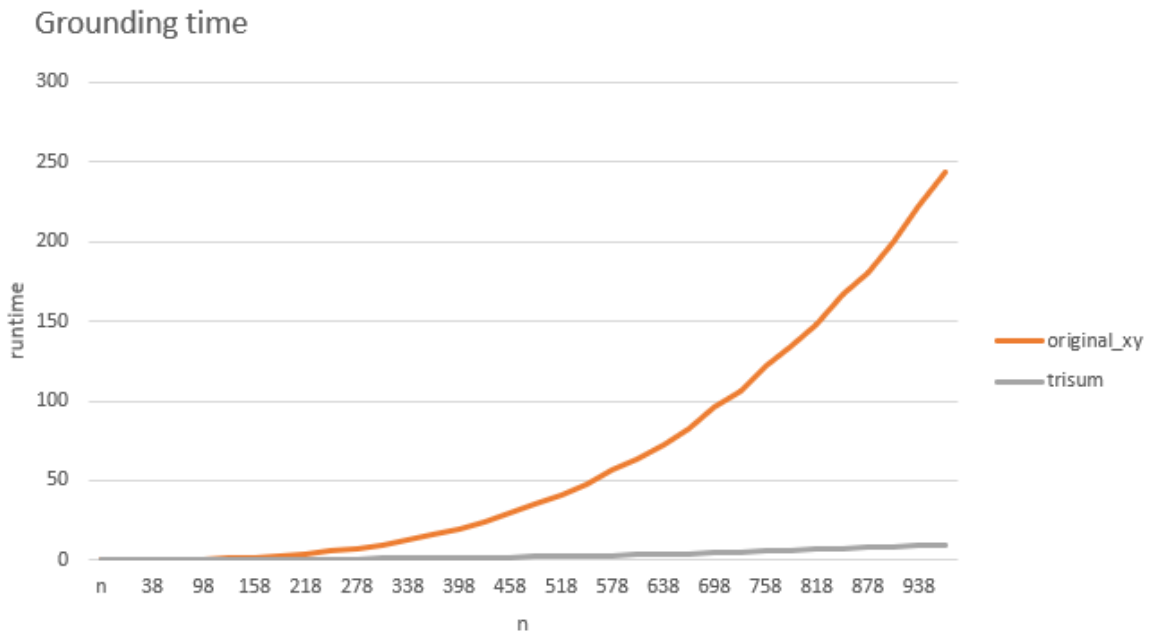


Figure 5.4: Grounding time of Schur encodings (original vs trisum)

```
:- partition(X,P), partition(Y,P), partition(Z,P),
    trisum(X,Y,Z), part(P).
```

Notice that we use the new predicate *trisum* to replace the square sum operation. With these pre-calculate predicates that satisfy $X + Y = Z$, the results of the sum relationship are passed as facts into the ASP system. Since there is no need to calculate the sum operation, we expect the grounding time would reduce accordingly. Indeed, the grounding time comparison of original encoding and new encoding (I call trisum) is shown in Figure 5.4. We observed that the grounding time reduces dramatically as a result of the introduction of the pre-calculated trisum predicate. The grounding time of the original encoding increases dramatically from 0s to 250s as the size n grows from 38 to 938. On the other hand, the grounding time of the trisum encoding does not change too much over time and is always less than 10s. Need to notice that the time to predicate *trisum*s outside the ASP problem using python is less than 0.2s for any n in the Figure 5.4.

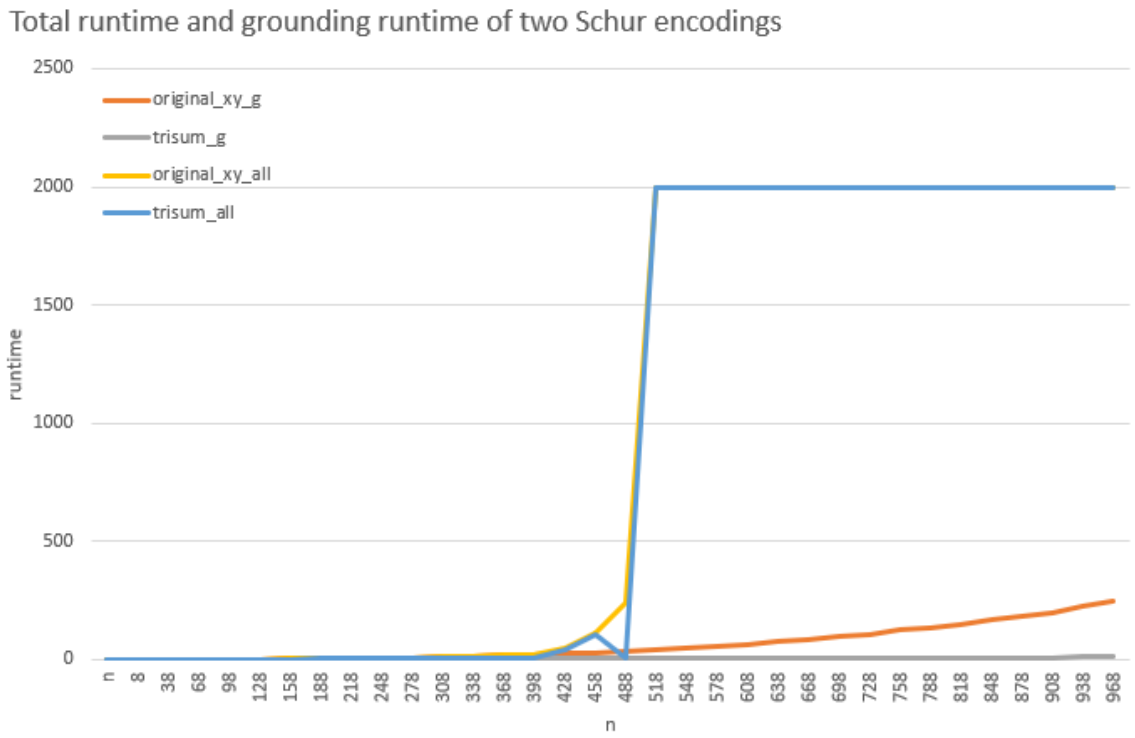


Figure 5.5: Total runtime compared with grounding time of two Schur encodings

The result of aggregate introduction on the Schur number problems is not as promising as the Pythagorean triple problems, mainly because solving time dominates grounding time when n is large. As is shown in Figure 5.5, the total runtime (grounding and solving together) grows much faster than the grounding time for any of the two encodings. While it does not help with solving, our results in the Figure 5.4 showed the grounding process gets much faster. When n is 518, no encoding can solve the Schur number problem within 2000s even if the grounding time is less than 50s for either encoding.

I note that while our work clearly shows that in many cases such rewritings will dramatically shorten the grounding time, they are not yet implemented. The key barriers are: the detection of arithmetic atoms amenable to pre-computation and the automated generation of code for the computation of tuples of variables for which these atoms are true.

5.2 Encoding Rewriting by Aggregates Introduction

Another way to generate equivalent encodings is through encoding rewriting automation. One can also write one or more encodings, and then use automated encoding rewriting tools based on principles such as projection and aggregate rewriting to generate a group of equivalent encodings. We hope to generate as many encodings as possible so that a large runtime diversity can be discovered and then exploited by the encoding selection method. When tested on a set of instances, some of these equivalent encodings are more efficient than others, and a set of efficient encodings are selected to perform encoding selection. Here the term *efficient* is defined in terms of a set of encodings, a set of instances, and some criteria such as run time and solving percentage. An encoding is considered efficient among a set of encodings when it provides the best solving time for some instances, or when it solves (terminates before the time out) much more instances than other encodings when processed by a grounder/solver system. On the other hand, encodings that never work as the best solution for any instance and only solve a small portion of instances are considered inefficient. I discuss how to select efficient encodings in Chapter 6.4

Here, I discuss automated encoding rewriting techniques and software I developed. These techniques are based on using the counting aggregate and exploiting methods to eliminate it. My work expands an earlier system AAgg [12]. The original AAgg worked by rewriting rules into ones that used the aggregate *count*. My work expanded the scope of input formulas to that older system. In addition, I also designed and implemented the reverse process of eliminating the aggregate. For example, when modeling a rule representing a constraint that each node can get at most one color in graph coloring problems, one can use the following rule (it is impossible for any node N to be colored with two different colors $C1$ and $C2$)

```
:- colored(N,C1), colored(N,C2), C1<C2.
```

Alternatively, one can also use a rule in which the same constraint is expressed by a counting aggregate:

```
colored_aagg(N) :- colored(N,C).
:- 2<= #count{C: colored(N,C)},  colored_aagg(N).
```

These two graph coloring encodings are represented in Appendix 9.B as enc1 and enc2 respectively. By introducing or eliminating counting aggregates, my automated encoding rewriting tool can convert one of such inputs into another to provide new candidate encodings. Although it is not guaranteed that certain types of rewrites perform better than others, my experiments show that typically they produce a family of equivalent encodings with complementary performance when tested on a specific instance set (see performance data of enc1 and enc2 in Table 7.2).

The original AAgg supports one input form and three output forms. It can be used to introduce counting aggregates when the original rules contain variables that are explicitly counted. Specifically, the input form is a rule that expresses a constraint that there are b different objects with a certain property by explicitly introducing b different variables to name these objects. The outputs generated by the original AAgg model have the same property by relying, in some way, on a counting aggregate [12].

Based on the original work of AAgg, I extended the scenarios where the counting aggregate can be used and also implemented new features to eliminate counting aggregates. Now I explain the input and output forms of the version of AAgg as extended by my work.

The first input form for the (expanded) AAgg is the original input form, specified by the work by Dingess and Truszczynski [12]:

$$H \leftarrow \bigwedge_{1 \leq i \leq b} F(X_i, Y), \bigwedge_{1 \leq i < j \leq b} X_i \neq X_j, G, \tag{5.3}$$

where

- H is the head of a rule (it can be empty to express a constraint).

- $X_1, X_2, \ldots, X_b$ are variables. They are not necessarily the first variables in F, but all should be in the same position in all occurrences of the predicate F in the rule.
- Y is a list of variables other than $X_1, X_2, \ldots, X_b$.
- F is a predicate with arity $1 + |Y|$.
- G is a list of atoms and can be empty.

In order to use AAgg, the following properties must be satisfied:

- $b > 2$.
- H and G have no occurrences of variables $X_1, X_2, \ldots, X_b$.
- The conjunction of terms $\bigwedge_{1 \leq i \leq j \leq b} X_i \neq X_j$ can be replaced by a conjunction of terms involving a continuous chain of comparisons $<$ or $>$ and other forms that are logically equivalent, such as $X_i + a \neq X_j + a$, where a is an integer.

The first output form deals with the case when Y is empty. The output form is

$$H \leftarrow b \leq \#count\ \{X : F(X)\},\ G. \tag{5.4}$$

- H, b, F, and G are the same as in the form (5.3)
- $\#count$ is the introduced aggregate. The aggregate element is $X : F(X)$, where X is a tuple of variables and $F(X)$ is a literal.

For example, the following rule satisfies the conditions AAgg checks to determine if a rewriting can be applied:

```
:- q(Y), q(Z), Y < Z.
```

Comparing the form (5.3) with the rule above, we can see here

- the head of the rule H is empty, meaning the rule is a constraint.
- $F(X_i, Y)$ is now the predicate q with arity 1; moreover, Y is empty.
- Variables $X_1, X_2, \ldots$ are now Y and Z. They appear in the same position in all occurrences of q.
- $b = 2$ as the variables used to implement counting are Y and Z.
- G is empty.
- H and G have no occurrences of Y and Z.

The rule above states that an occurrence of two different ground atoms $q(x)$ and $q(z)$ in an answer set is a contradiction. That is to say, the number of values in the extension of the predicate q cannot be greater than one. The first output form AAgg generates for this input is

```
:- 2 <= #count { Y : q(Y) }.
```

that expresses the same constraint using the aggregate *#count*.

The example above explains how an AAgg rewriting tool is used to rewrite a rule containing predicates with an arity one. When the list Y in $F(X_i, Y)$ is not empty, the arity of F is greater than one. In such cases, a projection will be performed to project out X. The output form has two rules in the form

$$\begin{aligned} H \ &\leftarrow \ b \ \leq \ \#count \ \{X : F(X, Y)\}, \ G, \ F'(Y). \\ F'(Y) \ &\leftarrow \ F(X, Y). \end{aligned} \tag{5.5}$$

- H, b, F, and G are the same as in the form (5.3)
- *#count* is the introduced aggregate. The aggregate element is $X : F(X, Y)$, where X is a tuple of variables and $F(X, Y)$ is a literal.

- F' is the predicate generated by applying a projection. The arity is equal to the size of Y.

An example of such a situation is the rule of the form

```
:- u(X,Y), u(X',Y), X < X'.
```

Comparing the form (5.3) with the rule above, we see that here

- the head of the rule H is empty, meaning the rule is a constraint.
- $F(X_i, Y)$ is now the predicate u with arity 2 and the list Y consists of one variable denoted here, with some abuse of notation, by the same symbol Y.
- Variables $X_1, X_2, \ldots$ are now X and X'. They appear in the same position in all occurrences of u.
- b = 2 as the variables used in counting are X and X'.
- G is empty.
- H and G have no occurrences of variables.

The rule above is rewritten by AAgg into two rules. The first rule projects away variable X defining a new predicate that collects all relevant values of Y, that is, those values of Y that together with some values of X are in the extensions of u. The projection creates a new auxiliary predicate *u_aagg*. The second rule enforces that the number of values of X that appear in the extension of u with any relevant Y is at most 2. To summarize, the rule

```
:- u(X,Y), u(X',Y), X < X'.
```

is replaced with the following two rules.

```
u_aagg(Y) :- u(X,Y).
:- 2 <= #count{ X: u(X,Y) }, u_aagg(Y).
```

The correctness of the aggregate introduction was proved in the original paper by Dingess and Truszczynski. We first recall the theorem concerning new predicate introduction.

Theorem 3. *Let* Π *be a program containing a rule of the form*

$$H \leftarrow \bigwedge_{1 \leq i \leq b} F(X_i, Y),\ Q(X),\ G. \tag{5.6}$$

where H *is the head of a rule and can be empty to express a constraint,* X *is a tuple of variables* $X_1, X_2, \ldots, X_b$*,* Y *is a tuple of variables disjoint with* X*,* F *is a predicate,* $Q(X)$ *is a list of literals over variables* $X_1, X_2, \ldots, X_b$*, and* G *is a list of literals. Moreover, we assume that* H *and* G *contain no variables from* X*.*

Let Π' *be a program obtained by replacing rule (5.6) with the following ones*

$$\begin{aligned} H &\leftarrow \bigwedge_{1 \leq i \leq b} F(X_i, Y),\ Q(X),\ G,\ F'(Y). \\ F'(Y) &\leftarrow F(X, Y). \end{aligned} \tag{5.7}$$

where F' *is a new predicate symbol not occurring in* Π*.*

The programs Π*(5.6) and* Π'*(5.7) have the same answer sets modulo ground atoms of the form* $F'(y)$*.*

Next, we recall the aggregate equivalency theorem proved by Lierler[39].

Theorem 4. *Let* Π *be a program containing a rule of the form*

$$H \leftarrow \bigwedge_{1 \leq i \leq b} F(X_i, Y),\ \bigwedge_{1 \leq i < j \leq b} X_i \neq X_j,\ G, \tag{5.8}$$

where H *is the head of a rule and can be empty to express a constraint,* G *is a list of literals,* $X_i, X_2, \ldots, X_b$ *are variables,* Y *is a tuple of variables, each different from* X *and each with at least one occurrence in a literal in* G*, and* F *is a predicate with arity* $1 + |Y|$*.*

Let Π' *be a program obtained by replacing the rule (5.8) with*

$$H \leftarrow b \leq \#count\ \{X\ :\ F(X, Y)\},\ G. \tag{5.9}$$

where H, b, F, and G are the same as in the form (5.8), and #count is the introduced aggregate. The aggregate element is $X : F(X,Y)$, where X is a tuple of variables and $F(X,Y)$ is a literal.

The programs Π(5.8) and Π'(5.9) are strongly equivalent if b is an integer, and H and G contain no X .

With these two theorems, the correctness of the first form of the AAgg rewriting follows.

Theorem 5. *Let Π be a program containing a rule of the form*

$$H \leftarrow \bigwedge_{1 \leq i \leq b} F(X_i, Y), \bigwedge_{1 \leq i < j \leq b} X_i \neq X_j, G, \tag{5.10}$$

where the definition of X, Y are the same as the rule (5.6) and $\bigwedge_{1 \leq i < j \leq b} X_i \neq X_j$ is a replacement to $Q(x)$. If Y is empty, the rule (5.10) is equivalent to the rule of the form

$$H \leftarrow b \leq \#count\ \{X : F(X)\}, G. \tag{5.11}$$

Otherwise, it is equivalent to the rules

$$\begin{aligned} H &\leftarrow b \leq \#count\ \{X : F(X,Y)\}, G, F'(Y). \\ F'(Y) &\leftarrow F(X,Y). \end{aligned} \tag{5.12}$$

If Y is empty, rule (5.10) can be directly replaced by the corresponding rule in the form (5.9) with aggregates, and the resulting form is rule (5.11). By Theorem 4, rule (5.10) and rule (5.11) have the same answer set.

Otherwise, rule (5.10) can be replaced by two rules in the form (5.7), the first of which then can be replaced by the aggregate form (5.9), and the resulting form is a set of rules (5.12). By Theorem 3 and Theorem 4, rule (5.10) and the set of rules (5.12) have the same answer set.

The second input form of AAgg is an extension of the form (5.3):

$$H \leftarrow \bigwedge_{1 \leq i \leq b, 1 \leq k \leq b} F(X_i, Y_k), \bigwedge_{1 \leq i < j \leq b} X_i \neq X_j, \bigwedge_{1 \leq p < q \leq b} Y_p = Y_q, G, \tag{5.13}$$

where

- H, F, X_i, and G have the same meaning as in the form (5.3).
- Y_k is a list variables other than X in F and appear in the same positions in every occurrence F.

Compared with form (5.3), where the same list Y of variables was used with all occurrences of F in the rule, this form allows different lists of Y_k in F as long as these lists are the same and their variables occur in the same positions in the atoms defined by the occurrences of F.

To illustrate, I will consider the *n-queens* problem. The n-queens problem asks for an assignment of n queens in a $n \times n$ chessboard so that no two queens attack each other. A placement of a single queen on the board can be modeled by an atom $queen(X_i, Y_i)$, where variables X_i and Y_i represent row and column of the placement cell, respectively. A constraint that there are no two queens in the same column can be modeled by a rule:

```
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, Y1=Y2.
```

Here even though the predicate *queen* contains different variables Y_i in the second position, aggregate introduction is applicable, as these variables are equal and appear in the same position in all occurrences of *queen*. Rewriting the rule above involves two steps. First, we remove all equality operators and replace the involved variables with one variable. Then we check if the new rule accords with form (5.3) to decide whether an aggregate rewriting can be applied.

By removing equality operators on Y and replacing all Ys with the same variable Y', we generate an intermediate rule:

$$H \leftarrow \bigwedge_{1 \leq i \leq b} F(X_i, Y'), \bigwedge_{1 \leq i < j \leq b} X_i \neq X_j, G, \tag{5.14}$$

In the example above, removing equality operators yields the rule

```
:- queen(X1,Y1), queen(X2,Y1), X1 < X2.
```

We notice the new rule has the same form as the one we considered in the previous example. Thus, aggregate introduction is possible. The final outcome of rule rewriting in this case consists of the two new rules as discussed above.

Theorem 6. *Let* Π *be a program containing a rule* r *of the form (5.13),* Π' *be a program obtained by rewriting the rule* r *into* r'*, through removing equality operators on* Y *and replacing all* Y*s with the same variable* Y' *in the form (5.14), and* Π'' *be a program applying an AAgg rewriting of any possible form on the rule* r'*. The programs* Π*,* Π'*, and the corresponding AAgg rewriting form* Π'' *and have the same answer sets.*

Proof. A ground instance of the rule (5.13) is not included in the grounding of the program Π if any Y_i and Y_j, $i \neq j$, are replaced with different constants. Thus, the only ground instances of the rule (5.13) considered for inclusion in the grounding of Π are rules of the form $F(x1, y), F(x2, y), \ldots, F(xb, y)$. This set of rules is the set of ground instances of the rule (5.14), when constructing the grounding of Π'. Thus, the grounding of Π and Π' are the same. So the programs Π and Π' have the same answer sets. Since Π' fits the input form (5.3), it can be rewritten by introducing aggregates. The resulting Π'' has the same answer sets as Π'. As a result, Π, Π', and Π'' have the same answer sets. □

I expanded the scope of usage of AAgg further. Sometimes more predicates are involved in the definition of a property for which a bound on the number of different values in its extensions is imposed. More formally, instead of a single predicate F, we have in its place a conjunction of predicates. A rule given below illustrates this situation in the case of the conjunction of two predicates:

$$H \leftarrow \bigwedge_{1 \leq i \leq b} F(X_i, Y),\ E(X_i, Y),\ \bigwedge_{1 \leq i < j \leq b} X_i \neq X_j,\ G., \tag{5.15}$$

where

- H, F, X_i, Y, and G have the same meaning as in the form (5.3).
- E is a predicate sharing the same variables as F. For each i, the corresponding variable X_i and the variables in Y appear in the same positions in both $E(\cdot), F(\cdot)$.

This extension does not fall under the scope of the form (5.3). However, I propose to deal with F and E as a group and apply aggregate introduction to the group. There are two corresponding output forms depending on Y:

If Y is empty, the rule (5.15) is equivalent to

$$H \ \leftarrow \ b \ \leq \ \#count \ \{X \ : \ F(X), E(X)\}, \ G. \tag{5.16}$$

Otherwise, it is equivalent to the rules

$$\begin{aligned} H \ &\leftarrow \ b \ \leq \ \#count \ \{X \ : \ F(X,Y), E(X,Y)\}, \ G, \ D'(Y). \\ D'(Y) \ &\leftarrow \ F(X,Y), E(X,Y). \end{aligned} \tag{5.17}$$

To explain my approach, let us consider the rule of the following form

```
:- p(X,Y),p(X',Y),q(X,Y),q(X',Y),c(Y),X < X'.
```

The rule above bounds the number of values of X's that may occur with the same value of Y in atoms p and q. We cannot perform aggregate introduction to either one of the two predicates as the other one also contains X, which does not meet the requirement of form (5.3). However, we can project out X from the conjunction of $p(Y,X)$ and $q(Y,X)$, and then use it to introduce the counting aggregate (note that the counting aggregate in ASP allows us to use conjunctions of atoms). The resulting rewriting has the following form

```
u_proj_X'(Y) :- p(X,Y), q(X,Y).
#false :- c(Y); u_proj_X'(Y); 2 <= #count{X: p(Y,X), q(Y,X)}.
```

Theorem 7. *Let* Π *be a program containing a rule* r *of the form (5.15),* Π' *be a program obtained by rewriting the rule* r *into* r' *with the corresponding AAgg rewriting form (5.16) (when* Y *is empty) or form (5.17)) (when* Y *is not empty). The programs* Π *and the rewriting form* Π' *have the same answer sets.*

Proof. In order to prove the correctness, let Π'' be a program containing a rule grouping predicates F and E together and renaming the new predicate D,

$$H \leftarrow \bigwedge_{1 \leq i \leq b} D(X_i, Y), \bigwedge_{1 \leq i < j \leq b} X_i \neq X_j, G., \tag{5.18}$$

where $D(X_i, Y)$ is a replacement for $F(X_i, Y), E(X_i, Y)$. Since the rule (5.18) fits the input form (5.3), we can apply aggregate introduction to Π'' and the output form Π''' has two types:

If Y is empty, the rule (5.18) is equivalent to

$$H \leftarrow b \leq \#count\{X : D(X)\}, G. \tag{5.19}$$

Otherwise, it is equivalent to the rules

$$\begin{aligned} H &\leftarrow b \leq \#count\{X : D(X, Y)\}, G, D'(Y). \\ D'(Y) &\leftarrow D(X, Y). \end{aligned} \tag{5.20}$$

Since $D(X_i, Y)$ is a replacement for $F(X_i, Y), E(X_i, Y)$, we see that the rule (5.19) and (5.20) are exactly the same as rule (5.16) and (5.17), and thus Π''' is exactly the same as Π', so Π''' and Π' have the same answer sets. Since the AAgg output form Π''' and the original form Π'' have the same answer sets, Π''', Π'', and Π' have the same answer sets. Now we need to prove Π'' and Π have the same answer sets. Consider a ground instance of the rule r in Π and check the first two groups of atoms in the body, that is, $F(x1, y1)$ and $F(x2, y1)$, and $E(x1, y1)$ and $E(x2, y1)$. For each y, different x variables must appear in both predicate F and E at the same time. It is easy to check all $D(X_i, Y)$ in the ground instance of Π'' only contain these x and y that makes $F(x1, y1) \wedge F(x2, y1) \wedge E(\ x1, y1) \wedge E(x2, y1)$ true. As a result,

ground(Π) and ground(Π″) are the same, so the program Π and Π″ have the same answer sets. As result, Π, Π′, Π″, and Π‴ all have the same answer sets. □

The new AAgg also supports removing the count aggregate atoms from rules. These input forms are consistent with the outputs of AAgg when it introduces aggregates. Therefore, aggregate removing applies to any rules that can be syntactically matched with any of the output forms. Specifically, these input forms are

$$H \ \leftarrow \ b \ \leq \ \#count \ \{X \ : \ F(X,Y)\}, \ F'(Y), \ G. \tag{5.21}$$

$$H \ \leftarrow \ not \ \#count \ \{X \ : \ F(X,Y)\} \ < \ b, \ F'(Y), \ G. \tag{5.22}$$

$$\begin{aligned} H \ \leftarrow \ & not \ \#count \ \{X \ : \ F(X,Y)\} \ = \ 0, \ \ldots, \\ & not \ \#count \ \{X \ : \ F(X,Y)\} \ = \ b-1, \ F'(Y), \ G. \end{aligned} \tag{5.23}$$

To prove the correctness of removing aggregates, we first consider the first form (5.21) above.

Theorem 8. *Let* Π *be a program containing a rule* r *with the aggregates form (5.21), and* Π′ *be a program obtained by replacing* r *by a normal rule*

$$H \ \leftarrow \ \bigwedge_{1 \leq i \leq b} F(X_i,Y), \ \bigwedge_{1 \leq i < j \leq b} X_i \neq \ X_j, \ F'(Y), \ G, \tag{5.24}$$

The aggregate program Π *and the corresponding program* Π′ *with normal rules have the same answer sets if* b *is an integer, and* H *and* G *contain no* X*.*

Proof. According to Theorem 4, the program with aggregates and the corresponding normal rules have the same answer sets, and it is also true the other way around. □

Since my work also supports the replacement of one aggregate form to another, now I prove the correctness of such cases.

Recall that we defined the splitting of a program into two parts when there is no predicate appearing in the head of a rule from another in definition 4. According to the definition, a program Π can be split into a program Π' and a program Π'', where Π'' is a collection of rules such that no predicate symbol in the head of a rule in Π'' appears in Π' (hence, $\Pi = \Pi' \cup \Pi''$). Then an interpretation S of Π' is an answer set of Π if and only if S extended by all ground atoms in the heads of ground instances of rules in Π'' whose body is satisfied in S. We use this to prove the correctness of the replacement of aggregate forms.

Theorem 9. *We split a program P with aggregate into an aggregate rule P'' and the other part P' so that $P = P' \cup P''$ and no predicate symbol in the head of a rule in P'' appears in P'. Then we replace the aggregate rule with its classically equivalent form Q'', and the resulting program $Q (Q = P' \cup Q'')$ has the same answer sets as P.*

Proof. For the program P where $P = P' \cup P''$, it is clear that the answer set of P is the answer set of P' extended by all ground atoms in the heads of ground instances of rules in P'' whose body is satisfied in the answer set of P' . Let assume the answer set of P is I, the answer set of P' is I', and the ground instances of rules in P'' is $ground(P'')$. Now I is the set I' extended by satisfaction result of $ground(P'')$ through simplifying the bodies of the rules in $ground(P'')$, that is, $I = I' \cup ground(P'')$. When replacing the P'' with its classically equivalent from Q'', the resulting $ground(Q'')$ is the same, that is, $ground(P'') = ground(Q'')$. As a result, the answer set to the new program $Q(=P' \cup Q'')$ is $I' \cup ground(Q'')$, and equal to $I' \cup ground(P'')$, and consequently, equal to I. □

To summarize, the new AAgg rewrites by introducing the aggregate count for rules of three input forms (as opposed to just one originally), and by removing the aggregate from rules of three additional input forms.

The main tool to support generating rewriting automatically is the concept of a parse tree that is built for each ASP rule based on its structure. I use *clingo* python API to analyze the structure and perform automate rewriting. There are several key steps to the rewriting process: checking, parsing, preprocessing, processing, and saving.

1. Checking. Check if the name of file ends with '_aagg' to avoid repeating rewriting. An encoding ending with '_aagg' is treated as an output file and no further step is processed.

2. Parsing. All encodings are parsed by the clingo API, then corresponding abstract syntax trees are generated for each rule.

3. Preprocessing. General information are collected for the program. For example, the program uses predicate dependencies to determine how many output forms are appropriate.

4. Processing. Rules are processed one by one to generate its equivalence form according to information reflected in its abstract syntax tree. Here, an equivalence transformer class traverses the abstract syntax tree to check if a rule contains aggregate or not. In each case, comparisons variables, comparison predicates, and counting numbers are recorded and analyzed to figure out important information related to rewriting, such as x, y, F, b. Then some verification steps are executed to check if the rewriting is possible. For example, it verifies the counting variables are not used anywhere else excluding the predicates using them, or the location of variables appears in the fixed location of the counting predicates. If any verification fails no further step will process and no rewriting is performed on the rule. If verification succeeds, the user required corresponding rewriting form is generated. The program also checks for available new names for auxiliary predicates when a predicate projection is needed.

enc	better10%	worse10%
ham3-dup-rule1	40%	32%
ham3-dup-rule2	22%	17.5%
ham3-dup-rule3	0	0
ham3-dup-rule4	0	0
ham3-dup-rule5	0	0

Table 5.1: Single rule duplication to Hamiltonian cycle encoding 3

5. Saving. The program saves the corresponding rewritten encoding to a new file, appending '_aagg' to its original file name.

5.3 Encoding Rewriting by Structure Modification

An easy way to modify a given encoding is to repeat some or all its rules. I tested the effectiveness by repeating all rules of encodings to Hamiltonian cycle, snake, and graceful graph problems. The results show that almost all generated encodings perform differently when tested on a large set of instance sets. These new encodings exhibit performance advantage over the originals on a portion of instances. A further investigation revealed that not all repeated rules affect the runtime of the encodings. This observation lead to an approach to encoding rewriting in which one finds rules that influence the performance and generates new encodings by repeating one, some or all of the "influential" rules. I performed a series of experiments by repeating only one rule of the original encoding. I investigated the encodings that show performance diversity from the original encodings and found that most of the diversity comes from encodings with repeated choice rules. This suggests that given one encoding I could first check if choice rules are involved and then simply repeat each relevant rule to generate a group of new encodings.

Table 5.1 summarizes the results of different single rule duplication to Hamiltonian cycle encoding *ham3* (see Appendix 9.A), when tested on a specific group of

Hamiltonian cycle instances[3]. The first column lists the new encodings, identified by the rule in the original encoding that is duplicated. For example, ham3-dup-rule1 is the encoding obtained from the encoding ham3 by duplicating the first rule in this encoding. The second column shows the percentage of instances with which the new encoding performs 10% better than the original one, that is, the runtime of clasp on the program using the original encoding is 10% better than when the new encoding is used. The third column shows the percentage of instances with which the new encoding performs 10% worse than the original one, that is, the runtime of clasp on the program using the original encoding is 10% worse than when the new encoding is used. In each case, the duplicated rule was a choice rule. We can see from table that the encoding ham3-dup-rule1 obtained by duplicating rule1 in the encoding ham3 solves 40% of the instance set at least 10% faster than the original, and ham3-dup-rule2, obtained by duplicating rule2 in the encoding ham3, solves 22% of the instance set at least 10% faster than the original. An observation is that the common property is these two rules are all choice rules. The results show that single rule duplication on a choice would produce an encoding perform better than the original on some instances. At the same time, we also note the duplication nevertheless produce much worse results on other instance. However, we are only interested in the performance improvement the duplicated encodings provide on some instances. The performance improvement can be utilized with the help of the encoding selection tool we discuss later so that the best encoding is selected on a per-instance basis.

A similar result were observed from the duplication experiment on encodings of the graceful graph problem. Table 5.2 summarizes the results of different single rule duplication applied to the graceful graph problem encoding graceful1, when tested on a specific group of graceful graph instances. We can see the new encoding graceful1-dup-l6 solves 35.5% of instances 10% better than the original, and it is the

[3] `https://drive.google.com/drive/folders/1DAiCQmsmrDmDJ8N3nsJ3C8YDNhddoRCX?usp=sharing`

enc	better10%	worse10%
graceful1-dup-l1	0	0
graceful1-dup-l2	0	0
graceful1-dup-l3	0	0
graceful1-dup-l4	0	0
graceful1-dup-l5	0	0
graceful1-dup-l6	35.5%	59.5%
graceful1-dup-l7	0	0
graceful1-dup-l8	0	0
graceful1-dup-l9	0	0
graceful1-dup-l10	0	0
graceful1-dup-l11	0	0

Table 5.2: Single rule duplication to graceful graph encoding 1

enc	better10%	worse10%
snake-dup-l1	0	0
snake-dup-l2	0	0
snake-dup-l3	34%	59%
snake-dup-l4	34%	60%
snake-dup-l5	0	0

Table 5.3: Single rule duplication to snake encoding

only encoding that perform much differently than the original one. It is important to stress that not all duplicated choice rules cause result in significantly different performance compared with the original. The encoding graceful1-dup-l7 in this table is also derived from a choice rule, but it does not result in any at least 10% better or worse performance.

Rule duplication also worked for the encodings of the snake problem (see Table 5.3). The rule duplications on rule 3 and rule 4 both help solve 34% of instances 10% better than the original and they are all choice rules.

I also experimented with encodings obtained by repeating rules three times. In all of the experiments, the resulting encodings performed worse than the duplication one. Table 5.4 list all the performance gain and loss of single rule triplication compared with corresponding duplication for snake encodings. For exmple, the encoding snake-triple-l2 is obtained by repeating rule 2 of the snake encoding 3 times. When

enc	better10%	worse10%
snake-triple-l0	0	0
snake-triple-l1	0	0
snake-triple-l2	0	32%
snake-triple-l3	0	43%
snake-triple-l4	0	0
snake-mt-triple-l0	0	0
snake-mt-triple-l1	0	0
snake-mt-triple-l2	0	37%
snake-mt-triple-l3	0	0
snake-mt-triple-l4	0	0
snake-mt-triple-l5	0	0
snake-rew-triple-l0	0	0
snake-rew-triple-l1	0	0
snake-rew-triple-l2	0	28%
snake-rew-triple-l3	0	32%
snake-rew-triple-l4	0	0
snake-vl-rc-triple-l0	0	4%
snake-vl-rc-triple-l1	0	100%
snake-vl-rc-triple-l2	0	37%
snake-vl-rc-triple-l3	0	100%
snake-vl-rc-triple-l4	0	100%
snake-vl-rc-triple-l5	0	100%
snake-vl-rc-triple-l6	0	100%
snake-vl-rc-triple-l7	0	100%

Table 5.4: Single rule triplication compared with duplication to all snake encodings

compared with its corresponding duplication, the one repeating rule 2 of the snake encoding 2 times, snake-triple-l2 solves 0% of instances 10% better, but 32% of instances 10% worse. All the data shows the duplication does not provide better results than its corresponding duplication. Some are even a hundred percent 10% worse. As a result, the rule duplication is enough for rewriting an encoding in order to provide performance diversity to be used later by the encoding selection tool.

Chapter 6 Encoding Selection Platform

It has been known in the literature that different encodings exhibit complementary performance when tested on a set of instances, and typically there is no single best encoding for the problem being solved. My experimental results I discussed so far provide further evidence for this claim. Selecting the correct encoding for a given instance is a challenge for ASP programmers. In this chapter, I propose an approach to address the problem. The idea is to design a system that will maintain a set of encodings of the problem. Given an instance, the system will estimate the performance of each encoding in the set on that instance, and will use these predictions to select a single encoding for that instance, or to select several of them and run them according to some schedule. I will now discuss this approach in more detail.

The encoding selection platform I developed aims to automate the process of encoding-based optimization of ASP performance, from encoding rewriting, performance data generation, learning performance models, to encoding prediction and selection. Given a grounder/solver, a set of encodings of a problem (possibly consisting of just one encoding only), and a training set of instances, the system automatically generates additional encodings, selects some that promise good performance and jointly exhibit some additional properties such as complementarity of areas of good performance, and finally builds for each selected encoding its performance model. The model predicts for any instance the execution time that the solver will take to process the instance if that encoding is used. These performance models are then used to improve solving efficiency: whenever a new instance arrives, the system selects a way to solve the problem for this instance based on predicted run times for selected encodings. The system I developed supports two techniques of using sets of encodings and their performance models: *encoding selection* and *encoding portfolio*

solving. I call this system the encoding selection platform (ESP).

The ESP takes a set of encodings supplied by the user. It then generates several additional encodings. The system then collects the performance data for these encodings on a specific set of instances, also supplied by the user. Based on the performance data, the ESP selects a set of well-performing encodings with complementary areas of good performance. It then uses the performance data for the selected encodings to build for them their performance models by using machine learning techniques.

An important step in using machine learning techniques to build performance models is to extract features of instances. The system first uses *Claspre*[1] to extract static and dynamic features, and then combines domain-specific features to characterize instances. With data and features, ESP builds machine learning models which are then used to select the most promising encoding on a per-instance basis. The system builds machine learning models to learn the runtime of the provided solver on each encoding with any possible instance. The learnt models are used to estimate the runtime of the solver on a given instance on each of the encodings. These machine learning models are evaluated against other solutions, and when the machine learning model method is the winner, the platform relies on the predicted runtime of all encodings to solve new instances. The ESP extracts features, predicts the runtime for all encodings, and selects the encoding with the lowest predicted runtime to solve new instances.

In order to improve the robustness and reliability, the system also builds schedules running several encodings according to different time slots allocated to them. These schedules are used when encoding selection fails to work on a specific instance set.

Now, I describe the architecture of the ESP and explain each component in detail. I also present a case study to illustrate functions of the building blocks of the ESP architecture and its operation using Hamiltonian cycle (HC) problems. Throughout

[1] `https://potassco.org/labs/claspre/`

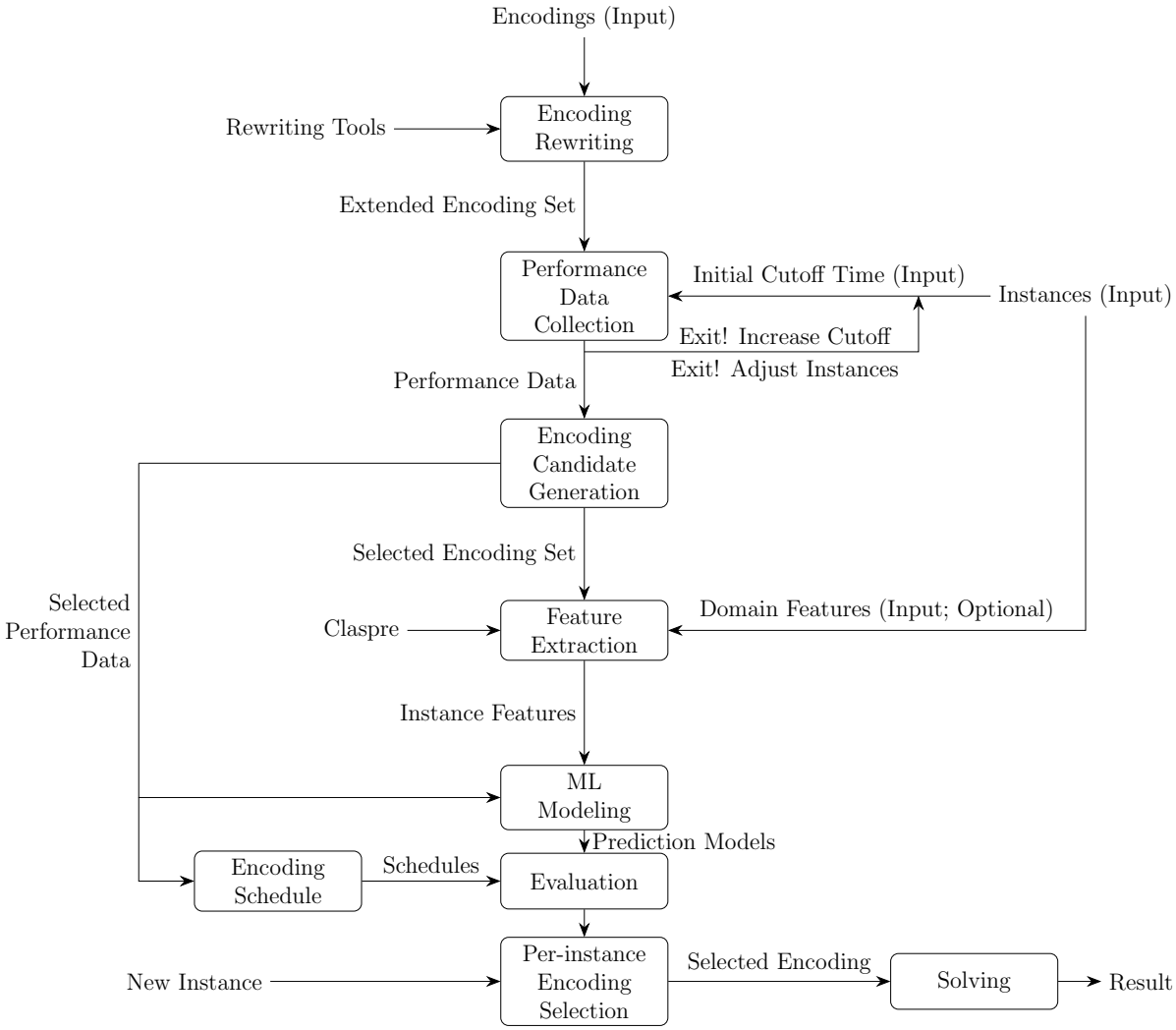


Figure 6.1: A flowchart to the encoding selection platform

my presentation I also present the insights and conclusions I arrived at while developing and using ESP. These insights offer empirical/practical tips in utilizing the introduced system by ASP practitioners. They can also help further advance the proposed technology in the future.

6.1 Platform Overview

The flowchart in Figure 6.1 shows the architecture and processes involved in the ESP. Occurrence of the word *Input* within the flowchart indicates input data and parameters that a user of the system must supply. In particular, the user must provide at least

one encoding for a problem to be solved, instances of this problem, and domain/application specific features if available. Components shown inside boxes denote processes implemented within the ESP platform. These include encoding rewriting, performance data collection, encoding candidate generation, feature extraction, machine learning modeling, per-instance encoding selection, and solving. Other annotations point at either outcomes of different processes or tools utilized by the system. The ESP uses such tools as encoding rewriting system *AAgg* (as extended in this thesis form the original design, see Section 5.2) and feature generator *claspre*[2] [19].

The ESP platform, a description of the system requirements, and instructions on how to use it are available for download.[3] Although the platform consists of several components, each part can be executed separately. Thus, users can either upload encodings and instances and run all the processes, or only run some selected ones.

In the remainder of this section, I review the key building blocks of the ESP architecture. I stress that the ESP platform is designed to assist the user with automatic improvement of the performance of an ASP-based solution to a problem at hand. In particular, it exploits the availability of distinct encodings for this problem. Improved performance means increased number of instances solved for an application and decreased time spent on these instances. The platform is general purpose and can be applied to arbitrary problems solved by means of ASP. However, any specific use of the ESP tool assumes a concrete problem at hand. In the narrative that follows I often use letter P to refer to a problem that the specific use of ESP targets.

6.2 Encoding Rewriting

Encodings The ESP expects the user to supply at least one ASP encoding for considered problem P. In most cases, the user will provide several encodings for the

[2] *claspre* is a sub-component of portfolio answer set solver *claspfolio*; it is available as a stand alone tool at `https://potassco.org/labs/claspre/`

[3] `http://www.cs.uky.edu/ASPEncodingOptimization/esp/`

problem. The supplied encodings are rewritten by means of an encoding rewriting tool AAgg available in the platform. The *extended set of encodings* (the input ones and these resulting from rewriting) are the basis for further processing that aims to select a subset of no more than six encodings that will be used when solving new instances of problem P. I comment on how performance data guides the selection of the subset of encodings implemented in ESP later in the thesis.

To show examples of possible input encodings that the user might supply to the ESP, I consider the graph coloring (GC) and the Hamiltonian cycle (HC) problems, two well-known application domains. The first encoding of the GC problem is presented in Listing 6.2. The lines starting with % are comments. The rule in line 2 forces each node to receive exactly one color; lines 4 and 5 ensures that no two adjacent nodes are colored the same. The second encoding of the GC problem is constructed from the one in Listing 6.2 by dropping its line 2 and including the rules presented in Listing 6.2. Thus, these two encodings differ in the way they implement the *chosencolor* relation. They also enforce differently the constraint that *each node is to be assigned exactly one color*. The first encoding uses a choice rule (line 2) to implement the requirement that each node is assigned exactly one color, while the second encoding uses two constraints to restrict that each node must be colored (line 5) and colored exactly once (line 7 and 8).

Listing 6.1: Graph coloring encoding 1

```
1 % Guess colors
2 {chosenColor(N,C):color(C)}=1:- node(N).
3 % No two adjacent nodes have the same color
4 :- link(X,Y), X<Y, chosenColor(X,C),
5                    chosenColor(Y,C).
6 #show chosenColor/2.
```

Listing 6.2: Part of graph coloring encoding 2

```
1 chosenColor(N,C) | notChosenColor(N,C):- node(N), color(C).
2 % At least one color per node.
3 colored(X):- chosenColor(X,_).
4 :- node(X), not colored(X).
5 % Only one color per node.
6 :- chosenColor(N,C1), chosenColor(N,C2), C1!=C2.
```

Listing 6.3: Hamiltonian cycle encoding 1

```
1 %Generator
2 { hcedge(X,Y) : link(X,Y) } =1:-node(X).
3 { hcedge(X,Y) : link(X,Y) } =1:-node(Y).
4 %Definition of reachability
5 reach(X) :- hcedge(1,X).
6 reach(Y) :- reach(X),hcedge(X,Y).
7 %test
8 :- not reach(X),node(X).
9 %show
10 #show hcedge/2.
```

The first encoding for the HC problem is shown in Listing 6.3. The first two rules model the requirement that the number of selected edges leaving and entering each node is exactly one. The rules in lines 5 and 6 define the concept of reachability from node 1. Constraint in line 8 guarantees that every node is reachable from node 1 by means of selected edges only. The second encoding for the HC problem is obtained by replacing line 5 in Listing 6.3 with the rule

```
reach(1).
```

Encoding rewriting tools Search and optimization problems typically admit many alternative equivalent encodings in ASP. Encoding rewriting tools are able

to detect if a different rewriting is possible and generate new encodings. These encodings may be syntactically different, but are semantically equivalent. When tested on a set of instances, these encodings always show varying performance. The ESP system is able to takes advantage of the varying performance of encodings and select the most effective encoding on a per-instance base. The *AAgg* system performs rewritings on non-ground programs. The current version of the ESP incorporates a rewriting tool *AAgg*. It is used to generate additional encodings based on the ones originally provided by the user. The original version of this system described by Dingess and Truszczynski [12] produced rewritings by discovering counting based rules that could be reformulated by means of cardinality aggregates. The present version, integrated into the platform, also supports rewritings that eliminate some cardinality aggregates. *AAgg* checks all encodings from the user input and generates new encodings if an aggregate related rewriting is possible. These new encodings together with all encodings from the user input are used to perform the following tasks, which include performance generation, encoding candidate selection, feature extraction, encoding selection, and schedule building. The detailed implementation is explained in Section 5.2.

6.3 Performance Data Collection

Instances Performance data reflects the effectiveness of different encodings on a given set of instances. To measure the performance of encodings on a set of instances, we run each encoding to solve the instances under a chosen ASP solving tool and record the running time. Sometimes the solving process can take 'forever', so a cutoff time is needed to terminate an unsuccessful run. As a result, if an encoding succeeds in solving an instance within the cutoff time, we record the real runtime; otherwise, we record the cutoff time as runtime and further process is needed to deal with the unsuccessful runs.

Benchmark instances must be provided by users. Benchmark instances are used to extract data on the performance of a solver on each of the selected encodings, to support feature extraction, and to form the training set used by machine learning tools to build encoding performance models. I discuss methods to generate problem instances in Section 7.2.

Users upload instances, encodings, and an initially estimated cutoff time to the platform, the platform aims to collect a meaningful performance data that show runtime diversity of different encodings. As a result, the instance set should not contain a large portion of instances that are either too easy or too hard. When a solver finds a solution to an instance in a short amount of time no matter what encoding is used, or when the solver times out no matter what encoding is used, the instance offers no insights that could inform encoding selection. Only instances that are not too easy and not too hard are meaningful. I will refer to such instances as *reasonably hard*, or just *hard*. So benchmark instances should be generated with care, and users may be requested to provide new data set according to the performance data collected by the platform.

The process of performance data collection works as follows. Users upload instances and encodings and set an initially estimated cutoff time. After encoding rewriting, the ESP starts to collect performance data of all encodings on the given instances set. It first estimates a suitable cutoff time by running all encodings on some randomly sampling instances. The ESP automatically increases cutoff time up to twice when most of the problems are extremely hard. When problems are still extremely hard after two adjustments, the ESP exits with the performance data and informs users to increase the initially estimated cutoff time according. If cutoff time is set correctly, The ESP collects performance data of the full instance set and verifies if the performance data is valid. The ESP only continues with a valid dataset. If the dataset is invalid, the ESP exits and informs users to provide a new instance set

referring to the collected performance data.

A dataset is *valid* when it contains a sufficient portion reasonably hard instances. The concept of a reasonably hard instance is determined by two parameters, the time T_e specifying when the execution time is long enough not to view an instance as easy, and the time T_{max} specifying the cutoff time. At present, the user inputs only the cutoff time T_{max}; the system then sets $T_e = T_{max}/7$. How to select the initial value of T_{max} depends on the capability of encodings, the available computing resources, as well as the time budget for solving incoming instances of the problem at hand.

Once the user provides the ESP with the initial set of instances, and the parameter T_{max}, and once the extended set of encodings is produced by rewriting, the ESP computes the performance data while automatically adjusting cutoff time T_{max} two times, each time doubling it, if too many time-outs occur. The ESP continues with the next step when the collected performance data suggests that the current instance set contains a sufficient proportion of problem instances that are reasonably hard. This decision is made based on processing a small random sample of all instances. Restricting the set of instances in this step limits the time needed to determine whether a given instance set is valid.

More specifically, the platform selects randomly a subset of

$$\min(\max(20, \min(\lfloor size/10 \rfloor, 100)), size)$$

instances to test the hardness and set the cutoff time accordingly (here $size$ denotes the size of the entire input set of instances). The formula calculates the number of selected instances according to the following cases:

$$sample(size) = \begin{cases} 100, & if\ size \geq 1000 \\ \lfloor size/10 \rfloor, & if\ 200 \leq size < 1000 \\ 20, & if\ 20 \leq size < 200 \\ size, & if\ 1 \leq size < 20 \end{cases}$$

In this way, normally 10% of instances are selected, unless the set of instances is extremely large ($size \geq 1000$) or extremely small ($size < 200$).

Once the subset of instances is selected, the ESP starts to estimate the correct cutoff time. The cutoff time is set based on the performance data of all encodings on the selected subset of instances. It automatically adjusts the cutoff time until the performance data on the subset of instances contains a certain portion of reasonably hard instances. An instance is considered *easy* when all encodings solve it within time T_e. An instance is *extremely hard* when it is not solved by any encoding within given the cutoff time T_{max}. All other instances are *reasonably hard*, or just *hard*. If at least 30% of instances in the selected subset are reasonably hard, the entire input data set is *valid*. If not and also no more than 30% of instances time out on each encoding, the ESP exits and declares the original input instance set "too easy." Otherwise, the selected subset is "too hard" and the system increases T_{max} by doubling it (and adjusting T_e accordingly). After doubling, the ESP again runs all encodings with all selected instances. If, with the new values for T_{max} and T_e, the number of reasonably hard instances becomes 30% or more, the ESP stops and declares the original input instance set as valid. Otherwise, the ESP doubles T_{max} one more time and repeats.

The possible outcomes of the process are then: "too easy, "too hard," and valid. In the first two cases, the user is informed and asked to adjust T_{max} and the hardness of the input instances accordingly. In the last case, the ESP checks if there are at least 500 reasonably hard instances in the entire input set. If not, the ESP exits and returns to the user the numbers of instances in the set that are easy, hard and extremely hard, and requests that the user updates the input instance set. Note that even if ESP requires at least 500 reasonably hard instances to move towards machine learning modeling steps, it still runs performance collection when the input instance set size is less than 500. The reason is that the collected performance can provide users with information about which kinds of instances are hard so that users can

Instance_id	ham1	ham2	ham3	ham4	ham5	ham6
insttri200_33_1	114.96	0.61	200.00	12.52	2.89	2.14
insttri200_41_2	15.22	49.10	200.00	200.00	0.65	0.49
insttri200_49_1	13.22	0.16	200.00	0.23	200.00	0.62
insttri200_57_1	47.86	200.00	0.45	7.85	200.00	200.00
insttri200_57_2	41.98	200.00	59.55	53.86	0.24	1.08
insttri200_65_2	15.61	1.02	200.00	26.42	45.46	25.65
insttri200_71_10	1.22	200.00	139.17	14.84	200.00	200.00
insttri200_81_8	200.00	38.08	200.00	32.40	200.00	200.00
insttri200_91_5	200.00	74.90	116.11	1.45	40.20	200.00
insttri200_131_10	8.31	132.25	2.85	22.46	42.22	58.86

Table 6.1: A list of valid structured dataset for Hamiltonian cycle problems: I report runtime for five encodings on these instances

easily generate more hard instances.

The process of subset aims to save time when there are too many extremely hard instances in the input instance set. In an extreme case when a large (say the size is 10,000) data set contains all extremely hard instances, only 100 are selected into subset by ESP, so ESP only times out 300 times (including the process of adjusting T_{max}) for each encoding and informs the user without having to process the entire data set.

Below, I illustrate the discussion using the HC problem as an example. This discussion assumes that a set of instances for the problem is available (see HC performance in Appendix 9.F). Graphs used in this example are built by removing a specified number of directed edges from triangle shaped grid graphs. I discuss in detail how to generate these instances in Section 7.2.

Table 6.1 shows performance data collected by running the *gringo/clasp* tools with six encodings of the HC problem on several instances of that problem, (in this case, directed graphs), coming from the set of instances I generated for the problem.

A valid instance set must evince complementary performance from the selected encodings. That is, no encoding must be uniformly better than others, in fact, *each* encoding must have its area of strength when it performs better than others. This

is the case for the set of instances in Table 6.1. For example, on the instances *insttri*200_33_1 and *insttri*200_57_1, *ham 2* and *ham 3* exhibit "opposite" performance: *ham 2* is the winner on the first instance while *ham 3* is the winner on the second one. Further, we can observe that each instance has its own best encoding and the order of per-instance best encodings in the table are 2, 6, 2, 3, 5, 2, 1, 4, 4, 3. In particular, each encoding is the winner on at least one instance. If a dominant encoding exists (performs best on all instances), encoding selection in such case is meaningless. The ESP will inform the user about it. The user will have an option to use the dominant encoding for all new instances, or to provide the system with a new set of input encodings.

Building a set of instances of those that are reasonably hard (with respect to T_e and T_{max}) may still yield a data set that is relatively easy (when execution times, while greater than T_e do not come close to the cutoff time). An additional requirement one could impose on a "good" set of instances is that each encoding must time out on at least some instances in the set.

In the thesis, I refer as the *oracle* to the non-deterministic algorithm that always selects the best encoding to run with a given instance. Typically, the oracle's performance is much better than the performance of any individual encoding. This is the case for the data set in Table 6.1. Thus, the task of selecting correct encodings on a per-instance basis becomes meaningful. Finally I note that to support encoding selection a large data set with at least 500 instances is needed. Although there is no standard rule on the size of data set, for a traditional regression model, one is suggested to have around 10 times as many data as the number of features. The features ESP use to build models consists of selected *claspre* features and domain specific features. The early work on HC problem [43] suggested that our models had better performance with 40 to 50 features. As a result, ESP requires that the instance set contains at least 500 elements to perform machine learning based encoding selection.

Cutoff time penalization Performance data represents the effectiveness of different encodings under a chosen ASP solving tool. Performance data is obtained by processing all encodings with all instances, using a selected solver (for instance specific versions of the *gringo* grounder and the *clasp* solver in some selected configuration). Each individual run should be limited to the selected cutoff time, since some encodings combined with some instances may take a large amount of time before terminating. As explained earlier in this section, the platform automatically adjusts cutoff time twice depending on the hardness of the problems, and then exits with an extremely hard instance set, or declares the instance set is valid when suitable cutoff time is set.

Once performance data set is collected, it is used to assess the quality of the considered encodings. To deal with execution time, the platform must account for timeouts. When an instance reaches timeout, the ESP considers the number of encodings reaching timeout for the instance, and a penalized runtime is given. The ESP uses an approach I call PARX, which takes for the runtime of a timeout instance the cutoff time multiplied by X, where X is the number of encodings that time out on this instance. For example, when this method is used, for the instances in Table 6.1, the penalized runtime for *insttri*200_33_1 is 200.00 for ham 3, and for *insttri*200_41_2 it is 400.00 for both ham 3 and ham 4.

6.4 Encoding Candidate Selection

In this stage of the process, the ESP analyzes the performance data obtained for the extended set of encodings. The system selects a subset of the extended encoding set that consists of encodings that are most effective and that together demonstrate *runtime complementarity*. At least two and no more than six encodings are selected. (If a particular encoding uniformly outperforms all other ones, the ESP exits; the user has an option to use this encoding or start anew with another set of input encodings.)

The idea of encoding candidate groups has the obvious advantage over one encoding group. Different encoding groups use different performance data and feature data, and are trained individually. Using different encoding groups leads to more candidate solutions.The ESP takes advantage of all performance models (built for all encoding groups) and selects the model with best validation result to make prediction. Selecting the best from several models instead of using one model allows for better generalization skill of the ESP to new instances.

Each encoding group consists of encodings that are most effective. To estimate the effectiveness of an encoding, I assign it a score. The score is affected by three factors: the number of instances for which the encoding provided the fastest solution, the percentage of the solved instances, and the average running time on all solved, called *winning score*, *solving percentage*, and *solving time*, respectively, each contributing to a portion of the score. Specifically, encodings with the most instances for which it provides the fastest solution have the largest winning score, encodings with largest solving percentage have the largest solving percentage score, and encodings with smallest running time have the largest solving time score. Adding up all the scores from three factors yields a ranking of the encodings according to their efficacy. The best of them are selected according to the ranking.

Based on the number of encodings, encodings are selected into groups of different size. If there are only two or three encodings, the ESP organizes all encodings in a single group. If there are i ($i > 3$) encodings, the ESP constructs serveral groups of encodings consisting of top3, ..., up to top6. That is, I consider one group that consists of the entire set of encodings, if only there are two or three encodings. Otherwise, the set of selected encodings has i encodings, where $i = 3, 4, 5$ or 6, and I consider the group of three top-scoring encodings (using the scoring method discussed in the section above), four top-scoring encodings etc., for the total of $i - 2$ groups (two groups if $i = 4$, three groups if $i = 5$ and four groups if $i = 6$).

6.5 Feature Extraction

In order to support machine learning of performance prediction models for each of the encoding groups, one needs to identify instances of problem P with the so called feature vectors. In other words, each instance-encoding pair needs to be mapped into an abstraction captured by a number of *features*, that is, properties that hold for this pair. My approach relies on two sets of features. First, it uses features that can be defined based on the generic structure of the propositional program obtained by grounding a given instance-encoding pair. In this, I take advantage of the system *claspre* [19]. Second, it uses domain specific features related to problem P that are supplied by the user.

Claspre features *Claspre* is a system designed to extract features of ground ASP programs. The extracted features fall into two groups: static and dynamic. Static ones contain features about atoms, rules, and constraints. For instance, they include such program properties as the number of rules, unary rules, choice rules, normal rules, weight rules, negative body rules, binary rules, ternary rules, etc. In total, *claspre* computes 38 static features. To extract dynamic features for a ground program, *claspre* runs *clasp* on it for some short amount of time, and has *clasp* return the information about the solving process. This information is then turned into (dynamic) features of the program. The ESP uses these features for the instance-encoding pair that defined the program processed by *claspre*. These features are based on information collected after each restart performed by *clasp*, with the number of restarts being a parameter of the process. Allowing for more restarts results in features that usually more accurately represent a problem, but the process requires extra runtime. Overall, *claspre* computes 25 dynamic features per restart. The ESP uses features collected from two restarts. However, extremely easy instances have no *claspre* features since they are solved during the feature extraction process, and no information

can be collected for them.

Domain features *Claspre* features are oblivious to the nature of the problem being solved. Domain features relevant to the nature of problem P, attributed to an instance of P, often provide additional useful characteristics of the instance (note that these features are independent of properties of a particular encoding). For example, if instances for problem P are graphs, possible features may include the number of nodes in a graph, the number of edges, the minimum and maximum vertex degrees, as well as measures reflecting connectivity and reachability properties. Availability of domain features often improves the performance of the platform. The ESP framework allows its users to supply domain features for the problems at hand through uploading the domain features file into a domain feature folder. Obviously, the ultimate selection of such features as input to the platform depends on the problem being solved. Indeed, different features may be relevant to, say, the graph colorability and Hamiltonian cycle problems. In the HC problem, existence of long paths plays a role and several features related to this property may be derived from running the depth-first search on the instance. Some domain specific features for the case of the HC problem are

- the average outdegree of nodes;
- the depth of the node from which no new nodes are discovered (found when running a depth first search from a fixed start node); (average over all start nodes is another feature)
- the depth of a node from which an edge is discovered that connects back to the fixed start node (found when running a depth first search from a fixed start node); -1 when no such edge exists; (average over all possible start nodes is another feature)

- the depth of the first node that has no edges to new nodes (found when running a breadth first search from a fixed start node); (average over all start nodes is another feature)

I used these features in our running example. The results I discuss in this theses for the ESP when used on the Hamiltonian cycle problem, assume these and some additional domain specific features (a complete list of 39 domain specific features is presented in Appendix 9.E).

The output of this phase is a table in which each row corresponds to an instance-encoding pair and contains the values of all features of the corresponding pair.

6.6 Machine Learning for Performance Model Building

The goal of utilizing machine learning techniques within ESP is to build encoding performance predictors based on performance data and features explained above. Once these predictors are constructed for a problem P at hand, they are used to select a way to use the available encodings to process new instances problem P. To build machine learning models, one can use regression or classification approaches. The former predicts each encoding's performance expressed as the running time, and then selects the most promising one by comparing the predicted times. The latter method builds a multi-class machine learning model and directly selects the most promising encoding from a collection of candidate ones. Our earlier experimental analysis indicates that regression approach works better than classification (the comparison is discussed in the experimental results of the Hamiltonian cycle case study in Section 8.1). As a result, in this work I decided to focus on regression approach and the ESP platform at present only supports the construction of regression models.

The set of selected encodings (at least two and at most six arranged into one to four groups, as discussed in Section 6.4) is the basis for machine learning algorithms currently used by the ESP. The ESP performs learning for each of the groups based

on instance features and instance performance data restricted to encodings in the group. Supervised ML techniques that I use here are trained on ⟨ instance features, instance performance ⟩ pairs for each encoding in the group. Once a model is trained it yields a mapping from instance features to the estimated performance of a targeted encoding. The ESP builds runtime prediction models for each encoding and selects the encoding with minimum predicted runtime. I now explain the detailed design below.

Features selection As explained in Section 6.5, ESP collects *claspre* features for each instance-encoding pair. Assuming a fixed set of encodings, each instance to the problem is assigned *claspre* features collected when processing that instance with all encodings in the set, as well as its domain specific features. As result, the features representing an instance consist of the features of that instance when paired with all encodings in the group being considered (88 features for each instance-encoding pair possible within the group) and the domain specific features. This is a large number of features that may cause poor computational performance of machine learning algorithms. Moreover, many of these features may be of little value to the task of characterizing an instance. To address these issues, the ESP reduces the number of features by further processing. For *claspre* features, the ESP first performs feature selection inside features related to individual encoding. Several subsets (the ESP choose from 40% to 70%) of features are selected for each encoding based on standard deviation reduction [24]. To evaluate which subsets have the best generalization ability, we generate different data sets related to all these subsets of features, train these subsets, and compare their validation results. Data with each subset of selected features are trained and validated on different data splits from the whole dataset, and validation results are compared. The subset that provides the lowest average mean squared error is selected as the set of selected features for

the instance-encoding pair. When the validation results for all encodings within the group are compared, the best subset is selected as the *claspre* features of the group. A subset of domain specific features is selected separately and then combined with selected *claspre* features to form final set of features.

Hyper-parameters tuning At present, the ESP supports three well-known machine learning algorithms: k-Nearest Neighbors (kNN), Decision Tree (for the review of these two methods see, for instance [55]), and Random Forest [27]. In each case, the performance of the algorithm depends on the choice of hyper-parameters (the number k of nearest neighbors to consider for the kNN method; the maximum depth of the tree, the minimum number of samples still to split, and the mini-mum number of samples in a leaf node for the decision tree method; and the decision tree parameters plus the number of trees in a forest for the random forest approach).

Hyper-parameter tuning is an important step within the training phase of machine learning. A typical method to find the optimal hyper-parameters is grid search[35]. This method defines a range for each hyper-parameter (feasible here because these ranges are finite), and exhaustively searches through all the possible value of hyper-parameters. I implemented the grid-search method for hyper-parameter searching in the ESP and combined it with the 10-fold cross-validation (for the description of k-fold cross validation method see, for instance, [34]) to improve the generalization of the obtained model.

Assessment of learned models The result of the learning (for each group) is the collection of performance models obtained by applying each of the machine learning methods implemented in the ESP. These models are compared by evaluating their performance on the 5-fold cross validation approach. For each round, the platform trains models on the training set, predicts the runtime of the corresponding encoding for instances on the validation set, and selects the most promising encoding on a

Table 6.2: Instance set that could be better solved by encoding schedules

Instance_id	Runtime_A	Runtime_B
1	30	200+
2	200+	80
3	80	200+
4	200+	40

per-instance basis. Average solving percentage and average solved time for five runs are compared for all learned models for all groups, and the best model among them is selected by the ESP for use with the future instances of problem P. The average solving percentage is the primary criterion, and when there is a tie, the average solved time comes as the secondary criterion. The ESP select best models from groups of encodings instead of one group, and thus can exhibit a better generalization skill.

6.7 Schedules

Encoding schedules An alternative to selecting an encoding to use with a given instance based on the predicted running time is to use several encodings in some order, allocating to them some specific time budgets, so that the total time allocated equals the cutoff time in the encoding selection approach. I refer to this approach as *encoding portfolio*, or *encoding scheduling*. The benefit of scheduling is that it provides the chance to solve an instance when a selected encoding does not work well on the instance but some other encodings do. To explain how an encoding schedule works and why it might be beneficial, I assume that the performance of some two hypothetical encodings A and B for a certain problem, processed by a solver on four instances is as shown in Table 6.2 (I assume that the cutoff time is 200s).

I notice that in this situation any individual encoding only solves half of the instance in the set when the timeout is set to 200s (either [A: 200s, B: 0s] or [A: 0s, B:200s]), so the solving percentage is 50%. However, when I use [A: 100s, B:100s] schedule to solve the instance set, by running A for 100s followed by running B for

Table 6.3: Instance set that could be better solved by interleaving schedules

Instance_id	Runtime_A	Runtime_B
1	30	200+
2	200+	30
3	40	200+
4	200+	40

100s, I can solve all the instances, and the runtime for each of these four instances is 30s, 100+80s, 80s, 100+40s, respectively.

Interleaving schedules An *interleaving schedule method* is another way that takes advantage of the performance diversity in a set of encodings. In an interleaving schedule, encodings are run in order, each for a short amount of time. When the limit is reached, the current encoding is suspended and the next one in the order resumes. This continues until the problem is solved or the cutoff time is reached. The interleaving schedule performs well on those instances that can be solved in a short time by at least one of of the encodings used by the method.

I use again the scenario from Table (An artificially constructed example) to illustrate how the method works. The performance of encoding A and B on the given instance set is recorded and the cutoff time is set to 200s.

We know that by only running A or only running B, we can solve two instances, but with [A: 100s, B:100s] schedule, we can solve all four instances with the runtimes 30s, 130s, 40s, 130s, respectively. However, more time can be saved in this case by running an interleaving schedule [A–B,30s] (–means that the encodings are executed in an interleaving order explained in the definition of an interleaving schedule above). For instance 1, encoding A can solve within 30s. For instance 2, the interleaving schedule runs encoding A for 30s, suspends execution, runs encoding B for 30s and solves the problem. For instance 3, the interleaving schedule runs encoding A for 30s, suspends execution, runs encoding B for 30s, suspends its execution, resumes

execution of encoding A and solves the problem after it is run for 10s. For instance 4, the interleaving schedule runs encoding A for 30s, suspends execution, runs encoding B for 30s, suspends its execution, resumes execution of encoding A, suspends its execution, resumes execution of encoding B and solves the problem after it is run for 10s. The result of the interleaving schedule is 30s, 60s, 70s, 100s, respectively for these four instances. All instances are solved and the runtime is much smaller than that of the encoding schedule [A: 100s, B:100s] mentioned above. An interleaving schedule is calculated based on the performance data (training data) and this fixed schedule is then used on all future instances.

6.8 Per-instance Encoding Selection and Solving

The platform computes models and schedules, selects a solution based on performance of cross-validation results, and uses this solution to solve new instances of the problem P. The process of computing and selecting machine learning models has been covered in Section 6.6 *Assessment of learned models* above. The best machine learning model is the result of the average performance on the validation dataset of five rounds in terms of solving percentage and solved runtime. In the same time, two schedules (encoding schedules and interleaving schedules) are built on the same data sets used for machine learning training, and average solving percentage and solved runtime of the schedules are compared with the validation result of the best machine learning model. When the selected machine learning model is better, the system is able to predict the per-instance promising encoding based on the instance features. When a new instance comes, the system extracts corresponding instance features selected by the system, predicts the runtime of all encodings, and finally selects the enocoding with the minimum predicted runtime to solve the new instance. On the other hand, when any of the schedule is a winner, the machine learning method fails to work better for the dataset provided. The reason might be that the current features do not

characterise instances. To address these problem, we need to provide more domain specific features. Or it could be the size of valid instances is not enough. To this end, we need to check if all instances are easy or too hard and make sure there are enough reasonable hard instance. However, in this situation, an instance-based selection performs worse than a schedule method built on the runtime property only, so the system selects the schedule to solve new instances. When a new instance comes, no feature extraction is performed in this case, the system executes the schedule consisting execution order and time of involved encodings.

Chapter 7 Generating Instances of the Desired Hardness

Availability of methods to generate instances of the desired hardness to a given problem is important for experimental evaluation of algorithms developed to solve it. In my work, they are also essential for building performance prediction models. In this section, I discuss methods I developed for generating instances to the problems I used in the experimental studies of the encoding selection approaches. These problems are: Hamiltonian cycle, graph coloring, and graceful graphs problems. Clearly, in each case the instances to be generated are graphs. The methods I developed for these specific problems suggest a general methodology applicable to other problems and domains, as well.

Hard instances are crucial for both measuring the performance of encodings and building performance prediction models. Easy instances can be solved within seconds with any encoding, and thus cannot be used to measure how efficient an encoding really is. Extremely hard instances timeout at cutoff time for all encodings, and thus the collected runtime is not the real solving time. Moreover, such instances make collecting performance data time consuming, as each encoding needs to run up to the cutoff time before performance data is generated. Instances of intermediate hardness (I refer to them simply as hard), that is, instances neither easy nor extremely hard, can be used to measure the real performance of encodings without consuming too much time. Therefore, it is important to develop techniques to generate hard instances to problems.

No matter what problem one is considering, there is no clearcut definition of easy, hard and extremely hard. The definition may depend on the solving ability of encodings, the experimental setup, such as the cutoff time, and the computational power of a computer, so they may vary case by case.

I propose to define these concepts relative to a single parameter namely, the *cutoff* time T_{max}. The specific choice of T_{max} has to be based on the available computing resources, the capability of encodings available at hand, and other aspects such as the time available for generating the performance data and running machine learning algorithms. In my work, I generally set $T_{max} = 200$s (but this parameter can clearly be set to a different value). I then define the *easy* threshold T_e. With these parameters set, I define easy, hard, and extremely hard. An instance is easy when all encodings solve it within time T_e, and extremely hard when all encodings time out on it, that is, fail to terminate within time T_{max}. Otherwise, the instance is *hard*. For example, in the *Hamiltonian cycle* problem, I set hard instances to be those that are solved with runtime above the threshold, $T_e = T_{max}/7$ seconds, and below cutoff time T_{max}, 200 seconds, for at least one encoding (not necessarily the same one). The threshold value T_{max}/k for T_e was set by analyzing the performance data of initially generated instances, where k is the largest integer so that at least 50% of instances were easy.

7.1 Random Graphs

An easy way to generate graph instances is to create them at random, with just two input parameters, the number of nodes and the number of edges generated graphs should have.

However, our experiments show that this method does not always guarantee hardness. In the case of the HC problem, graphs with n nodes and e randomly selected edges did not yield interesting instances. Random instances are generated in the following way: Given the number of nodes n and edges e, one randomly generates edges to connect nodes until the number of edge reaches e. I used graphs with the number of nodes n ranging from 1000 to 4000. In each case, I ranged the number of edges e from $10n$ to $1000n$. For each value of n and e, I generated 20 instances, used Hamiltonian cycle encodings and ran gringo/clasp tools. The results showed

all random graphs were solved within 10 seconds. In conclusion, in my work, I did not find hard problems for random graphs even if I experimented with graphs with thousands of nodes.

7.2 Structured Graphs

A better approach is to generate instances at random but ensuring that they do have some degree of regularity structure in them. A general idea is to start with a regular structure that happens to be an instance satisfying the constraints of the problem, and keep on simplifying (or, alternatively, adding complexity to) so that at some point it no longer admits solutions to the problem. Need to mention that one could start with a structure that does not admit solutions and move towards structures that do. The reason for finding the boundary between having solutions and no solutions is the idea of *phase transition*, the corresponding phenomenon where problems transition between all having solutions and all having no solution. By setting correct parameters, we can control the property of randomly generated structured instances and find the phase transition region. For many problems that show phase transition, this is precisely the region where hard problems are located (cf. the results on propositional satisfiability [52]).

The modification of the structure is based on the monotone graph property. A property Φ of graphs is monotone if

1. for every two graphs G_1 and G_2 on the same set of vertices such that $G_1 \subseteq G_2$ and G_2 has property Φ, it follows that G_1 has property Φ; or if

2. for every two graphs G_1 and G_2 on the same set of vertices such that $G_1 \subseteq G_2$ and G_1 has property Φ, it follows that G_2 has property Φ.

Two examples of monotone properties are the existence of a k-coloring in a graph (the property is monotone because of the condition 1) and the existence of a Hamilto-

nian cycle in a directed graph (here the property is monotone because of the condition 2).

Consider a particular graph problem P defined by a monotone property Φ. Assume that the graph with no edges satisfies the property Φ and the graph with all edges does not. Then, (assuming monotonicity because of the condition 1) if we start with any graph that satisfies the property Φ and start adding edges to it, at some point we obtain a graph that no longer satisfies Φ. If we add edges at random then for some integer k, the probability that the graph obtained after k randomly selected new edges are added has a property Φ is $1/2$. This integer k determines the so called *phase transition* for the problem P (property Φ). The discussion if Φ is monotone because of the condition 2 is similar.

In many cases, the graphs generated in this way from a structured graph that has (or does not have) the property Φ, (depending on which condition determines monotonicity) that fall in the phase transition region turn out to be harder than those from the regions before and after the phase transition. I used graphs generated in this way to build data sets used in my experiments. I applied this method to graphs, but it can be adjusted to apply to any problem where instances are represented by relational structures.

7.2.1 Hamiltonian Cycle Instances

In this section I provide insights into instance generation process by focusing on the HC domain.

In view of the experimental result on hardness of the HC problem for randomly generated graphs, which did not yield hard instances, I developed methods to generate instances based on graphs with structure, following the approach described in the previous section. Specifically, algorithms are based on structured graphs of two types: triangle grid graphs and square grid graphs, along with their variations. (The

definitions of these graphs are shown below.)

These graphs have Hamiltonian cycles if the sides of the graph are of specific parities. To find phase transition, I start with some basic structured graphs that have Hamiltonian cycles and then randomly remove edges until the number of removed edges e reaches a value e' and the likelihood of the existence of a Hamiltonian cycle approaches 0. To get an accurate estimation, for each value e, a total of 20 samples are generated and the average runtime is calculated. To generate hard graph instances, I search for the information about removed edges near phase transition for each graph of a preset size.

For the grid graphs, I start with a preset size of a graph n, set the number of removed edge e to a small number, and gradually increase e, until all 20 samples are unsatisfiable. I experimentally determine the number of edges that need to be removed for the likelihood of the existence of solution to be about 1/2 (near phase transition). By observing the runtime and phase transition curve, I determine if there exist hard instances with reference to T_e and T_{max} for the graph of preset size. The problem is that when the preset size of a graph is small, hard instances may be rare both in the phase transition and outside it. SO if there are few hard instances, I increase n. Similarly, if most of all instances are extremely hard, I decrease n. If the frequency of hard instances near the phase transition for the preset size is between 0.3 and 0.7, I record n and the corresponding range for the values of e that subsumes the phase transition location. The graphs generated using values for n and e are more likely to be hard than using other parameters. To collect enough hard instances, I generate graphs using the parameters collected and test the performance on all encodings until the number of hard instances reaches a desired size.

For grid graphs, one of the basic structured graphs is generated in the following way: Given the numbers of nodes for both sides of a grid, I connect all the nodes with their neighbors vertically and horizontally to have a regular rectangular or square

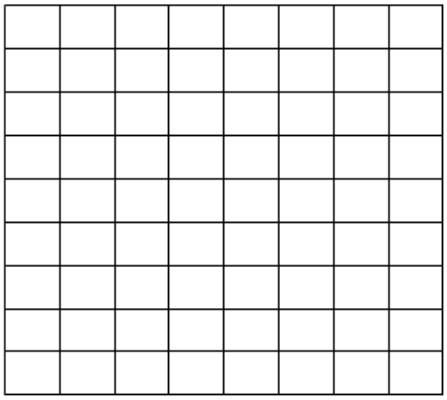


Figure 7.1: Basic structured grid graphs

graph with insides connected (shown in Figure 7.1). To define grid graphs formally: $R_{m,n}$ has mn vertices (i, j), $i = 1, 2, \ldots, m$ and $j = 1, 2 \ldots, n$, with two vertices (i, j) and $i', j')$ connected with an edge if and only if $|i - i'| + |j - j'| = 1$.

Proposition 1. *Basic structured grids have Hamiltonian Cycles as long as the number of nodes for any side is even.*

I prove with an illustration in Figure 7.2. A Hamiltonian cycle always exists as shown in the figure when we first connect the nodes in the first layer from left to right, then connect the nodes in the second layer back from right to left up to the second node, then connect the nodes in the next layer from left to right, repeat these steps until the last layer, where we connect all the nodes from right to left to the first node, and finally connect the first column from the bottom to up to meet with the starting node.

Proposition 2. *Basic structured grids have no Hamiltonian Cycle when the numbers of nodes for both sides are odd.*

Proof. Grid graphs are bipartite. If a bipartite graph has a Hamiltonian cycle, both bipartition classes are of the same size and the number of nodes in the graph is even. Grid graphs $R_{m,n}$, where both m and n are odd, have an odd number of nodes and, therefore, do not contain a Hamiltonian cycle.

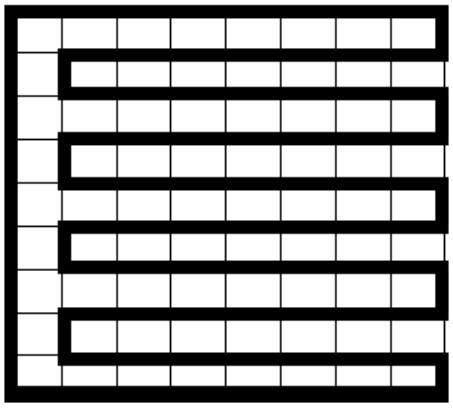
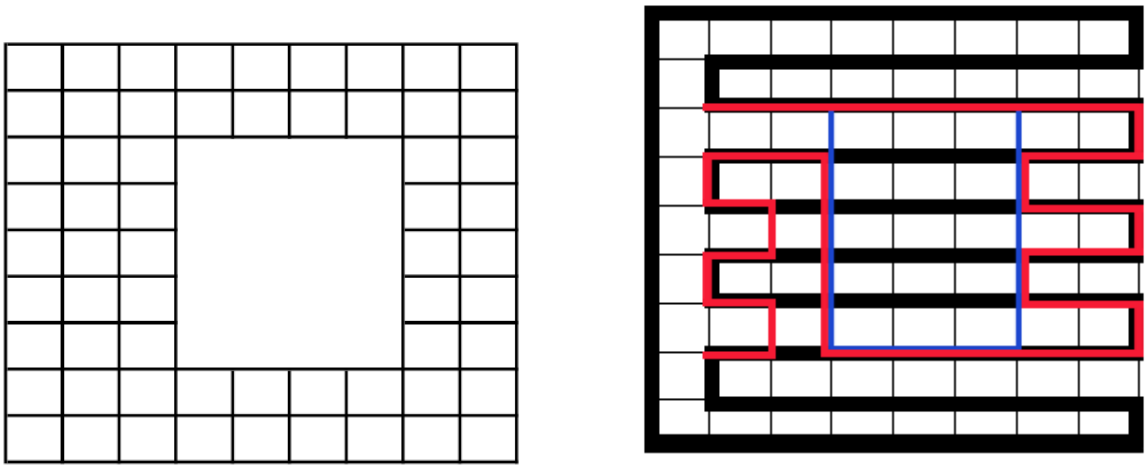


Figure 7.2: A solution to basic structured grid graphs



(a) Structured grid graphs with a hole

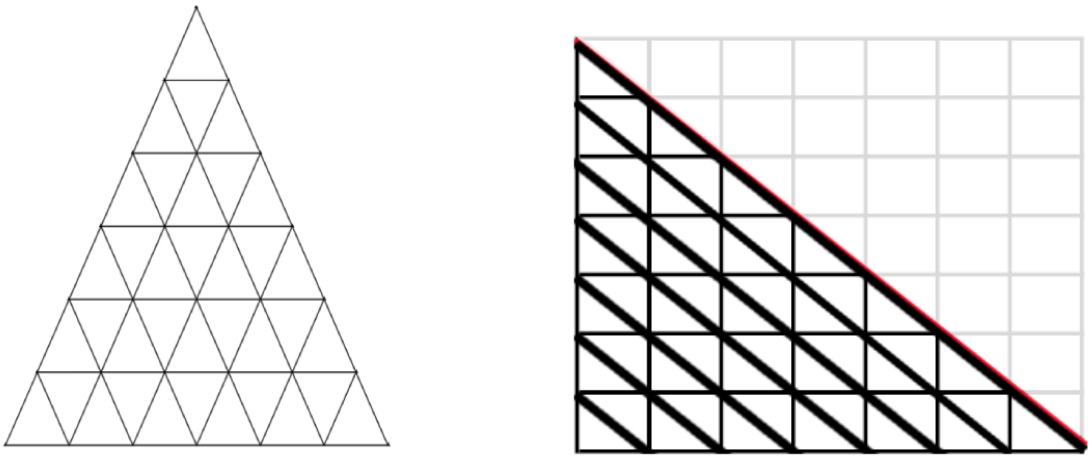
(b) Hamiltonian cycle

Figure 7.3: Basic Structured grid graphs

In my effort to generate hard graphs for experiments with the HC problem I also used rectangular grids with rectangular holes, see Figure 7.3a for an illustration.

Proposition 3. *Basic structured grids with hole inside have Hamiltonian Cycle when the hole is cut with even number of layers from the even number side of the grid.*

I prove this with an illustration in Figure 7.3b. As is shown in the graph, I draw a grid with hole (indicated by the blue boundary) on the basis of the basic grid. The original basic grid has a Hamiltonian cycle shown in black color (I call original path). A Hamiltonian solution for a grid with hole (I call new path) can be drawn on the basis of the original Hamiltonian cycle, only with some modification marked



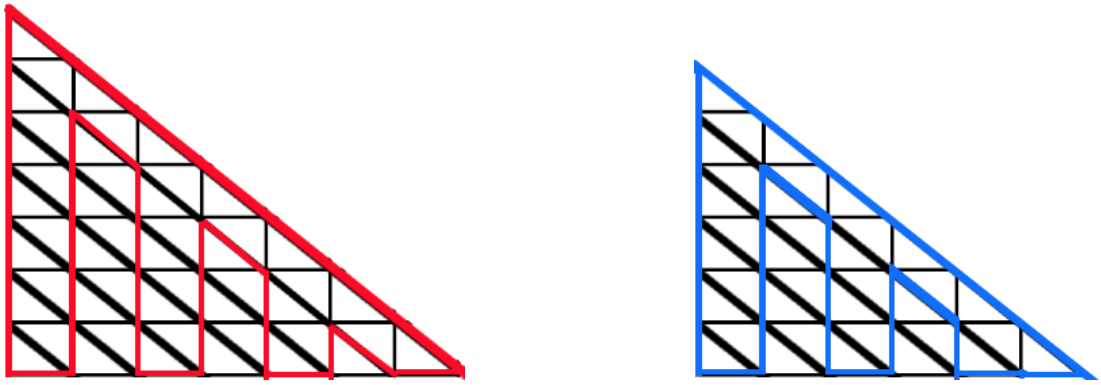
(a) General triangular graphs (b) Triangular graphs in a grid

Figure 7.4: Basic structured triangular graphs

in red. The top and bottom layers of the hole are covered by the original path. Since the number of the layers is even, these two layers are guaranteed to be covered by the original path. The right side of the hole blocks the original path. Whenever the original path encounters the right side of the hole boundary, the new path is created by moving down and changes direction. The left side of the hole boundary is connected by a path connecting from the bottom up to the second layer. The rest of the path is almost the same as the original path except that nodes on the left side of the hole boundary are already reached, so the new path only connects to the column next to the left boundary of the hole. After the new path covers the layer where the hole is created, the remaining path is as the same as the original path.

A similar way is used for triangular graphs. In a triangular graph, nodes are arranged in a triangle shape, so that the first layer consists of one node, next layer consists of two nodes, and so on. Nodes are connected with their neighbors of distance one by bidirected edges (see Figure 7.4a).

Formally, I define the basic triangular graphs in the following way: A triangular graph S_n with layer n can be generated from a grid graph of size $n \times n$ by cutting the



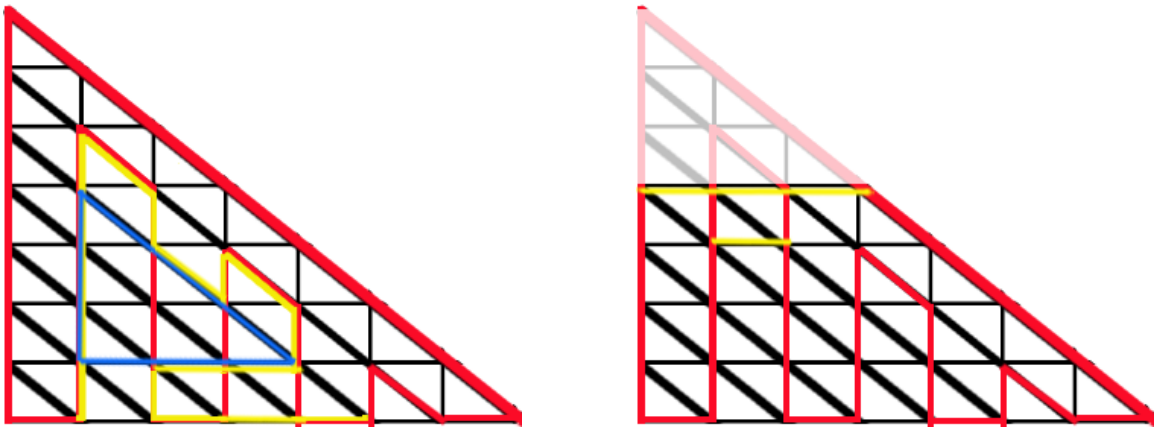
(a) S_n with an even n (b) S_n with an odd n

Figure 7.5: triangular graphs with even and odd number of layers

grid diagonally and adding necessary edges in the diagonal direction (shown in Figure 7.4b). To define such triangular graphs formally: R_m has vertices (i, j), $i = 1, 2, \ldots, n$ and $j = 1, 2 \ldots, n$, where $i >= j$, with two vertices (i, j) and (i', j') connected with an edge if and only if $|i - i'| + |j - j'| = 1$ for all vertices or $|i - i'|^2 + |j - j'|^2 = 2$ for $(i - i')(j - j') = 1$.

Proposition 4. *Basic triangle graphs have Hamiltonian Cycles.*

I prove with an illustration for both even and odd number of layers (shown in Figure 7.5) that a Hamiltonian cycle exists for such triangle graphs. A Hamiltonian path is constructed as follows. It first connects all nodes in the first column vertically from top to bottom, then connects the second column from bottom up to the second node on the top, and moves to the second node on the third column. On the odd number of column except for the first, the path moves from the second node down to the end, and the path only moves upwards on the even number of column when there are more than two nodes in that column. For each moving direction except for the first column, the starting position for the odd number of column and ending position for the even number of column are both the second node. When there are only two nodes in a column, the process stops, as the starting position (the second node) for moving down is the last node, and the ending position (the second node) for moving



(a) triangular graphs with a hole (b) Trapezoid shape

Figure 7.6: Two variations for structured triangular graphs

up is the also last node. For triangle graphs with n layers, the column number with only two nodes is $n-1$. When n is even, $n-1$ is odd, where the path is typically moving down, but since the column number only has two nodes, and the starting position is the second node, it cannot move down. It then connects to the only node in the last layer and connects back to the first node diagonally. When n is odd, $n-1$ is even, where the path is typically moving up, but since the ending position is the second node, it cannot move up. It then also connects to the only node in the last layer and connects back to the first node diagonally.

There are two kinds of variations to the triangular graphs. One is to delete all the nodes and edges in the center to have a hollow triangle, and another is to cut the top to have a trapezoid shape.

Proposition 5. *Basic structured triangular graphs with an inside hole have a Hamiltonian Cycle when the hole is in a triangular shape with an even number of layers.*

As is shown in the Figure 7.6a, the triangular hole is drawn in blue. The red path is a Hamiltonian path for the original basic triangular graph. I will show a Hamiltonian solution for triangular graphs with a inside hole on the basis of original solution. There are few changes to the Hamiltonian path near the hole area, which

are marked in yellow. The left side of the triangular hole is covered by the original Hamiltonian path. The hypotenuse of the triangular hole blocks the original path, so I use the yellow path to replace the original Hamiltonian path. Whenever the original path encounters the hypotenuse, it treats the node on the hypotenuse as the last node of the column and executes the original path finding process from node on the hypotenuse in the next column. When the new path reaches the last node on the hypotenuse, it connects all nodes on the bottom of the hole, except for the first nodes, which is already reached before. The remaining path finding starts from the last reached node on the hole boundary and executes the original path finding process, except for the columns with nodes in the hole, where it treats the node already reached as the first node in the column.

Once I make sure these variations have Hamiltonian cycles, I start to randomly remove edges until all 20 samples become unsatisfiable. When the number of removed edges is small, the graphs have Hamiltonian cycle solutions with the probability close to 1. But as the number of removed edges grows, I observe the phase transition, where this probability drops quickly and becomes close to 0. In the phase transition region, I see graphs for which the HC problem has a solution and also for which it does not have one. I also observe that the solving time grows and becomes significant. For structured instances, extremely hard instances exist near phase transition.

Figure 7.7 shows the result of satisfaction and runtime of one Hamiltonian cycle encoding for instances generated from deleting edges from a 14x12 grid graph. The first graph shows the relationship of satisfaction with the number of deleted edges. When there are few than 20 edges removed, the satisfaction rate is one, meaning almost all graphs have Hamiltonian cycles. With the increment of the deleted edges, the satisfaction decreases, and when the number of removed edges reaches 140, the satisfaction rate is zero, meaning no graph has a Hamiltonian cycle beyond this point. The second graph shows the relationship of running time with the number of deleted

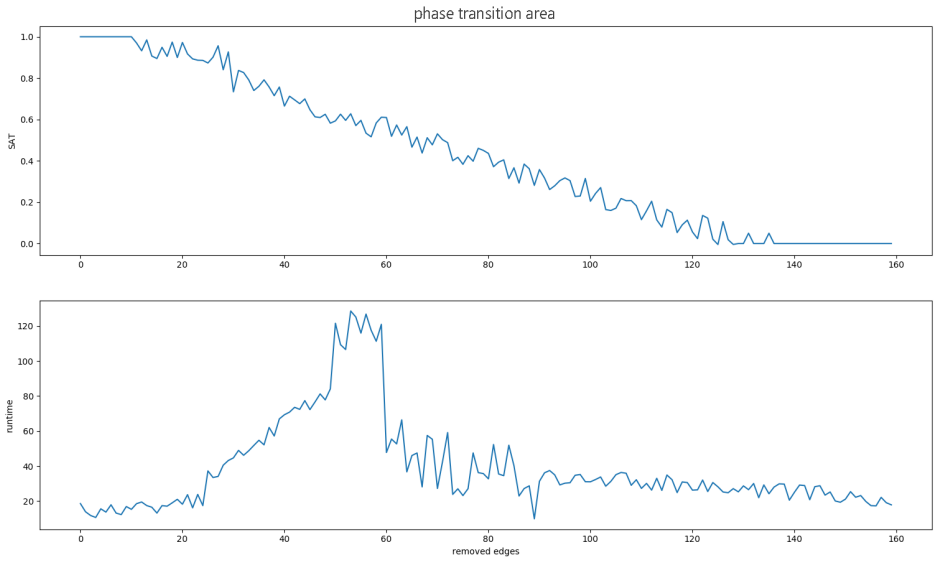


Figure 7.7: Phase transition and hard instances for 14x12 grid instances

edges. On both sides, running time is low, meaning it is easy to answer if a graph has Hamiltonian cycles. Specifically, when the graph is almost full, Hamiltonian cycles can be easily found and when there are few edges left, it is easy to check there is no Hamiltonian cycle in such graphs. However, when the deleted numbers reach 55, it becomes hard to check if such graphs have Hamiltonian cycles or not. A much longer time is needed to check all the possibilities to find answers. At this point, by referring to the first graph, I see the satisfaction rate is close to 0.5, which means half of the instances are satisfiable and half are unsatisfiable.

To support encoding selection on the HC problem, I need to generate a valid reasonably hard instance dataset. By controlling parameters n and e, I generate groups of structured graphs mentioned above. As I try to search phase transition regions with respect to each group, I find that a majority (roughly 60% to 70%) of instances can be solved within 200.00 seconds, and among these instances, around 50% of instance are above 25 seconds. As a result, I set T_{max}=200s, and T_e= $200s/7 \approx 28.6s$. I then experimentally increase the size of the starting graphs and search for

Table 7.1: Summary of the performance for Hamiltonian cycle encodings

enc	wins	Solving	Avg runtime
ham1	142	0.717949	89.86645
ham2	110	0.553846	105.6919
ham3	155	0.761538	80.39986
ham4	120	0.553846	106.8999
ham5	152	0.774359	82.05853
ham6	101	0.823077	104.6282
soa		0.979487	26.42333

correct phase transition regions where a reasonable number of hard instances are located. Once I determine the size for starting graph and its corresponding hard instance area, I generate a large number of instances and solve them by running all encodings to collect engough valid hard instances (instances that are neither too easy nor extremely hard for all encodings) for encoding selection. The truth is that even in phase transition area, hard instances are rare, so I need to test many groups of start graph to generate a large amount of instances in order to obtain enough hard instances.

After collecting enough hard instances, we test the performance of all the encodings to see if they are complementary. Table 7.1 summarizes the performance of six Hamiltonian cycle encodings on the valid instances (the table also contains a few extremely hard instances) in terms of the number of wins, the solving percentage, and the average solved runtime. The first column lists all the encodings and oracle (soa). The second column, which records the times each encoding serves as the best encoding, shows that all encodings have the opportunity to perform as the best. The third column shows that the best individual encoding ham6 solves 82% of instance, while the soa, by always selecting the best, solves around 98%. The table indicates that these encodings shows performance diversity, and when combined they can solve much more instances.

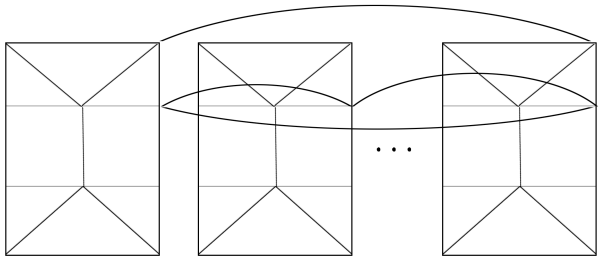


Figure 7.8: A graph coloring instance of basic grid structure

7.2.2 Graph Coloring Instances

In this section I discuss how the methodology I developed can be used to generate hard graphs for the graph coloring problem. Following the approach described in the previous section, I start with some basic structured graphs the graph coloring problems are solvable and then gradually adding edges to the graphs so that the graph coloring problems are unsolvable. Specifically, algorithms to generate hard instances are based on structured graphs of two types. The first type of structured graphs is based on some connected grids, as is shown in Figure 7.8. Several grids of the same type are connected by edges between them to form a whole graph. We can control the number of connected grids to increase the size of the problem. The connected grids form basic structural graphs to a 3-colorable problem, and they always have solutions. Then we gradually adding edges between nodes of each two grids graphs until the problems do not have solution when all the graphs are added with enough edges. We hope to find the phase transition and observe if hard instances exist near the phase transition region. There are two parameters to the graph of this type. The first is n, the number of connected grids. The second is e, the number of added edges between nodes of grids graphs.

The second type of structured graphs is based on some connected wheels. Several wheels of the same type are connected by edges between them to form a whole

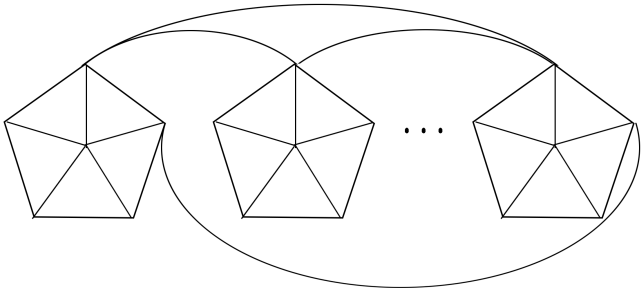


Figure 7.9: A graph coloring instance of basic wheel structure 1

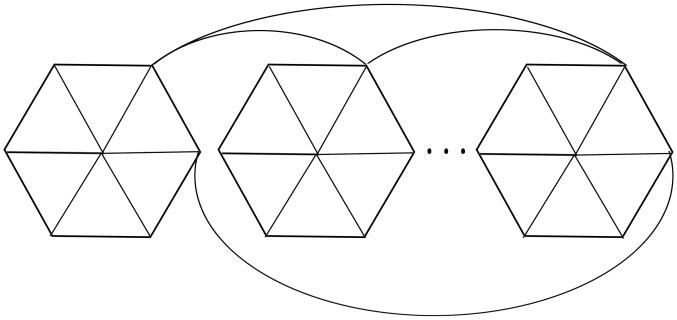


Figure 7.10: A graph coloring instance basic wheel structure 2

graph. We can also control the number of connected wheels to increase the size of the problem. Need to mention is that there are two types of wheels, the one with the odd number of nodes on the border of a wheel (as is shown in Figure 7.9), the one with the even number of nodes on the border of a wheel (as is shown in Figure 7.10). The first type is a 4-colorable problem, while the second is a 3-colorable problem, as the basic graphs with the odd number of nodes on the border of a wheel do not have 3-colorable solutions, while the ones with the even number of nodes on the border of a wheel have 3-colorable solutions. We gradually adding edges between nodes of each two wheel graphs until the problems do not have solution when all the graphs are added with enough edges. For each type, there are three parameters to control the graphs of these types. The first is e, the number of nodes on the border of a wheel, the second is n, the connected wheels, and the last is e, the number of added edges

Table 7.2: Summary of the performance for Graph coloring encodings on wheel structures

enc	Solving	Avg runtime
encoding1	0.629268	177.4688
encoding2	0.513821	218.1908
encoding3	0.660163	165.734
encoding4	0.676423	161.7599
soa	0.773984	123.8226

between nodes of wheel graphs.

Table 7.1 summarizes the performance for four Graph coloring encodings on wheel structures, in terms of the number of wins, the solving percentage, and the average solved runtime. The wins columns shows that all encodings have the opportunity to perform as the best, but the last two have more potentials. The third column shows that the best individual encoding encoding4 solves 67.6% of instance, while the soa solves 77.4%. The table also indicates that these encodings shows performance diversity, and they can solve more instances when combined.

7.3 Hard Instances without Phase Transition

For problem such as Hamiltonian cylce problems and Graph coloring problems, I can control parameters to the property of the instance set, such as the number of removed or added edges, to find phase transition. However, not all problems come with parameters. I now discuss how to generate structural hard instances when phase transition is not easily discovered.

7.3.1 Graceful Graph Instances

Let $G = (V, E)$ be an undirected graph with n nodes and e edges. Consider a labeling of nodes in the graph with distinct integers from $\{0, 1, 2, \ldots, e-1\}$. For each edge uv in E define its label as the absolute value of the difference between the labels of u and v. Such labeling of nodes is graceful if edge labels form the set $\{1, 2, \ldots, e\}$

(in particular, this means that the edge labels are pairwise distinct. The problem to find a graceful coloring for a graph (or determine that none exists) is known as the Graceful Graph problem.

The Graceful graph instances in my experiments all come from basic graphs with structures. The fist class of graphs for the graceful labeling problem used a grid graph as the basis. Here, I refer to graphs defined earlier $R_{m,n}(e)$ as a grid graph of the layer m, nodes in each layer n, and the number of deleted edge e. For a fixed $m * n$ grid graph, when deliberately controlling the increment of the number e, I can observer the satisfaction and the runtime curve and decide where the hard instances are most likely located. However, there are problems with this method. This method does not guarantee the graphs are connected after edges are removed, as removing edges may split a whole graph into several isolated subgraphs. Since reachability and node number assignment are two different problems, this type of graph generation method is not applicable.

I develop another method based on tree structures, the random tree method based on prufer sequence[1]. A tree with N nodes is generated in the following steps:

1. Generate the first set: a random sequence S of size $N-2$ from $1, \ldots, N$, repetition is allowed.
2. Generate the second set: the vertex set $V = \{1, \ldots, N\}$.
3. Find smallest element x such that $x \in V$ and $x \notin S$.
4. Join Node with value x to the node with first node in S.
5. Delete x from V, delete first node in S.
6. Repeat 3 to 5 until S is empty.
7. Connect the two nodes left in V.

[1] `https://en.wikipedia.org/wiki/Pr%C3%BCfer_sequence`

For example, a prufer sequence $S = \{2, 2, 4\}$ on the vertex set $V = \{1, \ldots, 5\}$ results in a tree $\{(1, 2), (3, 2), (2, 4), (4, 5)\}$.

Given the size n of a tree, I can obtain a set of tree instances with some randomness. However, the hardness of these instances cannot be predictable by the parameter size n. To generate a set of tree instances, I add new tree nodes to an existing tree instance to construct new tree instances that are more likely to be harder. Specifically, I generate valid instances by the following steps:

1. First generate a tree of size n according to the random tree process above.

2. Connect the new node number $n+1$ to one of the nodes in the tree of size n to form trees of size $n+1$. I will have n trees of size $n+1$.

3. Connect the new node number $n+2$ to the trees of size $n+1$. This step will generate $n+1$ trees of size $n+2$ for each tree of size $n+1$, and in total $(n+1)*n$ trees of size $n+2$.

4. Start with a new tree of size n and repeat all the above steps k times to have $k * (n+1) * n$ trees of size $n+2$.

The process above helps generate enough instances, but it may nevertheless bring so many easy and extremely hard instances. To find the reasonable hard instances of graceful graph problems, I filter out the easy and extremely hard instances in the first step above and only connect the new nodes to the tree when the first generation is tested reasonably hard. The performance diversity can be observed in the case study of the graceful graph problem in Section 8.2.

Chapter 8 Case Study

In this chapter, I discuss several case studies to evaluate the Machine Learning based encoding selection platform in terms of performance improvement over individual encoding. They are the Hamiltonian cycle problem and the graceful graph problem.

8.1 Hamiltonian Cycle Problem

A Hamiltonian cycle is a cycle through a graph that visits each node exactly once. A Hamiltonian cycle problem is to find such a cycle in a graph if at least one exists. The HC problem is an abstraction of problems of practical importance and has been often used in the past for solver benchmarking.

Instances of the HC problem are directed graphs. In ASP, directed graphs are represented as sets of facts enumerating all nodes and edges of the graph. An example graph and a collection of facts representing it are shown in Figure 8.1.

```
node(1..4).
edge(1,2).edge(2,3).edge(2,1).edge(3,4).edge(4,1),edge(4,3).
```

Encodings To use the ESP on the HC problem, I supplied the system with six encodings constructed by hand. Two of them are listed and explained below. All

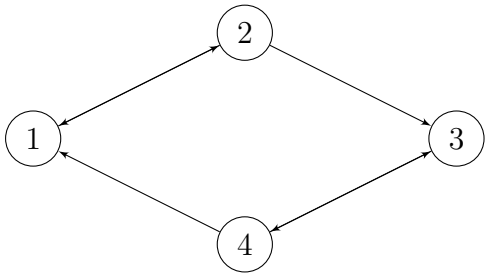


Figure 8.1: A directed graph with four nodes and six edges

encodings used in my study are presented in Appendix 9.A. To solve the HC problem in ASP, one can use different encodings based on different representation ideas, such as reachability, permutation, and cycle building and their rewritten forms. The encoding below is based on the idea of reachability.

The first three lines of the program use choice rules to select edges for candidates for a Hamiltonian cycle with each node having exactly one edge starting in it and one edge ending in it. The edges selected in this way form a collection of node-disjoint cycles covering all vertices in the graph. These may result in several isolated cycles that satisfy the constraints, so we need to impose the constraint that every two nodes are reachable from each other. I do it by defining an auxiliary notion of reachability, represented by a binary relation reach.

```
%Generator
{ hcedge(X,Y) : edge(X,Y) }=1:-node(X).
{ hcedge(X,Y) : edge(X,Y) }=1:-node(Y).
%rule
reach(X,Y):- hcedge(X,Y).
reach(X,Z) :- reach(X,Y),hcedge(Y,Z).
%test
:- not reach(X,Y),node(X),node(Y).
```

Alternatively, we can only check if each node is reachable from node 1 by a non-empty path (here, we assume that 1 is one of the nodes of the graph). This yields the following program (note that the definition of reachability and the test component are changed).

The benefit of rewriting in this way is that the size of the ground program and the search space will decrease dramatically.

```
%Generator
```

```
{ hcedge(X,Y) : edge(X,Y) }=1:-node(X).
{ hcedge(X,Y) : edge(X,Y) }=1:-node(Y).
%rule
reach(X):- hcedge(1,X).
reach(Y) :- reach(X),hcedge(X,Y).
%test
:- not reach(X),node(X).
```

Further rewritings can be obtained by observing that the choice rules (rule 1 and rule 2) in the program can be split into generation and elimination rules. We can first generate *hcedge* candidates without any constraints and then limit the number of out-degree and in-degree of each node. The last encoding can be rewritten as follows.

```
%Generator
{ hcedge(X,Y) : edge(X,Y) } :- node(X).
{ hcedge(X,Y) : edge(X,Y) } :- node(Y).
:-  2{ hcedge(X,Y) },node(X).
:-  2{ hcedge(X,Y) },node(Y).
:-  { hcedge(X,Y) }0,node(X).
:-  { hcedge(X,Y) }0,node(Y).
%rule
reach(X) :- hcedge(1,X).
reach(Y) :- reach(X),hcedge(X,Y).
%test
:- not reach(X),node(X).
```

I provide all the encodings in Appendix 9.A.

Experimental setup All my experiments were performed on a computer with Intel (R) Core (TM) i7-7700 CPU and 16 GB Memory, running on Linux 5.4.0-91-generic x86_64.

The input to the platform consists of *six* HC encodings and *one thousand* structured graph instances. In this case, the rewriting tool *AAgg* is not applicable to any of the mentioned encodings, so no new encodings were introduced by ESP. The instance set consists of graphs generated by following the methodology I developed and presented in Section 7.2. The cutoff time is initially set to 200.00 CPU seconds. The ESP system determined that the original cutoff time was appropriate and the cutoff time was not increased. Each encoding is run on all instances and runtime is recorded. All instances are grounded with *gringo* (versions 5.2.2) and solved by *clasp* (versions 3.3.3) with default configuration.

It took ten days to collect the performance data for all six encodings. Six encodings are ranked according to their performance. They give rise to four encoding groups (top three, top four, top five, and top six). For all the instances, *claspre* features are extracted, and graph-specific features are provided. Out of 1000 originally provided graph instances, the ESP platform determined 775 to be reasonably hard (instances that are too easy will be automatically deleted due to lack of *claspre* features). The data set is split into the training and the validation set (80% of instances) and the test set (20% of instances). The former is used by the ESP to build performance models and select the best solution with respect to cross-validation results. The test set is used in the experiments to evaluate the performance of the platform.

Experimental results Table 8.1 summarizes the average validation results of all the machine learning methods and the schedules. From the result, we observed that the method with the best average validation result is RF_group3, the random forest machine learning model built on group3 consisting of the top five encodings. It solves

method	solving	time
RF_group4	85.80	36.17
RF_group3	85.96	38.75
RF_group2	79.19	50.65
DT_group1	78.70	51.82
schedule_group4_ham5-ham4-175-25	81.00	71.2
schedule_group3_ham5-ham1-175-25	81.00	70.52
schedule_group1_ham3-ham5-50-150	81.00	54.49
schedule_group2_ham3-ham5-50-150	81.00	54.49
interleaving_group4_ham3-ham2-ham6-ham5_32	78.00	43.73
interleaving_group3_ham5-ham6-ham3-ham2_39	79.00	58.23
interleaving_group1_ham3-ham5-ham2_50	79.00	52.01
interleaving_group2_ham3-ham5-ham6-ham2_40	79.00	52.86

Table 8.1: Best validation results for each group - HC problem

85.96% of instances on average. As a result, it was chosen as the system solution to solve the new instance. Specifically, the system extracts relevant features of the instance, applies the learned RF_group3 models to predict runtime for all involved encodings in group3, selects the one with the lowest estimated run time, and applies the solver to the instance combined with the selected encoding.

The test results are shown in Table 8.2. Instances from the test set (in other words, instances that the platfrom has never seen during its ML modeling phase) are used to compile this table. This assessment is part of the ESP functionality.

The top part of the table shows the performance of individual encodings: solving percentage (solving%) and average solved runtime (avg_solved_t) are reported. The solving percentage records the percentage of instances each encoding can solve, and the average solved time counts the average runtime on instances that were solved within the cutoff time. The average solved runtime does not account for unsolved instances, because different penalty methods may result in different average overall runtime. The second part reports the oracle performance, which selects the best encoding for each instance, representing the *upper bound* on what is possible with the encoding selection method. The third part shows the result for the method selected

	solving%	avg_solved_t
Individual performance		
ham1	61.93	34.09
ham2	74.83	54.31
ham3	74.19	55.37
ham4	58.06	35.63
ham5	**7**8.70	**7**1.35
ham6	68.38	45.80
Oracle performance		
Oracle	**9**5.48	**2**1.64
system solution		
RFgroup3	**8**8.38	**4**0.81
Other solutions		
DTgroup4	85.16	39.14
RFgroup4	87.09	40.80
kNNgroup4	80.00	40.88
DTgroup3	87.09	36.84
KNNgroup3	80.00	41.68
DTgroup2	73.54	57.74
RFgroup2	78.06	60.81
KNNgroup2	77.41	52.54
DTgroup1	78.06	61.74
RFgroup1	79.35	56.72
KNNgroup1	76.77	57.11
schedule_group4_ham5-ham1-175-25	82.00	75.80
schedule_group3_ham5-ham1-175-25	80.00	73.07
schedule_group2_ham3-ham5-50-150	78.00	59.15
schedule_group1_ham3-ham5-50-150	78.00	59.15
interleaving_group4_ham3-ham2-ham6-ham5_32	74.00	47.69
interleaving_group3_ham5-ham6-ham3-ham2_39	75.00	61.45
interleaving_group2_ham3-ham5-ham6-ham2_40	74.00	57.04
interleaving_group1_ham3-ham5-ham2_50	77.00	59.30

Table 8.2: Test set report of the platform: performance of individual encoding, oracle, system solution, and other solutions in terms of solving rate and average runtime for solved instances - HC problem

by the ESP. The last part shows the performance of other solutions (intermediate performance models), which are obtained by the system, but not selected as the best solution by ESP.

The individual performance shows that the best individual encoding *ham5* can solve 78.70% of all instances. Thus, I can use the performance of this encoding as

the *baseline* performance. Even though *ham5* solves the most instances, it does not have the lowest average solved running time. In fact, it has the largest average solved runtime. The encoding *ham1* is the fastest in terms of average solved runtime, but it only solves 61.93% of instances. The oracle results point to the fact that there is a huge performance gain by selecting the best encoding for each instance. It solves 95.48% of instances, with an average solving time of 21.64. Compared with *ham5*, the success percentage of the always-select-best oracle is 16.78 points percentage higher. Overall, the table shows the encodings in the test set have complementary strengths. Each of them can solve a certain fraction of instances, but when combined, they can solve much more.

The system solution the ESP derived with the best cross-validation result is RF-group3, the random forest model based encoding selection from encoding group 3, which consists of the top five encoding candidates. When tested on the test set, it solves 88.38% of instances, 9.68 percentage points more than the best individual encoding *ham5*, and is also the best among all solutions. This confirms that the platform is able to generate solutions that improve the performance of ASP. The results also show that several other solutions built by ESP also outperform the individual best (ham5) and only two are noticeably worse). For example, these machine learning based solutions built for group 4 and group 3, which consist of six and five encoding candidates respectively, all give better results than *ham5*. Solutions built for group 2 and group 1 are worse since, I conjecture, they are based only on the top four and top three encoding candidates. I also observe that group 3, which consists of five encoding candidates, provides better results for corresponding models than other groups. This result shows the concept of generating different groups of encoding candidates helps select encodings with better performance.

The table also shows the results of schedule based methods. In this experiment, the four encoding schedules above have better performance than the four interleaving

schedules below. However, although these encoding schedules are able to perform well in terms of solving percentage, some of them consume much more time than machine learning based selections. Let us consider the result of schedule 'schedule_group4_ham5-ham1-175-25', which can solve 82% of instances. In this schedule, encoding 5 is first executed for 175 seconds, followed by encoding 1 for the remaining 25 seconds. As shown in the table, encoding 5 solves the most instances individually. However, it nevertheless cannot solve all the instances. To solve an instance, encoding 1 needs to wait up to 175 seconds to be executed in cases when encoding 5 fails to solve the instance in this period. As a result, the average runtime grows up to 75.80 seconds. This is much larger than the average runtime of the best machine learning based solution, which takes 40.81 seconds. As explained earlier, interleaving schedules are useful when the encoding selection method does not work. In our case, since the system solution with the best cross-validation result is RFgroup3, the machine learning model RFgroup3 is used to solve new instances. In the test set result above, we see the result of RFgroup3 is better than the results of schedule based methods, which confirms the correctness of the system solution.

As part of my work, I also compared the regression models with classification models using the same performance data and selected features. The process to derive these models is the same in each case except that different mappings are used. Regression models build mappings from instance features to the performance data of each encoding, and then use the predicted runtime of each encoding to find the best encoding, while classification models build mappings from instance features to the best encoding and then directly predict the best encodings. My experimental results of building classification models for encoding selection are in Table 8.3. We see in the table that the model with best test result is RFgroup4 in terms of both solving percentage and average runtime. It is exactly the solution selected in the validation phase, which shows the generalization skill of the ESP. The best solving

	solving%	avg_solved_t
Individual performance		
ham1	61.93	34.09
ham2	74.83	54.31
ham3	74.19	55.37
ham4	58.06	35.63
ham5	**7**8.70	**7**1.35
ham6	68.38	45.80
Oracle performance		
Oracle	**9**5.48	**2**1.64
system solution		
RFgroup4	**8**3.87	**3**7.83
Other solutions		
DTgroup4	81.94	36.25
kNNgroup4	78.06	35.08
DTgroup3	80.00	38.93
RFgroup3	82.58	36.27
KNNgroup3	76.77	38.27
DTgroup2	73.55	56.05
RFgroup2	70.97	50.56
kNNgroup2	72.26	51.20
DTgroup1	76.13	57.06
RFgroup1	74.19	59.38
kNNgroup1	74.19	47.30

Table 8.3: Test set report of classification models: performance of individual encoding, oracle, system solution, and other solutions in terms of solving rate and average runtime for solved instances - HC problem

percentage is only 83.87%. Even if it is better than the best individual encoding, it is 5% less than the performance of the regression models in Table 8.2. What is more, we also observe that almost all models in these four groups perform worse than the corresponding regression models. The only exception is DTgroup2, where the classification model solves 73.54% and the regression model solves 73.54%, but the difference is so minimum that we can ignore it. The results explain why the ESP selects regression models instead of classification models.

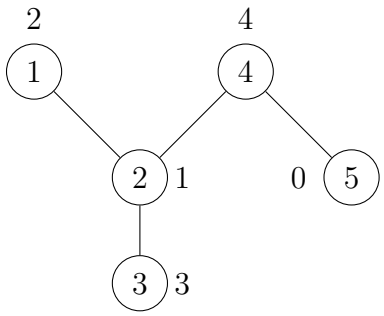


Figure 8.2: A tree instance for a graceful graph problem

8.2 Graceful Graph Problem

Let $G = (V, E)$ be an undirected graph with n nodes and e edges. Consider a labeling of nodes in the graph with distinct integers from $\{0, 1, 2, \ldots, e\}$. For each edge uv in E define its label as the absolute value of the difference between the labels of u and v. Such labeling of nodes is graceful if edge labels form the set $\{1, 2, \ldots, e\}$ (in particular, this means that the edge labels are pairwise distinct. The problem to find a graceful coloring for a graph (or determine that none exists) is known as the Graceful Graph problem.

An instance of the graceful graph problem can be a tree of the following structure (also shown in Figure 8.2)

```
edge(1,2).edge(2,4).edge(3,2).edge(4,5).
```

Note that we represent trees by listing their edges. The nodes of the trees can be determined from the edges as shown below.

Encodings Here I list one of the encodings that encode the graceful graph problem.

```
node(X) :- edge(X,Y).
node(Y) :- edge(X,Y).
num_edges(N) :- N = #count { X,Y : edge(X,Y) }.
num(0).
```

```
num(N) :- num(N1), N=N1+1, num_edges(E), N<=E.


{ value(X,N) : num(N) } = 1 :- node(X).
{ edge_value(edge(X,Y),N) : num(N), N>0 } = 1 :- edge(X,Y).


:- not edge_value(edge(X,Y),M-N), edge(X,Y), value(X,M), value(Y,N), N < M.
:- not edge_value(edge(X,Y),N-M), edge(X,Y), value(X,M), value(Y,N), N > M.
:- value(X,N), value(Y,N), num(N), X<Y.
:- edge_value(X,N), edge_value(Y,N), num(N), X<Y.
```

The encoding first gathers information about the node set, the total number of edges, and the number set in the first five lines. Then it generates a value for each node (from 0 to N) and for each edge (from 1 to N) in the following two lines. The last four lines are constraints. The first two constraints encode the absolute value of the difference of the connected nodes must be the same as the edge value. The last two constraints encode there is a unique value for each node and each edge.

The rewriting tool AAgg in the platform is able to detect the rule

```
:- value(X,N), value(Y,N), num(N), X<Y.
```

and rewrite it into rules of the following form

```
:- 2<= count {X : value(X,N) }, proj_value(N), num(N).
proj_value(N) :- value(X,N).
```

All the graceful graph encodings are presented in Appendix 9.C.

Experimental setup All my experiments were performed on a computer with Intel (R) Core (TM) i7-7700 CPU and 16 GB Memory, running on Linux 5.4.0-91-generic x86_64.

The input to the platform consists of *four* graceful graph encodings (see Appendix 9.C.) and *818* tree instances (see Appendix 9) following the methodology I presented in Section 7.3. The rewriting tool *AAgg* is able to rewrite all encodings. For the evaluation of *AAgg*, I designed two groups of experiments. I first skipped the encoding rewriting process and only used the original four encodings. Then I enabled the rewriting tool and compared the new performance with the old one without new encodings.

Experimental setup a. The ESP is run with the four original encodings, AAgg not activated. Four encodings are ranked according to their performance, and two groups of encodings are selected (top 3 and top 4) by the ESP.

Experimental setup b. The ESP is run with AAgg activated. When the original encodings are provided to the ESP, it generates four new encodings (all obtained by introducing the counting aggregate). Then the ESP performs encoding selection and encoding scheduling on the basis of the eight encodings. Since there are eight encodings, the encoding candidate generation process generates four groups of encodings (top 6, top 5, top 4, and top 3). This experiment shows that one can improve the performance of ASP solving by first rewriting encodings and then performing encoding selection.

All problems are grounded with *gringo* (versions 5.2.2) and solved by *clasp* (versions 3.3.3) with default configuration. Unlike the experiments on the HC problem, For both cases, the cutoff time was initially set to 200s and was adjusted to 400s. No graph-related features are supplied for the graceful graph problems.

Experimental result a Here I first report results without *AAgg* rewriting. The table 8.4 shows the best cross-validation results of machine learning models, encoding schedules, and interleaving schedules for each group. The group1 contains the top 3 encodings and group2 contains the top 4 encodings. The results show that all

method	solving	time
DT_group2	83.90	115.84
kNN_group1	85.86	117.30
schedule_group1_e2-e1-195-205	90.00	122.03
schedule_group2_e2-e1-195-205	90.00	122.03
interleaving_group1_e1-e3-e2_94	92.00	117.68
interleaving_group2_e4-e3-e1-e2_67	94.00	126.65

Table 8.4: Best validation results for each group - Graceful graph problem

scheduling based methods (with a minimum of 90.00%) work better than the machine learning models (with a maximum of 85.86%) for the graceful graph problems. What's more, the interleaving methods provide better solutions than the encoding scheduling methods. The method 'interleaving_group2_e4-e3-e1-e2_67' that runs an interleaving schedule with the order 'e4-e3-e1-e2' and time budget 67s for each encoding solves the most percentage of instances. The ESP chooses the solution with the largest solving percentage as the system solution to solve a new instance. Since the interleaving provides the best result, the ESP directly applies the schedule above to solve the new instances without computing instance features. The platform stores a leave-out test set to test if the schedule works for new instances.

The table 8.5 shows test results. Each of the four encodings solves between 81.81% and 83.03% of instances. But when combined, they contribute to the oracle that solves all the instances, roughly 17.00% more than the best individual encoding. We can check the system solution 'interleaving_group2_e4-e3-e1-e2_67' solves 96.00%, only 4.00% close to the oracle, and is much better than the results of all the machine learning models. Among these machine learning models, only $DTgroup1$ presents better results than the individual best. Also, the results of scheduling are all better than machine learning based models, and interleaving is the best among all solutions.

Experimental result b For the next, I present my results of the platform with the process AAgg rewriting enabled.

	solving%	avg_solved_t
Individual performance		
encoding1	81.81	109.99
encoding2	82.42	111.19
encoding3	81.21	118.38
encoding4	83.03	135.35
Oracle performance		
Oracle	100.00	31.18
system solution		
interleaving_group2_e4-e3-e1-e2_67	96.00	125.56
Other solutions		
DTgroup2	83.03	117.80
RFgroup2	81.81	115.87
kNNgroup2	79.39	120.77
DTgroup1	84.24	103.38
RFgroup1	83.03	110.09
kNNgroup1	83.03	112.38
schedule_group1_e2-e1-195-205	88.00	113.02
schedule_group2_e2-e1-195-205	88.00	113.02
interleaving_group1_e1-e3-e2_94	94.00	109.79

Table 8.5: Test set report of the platform: performance of individual encoding, oracle, system solution, and other solutions in terms of solving rate and average runtime for solved instances - Graceful graph problem

method	solving	time
DT_group4	82.70	120.09
DT_group3	84.06	113.97
RF_group2	84.81	112.13
kNN_group1	85.86	117.38
schedule_group4_encoding2-encoding1-195-205	90.00	122.03
schedule_group3_encoding2-encoding1-195-205	90.00	122.03
schedule_group1_encoding2-encoding1-195-205	90.00	122.03
schedule_group2_encoding2-encoding1-195-205	90.00	122.03
interleaving_group4_e3-e1-e1aagg-e2_65	93.00	117.15
interleaving_group3_e3-e1-e1aagg-e2_68	93.00	117.04
interleaving_group2_e3-e1-e1aagg-e2_68	93.00	117.04
interleaving_group1_e1-e3-e2_94	92.00	117.68

Table 8.6: Best validation results for each group - Graceful graph problem with AAgg

Table 8.6 reports the best cross-validation results for each group. There are eight encodings, the ESP selects top3, . . . ,top6 encodings into four different encodings. Same as the case when without the introduction of new encodings, interleaving results perform the best and the machine learning models perform the worst among these three kinds of methods. The best method is the interleaving schedule with the order 'e3-e1-e1aagg-e2' and interleaving time 68s (both group2 and group3 have the same interleaving schedule), which solves 93.00% of instances with 117.04s runtime on average.

Table 8.7 reports the performance of each solution on the test set. The individual performance only reports the top six encodings, which are all selected into group4. In contrast, group1 only contains the top three encodings. The table shows the best individual encoding is *encoding*2*aagg*, the AAgg rewritten form of *encoding*2, solves 84.24% of the instances. The worst individual, *encoding*1, solves 81.81%, almost the same as the best. However, when always selecting the best encoding from these six encodings, the Oracle solves all the instances. Need to note, the system solution, 'interleaving_group3_e3-e1-e1aagg-e2_68' also solves 97.00%, close to the Oracle in terms of solving percentage. It shows a significant improvement over any individual encoding. In this case, the schedule based methods also perform better than machine learning models, while some of the machine learning models perform much better than individual best. I also compare the performance of the oracle and system solution here with Table 8.5. When introducing new encodings, the runtime of Oracle is shortened from 31.18s to 22.01s, and the system solution solves 97% of instances, 1% better than the original solution. By enabling encoding rewriting, interleaving execution still works best and is selected as the ESP solution. When tested, this solution works slightly better than the one obtained from the four original ones. Therefore, working with larger sets of encodings in some cases at least is beneficial.

	solving%	avg_solved_t
Individual performance		
encoding1	81.81	109.99
encoding2	82.42	111.19
encoding3	81.21	118.38
encoding4	83.03	135.35
encoding1aagg	82.42	122.52
encoding2aagg	84.24	130.80
Oracle performance		
Oracle	100.00	22.01
system solution		
interleaving_group3_e3-e1-e1aagg-e2_68	97.00	127.28
Other solutions		
DTgroup4	82.42	112.59
RFgroup4	79.39	115.18
kNNgroup4	79.39	109.84
DTgroup3	83.63	99.83
RFgroup3	84.84	116.04
kNNgroup3	83.63	128.34
DTgroup2	86.06	112.39
RFgroup2	84.84	109.36
kNNgroup2	87.27	118.50
DTgroup1	84.24	103.38
RFgroup1	83.03	110.09
kNNgroup1	83.03	112.38
schedule_group1_encoding2-encoding1-195-205	88.00	113.02
schedule_group2_encoding2-encoding1-195-205	88.00	113.02
schedule_group3_encoding2-encoding1-195-205	88.00	113.02
schedule_group4_encoding2-encoding1-195-205	88.00	113.02
interleaving_group1_e1-e3-e2_94	94.00	109.79
interleaving_group2_e3-e1-e1aagg-e2_68	97.00	127.28
interleaving_group4_e3-e1-e1aagg-e2_65	96.00	124.42

Table 8.7: Test set report of the platform: performance of individual encoding, oracle, system solution, and other solutions in terms of solving rate and average runtime for solved instances - Graceful graph problem with AAgg

Chapter 9 Discussion

In the thesis, I discussed several techniques that could be used to improve the performance of ASP. The performance improvements of ASP can be obtained by rewriting answer-set programs both in the grounding stage and in the solving stage.

For the grounding stage, I proposed a manual method of rewriting to eliminate arithmetic atoms. This method introduces new atoms to replace the arithmetic ones, and uses procedural programming to precompute extensions of the new predicates. My experiments on the Pythagorean triple and Schur number problems showed that this approach may dramatically decrease the grounding time.

For the solving stage, I proposed methods to improve the performance of ASP through both encoding rewriting and encoding selection. For encoding rewriting, I introduced an automated encoding rewriting tool AAgg based on the aggregate introduction and implemented aggregate elimination. My work extended the original version of AAgg by expanding the scope of use when introducing aggregates, and by providing an option to convert aggregate rules to normal rules. In addition, I proposed and studied encoding rewriting through replicating rules with choice atoms in ASP encodings.

For encoding selection, I discussed methods of constructing groups of encodings with complementary behavior given a set of problem instances. For cases when such a group of encodings is available, I developed an approach to use machine learning to build performance models to support encoding selection on a per instance basis. Further, I developed a method to construct execution schedules for a given problem as an alternative processing approach.

I automated these processes by introducing the encoding selection platform. The new version of AAgg was used in the platform to support automatic encoding rewrit-

ing. Several other processes were incorporated, including performance data collection, candidate encoding selection, feature extraction, feature selection, machine learning modeling, and encoding scheduling, both assuming sequential execution of encodings according to some fixed time allocations as well as assuming interleaved execution.

To provide benchmarks for experimentation and to address a general need for constructing good data sets to support machine learning of performance models, I developed methods to randomly generate hard instances of graphs with some inherent structure (as just generating instances randomly often does not yield problems of sufficient hardness). I presented in detail algorithms I designed and implemented for generating three important graph problems: the Hamiltonian Cycle problem, the graph coloring problem, and the graceful graph labeling problem.

I demonstrated the efficacy of ESP on the Hamiltonian cycle problem and graceful graph problem. In case studies, I explained experiment designs, experiment data processing, and the encoding selection results. In the Hamiltonian cycle problem, the results show the machine learning models provide a much better result than the best individual encoding in terms of the solving percentage. In the graceful graph problem, the results show that when the learned performance models fail in the encoding selection approach, the ESP can still generate an effective encoding schedule to improve the solving ability of ASP.

To facilitate the experimental reproducibility, I attached in the appendix all the encodings for the problems I studied and the links to the generated hard instance sets, the performance data, and the algorithms for hard instance generation and the encoding selection platform.

My experimental study showed that the expanded AAgg tool and the rule duplication technique commonly produce new problem encodings that have their own areas of excellence and can be included in sets of encodings with complementary behavior.

The method to introduce new predicates that explicitly represent arithmetic rela-

tions and are precomputed outside of ASP was tested on two problems. The results showed that the method dramatically decreased the grounding time. This approach is at present not automated, but I conjecture it can be.

The processes of the encoding selection platform are all implemented in software. Given a set of input encodings of a problem and a set of instances, the platform can automatically finish all the processes and find the best encoding or an encoding schedule to solve new instances of similar types on a per-instance basis. Besides, all of the processes involved can be run separately. One can use the platform for performance data collection, since it deploys automatic cutoff time increments to deal with some harder instance sets than users may expect. Or one can skip over some parts of the overall process if the necessary inputs for later steps were already computed before. For instance, if the user already has generated instances and collected performance data, these steps can be omitted. The system provides a valuable tool for the ASP practitioners geared to assist them with performance analysis and encoding selection tasks in a systematic and principled manner.

In the hard instance generation chapter, I provided methods for generating hard structured graph instances and confirmed the relation between hard instances and phase transition.

In the case study, I showed that for the HC problem the platform ESP selected encodings and built performance prediction models that led to performance improvements over individual encoding. When the performance prediction models fail to work for the graceful graph problem, the ESP selected interleaving schedules as the system solution, and the results also showed performance improvement over the best individual.

There are limitations to my work. For encoding rewriting, the ESP platform only incorporates the extended tool of AAgg, while there are other tools to be incorporated. With more encodings available, we can expect a larger runtime diversity of

the encodings, which could be exploited by the ESP to build more effective solutions. The ESP requires more insights into fine-tuning machine learning methods for selecting encodings and building accurate performance predicting models. The ESP builds promising models on the HC problems, but in other cases, the models only performed comparably with the best individual encodings, and in some other cases, all ESP constructed solutions (model selection and two schedule-based ones) performed worse. Further, the ESP only works using the default setting of gringo and clasp. We know that specific settings of gringo/clasp parameters may have a huge impact on the performance of ASP. As a result, the selection from different configurations of a given solver is also of interest. My future work will aim to address the present shortcomings. First, I will expand the encoding rewriting module to generate more candidate encodings. Further, I plan to fine-tune the current machine learning methods to produce more accurate performance predictions and consider more complex machine learning models. Also, I plan to develop techniques combining encoding selection with the solver selection and solver configuration. In particular, I will study learning models to estimate for a given instance the performance of a pair (*clasp* configuration, problem encoding).

Appendices

Appendix A: Hamiltonian cycle encodings

```
ham1:
{ hcedge(X,Y) : link(X,Y) } :- node(X).
{ hcedge(X,Y) : link(X,Y) } :- node(Y).
:- 2{ hcedge(X,Y) : link(X,Y) },node(X).
:- 2{ hcedge(X,Y) : link(X,Y) },node(Y).
reach(X) :- hcedge(1,X).
reach(Y) :- reach(X),hcedge(X,Y).
:- not reach(X),node(X).
#show hcedge/2.

ham2:
{ hcedge(X,Y) : link(X,Y) } =1 :- node(X).
{ hcedge(X,Y) : link(X,Y) } =1 :- node(Y).
reach(X) :- hcedge(1,X).
reach(Y) :- reach(X),hcedge(X,Y).
:- not reach(X),node(X).
#show hcedge/2.

ham3:
{ hcedge(X,Y) : link(X,Y) }  =1 :- node(X).
{ hcedge(X,Y) : link(X,Y) }  =1 :- node(Y).
reach(1).
reach(Y) :- reach(X),hcedge(X,Y).
:- not reach(X),node(X).
```

```
#show hcedge/2.

ham4:
{ hcedge(X,Y) : link(X,Y) } 1 :- node(X).
{ hcedge(X,Y) : link(X,Y) } 1 :- node(Y).
reach(X) :- hcedge(1,X).
reach(Y) :- reach(X),hcedge(X,Y).
:- not reach(X),node(X).
#show hcedge/2.

ham5:
{ hcedge(X,Y) : link(X,Y) } :- node(X).
{ hcedge(X,Y) : link(X,Y) } :- node(Y).
:- 2{ hcedge(X,Y) : link(X,Y) },node(X).
:- 2{ hcedge(X,Y) : link(X,Y) },node(Y).
:- { hcedge(X,Y) : link(X,Y) }0,node(X).
:- { hcedge(X,Y) : link(X,Y) }0,node(Y).
reach(1).
reach(Y) :- reach(X),hcedge(X,Y).
:- not reach(X),node(X).
#show hcedge/2.

ham6:
{ hcedge(X,Y) : link(X,Y) } :- node(X).
{ hcedge(X,Y) : link(X,Y) } :- node(Y).
:- 2{ hcedge(X,Y) : link(X,Y) },node(X).
:- 2{ hcedge(X,Y) : link(X,Y) },node(Y).
:- { hcedge(X,Y) : link(X,Y) }0,node(X).
:- { hcedge(X,Y) : link(X,Y) }0,node(Y).
```

```
reach(X) :- hcedge(1,X).
reach(Y) :- reach(X),hcedge(X,Y).
:- not reach(X),node(X).
#show hcedge/2.
```

```
ham_xy:
{ hcedge(X,Y)} :- edge(X,Y).
:- hcedge (X,Y1), hcedge (X,Y2), Y1 != Y2.
:- hcedge (X1 ,Y), hcedge (X2 ,Y), X1 != X2.
reach (X,Y) :- hcedge (X,Y).
reach (X,Z) :- reach (X,Y), hcedge (Y,Z).
:- not reach (X,Y), node (X), node (Y).
```

```
ham_1x:
{ hcedge(X,Y)} :- edge(X,Y).
:- hcedge (X,Y1), hcedge (X,Y2), Y1 != Y2.
:- hcedge (X1 ,Y), hcedge (X2 ,Y), X1 != X2.
reach(X) :- hcedge(1,X).
reach(Y) :- reach(X), hcedge(X,Y).
:- not reach(X), node(X).
```

Appendix B: Graph coloring encodings

```
enc1:
% Guess colors.
chosenColor(N,C) | notChosenColor(N,C) :- node(N), color(C).
% At least one color per node.
```

```
:- node(X), not colored(X).
colored(X) :- chosenColor(X,Fv1).
% Only one color per node.
:- chosenColor(N,C1), chosenColor(N,C2), C1!=C2.
% No two adjacent nodes have the same color.
:- link(X,Y),  X<Y, chosenColor(X,C), chosenColor(Y,C).
#show chosenColor/2.

enc2:
% Guess colors.
chosenColor(N,C) | notChosenColor(N,C) :- node(N), color(C).
% At least one color per node.
:- node(X), not colored(X).
colored(X) :- chosenColor(X,Fv1).
% Only one color per node.
%:- chosenColor(N,C1), chosenColor(N,C2), C1!=C2.
:- 2<= #count{C: chosenColor(N,C)},proj_chosenColor(N).
proj_chosenColor(N) :- chosenColor(N,C).
% No two adjacent nodes have the same color.
:- link(X,Y),  X<Y, chosenColor(X,C), chosenColor(Y,C).
#show chosenColor/2.

enc3:
% Guess colors.
{ chosenColor(N,C) : color(C) } = 1 :- node(N).
% No two adjacent nodes have the same color.
:- link(X,Y),  X<Y, chosenColor(X,C), chosenColor(Y,C).
#show chosenColor/2.
```

```
enc4:
% Guess colors.
{ chosenColor(N,C) : color(C) } :- node(N).
:- { chosenColor(N,C) : color(C) }0,node(N).
:- 2{ chosenColor(N,C) : color(C) },node(N).
% No two adjacent nodes have the same color.
:- link(X,Y),  X<Y, chosenColor(X,C), chosenColor(Y,C).
#show chosenColor/2.
```

Appendix C: Graceful graph encodings

```
enc1:
% nodes and values
node(X) :- edge(X,Y).
node(Y) :- edge(X,Y).
num_edges(N) :- N = #count { X,Y : edge(X,Y) }.
num(0).
num(N) :- num(N1), N=N1+1, num_edges(E), N<=E.
% assignment to nodes
{ value(X,N) : num(N) } = 1 :- node(X).
% assignment to edges
{ edge_value(edge(X,Y),N) : num(N), N>0 } = 1 :- edge(X,Y).
% relates node values with edge values
:- not edge_value(edge(X,Y),M-N), edge(X,Y), value(X,M), value(Y,N), N < M.
:- not edge_value(edge(X,Y),N-M), edge(X,Y), value(X,M), value(Y,N), N > M.
% alldifferent values
:- value(X,N), value(Y,N), num(N), X<Y.
:- edge_value(X,N), edge_value(Y,N), num(N), X<Y.
```

```
#show value/2.

enc2:
% nodes and values
node(X) :- edge(X,Y).
node(Y) :- edge(X,Y).
num_edges(N) :- N = #count { X,Y : edge(X,Y) }.
num(0).
num(N) :- num(N1), N=N1+1, num_edges(E), N<=E.
% assignment to nodes
%{ value(X,N) : num(N) } = 1 :- node(X).
{ value(X,N) : num(N) } :- node(X).
:- 2{value(X,N) : num(N)},node(X).
:- {value(X,N) : num(N)}0,node(X).
% assignment to edges
%{ edge_value(edge(X,Y),N) : num(N), N>0 } = 1 :- edge(X,Y).
{ edge_value(edge(X,Y),N) : num(N), N>0 } :- edge(X,Y).
:- 2{ edge_value(edge(X,Y),N) : num(N), N>0 }, edge(X,Y).
:- { edge_value(edge(X,Y),N) : num(N), N>0 }0, edge(X,Y).
% relates node values with edge values
:- not edge_value(edge(X,Y),M-N), edge(X,Y), value(X,M), value(Y,N), N < M.
:- not edge_value(edge(X,Y),N-M), edge(X,Y), value(X,M), value(Y,N), N > M.
% alldifferent values
:- value(X,N), value(Y,N), num(N), X<Y.
:- edge_value(X,N), edge_value(Y,N), num(N), X<Y.

#show value/2.
```

```
enc3:
% nodes and values
node(X) :- edge(X,Y).
node(Y) :- edge(X,Y).
num_edges(N) :- N = #count { X,Y : edge(X,Y) }.
num(0).
num(N) :- num(N1), N=N1+1, num_edges(E), N<=E.
% assignment to nodes
{ value(X,N) : num(N) } = 1 :- node(X).
% assignment to edges
{ edge_value(edge(X,Y),N) : num(N), N>0 } = 1 :- edge(X,Y).
% relates node values with edge values
:- not edge_value(edge(X,Y),M-N), edge(X,Y), value(X,M), value(Y,N), N < M.
:- not edge_value(edge(X,Y),N-M), edge(X,Y), value(X,M), value(Y,N), N > M.
% alldifferent values
%:- value(X,N), value(Y,N), num(N), X<Y.
:- 2<= #count{ X:value(X,N) },proj_x(N),num(N).
proj_x(N) :- value(X,N).
:- edge_value(X,N), edge_value(Y,N), num(N), X<Y.
#show value/2.

enc4:
% nodes and values
node(X) :- edge(X,Y).
node(Y) :- edge(X,Y).
num_edges(N) :- N = #count { X,Y : edge(X,Y) }.
num(0).
num(N) :- num(N1), N=N1+1, num_edges(E), N<=E.
```

```
% assignment to nodes
{ value(X,N) : num(N) } = 1 :- node(X).
% assignment to edges
{ edge_value(edge(X,Y),N) : num(N), N>0 } = 1 :- edge(X,Y).
% relates node values with edge values
:- not edge_value(edge(X,Y),M-N), edge(X,Y), value(X,M), value(Y,N), N < M.
:- not edge_value(edge(X,Y),N-M), edge(X,Y), value(X,M), value(Y,N), N > M.
% alldifferent values
:- value(X,N), value(Y,N), num(N), X<Y.
%:- edge_value(X,N), edge_value(Y,N), num(N), X<Y.
:- 2<= #count{ X:edge_value(X,N) },proj_edge_valuex(N),num(N).
proj_edge_valuex(N) :- edge_value(X,N).
#show value/2.
```

Appendix D: Snake encodings

```
snake.lp
num(1..n).
entry(1..n*n).
{ filled(R,C,N) : entry(N) } =1 :- num(R), num(C).
:- { filled(I,J,N) : num(I), num(J) } 0 , entry(N).
(I1-I2)**2 + (J1-J2)**2 <= 2 :- filled(I1,J1,X), filled(I2,J2,X+1).

snake-mt:
num(1..n).
entry(1..n*n).
{ filled(R,C,N) : entry(N) } = 1 :- num(R), num(C).
used(Z) :- filled(X,Y,Z).
:- not used(Z), entry(Z).
```

```
:- filled(I1,J1,X), filled(I2,J2,X+1), (I1-I2)**2 + (J1-J2)**2 > 3.
#show filled/3.
```

```
snake-rew:
num(1..n).
entry(1..n*n).
{ filled(R,C,N) : entry(N) } =1 :- num(R), num(C).
:- { filled(I,J,N) : num(I), num(J) } 0 , entry(N).
%:- not filled(_,_,X), entry(X).
:- filled(I1,J1,X), filled(I2,J2,X+1), not neighbor(I1,J1,I2,J2).
```

```
snake-vl-rc:
num(1..n).
entry(1..n*n).
{ filled(R,C,N) : entry(N) } =1 :- num(R), num(C).
:- not filled(_,_,X), entry(X).
row(I,X) :- filled(I,J,X).
col(J,X) :- filled(I,J,X).
:- row(I,X), row(I1,X+1), (I-I1)**2 > 1.
:- col(J,X), col(J1,X+1), (J-J1)**2 > 1.
```

Appendix E: A list of domain specific features for the Hamiltonian cycle problem

1. num of nodes: the number of nodes in a graph
2. ratio node edge: the number of edges in a grpah

3. ratio node edge: the ratio of the number of nodes to the edge
4. bi edge: the number of bidirectional edges
5. ratio bi edge: the ratio of bidirectional edges over all edges
6. min out degree: the minimum of outdegree of nodes
7. max out degree: the maximum outdegree of nodes
8. avg out degree: the average outdegree of nodes
9. min in degree: the minimum indegree of nodes
10. max in degree: the maximum indegree of nodes
11. avg in degree: the average of indegree of nodes
12. num of odd out degree: the number of nodes with odd outdegree
13. ratio of odd out degree: the ratio of the number of nodes with odd outdegree over the total nodes
14. num of even out degree: the number of nodes with even outdegree
15. ratio of even out degree: the ratio of the number of nodes with even outdegree over the total nodes
16. num of odd in degree: the number of nodes with odd indegree
17. ratio of odd in degree: the ratio of the number of nodes with odd indegree over the total nodes
18. num of even in degree: the number of nodes with even indegree
19. ratio of even in degree: the ratio of the number of nodes with even indegree over the total nodes

20. num of odd degree: the number of nodes with odd degree (in+out)

21. ratio of odd degree: the ratio of the number of nodes with odd degree (in+out) over all nodes

22. num of even degree: the number of nodes with even degree (in+out)

23. ratio of even degree: the ratio of the number of nodes with even degree (in+out) over all nodes

24. out degree less than 3: the number of nodes with outdegree less than 3

25. ratio out degree less than 3: the ratio of the number of nodes with outdegree less than 3 over all nodes

    in degree less than 3: the number of nodes with indegree less than 3

26. ratio in degree less than 3: the ratio of the number of nodes with indegree less than 3 over all nodes

    degree less than 3: the number of nodes with degree(in+out) less than 3

27. ratio degree less than 3: the ratio of the number of nodes with degree(in+out) less than 3 over all nodes

28. depth dfs 1st backjump: run DFS from node 1, return the depth of the first backjump, where the algorithm discovers no new nodes.

29. sum of choices along path: when depth dfs 1st backjump, return the total number of choices of each discovered node along the DFS path.

30. depth avg dfs backjump: run DFS from node 1, return the average depth of all backjumps defined above.

31. depth back to root: run DFS from node 1, return the depth of a node that has a back edge to node 1.

32. depth back to any: run DFS from node 1, return the depth of a node that has a back edge to any discovered node.

33. depth one path: run DFS from node 1, return the depth of the first node with only one path, or one child.

34. min depth bfs: run BFS from node 1, record the depth of all the dead nodes, the node

    with no new nodes attached to it, and return the minimum.

35. max depth bfs: run BFS from node 1, record the depth of all the dead nodes and return the maximum.

36. avg depth bfs: run BFS from node 1, record the depth of all the dead nodes and return the average.

37. min depth beam: run beam search, a two branches BFS search, from node 1, record the depth of all the dead nodes, the node with no new nodes attached to it, and return the minimum.

38. max depth beam: run beam search from node 1, record the depth of all the dead nodes and return the maximum.

39. avg depth beam: run beam search from node 1, record the depth of all the dead nodes and return the average.

Appendix F: links to instance set and performance data

1. Hamiltonian cycle problems

    - Instance set[1]

[1] `https://drive.google.com/drive/folders/1rR9jJ47plqyK-VQUSkEbtdPjMvhNLj2Z?usp=sharing`

- Performance data[2]

2. Graceful graph problems

   - Instance set[3]
   - Performance data[4]

3. Graph coloring problems

   - Instance set[5]
   - Performance data[6]

4. Snake problems

   - Instance set[7]
   - Performance data[8]

Appendix G: links to instance generation software

1. Hamiltonian cycle problems[9]

   - Grid instances
   - Triangle instances

[2] https://drive.google.com/drive/folders/1yG1wIdKWlLfageIm252li1JWQEPWf17i?usp=sharing
[3] https://drive.google.com/drive/folders/10uueApRqAksHdyOWdDFT7YFSVfFwChUi?usp=sharing
[4] https://drive.google.com/drive/folders/1krAS28ai-gXBg32MLbWULS1UUpKBZq8c?usp=sharing
[5] https://drive.google.com/drive/folders/1V38C4V4YPGYzcC_-rWHik_swvBG8QTzA?usp=sharing
[6] https://drive.google.com/drive/folders/1q3BtlDs5QKBtody-CsfbF_cu9blAMM3f?usp=sharing
[7] https://drive.google.com/drive/folders/1eqPQLZxst67rYDpsaejoRMJgO6UbSgoU?usp=sharing
[8] https://drive.google.com/drive/folders/1W6Qk2abHbystw4r2ZhsxUimTRVNkrKfM?usp=sharing
[9] https://drive.google.com/drive/folders/1hHTjIZK2AI4LEZulMGHD2hqr3m1A_OKh?usp=sharing

2. Graceful graph problems[10]

    - Tree random generation
    - Adding a node to an existing tree

3. Graph coloring problems[11]

    - wheel structure
    - grid structure

4. Snake problems[12]

    - Grid instances

Appendix H: links to platform software

platform software with manual[13]

[10] https://drive.google.com/drive/folders/1H1-5pxmYKaaKGOph--emZZq2-bovEK-H?usp=sharing

[11] https://drive.google.com/drive/folders/1Gskhs6miAPcWaCylVn4vPzrauoQKCaiT?usp=sharing

[12] https://drive.google.com/drive/folders/1pyZiAwp1kTfmNeOnZTXpXJ6kio_XGwcv?usp=sharing

[13] http://www.cs.uky.edu/ASPEncodingOptimization/esp/

Bibliography

[1] M. Alviano, C. Dodaro, W. Faber, N. Leone, and F. Ricca. WASP: A native ASP solver based on constraint learning. In P. Cabalar and T. C. Son, editors, *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR-2013*, volume 8148 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 2013.

[2] M. Alviano, C. Dodaro, W. Faber, N. Leone, and F. Ricca. Wasp: A native asp solver based on constraint learning. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning - Volume 8148*, LPNMR 2013, page 54–66, Berlin, Heidelberg, 2013. Springer-Verlag.

[3] M. Bichler, M. Morak, and S. Woltran. lpopt: A rule optimization tool for answer set programming. In M. V. Hermenegildo and P. López-García, editors, *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR-2016*, volume 10184 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2016.

[4] G. Brewka, T. Eiter, and M. Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[5] M. Buddenhagen and Y. Lierler. Performance tuning in answer set programming. In F. Calimeri, G. Ianni, and M. Truszczynski, editors, *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR-2015*, volume 9345 of *Lecture Notes in Computer Science*, pages 186–198. Springer, 2015.

[6] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner,

N. Leone, M. Maratea, F. Ricca, and T. Schaub. Asp-core-2 input language format, 2019.

[7] P. C. Cheeseman, B. Kanefsky, W. M. Taylor, et al. Where the really hard problems are. In *Ijcai*, volume 91, pages 331–337, 1991.

[8] W. F. Clocksin and C. S. Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

[9] M. A. Covington, B. J. Grosz, and F. C. Pereira. *Natural language processing for Prolog programmers*. Prentice hall Upper Saddle River, 1994.

[10] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[11] B. De Cat, B. Bogaerts, M. Bruynooghe, G. Janssens, and M. Denecker. Predicate logic as a modeling language: the idp system. In *Declarative Logic Programming: Theory, Systems, and Applications*, pages 279–323. 2018.

[12] M. Dingess and M. Truszczynsk. Automated aggregator — rewriting with the counting. *J. ACM*, 7(3):201–215, July 1960.

[13] E. Erdem, E. Aker, and V. Patoglu. Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. *Intelligent Service Robotics*, 5(4):275–291, 2012.

[14] E. Erdem and U. Oztok. Generating explanations for biomedical queries. *Theory and Practice of Logic Programming*, 15(1):35–78, 2015.

[15] H. Fredricksen and M. M. Sweet. Symmetric sum-free partitions and lower bounds for schur numbers. *the electronic journal of combinatorics*, 7:R32–R32, 2000.

[16] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Challenges in answer set solving. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 2011.

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

[18] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. Schneider, and S. Ziller. A portfolio solver for answer set programming: Preliminary report. pages 352–357, 05 2011.

[19] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. Schneider, and S. Ziller. A portfolio solver for answer set programming: Preliminary report. In *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 352–357. Springer-Verlag, 2011.

[20] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp : A conflict-driven answer set solver. pages 260–265, 06 2007.

[21] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. pages 1070–1080. MIT Press, 1988.

[22] I. P. Gent and T. Walsh. The tsp phase transition. *Artificial Intelligence*, 88(1-2):349–358, 1996.

[23] C. P. Gomes and B. Selman. Algorithm portfolios. *Artif. Intell.*, 126(1-2):43–62, 2001.

[24] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3(null):1157–1182, mar 2003.

[25] M. J. Heule, O. Kullmann, and V. W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 228–245. Springer, 2016.

[26] N. Hippen and Y. Lierler. Automatic program rewriting in non-ground answer set programs. In J. J. Alferes and M. Johansson, editors, *Proceedings of the 21th International Symposium on Practical Aspects of Declarative Languages*, volume 11372 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2019.

[27] T. K. Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[28] H. H. Hoos, R. Kaminski, M. T. Lindauer, and T. Schaub. aspeed: Solver scheduling via answer set programming. *TPLP*, 15(1):117–142, 2015.

[29] H. H. Hoos, M. T. Lindauer, and T. Schaub. claspfolio 2: Advances in algorithm selection for answer set programming. *Theory and Practice of Logic Programming*, 14(4-5):569–585, 2014.

[30] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 507–523, Berlin, Heidelberg, 2011. Springer-Verlag.

[31] F. Hutter, H. H. Hoos, and K. Leyton-Brown. *ParamILS: An Automatic Algorithm Configuration Framework*. 2018.

[32] F. Hutter, H. H. Hoos, K. Leyton-Brown, and Thomas. Paramils: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, October 2009.

[33] P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary Computation*, 27(1):3–45, 2019.

[34] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. 14, 03 2001.

[35] S. M. LaValle, M. S. Branicky, and S. R. Lindemann. On the relationship between classical grid search and probabilistic roadmaps. *The International Journal of Robotics Research*, 23(7-8):673–692, 2004.

[36] E. Lawler. *The Travelling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley-Interscience series in discrete mathematics and optimization. John Wiley & Sons, 1985.

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

[38] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562, 2006.

[39] Y. Lierler. Strong equivalence and program's structure in arguing essential equivalence between first-order logic programs. In *International Symposium on Practical Aspects of Declarative Languages*, pages 1–18. Springer, 2019.

[40] Y. Lierler and M. Maratea. Cmodels-2: Sat-based answer set solver enhanced to non-tight programs. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 346–350, 2004.

[41] Y. Lierler and M. Maratea. Cmodels-2: Sat-based answer set solver enhanced to non-tight programs. In *International Conference on Logic Programming and NonMonotonic Reasoning*, pages 346–350. Springer, 2004.

[42] M. Lindauer, H. H. Hoos, F. Hutter, and T. Schaub. Autofolio: An automatically configured algorithm selector. *J. Artif. Int. Res.*, 53(1):745–778, May 2015.

[43] L. Liu and M. Truszczynski. Encoding selection for solving hamiltonian cycle problems with asp. *arXiv preprint arXiv:1909.08252*, 2019.

[44] M. Manna, F. Scarcello, and N. Leone. On the complexity of regular-grammars with integer attributes. *Journal of Computer and System Sciences*, 77(2):393–421, 2011.

[45] M. Maratea, L. Pulina, and F. Ricca. A multi-engine approach to answer-set programming. *Theory and Practice of Logic Programming*, 14(6):841–868, 2014.

[46] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K. Apt, W. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer, Berlin, 1999.

[47] J. P. Marques Silva and K. A. Sakallah. Grasp-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, Nov 1996.

[48] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, Feb. 1999.

[49] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25:241–273, 11 1999.

[50] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An a-prolog decision support system for the space shuttle. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL '01, pages 169–183, London, UK, UK, 2001. Springer-Verlag.

[51] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[52] B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artif. Intell.*, 81(1-2):17–29, 1996.

[53] G. Terracina, A. Martello, and N. Leone. Logic-based techniques for data cleaning: an application to the italian national healthcare system. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 524–529. Springer, 2013.

[54] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, Oct. 1976.

[55] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip, et al. Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37, 2008.

[56] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *CoRR*, abs/1111.2249, 2011.