Computing Secure Sets in Graphs using Answer Set Programming

Michael Abseher, Bernhard Bliem, Günther Charwat,
Frederico Dusberger and Stefan Woltran

Institute of Information Systems 184/2
Vienna University of Technology
Favoritenstrasse 9–11, 1040 Vienna, Austria
[abseher,bliem,gcharwat,dusberg,woltran]@dbai.tuwien.ac.at

Abstract. Problems from the area of graph theory always served as fruitful benchmarks in order to explore the performance of Answer Set Programming (ASP) systems. A relatively new branch in graph theory is concerned with so-called secure sets. It is known that verifying whether a set is secure in a graph is already co-NP-hard. The problem of enumerating all secure sets thus is challenging for ASP and its systems. In particular, encodings for this problem seem to require disjunction and also recursive aggregates. In this paper, we provide such encodings and analyze their performance using the Clingo system.

Keywords: secure sets, security in graphs, alliances in graphs

1 Introduction

The paradigm of Answer Set Programming (ASP) [5] has been shown to be very effective for tackling computationally complex problems [1, 21, 22]. Its *Guess & Check* approach allows for easy specification and often very efficient solving of NP-complete problems in practice. By allowing disjunctive logic programs, ASP can even capture problems up to the second level of the polynomial hierarchy [9]. However, compared to the vast amount of benchmarks for problems in NP, problems at the second-level of the polynomial hierarchy have not been investigated at a large scale (with some exceptions like [1], [4] or [17]). To tune the performance of ASP solvers for these problems, the community requires further benchmarks from this category.

In this paper, we consider *secure sets*, a relatively new concept from graph theory that was introduced by Brigham et al. extending the weaker notion of defensive alliances [6]. The question of alliances has several practical applications. For instance, it arises naturally when asking for common relations between people, possible coalitions of political parties, or companies sharing the same economic interests. It is, however, not limited to these examples but can generally be applied when a classification of objects with respect to some common property is necessary [23].

The secure set problem asks for a set of vertices S in a graph G such that for each subset $X \subseteq S$, $|N[X] \cap S| \geq |N[X] \setminus S|$ holds, where $N[X]$ is the closed neighborhood of X in G, i.e., the set X together with all vertices adjacent to some vertex in X. It is known that checking if a given set S is a secure set of G is co-NP-complete [20]. This leads to the following observations regarding an encoding for enumerating all secure sets:

- Since the verification problem is already co-NP-hard, suitable encodings require the full power of ASP, in particular disjunction (unless NP = co-NP).
- Such encodings typically rely on a recursive schema (the so-called saturation technique) for the verification part.
- From the definition of secure sets we can conclude that aggregates have to be employed also in the recursive part.

Aggregates in ASP have been thoroughly investigated [12, 13, 19, 24]. Due to various underlying semantics, the implementation of aggregates differs between individual ASP systems. Moreover, for the sake of simplicity, some systems impose restrictions on aggregates such that not everything that is syntactically and semantically expressible can also be used in practice. Therefore, we give a detailed account of how our desired encodings can be realized using Clingo 4, one of the prominent ASP systems.

To summarize, our main contributions are as follows:

- We provide encodings for finding secure sets in graphs, a problem which to the best of our knowledge has not been tackled with ASP yet. Our encodings require the "full power" of ASP, i.e., disjunction (in combination with the saturation technique) plus aggregates.
- We present several optimizations for our encodings and provide an experimental analysis, thereby comparing the run-time performance of the encoded variants.
- We additionally state theoretical observations in form of new characterizations for secure sets.

This paper is structured as follows. Section 2 covers the required background on secure sets and ASP. In Section 3 we present our ASP encodings for the problems under consideration. Experimental results for these encodings are reported in Section 4. Section 5 concludes the paper with a discussion of our results.

2 Preliminaries

In this section, we define the notions of secure sets and then give a brief introduction to Answer Set Programming (ASP). Beside the basic ASP syntax and semantics we further describe the concept of the saturation technique and the commonly used extension of the language for aggregates.

2.1 Secure Sets

Let $G = (V, E)$ be a simple graph with vertex set V and edge set E. The set of vertices adjacent to a vertex $v \in V$, the *open neighborhood* of v, is denoted by $N(v)$. The *closed neighborhood* $N[v]$ of a vertex $v \in V$ is the open neighborhood of v together with the vertex v itself, formally $N[v] = N(v) \cup \{v\}$. For a set $S \subseteq V$, $N(S) = \bigcup_{v \in S} N(v)$ and $N[S] = \bigcup_{v \in S} N[v]$ define the open and the closed neighborhood of S.

A *defensive alliance* of $G = (V, E)$ is a subset $S \subseteq V$ such that for any $x \in S$ the inequality $|N[x] \cap S| \geq |N[x] \setminus S|$ is satisfied. For a better understanding, one can think of the vertices of $N[x] \cap S$ as the defenders of x and those of $N[x] \setminus S$ as the attackers of x. This means that for any vertex contained in a defensive alliance there are at least as many defenders as there are attackers and so any attack on a single vertex can be repelled.

A more restrictive concept than defensive alliances are *secure sets* [6] where simultaneous attacks on more than a single vertex have to be defended. A non-empty set $S \subseteq V$ is said to be secure in $G = (V, E)$ if and only if for *each* subset X of S the inequality $|N[X] \cap S| \geq |N[X] \setminus S|$ holds. In this paper we consider the following enumeration problems:

Problem 1. Secure Set: Given a graph $G = (V, E)$, what are the secure sets $S \subseteq V$, where a set S is secure if for all $X \subseteq S$ the inequality $|N[X] \cap S| \geq |N[X] \setminus S|$ holds?

Problem 2. Minimum Secure Set: Given a graph $G = (V, E)$, what are the cardinality-minimal secure sets $S \subseteq V$?

2.2 Answer Set Programming

ASP [5] is a declarative language where a *program* Π is a finite set of *rules*

$$a_1 | \ldots | a_k \leftarrow b_1, \ldots, b_m, \mathit{not}\, b_{m+1}, \ldots, \mathit{not}\, b_n.$$

where $a_1, \ldots, a_k, b_1, \ldots, b_n$ are *atoms*. The constituents of a rule $r \in \Pi$ are its *head* $h(r) = \{a_1, \ldots, a_k\}$, and its *body* consisting of $b^+(r) = \{b_1, \ldots, b_m\}$ and $b^-(r) = \{b_{m+1}, \ldots, b_n\}$. Intuitively, r states that if an answer set contains all of $b^+(r)$ and none of $b^-(r)$, then it contains some element of $h(r)$. An *interpretation* I is a subset of atoms over the domain. I satisfies a rule r iff $I \cap h(r) \neq \emptyset$ or $b^-(r) \cap I \neq \emptyset$ or $b^+(r) \setminus I \neq \emptyset$. I is a *model* of a set of rules iff it satisfies each rule. I is an *answer set* of a program Π iff it is a subset-minimal model of the program $\Pi^I = \{h(r) \leftarrow b^+(r) \mid r \in \Pi, b^-(r) \cap I = \emptyset\}$, called the *Gelfond-Lifschitz reduct* of Π with respect to I [18].

ASP systems allow first-order atoms of the form $p(t_1, \ldots, t_l)$ where p is a *predicate* of arity $l \geq 0$ and each t_i is either a variable or a constant. With $p(t_1; \ldots; t_j)$ we denote the sequence of unary atoms $p(t_1), \ldots, p(t_j)$. An atom is *ground* if it is free of variables. For any program Π, let U_Π be the set of all constants appearing in Π. For non-ground programs, the above semantics can be utilized by first applying, to each rule $r \in \Pi$, all possible substitutions from the variables in r to elements of U_Π.

The Saturation Technique. The *saturation technique* allows us to represent problems on the second level of the polynomial hierarchy by encoding the co-NP-check in ASP [10]. This technique relies on the fact that rule heads may contain disjunctions. This way, all atoms that are subject to a guess can be jointly contained in an answer set.

The idea how this step can be employed to compute solutions for problems on the second level of the polynomial hierarchy is the following. In the first place, we guess a solution candidate for which we want to know if all possibilities of this being in fact no valid solution fail. Therefore, at the same time we also guess (using disjunction) a potential witness of the solution candidate being invalid. If this second guess indeed does not yield such a witness, we derive a designated atom that causes all atoms in the disjunction for guessing witnesses to be set to true. By doing so, all models not amounting to a witness are "saturated" with all the atoms in this disjunction. So, if a valid solution has been guessed, all guesses of potential witnesses thus collapse to a unique maximal answer set. On the other hand, if we have managed to guess a witness, we kill the solution candidate by means of a constraint. Invalid solution candidates are then discarded by the minimal model semantics because each model of the program that does not encode a witness is saturated and is thus not a *minimal* model of the reduct. For more details, we refer the reader to [10].

A concept that is often applied in combination with saturation is that of a "loop" (see, e.g., [11]). A loop allows to avoid (unstratified) default negation on the saturated parts of the program by "iterating" over the witnesses to be checked.

Aggregates. Beside default negation, also recursive (or saturation-dependent) aggregates may impose non-obvious results on the ASP program. In particular, different ASP solvers handle aggregates differently. In the following we give an overview of the semantics of aggregates. In order to cope with simple arithmetic operations, the basic ASP language is commonly extended by aggregate functions (cf. the ASP-Core-2 input language [7] and, e.g., [12]).

Syntax. An *aggregate element* e is of the form

$$t_1, \ldots, t_m : l_1, \ldots, l_n$$

where $t_1, \ldots, t_m$ are terms (denoted by $\mathrm{Term}(e)$) and $l_1, \ldots, l_n$ are atoms, (denoted by $\mathrm{Conj}(e)$). An *aggregate atom* has the form

$$\texttt{\#aggr}\{e_1; \ldots ; e_n\} \prec t$$

where `#aggr`, called an *aggregate function*, is one of $\{\texttt{\#count}, \texttt{\#sum}, \texttt{\#max}, \texttt{\#min}\}$, $e_1, \ldots, e_n$ are aggregate elements, $\prec \in \{<, \leq, >, \geq, =, \neq\}$ is an *aggregate relation* and t is a term. ASP programs under this extension allow aggregate atoms in rule bodies.

In a given program Π an aggregate atom $\texttt{\#aggr}\{e_1; \ldots ; e_n\} \prec t$ is called *recursive* if it involves a cyclic dependency, i.e., if it contains an aggregate element

e with an atom $a \in \text{Conj}(e)$ and there is a rule r with an atom $b \in h(r)$ such that a and b cannot be stratified. Otherwise it is called *non-recursive*.

Example 1. The program

$$\Pi : \{p(1) \leftarrow \texttt{\#sum}\{X : p(X)\} > 0\}.$$

contains a recursive aggregate since the atom $p(1)$ in the head of the rule depends on an aggregate atom involving $p(1)$. △

Semantics. Aggregate functions are evaluated with respect to an interpretation I. A straightforward semantics for aggregates was proposed by Dell'Armi et al. [8] and Faber et al. [12]. In the following we describe the semantics for the ground case. The valuation of an aggregate element $e = \langle t_1, \ldots, t_m : l_1, \ldots, l_n \rangle$ is defined by

$$I(e) = \begin{cases} t_1 & \text{if } I \models \text{Conj}(e) \\ 0 & \text{otherwise,} \end{cases}$$

i.e., the projection of $\text{Term}(e)$ on the first term, if $\text{Conj}(e)$ evaluates to true under I, and 0 otherwise. An aggregate function $\texttt{\#aggr}\{e_1; \ldots; e_n\}$ is now simply evaluated as the application of the function denoted by `#aggr` on $\bigcup_{i=1}^{n} I(e_i)$. If $\bigcup_{i=1}^{n} I(e_i)$ is not in the domain of `#aggr`, then $I(\texttt{\#aggr}\{e_1; \ldots; e_n\}) = \bot$. Based on these definitions, $\texttt{\#aggr}\{e_1; \ldots; e_n\} \prec t \in I$ holds iff

(i) $I(\texttt{\#aggr}\{e_1; \ldots; e_n\}) \neq \bot$ and
(ii) $I(\texttt{\#aggr}\{e_1; \ldots; e_n\}) \prec t$ holds.

Note that when aggregates are involved, the reduct of a program Π under an interpretation I is obtained by deleting the rules in which a body literal is false. For programs restricted to the basic ASP language, the semantics when using this notion of a reduct is equivalent to the semantics using the Gelfond-Lifschitz reduct [12].

Example 2. Consider the program

$$\Pi = \{q \leftarrow \texttt{\#sum}\{1 : q\} = 1.\},$$

and interpretations $\{\}$, $\{q\}$, for which we obtain:

$$\Pi^{\{\}} = \emptyset, \qquad \Pi^{\{q\}} = \Pi$$

Under the subset-minimal model semantics $\{\}$ is the only answer set of Π, since $\{q\}$ is not minimal and thus not a valid answer set. △

In contrast, a different way of defining the semantics of aggregates, which is implemented in, e.g., Clingo, is by translating them to another formula which is evaluated instead. Ferraris and Lifschitz proposed such a translation to evaluate so-called weight constraints [15], which can also be used to define the semantics of aggregates [13, 19]. In a nutshell, the aggregate is translated to a conjunction of implications reflecting the intuitive meaning of the aggregate.

It has been shown that both semantics are equivalent, at least when considering positive programs [14]. This result suggests that we can provide an encoding that can equivalently be run with DLV and Clingo. However, negative numbers within aggregates are neither allowed in the current version of DLV (Dec 16, 2012) [3], nor in the WASP system (Jun 12, 2013) [2], which utilizes a modified version of DLV. We therefore restrict ourselves to a realization using Clingo (4.3.0) [16].

3 Encodings

As the definition of secure sets suggests, recursive aggregates are required to cope with the co-NP-hard task of verifying whether a guessed set of vertices is secure. In what follows, we give two proposals for such encodings. The main obstacle here is to get the recursive aggregates to work. In Clingo, it is required that the value that the outcome of the aggregate is compared to is fixed, and moreover that negative parts in aggregates have to stem from "outside" the saturation, i.e., they are not allowed to be part of the recursion. Note that excluding (default-negated) negative parts from the saturation is required in general.

3.1 Loop Encoding

In order to fulfill the requirements above, we give a slightly different characterization of secure sets:

Lemma 1. *Let $G = (V, E)$ be a graph. A set $S \subseteq V$ is secure in G iff for all $X \subseteq S$*

$$|N[X] \cap S| + |(N[S] \setminus N[X]) \setminus S| - |N[S] \setminus S| \geq 0.$$

Proof. Recall that S is secure iff for all $X \subseteq S$

$$|N[X] \cap S| \geq |N[X] \setminus S|.$$

We now add $|(N[S] \setminus N[X]) \setminus S|$ to both sides. Note that $J = (N[S] \setminus N[X]) \setminus S$ and $J' = (N[X] \setminus S)$ are obviously disjoint and that $J \cup J' = N[S] \setminus S$. It follows that $|J| + |J'| = |N[S] \setminus S|$ and thus we obtain

$$|N[X] \cap S| + |(N[S] \setminus N[X]) \setminus S| \geq |N[S] \setminus S|$$

yielding the desired result. □

Note that this characterization has a subtraction involved ($|N[S] \setminus S|$) but this number is independent from X, the set which has to be dealt with in the saturation part of the encoding.

We now take this result further by defining secure sets based on their *border*:

Definition 1. *Let $G = (V, E)$ and $S \subseteq V$. The* border $b(S)$ *of S is $b(S) = \{x \mid \{x, y\} \in E, x \in S, y \notin S\}$, i.e., the set of all vertices in S which are adjacent to at least one vertex $y \notin S$.*

Lemma 2. *Let $G = (V, E)$ be a graph. A set $S \subseteq V$ with border $b(S)$ is secure in G iff for all $X \subseteq b(S)$*

$$|N[X] \cap S| + |(N[S] \setminus N[X]) \setminus S| - |N[S] \setminus S| \geq 0.$$

Proof. Given a secure set S, since $b(S) \subseteq S$, the "only if" direction follows directly from Lemma 1. As for the other direction: Since the only negative part of the inequality ($|N[S] \setminus S|$) is independent of all $x \in S \setminus b(S)$, checking the inequality for all $X \subseteq b(S)$ suffices to verify that S is secure. □

In the following we present the ASP encoding for enumerating all secure sets of a given graph using the concept of *loops*. The input graph $G = (V, E)$ is specified by means of the predicates `vertex/1` and `edge/2`. We assume that for every input fact `edge(x,y)` also `edge(y,x)` is given. Furthermore, let the predicates `inf/1` (infimum), `succ/2` (successor), and `sup/1` (supremum) specify the lexicographical ordering over all vertices. Note that this ordering can be obtained in polynomial time by a positive logic program.

In Encoding 1, a (non-empty) set S, denoted by `inS/1`, is guessed[1] (rules 1–3). Predicate `attackSet/1` represents the vertices in $N[S] \setminus S$ (rule 4), and `border/1` the set $b(S)$ (rule 5). Following Lemma 2, it suffices to guess all sets $X \subseteq b(S)$, denoted by `inX/1`. Vertices in $S \setminus X$ are given by `outX/1` (rules 6–7). Predicate `defendSet/1` specifies the set $N[X] \cap S$ (rules 8–9).

Rules 10–17 define the set $(N[S] \setminus N[X]) \setminus S$, that is, the set of *inactive attackers*. Intuitively, an inactive attacker is an attacker (i.e., a vertex in $N[S] \setminus S$) that is not adjacent to any vertex in X. In order to obtain the set of inactive attackers (without using default negation on saturation-dependent atoms), we have to loop. To be more precise, to determine for a vertex $u \in V$ whether $u \in (N[S] \setminus N[X])$ holds, we "loop" over all vertices $v \in V$ and check if one of the following conditions holds: (a) $v \in S \setminus X$ (rules 10,13); (b) $v \in V \setminus S$ (rules 11,14); or (c) $v \in X, \{u, v\} \notin E$ (rules 12,15). Finally, the predicate `inactAtt/1` denotes the set of inactive attackers (rule 17).

The set $X \subseteq S$ is defended if X satisfies the inequality given in Lemma 2 (rule 18). In case `defended/0` is obtained, we *saturate* by setting all vertices in S to both `inX/1` and `outX/1` (rules 19–20); On the other hand, if X is not defended against the attackers, the answer set is removed (rule 21). In that case, due to the minimal model semantics, no answer set containing S is returned. Specifically, in case any $X \subseteq S$ cannot be defended, S is not secure and is therefore not returned as a solution.

Example 3. Figure 1 shows an example graph G_{ex} with currently selected sets $S = \{a, b, c, f\}$ and $X = \{c\}$. The figure illustrates the corresponding defend set $N[X] \cap S = \{b, c\}$, attack set $N[S] \setminus S = \{d, e, g, h\}$ and the set of inactive attackers $(N[S] \setminus N[X]) \setminus S = \{e, g\}$. Since $|\{b, c\}| + |\{e, g\}| - |\{d, e, g, h\}| = 2 + 2 - 4 = 0 \geq 0$ holds, X is defended. For S to be secure, all $X \subseteq b(S)$, i.e., $X \subseteq \{c, f\}$

[1] We remark that this guess can alternatively be specified by using disjunction or a so-called *choice rule* (see, e.g., [7]), which we omit here for legibility reasons.

have to be defended. For $X = \{c, f\}$, we have $N[X] \cap S = \{b, c, f\}$, $N[S] \setminus S = \{d, e, g, h\}$ and $(N[S] \setminus N[X]) \setminus S = \{\}$. Here, $|\{b, c, f\}| + |\{\}| - |\{d, e, g, h\}| = 3 + 0 - 4 = -1 \not\geq 0$, and therefore S is not secure. △

Encoding 1: Secure Sets (Loop Encoding π_{loop})

$$\texttt{inS}(V) \leftarrow \texttt{vertex}(V), \mathit{not}\ \texttt{outS}(V). \tag{1}$$
$$\texttt{outS}(V) \leftarrow \texttt{vertex}(V), \mathit{not}\ \texttt{inS}(V). \tag{2}$$
$$\leftarrow \#\text{count}\{V : \texttt{inS}(V)\} = 0. \tag{3}$$
$$\texttt{attackSet}(V) \leftarrow \texttt{inS}(U), \texttt{edge}(U, V), \texttt{outS}(V). \tag{4}$$
$$\texttt{border}(U) \leftarrow \texttt{inS}(U), \texttt{outS}(V), \texttt{edge}(U, V). \tag{5}$$
$$\texttt{inX}(V) \mid \texttt{outX}(V) \leftarrow \texttt{border}(V). \tag{6}$$
$$\texttt{outX}(V) \leftarrow \texttt{inS}(V), \mathit{not}\ \texttt{border}(V). \tag{7}$$
$$\texttt{defendSet}(V) \leftarrow \texttt{inX}(V). \tag{8}$$
$$\texttt{defendSet}(V) \leftarrow \texttt{inX}(U), \texttt{edge}(U, V), \texttt{inS}(V). \tag{9}$$
$$\texttt{okupto}(U, V) \leftarrow \texttt{vertex}(U; V), \texttt{inf}(V), \texttt{outX}(V). \tag{10}$$
$$\texttt{okupto}(U, V) \leftarrow \texttt{vertex}(U; V), \texttt{inf}(V), \texttt{outS}(V). \tag{11}$$
$$\texttt{okupto}(U, V) \leftarrow \texttt{vertex}(U; V), \texttt{inf}(V), \texttt{inX}(V), \mathit{not}\ \texttt{edge}(U, V). \tag{12}$$
$$\texttt{okupto}(U, W) \leftarrow \texttt{vertex}(U; V; W), \texttt{okupto}(U, V), \texttt{succ}(V, W), \texttt{outX}(W). \tag{13}$$
$$\texttt{okupto}(U, W) \leftarrow \texttt{vertex}(U; V; W), \texttt{okupto}(U, V), \texttt{succ}(V, W), \texttt{outS}(W). \tag{14}$$
$$\texttt{okupto}(U, W) \leftarrow \texttt{vertex}(U; V; W), \texttt{okupto}(U, V), \texttt{succ}(V, W), \texttt{inX}(W), \mathit{not}\ \texttt{edge}(U, W). \tag{15}$$
$$\texttt{ok}(U) \leftarrow \texttt{okupto}(U, V), \texttt{sup}(V). \tag{16}$$
$$\texttt{inactAtt}(U) \leftarrow \texttt{ok}(U), \texttt{outS}(U), \texttt{edge}(U, V), \texttt{inS}(V). \tag{17}$$
$$\texttt{defended} \leftarrow \#\text{sum}\{1, V, \mathit{pos} : \texttt{defendSet}(V); 1, V, \mathit{pos} : \texttt{inactAtt}(V); -1, V, \mathit{neg} : \texttt{attackSet}(V)\} \geq 0. \tag{18}$$
$$\texttt{inX}(V) \leftarrow \texttt{defended}, \texttt{inS}(V). \tag{19}$$
$$\texttt{outX}(V) \leftarrow \texttt{defended}, \texttt{inS}(V). \tag{20}$$
$$\leftarrow \mathit{not}\ \texttt{defended}. \tag{21}$$

3.2 Alternative Secure Set Characterization

In order to avoid the loop from above, we provide an alternative characterization for secure sets, which to the best of our knowledge has not appeared in the literature yet. The intuition is to guess instead of the set $X \subseteq S$ a partition of the attack set $N[S] \setminus S$ into active and inactive attackers, and then check whether for such a partition a suitable set X exists.

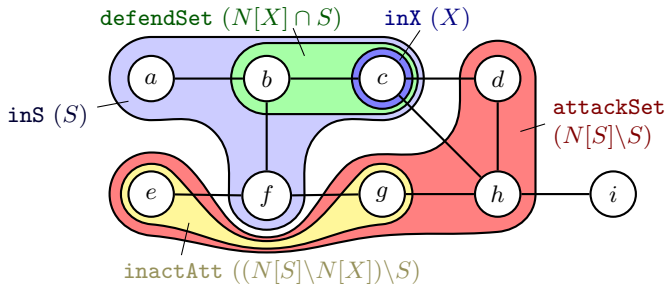


Fig. 1: Example graph G_{ex} with $S = \{a, b, c, f\}$ and $X = \{c\}$.

Definition 2. *Let $G = (V, E)$ and $S \subseteq V$. We call any subset $B \subseteq N[S] \setminus S$ a* partial boundary *of S and identify by $\chi(B) = \{X \subseteq S \mid (N[X] \setminus S) = B\}$ the subsets of S that cover B. B is called a* valid partial boundary *if $\chi(B) \neq \emptyset$.*

Lemma 3. *Let $G = (V, E)$ and $S \subseteq V$. Then S is secure in G iff for all valid partial boundaries B and all $X \in \chi(B)$, $|N[X] \cap S| \geq |B|$.*

Proof. "If" direction: Suppose S is not secure. Then there exists $X \subseteq S$, such that $|N[X] \cap S| < |N[X] \setminus S|$. Let $B = N[X] \setminus S$. We show that B is indeed a valid partial boundary. Obviously, $B \subseteq (N[S] \setminus S)$. Furthermore, $X \in \chi(B)$ and by assumption $|N[X] \cap S| < |B|$. "Only if" direction: Suppose there is a valid partial boundary B and an $X \in \chi(B)$ such that $|N[X] \cap S| < |B|$. By definition $X \subseteq S$ and $B = N[X] \setminus S$. Then S is not secure. □

We can strengthen the result as follows.

Definition 3. *Let $G = (V, E)$ and $S \subseteq V$, and $B \subseteq N[S] \setminus S$. Moreover, let $b(S)$ be the border of S. Then $\chi'(B) = \{X \subseteq b(S) \mid (N[X] \setminus S) = B\}$.*

Lemma 4. *Let $G = (V, E)$ and $S \subseteq V$. Then S is secure in G iff for all valid partial boundaries B and all $X \in \chi'(B)$, $|N[X] \cap S| \geq |B|$ holds.*

Proof. The "only if" direction follows directly from Lemma 3 and the fact that $\chi'(B) \subseteq \chi(B)$ for all partial boundaries B of S. For the other direction note that $|N[X] \cap S| \geq |B|$ holds for $X \in \chi(B) \setminus \chi'(B)$ since for each $Y \in \chi(B)$ there is an $Y' \in \chi'(B)$ with $Y' \subseteq Y$ and that in this case $|N[Y] \cap S| \geq |N[Y'] \cap S|$. □

The main advantage of this definition is that we have for each relevant $X \subseteq S$, i.e., for each $X \subseteq b(S)$, the set $(N[S] \setminus N[X]) \setminus S$ directly given via $N[S] \setminus B$ where B is some partial boundary of S with $X \in \chi(B)$.

In Encoding 2 we again first guess S (rules 1–3), and obtain the (directed) edges from S to $N[X] \setminus S$, denoted by `borderEdge/2`, as well as the `attackSet/1` (rules 4–5). Next, the partial boundaries $B \subseteq N[S] \setminus S$, or active attackers (denoted by `actAtt/1`), are guessed (rule 6). In order to obtain the sets X where $\chi'(B) = \{X \subseteq b(S) \mid (N[X] \setminus S) = B\}$, $\chi' \neq \emptyset$ holds, each vertex in B must

be adjacent to at least one vertex in $X \subseteq b(s)$, denoted by `inX/1` (rule 7). Furthermore, if there is a vertex $v \in N[X] \backslash S$ with $v \notin B$, then we can set X to be defended, since $X \notin \chi'(B)$ but $X \in \chi'(B')$ with $B \subset B'$ (rule 8). Note that this is required to be able to saturate. Rules 9–11 of Encoding 2 correspond to rules 8, 9 and 18 of Encoding 1, i.e., they define the defend set and check whether the inequality given in Lemma 2 holds. Analogous to the loop encoding, we saturate on the guessed `inX/1`, `actAtt/1` and `inactAtt/1` predicates. Due to Lemma 4, we thus obtain the secure sets.

Encoding 2: Secure Sets (Alternative Encoding π_{alt})

$$\texttt{inS}(V) \leftarrow \texttt{vertex}(V), \mathit{not}\ \texttt{outS}(V). \tag{1}$$
$$\texttt{outS}(V) \leftarrow \texttt{vertex}(V), \mathit{not}\ \texttt{inS}(V). \tag{2}$$
$$\leftarrow \#\text{count}\{V : \texttt{inS}(V)\} \leq 0. \tag{3}$$
$$\texttt{borderEdge}(U,V) \leftarrow \texttt{inS}(U), \texttt{outS}(V), \texttt{edge}(U,V). \tag{4}$$
$$\texttt{attackSet}(V) \leftarrow \texttt{borderEdge}(U,V). \tag{5}$$
$$\texttt{actAtt}(V) \mid \texttt{inactAtt}(V) \leftarrow \texttt{attackSet}(V). \tag{6}$$
$$\texttt{inX}(U) : \texttt{borderEdge}(U,V) \leftarrow \texttt{actAtt}(V), \texttt{outS}(V). \tag{7}$$
$$\texttt{defended} \leftarrow \texttt{inactAtt}(U), \texttt{inX}(V), \texttt{edge}(U,V). \tag{8}$$
$$\texttt{defendSet}(V) \leftarrow \texttt{inX}(V). \tag{9}$$
$$\texttt{defendSet}(V) \leftarrow \texttt{inX}(U), \texttt{edge}(U,V), \texttt{inS}(V). \tag{10}$$
$$\begin{aligned}\texttt{defended} \leftarrow\ \#\text{sum}\{&1, V, \mathit{pos} : \texttt{defendSet}(V);\\ &1, V, \mathit{pos} : \texttt{inactAtt}(V);\\ &-1, V, \mathit{neg} : \texttt{attackSet}(V)\} \geq 0.\end{aligned} \tag{11}$$
$$\texttt{inX}(V) \leftarrow \texttt{defended}, \texttt{inS}(V). \tag{12}$$
$$\texttt{actAtt}(V) \leftarrow \texttt{defended}, \texttt{attackSet}(V). \tag{13}$$
$$\texttt{inactAtt}(V) \leftarrow \texttt{defended}, \texttt{attackSet}(V). \tag{14}$$
$$\leftarrow \mathit{not}\ \texttt{defended}. \tag{15}$$

3.3 Problem Variants and Optimizations

Optimization problem. In order to solve the Minimum Secure Set problem, it is sufficient to add the minimize statement

$$\#\text{minimize}\{1, X, \mathit{inS} : \texttt{inS}(X), \texttt{vertex}(X)\}.$$

to π_{loop} and π_{alt}. With this, only cardinality-minimal secure sets are obtained, i.e., the answer sets with a minimal number of `inS/1` occurrences are returned.

Counting alternative. The encodings presented so far rely on the #sum aggregate for checking whether a set X is defended (see π_{loop} rule 18, and π_{alt} rule 11). The

#sum aggregate is required because the size of the attack set is subtracted from the combined size of the defend and inactive attacker sets. By adding $|V|$ to both sides of the inequality in Lemma 2, and since $|V| - |N[S]\backslash S| = |V\backslash(N[S]\backslash S)|$, we get rid of the negative part in the inequality. Thus, it is possible to use the #count aggregate by replacing rule 18 (π_{loop}) and rule 11 (π_{alt}) with the following rules:

$$\begin{aligned}
\texttt{size}(N) \leftarrow\ & N = \#\text{count}\{V : \texttt{vertex}(V)\}. \\
\texttt{defended} \leftarrow\ & \#\text{count}\{V, pos : \texttt{defendSet}(V); \\
& \qquad V, pos : \texttt{inactAtt}(V); \\
& \qquad V, neg : \texttt{vertex}(V), \mathit{not}\ \texttt{attackSet}(V)\} \geq N, \texttt{size}(N).
\end{aligned}$$

Note that default negation is allowed here, since `attackSet/1` is independent of the saturation. Furthermore, instead of adding $|V|$, $|N[S]\backslash S|$ could also simply be shifted to the right side of the inequality. However, this drastically increases the size of the grounding.

Search-space pruning. In order to improve the performance of our encodings, it is desirable to define "strengthening" constraints. Here, we define constraints that kill insecure sets $S \subseteq V$, independent of the guess for $X \subseteq S$. A vertex $v \in S$ cannot be defended if $|N(v) \cap S| < \lfloor |N(v)|/2 \rfloor$, or, in other words, less than half of its neighbors are in S. In that case S cannot be secure. In ASP, this is represented as follows:

$$\begin{aligned}
\texttt{deg}(V, D) \leftarrow\ & \texttt{vertex}(V), D = \#\text{count}\{U : \texttt{edge}(U, V)\}. \\
\leftarrow\ & \texttt{inS}(V), \texttt{deg}(V, D), \#\text{count}\{U : \texttt{edge}(U, V), \texttt{inS}(U)\} < D/2.
\end{aligned}$$

Note that degree `deg/2` can be computed during grounding (i.e., it is independent of any guess). Furthermore, ASP implements integer arithmetic, therefore $D/2$ always corresponds to $\lfloor |N(v)|/2 \rfloor$.

4 Experimental Results

In this section we present the results of a preliminary performance analysis. In detail, we compare our original Loop Encoding (which is not presented in this paper), the Loop Encoding with Restriction to Border Vertices (Encoding 1) and the Alternative Encoding (Encoding 2) for both enumerating all secure sets and for enumerating all minimum secure sets. For all the encodings under investigation we use the search-space pruning approach described above to increase performance. Furthermore, for performance reasons, we replace rules 1-3 of the encodings with an appropriate choice rule. We omit a discussion of the counting alternative here as our first experiments showed no significant difference in performance between using `#sum` and `#count` in the encodings.

4.1 Benchmark Setup

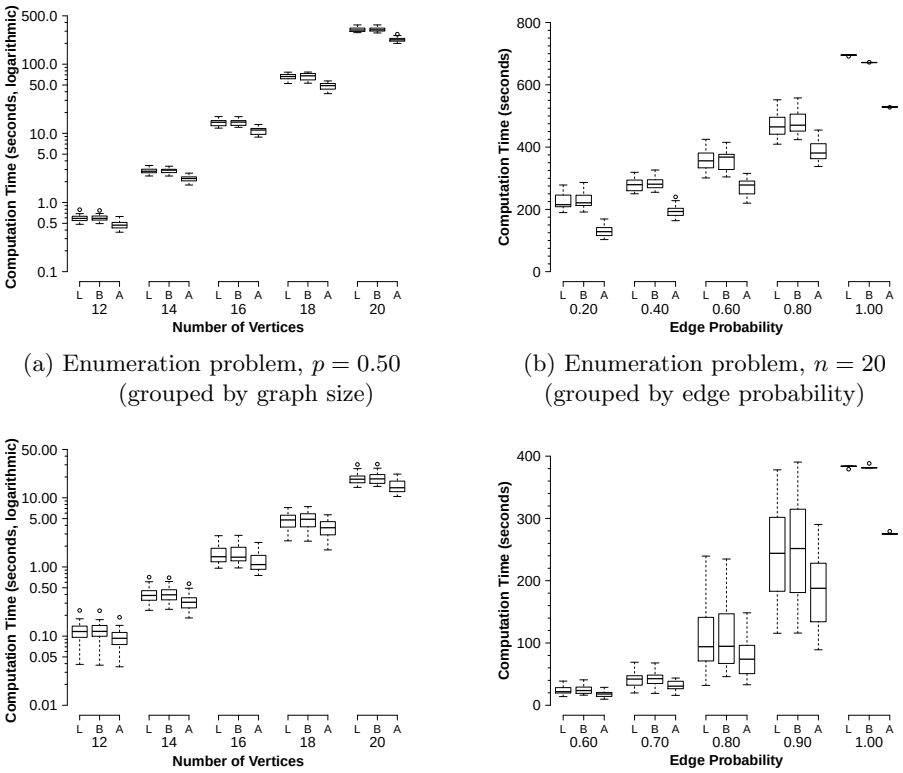
We evaluate our encodings based on a set of graphs (generated using the Erdös-Rényi random graph model) of different sizes n between 12 and 20 vertices. The graphs were generated using a fixed edge probability p (i.e., the probability of adding an edge between a pair of vertices). In our tests, we investigated the influence of the edge probability for a range between 20 and 100 percent. This allows us to analyze the impact of the size of the input graph and the impact of the graph density on the overall performance of our encodings separately. For each edge probability and for each graph size, 20 instances were generated and tested. The benchmark results were obtained using a machine with two Intel Xeon E5345 @ 2.33GHz processors and 48 GB RAM running openSUSE 11.4 x64. Each test run, using Clingo 4.3.0 [16], was limited to a single core and 256 MB RAM with a time limit of 15 minutes. Due to the fact that preliminary tests showed no significant variance between the results of repeatedly executed experiments, we only used one test run for each instance and encoding to obtain the computation times which are represented in Figure 2.

4.2 Benchmark Discussion

Performance comparison of the different encodings. When interpreting Figure 2, one can see that the original loop encoding and the loop encoding with restriction to the border vertices are almost equally efficient while the alternative encoding outperforms them in every case. We assume that the reason why the restriction to border vertices does not show a significant performance improvement is the fact that rules 4-6 of Encoding 1 introduce a kind of indirection that mitigates the desired performance gain. In contrast to this, the strengths of the alternative encoding (where, like in Encoding 1, only the border vertices are considered within the subset check) are visible over the whole range of instances under investigation and we assume that this also holds for larger graphs.

Influence of the graph size on computation time. Figures 2a and 2c show the program's runtime in the presence of different graph sizes. In particular note the exponential explosion of the computation time in relation to the graph size for both problem variants. Already for 22 vertices the time limit of 15 minutes was exceeded by almost every instance. For the optimization problem, the variation in runtime is much higher as the structure of the graph instance plays an important role (see also the discussion of the influence of the edge probability on the program runtime right below). Figures 2a and 2c only cover the experiments for instances of edge probability 0.5, but these observations also apply to other edge probabilities which are not depicted here.

Influence of the edge probability on computation time. Figures 2b and 2d illustrate the strong dependency of the computation time on the edge probability for a fixed graph size of 20 vertices. We assume that this can be explained by the fact that "small" solutions (i.e., secure sets which contain a small number of vertices) become less likely when the degree of connectedness increases. Interestingly, the optimization encodings are much more sensitive to changes in the



(a) Enumeration problem, $p = 0.50$ (grouped by graph size)

(b) Enumeration problem, $n = 20$ (grouped by edge probability)

(c) Optimization problem, $p = 0.50$ (grouped by graph size)

(d) Optimization problem, $n = 20$ (grouped by edge probability)

Fig. 2: Performance results. The following abbreviations are used for the different encoding variants: L - "Loop Encoding", B - "Loop Encoding with Restriction to the Border Vertices" and A - "Alternative Encoding".

edge probability than the enumeration variant. This is because the solver does not have to check any larger candidates for secure sets in the encodings of the optimization problem after a solution of smaller size has been found, while it still has to check the whole search space when solving the enumeration variant. Finally, note that for $p = 1$ all generated instances represent the same graph (i.e. a clique), which explains the very small difference in runtime.

5 Conclusion

In this paper we have studied the problem of finding secure sets in graphs, which is relevant, for example, in the context of studying relations between groups of people. We have presented some alternative characterizations of secure sets that we put to work in an ASP-based implementation.

From an ASP modeling point of view, finding secure sets in graphs is an interesting problem as it requires disjunction as well as aggregates. It is therefore

an attractive candidate for testing the performance of ASP systems that allow solving of problems harder than NP. In this work we have presented different encodings that we believe to be useful for benchmarking ASP systems in the future. This is of particular interest because so far problems harder than NP are underrepresented among common collections of benchmark problems.

Recursive aggregates have posed a challenge in writing encodings for the secure set problem. Our work witnesses that even some natural problems seem to require relatively involved tricks in order to properly encode the required arithmetic by means of aggregates in ASP. Moreover, current restrictions in the DLV solver prevented us from gathering experimental data for this system.

Our experiments show a strong dependency of the execution time on the graph size and the edge probability for all of our encodings, and the problem of enumerating all secure sets in a graph is shown to be very challenging for today's ASP solvers even for relatively small instances. Our encoding that refrains from looping over vertices (Encoding 2) outperforms our other encodings in all cases and is therefore assumed to be a good starting point for further investigations.

We hope that in future versions of ASP systems restrictions with respect to (recursive) aggregates are alleviated allowing further research on the secure set problem as well as on similar problems requiring the full power of ASP. In future work, we want to perform further analyses for such problems, which should include other ASP systems (such as DLV) in order to obtain a more comprehensive picture of how ASP can cope with them.

Acknowledgments

The authors would like to thank Mario Alviano, Wolfgang Faber, Roland Kaminski and Francesco Ricca for their helpful comments about recursive aggregates and their implementation in various ASP systems. This work was supported by the Austrian Science Fund (FWF): P25607, and by the Vienna University of Technology special fund "Innovative Projekte" (9006.09/008).

References

1. M. Alviano, F. Calimeri, G. Charwat, M. Dao-Tran, C. Dodaro, G. Ianni, T. Krennwallner, M. Kronegger, J. Oetsch, A. Pfandler, J. Pührer, C. Redl, F. Ricca, P. Schneider, M. Schwengerer, L. K. Spendier, J. P. Wallner, and G. Xiao. The fourth answer set programming competition: Preliminary report. In *Proc. of LPNMR'13*, volume 8148 of *LNCS*, pages 42–53. Springer, 2013.
2. M. Alviano, C. Dodaro, W. Faber, N. Leone, and F. Ricca. WASP: A native ASP solver based on constraint learning. In *Proc. of LPNMR'13*, volume 8148 of *LNCS*, pages 54–66. Springer, 2013.
3. M. Alviano, W. Faber, N. Leone, S. Perri, G. Pfeifer, and G. Terracina. The disjunctive Datalog system DLV. In *Proc. of Datalog'10*, volume 6702 of *LNCS*, pages 282–301. Springer, 2011.
4. Asparagus – A web-based benchmarking environment for answer set programming. `http://asparagus.cs.uni-potsdam.de`. Accessed May 19, 2014.

5. G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.
6. R. C. Brigham, R. D. Dutton, and S. T. Hedetniemi. Security in graphs. *Discrete Applied Mathematics*, 155(13):1708–1714, 2007.
7. F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub. ASP-Core-2: 4th ASP competition official input language format. `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.01c.pdf`, 2013.
8. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate functions in DLV. In *Proc. of ASP'03*, volume 78, pages 274–288. CEUR-WS, 2003.
9. T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3–4):289–323, 1995.
10. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.
11. T. Eiter, G. Ianni, and T. Krennwallner. Answer set programming: A primer. In *Proc. of Summer School on Reasoning Web 2009*, volume 5689 of *LNCS*, pages 40–110. Springer, 2009.
12. W. Faber, G. Pfeifer, and N. Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011.
13. P. Ferraris. Answer sets for propositional theories. In *Proc. of LPNMR'05*, volume 3662 of *LNCS*, pages 119–131. Springer, 2005.
14. P. Ferraris. Logic programs with propositional connectives and aggregates. *ACM Trans. Comput. Log.*, 12(4):25, 2011.
15. P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5(1–2):45–74, 2005.
16. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.
17. M. Gebser, T. Schaub, S. Thiele, and P. Veber. Detecting inconsistencies in large biological networks with answer set programming. *Theory and Practice of Logic Programming*, 11(2–3):323–360, 2011.
18. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3–4):365–386, 1991.
19. A. Harrison, V. Lifschitz, and F. Yang. On the semantics of Gringo. In *Proc. of ASPOCP'13*, pages 129–142, 2013.
20. Y. Y. Ho. *Global Secure Sets of Trees and Grid-like Graphs*. PhD thesis, University of Central Florida, Orlando, USA, 2011.
21. M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog decision support system for the space shuttle. In *Proc. of PADL'01*, pages 169–183. Springer, 2001.
22. F. Ricca, G. Grasso, M. Alviano, M. Manna, V. Lio, S. Iiritano, and N. Leone. Team-building with answer set programming in the Gioia-Tauro seaport. *Theory and Practice of Logic Programming*, 12(3):361–381, 2012.
23. K. H. Shafique. *Partitioning a Graph in Alliances and its Application to Data Clustering*. PhD thesis, University of Central Florida, Orlando, USA, 2004.
24. T. C. Son and E. Pontelli. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming*, 7(3):355–375, 2007.