Institute for Artificial Intelligence

Department Analytic Computing

University of Stuttgart
Universitätsstraße 32
D-70569 Stuttgart

Bachelorarbeit

Generating Random Knowledge Graphs from Rules

Gabriel Timon Glaser

Course of Study:	Informatik
Examiner:	Prof. Dr. Steffen Staab
Supervisor:	Dr. Daniel Hernández Jiaxin Pan, M.Sc.
Commenced:	April 07, 2024
Completed:	July 31, 2024

Abstract

A knowledge graph is a datastructure that is capable of storing knowledge. Besides that, there are several methods that use knowledge graphs to derive more information. These methods need to be validated with example knowledge graphs. However, real data might not be available or not contain desired properties. Thus, there are use cases that benefit from the generation of synthetic knowledge graphs. To define a synthetic knowledge graph, there is the need of a characterization that expresses how the synthetic data should look. In this thesis, I use Horn clauses for this characterization because of their good balance of expressiveness and complexity, their use in the field of rule mining, and their base role in the logical language Datalog. As clauses are usually not represented perfectly in real data, the goal of this thesis is to generate a knowledge graph that does not perfectly fulfil given Horn clauses, but in a desired degree of fulfillment.

During the thesis, I developed and implemented two modifiable algorithms to generate knowledge graphs. On the one hand, I adapted the general hill climbing technique to generate knowledge graphs. On the other hand, I implemented a greedy algorithm which orders a given set of Horn clauses using logical subsumptions between their bodies, and then add edges to fulfil one Horn clause after the other, in the computed order. Both algorithms aim to fulfil the goal of this thesis by generating synthetic knowledge graphs according to given Horn clauses, each with a degree of fulfilment. The degree of fulfilment of any Horn clauses is characterized by body support, the number of times the premise of the Horn clauses is fulfilled, and support, the number of times the premise and conclusion of the Horn clauses are fulfilled. Additionally, there is the confidence which is the fraction of support and body support, i.e., the percentage of cases the Horn clause is fulfilled. All code is published such that anyone can try it.

During the evaluation, random sets of Horn clauses were produced and the implementations generated corresponding knowledge graphs. Generated knowledge graphs were compared by considering the difference between the expected and the actual degree of fulfilment for each Horn clause. The result is that generation variant hill climbing with the initial state set to the result of the greedy algorithm with rule order based on subsumption yields the best results. Also, the difficulty of generating a good knowledge graph increases along with the overlapping degree of the input set of Horn clauses. Note that the overlapping degree reflects how many relation names occur in how many Horn clauses of the set. Lastly, the state-of-the-art mining tool AMIE found many Horn clauses in the generated graphs which were not intended by the set given to the generator.

Kurzfassung

Ein Wissensgraph ist eine Datenstruktur mit der Möglichkeit Wissen zu speichern. Zudem gibt es einige Techniken, die Wissensgraphen nutzen, um neue Informationen zu folgern. Um solche Techniken zu validieren, benötigt man beispielhafte Wissensgraphen. Jedoch sind echte Daten oft nicht verfügbar oder verfügbare Daten enthalten nicht die erwünschten Eigenschaften. Deswegen gibt es Anwendungsfälle, die von synthetischen Wissensgraphen profitieren können. Dazu braucht man Charakterisierungseigenschaften von Wissensgraphen, dass Nutzer definieren können, wie die synthetischen Daten aussehen sollen. In dieser Bachelorarbeit nutze ich dafür Horn-Klauseln, welche ein gutes Gleichgewicht zwischen Ausdrucksfähigkeit und Komplexität darstellen, bereits im Gebiet Rule Mining verwendet werden und die Basis der logischen Sprache Datalog darstellen. Da Klauseln in echten Daten meistens nicht perfekt erfüllt sind, ist das Ziel dieser Bachelorarbeit Wissensgraphen zu generieren, die gegebene Horn-Klauseln kontrolliert nicht perfekt erfüllen.

Im Verlauf der Bachelorarbeit habe ich zwei konfigurierbare Verfahren, die Wissensgraphen generieren, entwickelt und implementiert. Einerseits habe ich die allgemeine Hill Climbing Methode adaptiert, um Wissensgraphen zu generieren. Andererseits habe ich einen greedy Algorithmus entwickelt, der die gegebenen Horn-Klauseln anhand logischer Teilmengen sortiert und daraufhin Graphkonstrukte für jede Horn-Klausel der Reihe nach dem Wissensgraphen hinzufügt. Beide verfolgen das Ziel synthetische Wissensgraphen zu generieren, die gegebene Horn-Klauseln mit Erfüllungsgrad repräsentieren. Der Erfüllungsgrad ist charakterisiert durch den Body Support, die Anzahl an Fällen, in denen die Vorbedingung der Horn-Klausel erfüllt ist, und dem Support, der Anzahl an Fällen, in denen die Vorbedingung und Nachbedingung der Horn-Klausel erfüllt ist. Zusätzlich gibt es noch die Confidence, die als Quotient von Support und Body Support definiert ist. Sie repräsentiert den Prozentsatz an Fällen wie oft die Horn-Klausel im Wissensgraphen erfüllt ist. Der gesamte Programmiercode ist öffentlich, dass jeder diesen testen kann.

Während der Evaluation wurden zufällige Mengen von Horn-Klauseln generiert und als Eingabe an die Implementierungen gegeben, um daraus Wissensgraphen zu generieren. Generierte Wissensgraphen wurden anhand der Differenz des erwartetem und tatsächlichem Erfüllungsgrad jeder Horn-Klausel verglichen. Das Ergebnis ist, dass die Hill Climbing Methode, mit Initialzustand gesetzt auf die Generierung des greedy Algorithmus, die besten Ergebnisse liefert. Außerdem steigt die Schwierigkeit für den Generator einen guten Wissengraphen zu generieren, wenn der Überlappungsgrad der gegebenen Horn-Klauseln hoch ist. Der Überlappungsgrad einer Menge von Horn-Klauseln zeigt auf wie viele Relationsnamen in wie vielen der Horn-Klauseln vorkommen. Zuletzt kam heraus, dass das Rule Mining Tool AMIE viele Horn-Klauseln in generierten Graphen gefunden hat, welche nicht in der dem Generator gegebenen Menge von Horn-Klauseln enthalten waren.

Contents

1 Introduction **13**

2 Background **17**
2.1 Knowledge Graph 17
2.2 First-Order Formula 18
2.3 Rule 18
2.4 Hill Climbing 19

3 Related Work **23**
3.1 Algorithms which generate Random Graphs 23
3.2 Algorithms which generate Graphs from Constraints 24

4 Problem Statement **27**

5 Knowledge Graph Generation Algorithms **29**
5.1 Hill Climbing with Knowledge Graph Updates 29
5.2 Greedy Edge Generation based on Rule Body Subsumption 32

6 Implementation **35**
6.1 Knowledge Graph Datastructure 35
6.2 Computing the Loss Function 36
6.3 Computing Rule Body Subsumption 40

7 Evaluation **43**
7.1 Setting 43
7.2 Algorithm Comparison 44
7.3 Follow-Up Analysis 46

8 Conclusion **55**

Bibliography **57**

List of Figures

1.1 Example synthetic knowledge graphs that aim to fulfil example rules 14

2.1 Example knowledge graph which contains knowledge about people and locations [Don21] 17
2.2 Example knowledge graph to illustrate how the body support is counted 19
2.3 Example illustration of an objective function for hill climbing [RRN20] 20

5.1 Initial graph to illustrate how hill climbing generates a graph 31
5.2 Example knowledge graph generated by the greedy edge generation algorithm . . 33

6.1 Visualization of how edges are stored in the implemented knowledge graph data structure for a single relation name 35
6.2 Example knowledge graph to illustrate how the body support is computed 39

7.1 Visualization of number of relation names to sample from with resulting overlapping degree of generated set of rules 47
7.2 Plot of overlapping degree of any generated rule set with corresponding loss function of knowledge graph generation 48
7.3 Visualization of number of rules in a rule set with loss function of resulting knowledge graph generation 48
7.4 Visualization of number of overlapping degree and loss function of resulting knowledge graph generation, only considering sets of rules C with $|C| = 18$ . . . 49
7.5 Visualization of number of overlapping degree and loss function of resulting knowledge graph generation, only considering sets of rules C with $|C| = 14$ . . . 50
7.6 Visualization of number of overlapping degree and loss function of resulting knowledge graph generation, only considering sets of rules C with $|C| = 3$ 50
7.7 Knowledge graph generated by hill climbing algorithm with greedy initial graph (subsumption order) and the rules of Table 7.3 as input 51

List of Tables

1.1 Example rules, each with body support and confidence 14

5.1 Example rules, each with body support and support, for which no graph G with exactly matching support values can be generated 29

6.1 Example initial table to illustrate how the body support of a rule is computed . . 39
6.2 Example illustration of how Algorithm 6.3 extends a table of edges by a new set of edges 39

7.1 Aggregated performance metric measurements of running each knowledge graph generation variant on 97 sets of rules 45
7.2 Number of climbing steps for hill climbing variants 46
7.3 Rule set generated by GenerateRuleSet$(4, 3, 4)$ (Algorithm 7.1) with expected body supports chosen randomly from $\{2, 3, 4, 5\}$ and expected support chosen randomly from 0 between the previous body support 51
7.4 Set of rules randomly generated by GenerateRuleSet$(8, 5, \{r_1, r_2, \cdots, r_5\})$ (Algorithm 7.1) with their corresponding body support and support 52
7.5 Loss function values of all generation variants on a set of eight rules while the loss of climbing variants is aggregated from ten climbing executions 52
7.6 Number of mined rules for each generation variant while results of hill climbing variants are aggregated by average ± standard deviation across ten executions . . 53
7.7 Graph statistics for graphs generated by each generation variant while the values for hill climbing variants are aggregated by average ± standard deviation across ten executions 53

List of Algorithms

2.1 General hill climbing algorithm to solve discrete problems 21
5.1 Initial graph for the hill climbing graph generator 30
5.2 Hill climbing graph generator 31
5.3 Algorithm that orders a rule set 33
5.4 Greedy graph generator that fulfils one rule after the other 34
6.1 Naive algorithm to compute the body support of a rule c measured in graph G . . 36
6.2 Algorithm to compute all valid groundings of a rule body in a graph 37
6.3 Algorithm to extend valid groundings for a conjunction of atoms by the edges of another atom 38
6.4 Recursive algorithm to check whether a rule body subsumes another rule body . . 40
7.1 Algorithm that generates a random set of rules 44

1 Introduction

Knowledge graphs play a key role in representing knowledge within organizations and communities, in several application scenarios and in integrating data at a large scale [HBC+21; TKBG23]. Developing methods to learn models from knowledge graphs, e.g., knowledge graph embeddings [WMWG17], require example knowledge graphs to conduct a validation. Using real-world knowledge graphs on this validation provides a realistic insight into the performance of these methods. However, using real-world knowledge graphs can be difficult. For example, real-world graphs often contain private information, so their access is restricted [LLP+15]. Moreover, available real-world graphs may not include desired properties for the evaluation of a method or their complexity makes the understanding of the method's performance difficult. Therefore, there are use cases for which it is worth generating synthetic data.

Real knowledge graphs follow constraints like "people having the same parents are siblings" or "a professor is a person". Developing a tool that generates random knowledge graphs that satisfy constraints requires a type of constraint such that the problem remains tractable and interesting. The problem of determining whether a graph fulfils a set of first-order logic formulas is undecidable [Tur37]. Nevertheless, considering only *Horn clauses* out of first-order logic makes the decision problem (*Horn-satisfiability*) decidable and tractable (P-complete) [CN10]. Also, Horn clauses are the base of logical languages such as Datalog [CGT89], they have been used to describe the expressive power of knowledge graph embedding models and are called *logical patterns* in the knowledge graph embedding literature [GNHS23].

A Horn clause has the form $c = \forall x \forall y(\text{parent}(x, y) \rightarrow \text{child}(y, x))$. In the context of knowledge graphs, c means that if there is an edge from a node A to node B labeled with the relation name "parent", then there must be an edge from B to A labeled with the relation name "child". These clauses can be unknown in real knowledge graphs and extracting them is a problem for which there exist several tools called *rule miners* [GTHS13]. In what follows the word *rule* is considered a synonym of Horn clause.

In real knowledge graphs, rules are not always satisfied because real data usually is incomplete or sometimes may even be contradictory. For example, a graph may contain the "parent"-edge from A to B, but the inverse "child"-edge is missing. Rule miners annotate extracted rules with statistics as the *body support* which is the number of cases where the premise of the rule is satisfied. Also, there is the *confidence* which is the proportion of cases where the premise and conclusion are satisfied. Thus, the goal of this thesis is not to produce a knowledge graph that fulfils all given rules perfectly, but a number of times that follows a given body support and confidence.

As a result, the problem of this thesis is to create a tool that generates a knowledge graph such that each given rule's body support as well as the rule's confidence (measured in the graph) is as close as possible in comparison with its expected body support and confidence. Notice that this problem differs from Horn-satisfiability, because the tool of this thesis needs to approximately satisfy the rules based on given body supports and confidences.

Note that this problem cannot be solved with a naive approach that fulfils one rule after the other, because adding edges to a graph that fulfil one rule could violate another rule. This can be seen by considering the rules c_1, c_2 with their expected body support and confidence in Table 1.1.

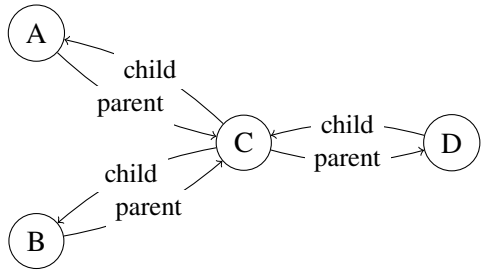
Rule	Body Support	Confidence
$c_1 = \forall x \forall y (\text{parent}(x, y) \rightarrow \text{child}(y, x))$	3	1
$c_2 = \forall x \forall y \forall z ((\text{parent}(x, y) \wedge \text{parent}(z, y)) \rightarrow \text{sibling}(x, z))$	6	$\frac{1}{2}$

Table 1.1: Example rules, each with body support and confidence

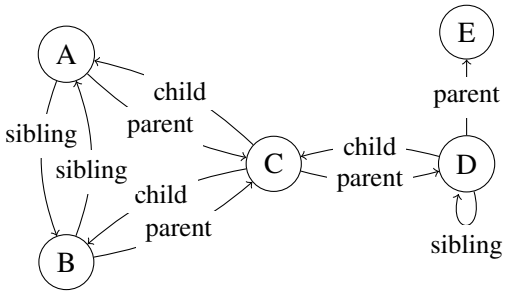
Aiming to fulfil the body support and confidence of rule c_1, a generator could generate the knowledge graph depicted in Figure 1.1a which exactly matches the expected body support and expected confidence. However, the body support of c_2 is 5 instead of the expected 6. Indeed, the set of tuples supporting the body of rule c_2 is

$$\{(A, B), (B, A), (A, A), (B, B), (C, C)\}.$$

Intuitively, each of these tuples corresponds to a pair of variable instances (x, z) that have a common parent, i.e., each two nodes semantically are siblings. A generator could address this mismatch by adding a node E and the edge (D, parent, E) to the firstly generated knowledge graph. Since (D, D) will be another valid tuple for c_2, the expected body support of 6 is reached. To match the confidence of rule c_2, which is 0 so far, the generator could add three edges with the relation name "sibling", e.g., $(A, \text{sibling}, B)$, $(B, \text{sibling}, A)$ and $(D, \text{sibling}, D)$ which results in the knowledge graph depicted in Figure 1.1b.



(a) Generated knowledge graph to fulfil the expected body support and confidence of rule c_1



(b) Expansion of graph depicted in Figure 1.1a to fulfil the expected body support and confidence of rule c_2

Figure 1.1: Example synthetic knowledge graphs that aim to fulfil example rules

While the second knowledge graph fulfils the expected body support and confidence of rule c_2, it does no longer fulfil the expected body support and confidence of rule c_1. Indeed, the body support of rule c_1 changed to 4 and the confidence to $\frac{3}{4}$. Hence, the problem of this thesis cannot be trivially solved, because trying to match the expected values for one rule may affect the values of another rule.

Previously developed graph generation algorithms often do not consider constraints which have a similar expressiveness than Horn clauses [DRW98; ER59; KKS+23; KSY23; TK02; YW09]. For example, graph generators that aim to yield graphs with a certain node degree distribu-

tion [DRW98; KKS+23; KSY23; YW09]. Besides that, graph generation algorithms that consider similar constraints than Horn clauses are either limited, e.g., no Horn clauses with more than four atoms [HMd+24], or they do not allow to specify a metric similar to *body support* and *confidence* [Coo23; ON16; TKBG23; VJ24], i.e., the generators do not allow to control the degree of fulfilment of a constraint.

Besides, the problem of this thesis is further specified as an optimization problem with a loss function that indicates the level of satisfaction of the rules regarding the expected body support and confidence. During the thesis, two algorithms were developed. The first algorithm is an adaptation of the general discrete optimization technique hill climbing. During the climbing, the graph is randomly changed many times and the improvements (with respect to the loss function) are saved. The second algorithm fulfils the rules in a greedy way, i.e., one after the other. To this end, the rules are ordered according to subsumption relations between their bodies. If the body of a rule c_i subsumes the body of a rule c_j, adding edges that fulfil c_i before adding edges that fulfil c_j would mean that generated instances of c_i can be reused for c_j.

Both algorithms are evaluated to answer the following research questions:

RQ1. What is the performance of the knowledge graphs produced by the developed generation algorithms regarding the fulfilment of the input rules? How is this performance affected by modifications in the algorithms? On the one hand, the initial graph can be changed for the adaption of hill climbing. On the other hand, the order of rules can be changed for the greedy algorithm.

RQ2. Do the developed algorithms generate graphs that fulfil unintended rules, i.e., rules that are not equivalent to any of the input rules?

Outline. The remaining thesis is structured as follows. Firstly, Chapter 2 explains terms and definitions to understand subsequent chapters. Then, Chapter 3 presents related works which generate graphs of some kind. After that, the problem addressed by this thesis is formally defined in Chapter 4. Then, the developed algorithms are discussed in Chapter 5 while key parts of their implementation are explained in Chapter 6. Next, Chapter 7 depicts the results of the implemented tools to answer the proposed research questions. Finally, Chapter 8 summarizes the key findings of this thesis and provides an outlook for further research.

2 Background

This chapter introduces definitions and terminology to understand the remaining parts of the thesis. First, there is the definition of a knowledge graph, which is the datastructure the tool of this thesis should generate, in Section 2.1. After that, Section 2.2 introduces first-order formulas and how it is defined whether a knowledge graph fulfils such a formula. Then, Section 2.3 introduces rules and the related terms body support, support and confidence. Finally, how general hill climbing addresses discrete problems is explained in Section 2.4.

2.1 Knowledge Graph

A knowledge graph G is a triple (N, R, E) while N and R are two finite disjoint sets whose elements are called *nodes* and *relation names*, and $E \subseteq N \times R \times N$ is the set of *edges*. In the context of this thesis, an edge represents a relationship between a pair of nodes. For instance, if painted $\in R$, DA VINCI $\in N$ and MONA LISA $\in N$, then (DA VINCI, painted, MONA LISA) $\in E$ is interpreted as the relationship stating that Da Vinci painted Mona Lisa.

Example 2.1
An example knowledge graph can be seen in Figure 2.1.

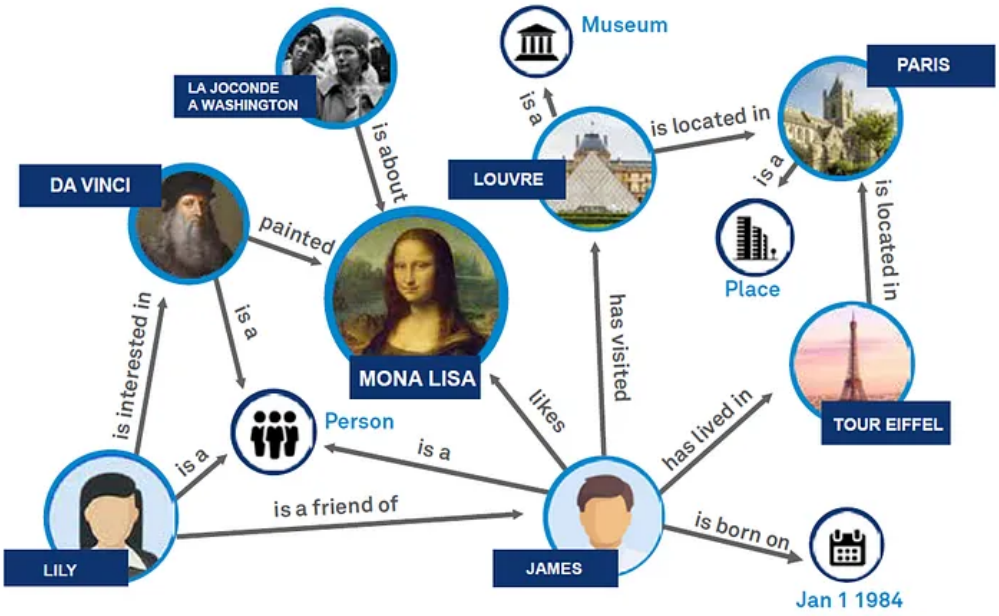


Figure 2.1: Example knowledge graph which contains knowledge about people and locations [Don21]

2.2 First-Order Formula

In this thesis, first-order formulas are defined over the vocabulary of knowledge graphs $G = (N, R, E)$ with an infinite set of variables W disjoint with $N \cup R$. Therefore, atoms $r(u, v)$ consist of a relation name $r \in R$ and terms $u, v \in N \cup W$. Then, a *first-order formula* is defined recursively as to be either an atom $r(u, v)$ or one of the following: $\varphi \wedge \psi$, $\neg\varphi$, and $\exists x\varphi$ while $x \in W$ and φ, ψ are first-order formulas. The formulas $\varphi \vee \psi$, $\forall x\varphi$ and $\varphi \rightarrow \psi$ are written as an abbreviation for $\neg(\neg\varphi \wedge \neg\psi)$, $\neg(\exists x \neg\varphi)$ and $\neg\varphi \vee \psi$, respectively. Given a finite set of variables $X = \{x_1, \ldots, x_n\}$, $\exists X\, \varphi$ and $\forall X\, \varphi$ as are seen as abbreviations for $\exists x_1 \ldots \exists x_n \varphi$ and $\forall x_1 \ldots \forall x_n \varphi$, respectively. A variable x is called *quantified* in a formula, if the formulas has either the form $\exists x\varphi$ or one of the following: $\psi_1 \wedge \psi_2$, or $\neg\psi_1$, where x is quantified in formulas ψ_1 and ψ_2. In contrast, a variable is called *free* if it is not quantified in formula. Given a formula φ with free variables, an *instance* of φ is a formula ψ resulting from consistently replacing each free variable x in φ with a node $n \in N$. A formula without free variables is called *closed*. A *first-order theory* is a set of closed formulas. Given two *first-order theories* Σ_1 and Σ_2, the entailment relationship $\Sigma_1 \models \Sigma_2$ is defined as usual in function of the models of the formulas, see [Lib04].

The theory of a graph is the set Σ_G of first-order logic atoms $\{r(a, b) \mid (a, r, b) \in G\}$. Given a first order formula φ and a graph G, G *satisfies* φ if and only if $\Sigma_G \models \varphi$, where $\models$ denotes the usual first-order entailment relation. Given a first-order formula φ, $\mathrm{var}(\varphi)$ denotes the set of *free variables* (non-quantified) of formula φ. Given a knowledge graph $G = (N, R, E)$, a first-order formula φ and a mapping $\mu : \mathrm{var}(\varphi) \rightarrow N$, $\mu(\varphi)$ represents the formula that results of replacing each free variable $x \in \mathrm{var}(\varphi)$ with $\mu(x)$. The mapping μ *satisfies* formula φ over G, i.e., $\mu \models_G \varphi$, if and only if $\Sigma_G \models \mu(\varphi)$. Finally, $\mathrm{dom}(\mu)$ and $\mathrm{range}(\mu)$ are defined as the domain and range of a mapping μ.

2.3 Rule

A rule is a first-order formula of the form $\forall x_1 \ldots \forall x_n (B \rightarrow H)$ while B is a conjunction of atoms, called the *body*, and H is a single atom, called the *head*. Furthermore, all atoms in c have the form $r(u, v)$ where each u, v is either a node or a variable. Lastly, all variables occurring in B are universally quantified, i.e., $\mathrm{var}(B \rightarrow H) = \{x_1, \ldots, x_n\}$, and all variables occurring in H occur in B, too. For simplicity, rule quantifiers are usually omitted.

Example 2.2
For instance, given a graph $G = (N, R, E)$, the rule

$$c_{ex} = (\mathrm{parent}(x, y) \wedge \mathrm{parent}(z, y)) \rightarrow \mathrm{sibling}(x, z)$$

states that nodes who have the same parents are siblings. Hence, the existence of the edges $(a, \mathrm{parent}, b) \in E$ and $(c, \mathrm{parent}, b) \in E$ for nodes $a, b, c \in N$ implies the existence of the edge $(a, \mathrm{sibling}, c) \in E$.

Let rule c be a rule of the form $B(u, v) \rightarrow H(u, v)$ where u and v denote the terms occurring in the head of the rule. Additionally, let $\{x_1, \ldots, x_n\} = \text{var}(B) \setminus \text{var}(H)$ be variables that are not in $H(u, v)$. Then, the *body support*, *support* and *confidence* of c are defined as follows:

$$\begin{aligned} \text{BodySupp}(c) &= |\{(a, b) \mid \exists x_1 \ldots \exists x_n B(a, b)\}|, \\ \text{Supp}(c) &= |\{(a, b) \mid \exists x_1 \ldots \exists x_n (B(a, b) \wedge H(a, b))\}|, \\ \text{Conf}(c) &= \frac{\text{Supp}(c)}{\text{BodySupp}(c)}. \end{aligned}$$

Intuitively, the body support represents the number of times the body of c is fulfilled in the knowledge graph. However, edges that correspond to an instance of c's body contributes plus 1 to the body support score only if the replacements of the variables in the head are unique.

Example 2.3
For instance, consider the rule c_{ex} of Example 2.2 and the knowledge graph in Figure 2.2.

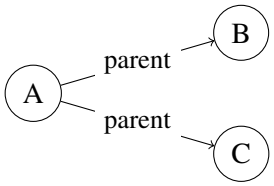


Figure 2.2: Example knowledge graph to illustrate how the body support is counted

One could think that the body support is 2, because it is possible to replace the variables (x, y, z) of rule c_{ex} by (A, B, A) and (A, C, A). However, the body support only considers the variables of c_{ex}'s head, i.e., (x, z). Therefore, there is only the possibility (A, A) and thus, the body support is 1. Moreover, the support of c_{ex} is 0, because out of all tuples (a, b) which count towards the body support score (which is only (A, A) here), no corresponding edge $(a, \text{sibling}, b)$ is present in the graph, i.e., $(A, \text{sibling}, A)$ is not present. Finally, the confidence is 0, because $\frac{0}{1} = 0$.

2.4 Hill Climbing

Hill climbing is a technique which can be applied to solve discrete problems [RRN20]. A discrete problem is characterized by a *state space* $\mathcal{S}$ which is the set of all possible states, including solutions. Then, each state $s \in \mathcal{S}$ has a neighbourhood $\textsc{Neighbours}(s) \subseteq \mathcal{S}$ which is a finite set of states that are reachable from s. Next, there is an *objective function* $o : \mathcal{S} \rightarrow \mathbb{R}$ which assigns each state s a real number $o(s)$. $o(s)$ represents the quality of s, i.e., $o(s) > o(s')$ means that s is considered to be closer to a solution than state s'. Finally, hill climbing starts in an initial state and moves to neighbouring states that have a better quality until there is no better neighbour anymore.

Example 2.4
The problem which should be solved is finding a chess permutation that contains 8 queens on a chess board while no queen attacks another. Thus, $\mathcal{S}$ can be defined as all figure permutations of 8 queens. Afterwards, $\textsc{Neighbours}(s)$ is defined as the set of all 8-queens-permutations $s' \in \mathcal{S}$ that

are a single, legal chess move away of s. Then, $o(s)$ can be defined as the number of queens which are not attacking any other. Therefore, $o(s) = 8$ would mean that s represents a solution, $o(s') = 7$ means s' is close to a solution and $o(s'') = 2$ means s'' is not close to a solution.

Consider Figure 2.3 which illustrates how the objective function o can look like.

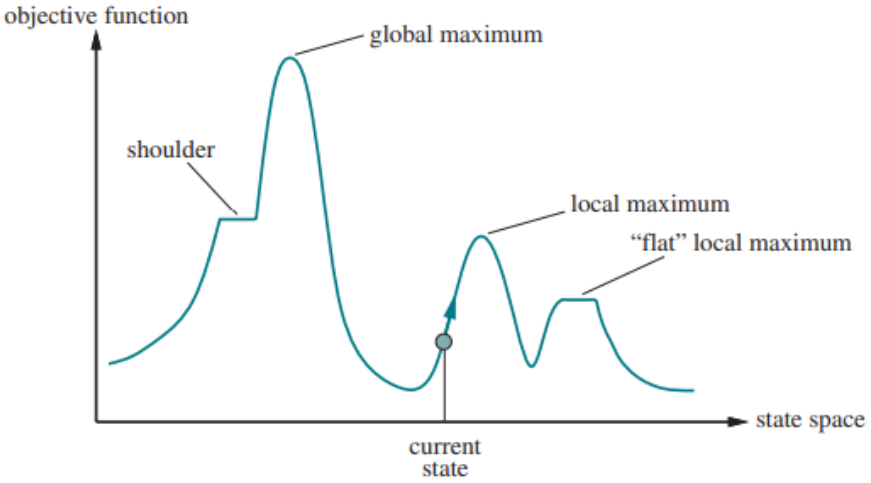


Figure 2.3: Example illustration of an objective function for hill climbing [RRN20]

When starting the climbing process at "current state", the algorithm will move towards the right, because the local neighbourhood contains states with a higher objective function value in that direction. Hill climbing will continuously move to the right until the local maximum is reached. Then, hill climbing terminates and yields the state at the optimum. Therefore, hill climbing is not able to reach the global maximum.

Hill climbing is classified as a local search algorithm, because at each algorithm step (being at state s_{current}), hill climbing only considers the neighbouring states $\textsc{Neighbours}(s_{\text{current}})$. The default hill climbing variant will replace s_{current} with

$$s_{\text{next}} = \underset{s \in \textsc{Neighbours}(s_{\text{current}})}{\arg\max} \; o(s)$$

at each step. If $s_{\text{current}} = s_{\text{next}}$, the algorithm terminates and yields s_{current}. The pseudo code of hill climbing is depicted in Algorithm 2.1. Intuitively, the algorithm replaces the current state s with the neighbour with maximal value of o at each iteration step (lines 3 to 13). In case there is no neighbour which has a higher value (line 9), the algorithm stops and yields the current state s.

Algorithm 2.1 General hill climbing algorithm to solve discrete problems

Require: Set of states $\mathcal{S}$, objective function $o : \mathcal{S} \to \mathbb{R}$, neighbourhood function NEIGHBOURS : $\mathcal{S} \to 2^{\mathcal{S}}$ such that each NEIGHBOURS(s) is finite and an initial state $s_0 \in \mathcal{S}$.
Ensure: A state $s' \in \mathcal{S}$ such that $o(s') \geq o(s_0)$.

```
1:  Let s ← s_0.
2:  while true do
3:      Let s_max ← s.
4:      for all s' ∈ NEIGHBOURS(s) do
5:          if o(s') > o(s_max) then
6:              s_max ← s'
7:          end if
8:      end for
9:      if s = s_max then
10:         return s
11:     else
12:         s ← s_max
13:     end if
14: end while
```

Example 2.4 shows that hill climbing does not reach the global optimum in general. A possibility to address this issue could be to increase the size of the neighbourhood. However, bigger neighbourhoods increase the complexity which could lead to an infeasible runtime. Nevertheless, there are variants of hill climbing which also address the issue of "getting stuck in a local optimum". For example, *stochastic hill climbing* chooses any random neighbour that has a higher objective function value, *simulated annealing* inserts down-hill moves, i.e., replace the current state with a state that has a lower objective function value, or *local beam search* which concurrently performs hill climbing n times and shares the found states across all n climbing processes.

3 Related Work

Related work is organized in two groups. On the one hand, the first group consists of graph generation algorithms that generate nodes and then add edges according to given distributions or procedures which try to generate the expected distributions. On the other hand, the second group consists of algorithms that generate graphs that satisfy a set of formally defined constraints.

3.1 Algorithms which generate Random Graphs

These algorithms generate *random graphs*, not knowledge graphs, that satisfy general properties of the graph, as the number of nodes, the probability of edges, and the in- and out-degree of nodes.

Random graphs were introduced by Erdös and Rényi [ER59] in 1959 when they proposed graphs $G(n, p)$ which consist of n nodes where each node has an independent probability p. A variant, $G(n, m)$ chooses uniformly m edges from the set of edges of the clique of n nodes.

Tobita and Kasahara [TK02] proposed graphs $G(n, k, p)$ which consist of n nodes divided in k layers $L_1, \ldots, L_k$. The probability of edges (u, v) where $u \in L_i$ and $v \in L_j$ is p if $i < k$, and 0 if $i \geq k$.

Dick et al. [DRW98] propose the *Fan-in/Fan-out* graph $G(n, id, od)$, which constructs the graph incrementally. This algorithm allows control over the in-degree, $\text{in}(n_i)$ and the out-degree $\text{out}(n_i)$ of each node n_i. First, the graph is initialized with a single node. Then, one of two phases are chosen with equal probability. In the fan-out phase, the algorithm chooses a node n_i with the biggest difference $od - \text{out}(n_i)$, adds $od - \text{out}(n_i)$ new nodes u_j and edges (n_i, u_j). In the fan-in phase, it finds the set S of all nodes with $\text{out}(n_i) < od$, computes a subset $T \subseteq S$ such that $|T|$ is at most id and adds a new node v plus edge (v, t) for each node $t \in T$. The procedure of choosing a phase and executing it is repeated until the graph has at least n nodes.

Ying and Wu [YW09] propose the *Switching Algorithm* which produces a new graph G out of a given an initial graph G_0. G contains the same nodes than G_0 and preserves the degrees, too. In each iteration, two edges (a, b) and (c, d), are selected to be switched to be replaced with either the edges $(a, d), (b, c)$ or $(a, c), (b, d)$. The algorithm only performs the switch if it doesn't result in a multi- or self-edge. As each node, which takes part in the switch, loses one neighbour node and gains one neighbour node, the degrees are preserved.

Kunegis et al. [KSY23] generate graphs that follow properties from network statistics, e.g., clustering coefficient, connectivity or number of cycles of a certain length. Another work by Kunegis et al. [KKS+23] proposes an algorithm to summarize a graph, i.e., create a small graph that preserves the properties of an input graph. The discussed properties are global graph measures, e.g., number of triangles or number of wedges. The generation algorithm starts with an Erdös-Rényi [ER59]

graph and then performs hill climbing. During hill climbing, small variations of the current graph are evaluated with respect to how good the measures are fulfilled. The algorithm terminates if 90% of nodes were visited without finding an improvement.

All these approaches could easily be extended to incorporate edge labels, i.e., relation names. However, none of these approaches support constraints which could express rules, which is the goal of this thesis.

3.2 Algorithms which generate Graphs from Constraints

These algorithms generate graphs that aim to satisfy a set of given constraints.

Cooper [Coo23] proposes a technique which uses example graphs as constraints to generate similar graphs. Each node and edge in a generated graph can have a label which can be the same for different nodes and different edges. Generating a graph has two steps. First, a set of small graph structures (patterns) is learned from the examples. A single graph pattern consists of a node label (also called key node label), labels of incident nodes and labels of edges. Second, a *Constraint Satisfaction Problem* (CSP) is created from the learned graph patterns. A general CSP contains a set of variables, a domain of possible values for each variable and a set of constraints while each restricts the possible values of multiple variables. A solution of a general CSP is a value for each variable out of the corresponding domain such that no constraint is violated. The CSP to generate a graph contains a variable for each combination of node and label, i.e., $n \cdot (m + 1)$ variables for n nodes and m labels. Note that $m + 1$ denotes m labels plus the possibility to have a blank label. Each variable is binary and reflects whether the corresponding node with label will exist in the generated graph. Edges are reflected via CSP-variables in an analogous way. Afterwards, the previously learned patterns are encoded into propositional logic formulas in such a way that they express the semantic as follows. If a node with a specific label exists (in the generated graph), then at least one pattern has to occur around the node. A "pattern occurs around a node" means that the there are incident nodes as if the node would be the key node of the pattern. Then, generating a graph is done by solving the CSP. However, solving a general CSP is NP hard [JLR17] which makes the technique inconvenient, because the usually big knowledge graphs would lead to high computing costs. Note that it could be the case that the complexity can be reduced, because the problem of this thesis is simpler than a general CSP. Nevertheless, Cooper's [Coo23] way of generating a graph does not support fulfilling constraints intentionally imperfect. Thus, it cannot be applied to the problem of this thesis.

Omrani and Naanaa [ON16] also generate random graphs by using the framework of constrained satisfaction problems. The constraints restrict the graphs to enforce chemical properties on the molecules these graphs represent. To generate the molecule $C_{35}O_{16}H_{44}$, their algorithm took about 3 minutes. This molecule corresponds to a graph with 95 nodes. Therefore, their algorithm does not scale for large knowledge graphs, which can contain billions of nodes [FŞA+20].

Thanapalasingam et al. [TKBG23] generate synthetic knowledge graphs from a set of first-order logic formulas. They describe their underlying generation algorithm as a sampling process from a probability distribution. During the sampling process, the algorithm ensures that all formulas are satisfied. However, the authors of the work do not propose a general approach to address the problem of generating synthetic knowledge graphs for first-order logic formulas. Instead, they

utilize a custom algorithm for each of the five fixed constraint sets they consider. Besides, imperfect fulfilment of constraint, e.g., assigning a body support and confidence, is not possible, either. Therefore, the algorithm of Thanapalasingam et al. [TKBG23] cannot be applied to the problem of this thesis.

Next, the tool by Vecovsca and Jovanovic [VJ24] generates RDF graphs based on SHACL constraints. On the one hand, RDF graphs are equivalent to knowledge graphs, because RDF is a framework that represents data as several combinations of subject, predicate and object, i.e., an edge in a knowledge graph. On the other hand, SHACL is a language that describes how triples for an RDF graph should look like. However, SHACL constraints do not share the same expressiveness as Horn clauses, which is the input of the problem addressed by this thesis. The idea of a SHACL constraint is describing an instance of anything using triples (i.e., edges), e.g., several triples to describe the properties of a book. To fulfil given SHACL constraints, the tool generates graph structures independently. This is in contrast to the goal of this thesis which is generating graph structures that fulfil related rules.

Lastly, Hubert et al. [HMd+24] present the tool *PyGraft* which is able to generate synthetic knowledge graphs, given "OWL" - and "RDFS" constructs. On the one hand, OWL is a language designated to represent relationships between terms of a vocabulary [VM04]. On the other hand, RDFS extends RDF by a typesystem which allows putting resources, which are described by several triples, in classes and relating the classes [GKHT11]. For example, class "man" is a subclass of "person". Hubert et al. [HMd+24] defined 9 constructs (i.e., constraints), e.g., the "owl:SymmetricProperty" $\forall x \forall y : p(x, y) \rightarrow p(y, x)$ which is a Horn clause. Also, there is a constraint which is not a Horn clause, namely the "owl:AsymmetricProperty" $\forall x \forall y : p(x, y) \rightarrow \neg p(x, y)$. Moreover, the tool allows to specify a number $\in [0, 1]$ to each construct. It represents the percentage of relation names for which the respective constraint holds. Note that the set of relation names is given by a user-defined number of relation names. Finally, they allow to set more general graph properties, e.g. number of nodes or number of edges. The generation process generates random edges and only adds an edge to the generated knowledge graph if it is consistent according to the relation name percentages of the user input. However, the tool cannot be used to solve the problem of this thesis. On the one hand, the available constraints are fixed, e.g., no Horn clause with a body of three atoms. On the other hand, it is not possible to influence for which relation name which constraints hold. For instance, the previously presented "owl:SymmetricProperty" could hold for the relation name p = "child" which would be semantically wrong.

4 Problem Statement

The goal of this thesis is to generate random knowledge graphs that fulfil multiple given Horn clauses, each with an expected degree of fulfilment. The expected fulfilment degree of a single rule is denoted via body support, the number of time the premise of a Horn clause is fulfilled, and support, the number of times the premise and conclusion of a Horn clause is fulfilled. A generated graph should approximate the expected values for body support and support. Formally, the problem is defined as follows.

Input:

- a set $C = \{c_1, \ldots, c_k\}$ where each $c_i \in C$ is a Horn clause $B_i \rightarrow H_i$ including only binary predicates, all variables are universally quantified, and all variables in H_i are also in B_i,
- a function $\text{EBodySupp} : C \rightarrow \mathbb{N}$ and
- a function $\text{ESupp} : C \rightarrow \mathbb{N}$ with $\text{ESupp}(c_i) \leq \text{EBodySupp}(c_i)$.

Output: a knowledge graph $G = (N, R, E)$ which minimizes the loss function

$$\ell(G) = \sum_{i=1}^{k} \left(\left(\frac{\text{EBodySupp}(c_i) - \text{BodySupp}_G(c_i)}{\text{EBodySupp}(c_i)} \right)^2 + \left(\frac{\text{ESupp}(c_i)}{\text{EBodySupp}(c_i)} - \frac{\text{Supp}_G(c_i)}{\text{BodySupp}_G(c_i)} \right)^2 \right)$$

Intuitively, the tool should generate a graph G such that for every rule $c_i \in C$ the body support and confidence of c_i in the graph are as close as possible to the expected values, namely the *expected body support* ($\text{EBodySupp}(c_i)$) and the *expected confidence* ($\text{EConf}(c_i) = \frac{\text{ESupp}(c_i)}{\text{EBodySupp}(c_i)}$).

5 Knowledge Graph Generation Algorithms

This chapter explains two algorithms that generate random knowledge graphs. Given a set of rules, each annotated with body support and support, both algorithms aim to solve the optimization problem introduced in Chapter 4. Likely, neither algorithm will generate a graph G such that $\ell(G) = 0$. Indeed, finding such a G for any set of rules C is not possible in general. For example, consider the rules depicted in Table 5.1.

Rule	Body Support	Support
$c_1 = \forall x \forall y (\mathrm{r}(x, y) \rightarrow \mathrm{r}(y, x))$	3	1
$c_2 = \forall x \forall y (\mathrm{q}(x, y) \rightarrow \mathrm{r}(x, y))$	4	4

Table 5.1: Example rules, each with body support and support, for which no graph G with exactly matching support values can be generated

Fulfilling c_2 perfectly requires generating four edges (a, r, b), each with different node pairs (a, b). However, this means that the body support of c_1 is 4, which exceeds the expected body support. Fulfilling c_1 requires removing an edge which consequently results in a knowledge graph that violates c_2. Thus, the rules (with body support and support) contradict and it is not possible to find a G such that each rule is fulfilled. Ultimately, both algorithms are heuristics which yield approximate solutions for the optimization problem.

5.1 Hill Climbing with Knowledge Graph Updates

This algorithm uses discrete hill climbing to solve the optimization problem. The *state space* is defined as the set of all knowledge graphs $G = (N, R, E)$ while R are the relation names of the given rules, N is any finite set of nodes and E is any subset of $N \times R \times N$. Then, the *objective function* is defined as $o(G) = -\ell(G)$. Intuitively, graphs which have a lower loss are considered to be "better" by o. Lastly, the *neighbourhood function* maps each $G = (N, R, E)$ to all knowledge graphs that result in adding an edge to E which was not present before or by removing an existing edge. Standard hill climbing would consider every $G' \in \textsc{Neighbours}(G)$ at each step. However, as $\textsc{Neighbours}(G)$ is large (linear with respect to graph size), the used approach to generate a graph is the hill climbing variant *stochastic hill climbing*. This variant chooses a random neighbour with a higher value of the objective function o at each step. Thus, the algorithm has to only consider a few neighbours during each climbing step. Additionally, the number of climbing steps is fixed at the beginning by setting them to the product of number of the initial graph's edges and the number of input rules. Thus, there are enough climbing steps available to change the entire graph for each rule, but only the overall improvements are saved.

Then, an initial graph is generated by Algorithm 5.1 such that the body support and support of each rule are reached. The algorithm generates as many nodes as needed (line 4). Then, it samples instances of rule's variables (line 7) while ensuring that adding the edges corresponding to the mapping will actually affect the body support and support. Intuitively, $\mu_1|_{Y_c} \neq \mu_2|_{Y_c}$ means adding the edges corresponding to the mappings μ_1, μ_2 to the graph result in a different rule head for μ_1, μ_2, respectively. Thus, both μ_1, μ_2 contribute to the body support and support score of c. Lastly, the instance of c's body (corresponding to μ) is added (lines 10 to 12) and the edge corresponding to the c's head is added, if the support is not reached so far (lines 13 to 15).

Algorithm 5.1 Initial graph for the hill climbing graph generator

Require: A set of rules C and two functions $\text{ESupp} : C \to \mathbb{N}$ and $\text{EBodySupp} : C \to \mathbb{N}$ such that, for each rule $c \in C$, $\text{ESupp}(c) \leq \text{EBodySupp}(c)$.

Ensure: A graph $G = \textsc{SuppReachInitialGraph}(C, \text{ESupp}, \text{EBodySupp})$ such that, for each rule $c \in C$, $\text{Supp}_G(c) \geq \text{ESupp}(c)$ and $\text{BodySupp}_G(c) \geq \text{EBodySupp}(c)$.

1: Let G be the graph (N, R, E) where set R consists of all relation names in rules $c \in C$ and the sets N, E are empty.
2: For each rule $c \in C$, let X_c be the set of variables in c, and Y_c be the set of variables in the head of c.
3: Let $q \leftarrow \sum_{c \in C}(\text{EBodySupp}(c) \cdot |X_c|)$.
4: $N \leftarrow N \cup \{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_q\}$ while $\{a_1, \ldots, a_m\}$ are all nodes occurring in rules $c \in C$, and $\{b_1, \ldots, b_q\}$ is a set of fresh nodes.
5: **for all** $c \in C$ **do**
6: Let $k \leftarrow 0$.
7: Let Ω_c be a set of $\text{EBodySupp}(c)$ randomly selected mappings $\mu : N \cup X_c \to N$ such that $\forall n \in N : \mu(n) = n$ and for every pair of mappings $\mu_1, \mu_2 \in \Omega_c$, $\mu_1 \neq \mu_2 \Rightarrow \mu_1|_{Y_c} \neq \mu_2|_{Y_c}$.
8: **for all** $\mu \in \Omega_c$ **do**
9: $k \leftarrow k + 1$
10: **for all** atoms $r(u, v)$ in the body of rule c **do**
11: Add edge $(\mu(u), r, \mu(v))$ to set E.
12: **end for**
13: **if** $k < \text{ESupp}(c)$ **then**
14: Given the head $s(u', v')$ of rule c, add edge $(\mu(u'), s, \mu(v'))$ to set E.
15: **end if**
16: **end for**
17: **end for**
18: **return** G

This graph is also called "SuppReach".

Finally, the stochastic hill climbing procedure is adapted to generate a graph by Algorithm 5.2. Lines 1 and 2 retrieve the initial graph and compute the number of climbing steps. Then, lines 5 to 8 represent a single climbing step consisting of choosing a random neighbour of the current graph and updating the current graph, if the neighbour has a higher value for $-\ell$.

Algorithm 5.2 Hill climbing graph generator

Require: A set C of rules, and two functions $\text{ESupp} : C \to \mathbb{N}$ and $\text{EBodySupp} : C \to \mathbb{N}$ such that, for each rule $c \in C$, $\text{ESupp}(c) \leq \text{EBodySupp}(c)$.
Ensure: A graph G.

```
Let G ← SuppReachInitialGraph(C, ESupp, EBodySupp).
Let S ← |C| · #edges(G).
Let i ← 1.
while i ≤ S do
    Let G′ be a random element of the set Neighbours(G).
    if −ℓ(G′) > −ℓ(G) then
        G ← G′
    end if
    i ← i + 1
end while
return G
```

Example 5.1
Consider the set of rules $C = \{c_i, c_j\}$ *while* $c_i = \forall x \forall y (p(x, y) \to q(x, y))$ *and* $c_j = \forall u \forall v (p(u, v) \to r(v, u))$. *The functions* EBodySupp *and* ESupp *are defined by*

$$\text{EBodySupp}(c_i) = 2, \qquad \text{ESupp}(c_i) = 1,$$
$$\text{EBodySupp}(c_j) = 2, \qquad \text{ESupp}(c_j) = 2.$$

First, the initial graph is built by Algorithm 5.1 which results in graph G_0 *depicted in Figure 5.1.*

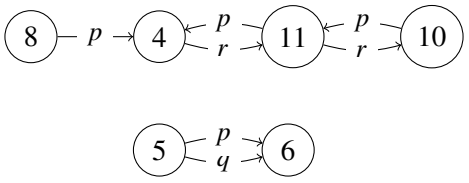


Figure 5.1: Initial graph to illustrate how hill climbing generates a graph

Initially, the body-support and support of c_i, c_j *measured in* G_0 *are*

$$\text{BodySupp}_{G_0}(c_i) = 4, \qquad \text{Supp}_{G_0}(c_i) = 1,$$
$$\text{BodySupp}_{G_0}(c_j) = 4, \qquad \text{Supp}_{G_0}(c_j) = 2.$$

Therefore,

$$-\ell(G_0) = -\left(\left(\frac{2-4}{2}\right)^2 + \left(\frac{1}{2} - \frac{1}{4}\right)^2 + \left(\frac{2-4}{2}\right)^2 + \left(\frac{2}{2} - \frac{4}{2}\right)^2\right) = -\frac{37}{16}.$$

After that, Algorithm 5.2 computes the number of climbing steps by multiplying the number of rules by the number of edges of the initial graph which is $2 \cdot 7 = 14$. *Using that, the hill climbing procedure could perform the following climbing steps.*

1. *Randomly choose to add* $(8, q, 4)$ *to* G_0 *which results in graph* G_1 *and affects* $\text{Supp}_{G_1}(c_i) = 2$. *Therefore,* $-\ell(G_1) = -\frac{9}{4}$ *which is bigger than* $-\ell(G_0)$ *and thus,* G_1 *is considered for further hill climbing steps. Note that increasing the previously correct support value* $\text{Supp}_{G_0}(c_i) = 1$ *to* $\text{Supp}_{G_1}(c_i) = 2$ *was an improvement, because the confidence in the new graph* $\text{Conf}_{G_1}(c_i) = \frac{2}{4} = \frac{1}{2}$ *matches the expected confidence* $\frac{\text{ESupp}(c_i)}{\text{EBodySupp}(c_i)}$ *while the confidence initially was* $\text{Conf}_{G_0}(c_i) = \frac{1}{4}$.
2. *Randomly choose to remove* $(4, r, 11)$ *from* G_1 *which results in graph* G_2 *and affects* $\text{Supp}_{G_2}(c_j) = 1$. *Besides,* $-\ell(G_2) = -\frac{41}{16}$ *which is smaller than* $-\ell(G_1)$, *hence,* G_2 *is discarded.*
3. *Randomly choose to add* $(4, r, 8)$ *to* G_1 *which results in graph* G_3 *with* $-\ell(G_3) = -\frac{33}{16}$. *Due to the fact that* $-\ell(G_3) > -\ell(G_1)$, G_3 *is considered for further hill climbing steps.*
4. *Finally, hill climbing would try to find improvements for 11 more steps, possibly finding more improvements or not, and then yield a knowledge graph.*

5.2 Greedy Edge Generation based on Rule Body Subsumption

This algorithm orders the given rules and then fulfils one rule after the other (in the computed order). Fulfilling a rule means adding as many edges as needed to fulfil its expected body support and support to an initially empty graph. The amount of edges needed is determined by computing the actual body support and support of the current rule in the current graph, First, rules are sorted in ascending order regarding the expected body support. Then, given two rules c_i, c_j with expected body supports k_i, k_j such that every instance of c_i's body B_i also includes an instance of c_j's body B_j, it is said that B_i *subsumes* B_j. For example, $p(x, y) \wedge r(x, z)$ subsumes $r(x, y)$. Hence, k_i instances of formula B_i can be constructed first, which also include k_i instances of B_j. Then, the algorithm adds the remaining $k_j - k_i$ instances of formula B_j that are not instances of formula B_i. In the case $k_j - k_i \leq 0$, the algorithm adds no instances for the body of c_j.

Example 5.2
Consider two rules $c_i := p(x, y) \wedge q(y, z) \rightarrow r(x, z)$ *and* $c_j := p(u, v) \rightarrow s(u, v)$, *whose expected body supports are respectively* 2 *and* 3. *Furthermore, their expected supports are both* 1 *and the algorithm recognizes that the body of* c_i *subsumes the body of* c_j. *Therefore, it will address* c_i *first, then* c_j.

Firstly, the algorithm creates 2 *instances to reach the body support of rule* c_i, *namely* $B_i^1 := p(A_1, B_1) \wedge q(B_1, C_1)$ *and* $B_i^2 := p(A_2, B_2) \wedge q(B_2, C_2)$. *Then, the algorithm fulfils the support of* c_i *by adding* $H_i^1 := r(A_1, C_1)$. *Regarding* c_j, *the algorithm creates just one instance to reach the body support of rule* c_j, *namely* $B_j^3 := p(A_3, B_3)$ *and one instance,* $H_j^1 := u(A_2, B_2)$, *to reach the rule support. Only one is needed, because 2 instances of* B_j *were already added. Let* G *be the graph that includes all edges corresponding to the atoms in the instances of* B_i, H_i, B_j *and* H_j. *Then,* $\text{BodySupp}_G(c_i) = 2$, $\text{Supp}_G(c_i) = 1$, $\text{BodySupp}_G(c_j) = 3$ *and* $\text{Supp}_G(c_j) = 1$. *Hence,* G *perfectly fulfils the expected support values of each rule. A visualization of* G *is depicted in Figure 5.2.*

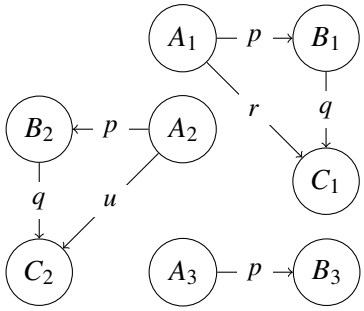


Figure 5.2: Example knowledge graph generated by the greedy edge generation algorithm

Ordering the rules C by OrderRules$(C, \text{EBodySupp})$ is done by Algorithm 5.3. At each iteration (lines 3 to 9) the algorithm determines the "best" rule and appends it to the order. Line 4 computes the maximum number of subsumptions s any rule has which was not ordered before. Afterwards, $C_{\max}$ denotes the set of rules while each element subsumes s other rules (line 5). Finally, the next rule appended to C_{Ordered}, i.e., the "best" rule, is the one of $C_{\max}$ with maximum expected body support (line 6).

Algorithm 5.3 Algorithm that orders a rule set

Require: Set of rules C and function $\text{EBodySupp} : C \to \mathbb{N}$.
Ensure: Ordered list of rules $C_{\text{Ordered}} = \text{OrderRules}(C, \text{EBodySupp})$.
1: Let C' be a copy of C.
2: Let C_{Ordered} be an empty list.
3: **for all** $i \in \mathbb{N}, 1 \leq i \leq |C|$ **do**
4: Let $s \leftarrow \max\{|\,\text{SubsumesAll}(c, C'\backslash\{c\})| : c \in C'\}$ while $\text{SubsumesAll}(c, C'\backslash\{c\})$ denotes the set of rules of all $c' \in C'\backslash\{c\}$ which have a body that is subsumed by c's body.
5: Let $C_{\max} \leftarrow \{c \in C' : |\,\text{SubsumesAll}(c, C'\backslash\{c\})| = s\}$.
6: Let $c_{\max} \leftarrow \arg\max_{c \in C_{\max}} \text{EBodySupp}(c)$.
7: Append $c_{\max}$ to C_{Ordered}.
8: $C' \leftarrow C' \setminus \{c_{\max}\}$
9: **end for**
10: **return** C_{Ordered}

Using the order, Algorithm 5.4 generates a graph by greedily fulfilling one rule after the other. Lines 1 and 2 introduce the initial graph and determine the order of rules. After that, the algorithm loops over each rule in the previously computed order (line 3). During each loop iteration, on the one hand, edges are generated to reach the expected body support of the rule (lines 6 to 12). Note that each new instance replaces the variables with fresh, distinct nodes. On the other hand, edges are added to reach the expected rule support (lines 13 to 16). Reaching the body support or support means, measuring the current body support or support and adding instances of the rule by considering the difference to the corresponding expected value. If the current body support or support already exceeded the expected value, the algorithm neither adds or removes edges.

Algorithm 5.4 Greedy graph generator that fulfils one rule after the other

Require: Set of rules C and two functions $\text{ESupp} : C \to \mathbb{N}$, $\text{EBodySupp} : C \to \mathbb{N}$ such that for each rule $c \in C$, $\text{ESupp}(c) \leq \text{EBodySupp}(c)$.

Ensure: A graph G such that for each rule $c \in C$, $\text{Supp}_G(c) \geq \text{ESupp}(c)$ and $\text{BodySupp}_G(c) \geq \text{EBodySupp}(c)$.

1: Let G be the graph (N, R, E) where set R consists of all relation names occurring in rules $c \in C$ and sets N, E are empty.
2: Let $C_{\text{Ordered}} \leftarrow \textsc{OrderRules}(C, \text{EBodySupp})$.
3: **for all** $c_i \in C_{\text{Ordered}}$ (taken in order) **do**
4: Let B_i be the body of c_i with variables X_i and H_i the head of c_i with variables $Y_i \subseteq X_i$.
5: $N \leftarrow N \cup \{a_1, \ldots, a_m\}$ while $\{a_1, \ldots, a_m\}$ are all nodes occurring in rule c.
6: **for all** $k \in \mathbb{N}, \text{BodySupp}_G(c_i) < k \leq \text{EBodySupp}(c_i)$ **do**
7: Let μ be a mapping with $\text{dom}(\mu) = N \cup X_i$ such that $\forall n \in N : \mu(n) = n$ and $\forall x \in X_i : \mu(x)$ is a distinct, fresh node ($\notin N$).
8: $N \leftarrow N \cup \text{range}(\mu)$
9: **for all** atoms $r(u, v)$ in the body of rule c_i **do**
10: Add edge $(\mu(u), r, \mu(v))$ to set E.
11: **end for**
12: **end for**
13: Let Ω be a set of $(\text{ESupp}(c_i) - \text{Supp}(c_i))$ sampled mappings $\mu : X_i \cup N \to N$ such that $\forall n \in N : \mu(n) = n$, $\mu \models_G B_i$ and $\forall \mu_1, \mu_2 \in \Omega : \mu_1 \neq \mu_2 \Rightarrow \mu_1|_{Y_i} \neq \mu_2|_{Y_i}$.
14: **for all** $\mu \in \Omega$ **do**
15: Given the head $s(u', v')$ of rule c_i, add edge $(\mu(u'), s, \mu(v'))$ to set E.
16: **end for**
17: **end for**

6 Implementation

The implementation is written using Java 17 with Maven as build tool and is publicly available in the repository of the University of Stuttgart under an open-source licence [Gla24]. The core parts of this chapter are how a knowledge graph is depicted in memory such that generation algorithms can operate on it efficiently. Then, how the loss function, introduced in Chapter 4, is computed which mainly requires an efficient computation of the body support and support of any rule in any knowledge graph. Lastly, there is an explanation how determining subsumption relationships between rule bodies is implemented.

6.1 Knowledge Graph Datastructure

Having an efficient knowledge graph implementation is crucial for graph generation and evaluation with body support and support. To this end, an edge (A, r, B) is stored in two different sorted sets designated to relation name r. One set sorts its edges according to the source node and the other according to the target node. For instance, a conceptual view of how the edges (A, r, A), (A, r, B) and (D, r, A) would be stored is depicted in Figure 6.1.

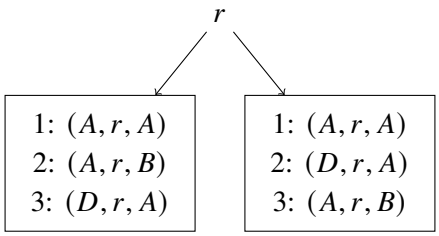


Figure 6.1: Visualization of how edges are stored in the implemented knowledge graph data structure for a single relation name

On the one hand, this requires at least double the storage than needed to correctly maintain the graph. On the other hand, this will be handy when computing the body support of a rule, because it requires sorting the edges, possibly according to the source node or target node. The hill climbing algorithm presented in Section 5.1 needs to compute the loss function many times. Sorting all triples after a change of a single edge is inefficient in comparison of preserving the sorted characteristic all the time. Fortunately, the implementation of both sorted sets is a tree already given by the Java Library [Blo98a]. Thus, basic operations, e.g., adding or removing an edge, costs $2 \cdot O(\log n)$ while the order is preserved for both directions. Finally, a consistent order of nodes is defined by assigning a unique integer $i \in \mathbb{N}$ to each distinct node.

6.2 Computing the Loss Function

Computing the loss function ℓ presented in Chapter 4 requires computing the actual body support and support of any rule c_i in any knowledge graph G. As previously introduced in Chapter 2, the body support of a rule $c_i = B(u, v) \rightarrow H(u, v)$ measured in graph $G = (N, R, E)$ is defined as the number of different node pairs $(a, b) \in N^2$ such that the remaining variables in $B(a, b)$ can be set to nodes in a way that each atom in the fully grounded $B(a, b)$ corresponds to an existing edge in G.

Algorithm 6.1 shows a naive way of computing the body support. Line 1 defines the set P which contains all permutations of edges with a length of the number of atoms in $B(u, v)$. Each $p \in P$ contains an edge at position i that corresponds to $B(u, v)$'s atom at position i. Thus, P contains all valid groundings of B, but also invalid groundings. For example,

$$p_1 = ((A, r, B), (B, r, C)), p_2 = ((A, r, B), (A, r, B))$$

would be elements of P, if the four listed edges are in G, $B(u, v)$ consists of exactly 2 atoms and both atoms of $B(u, v)$ have relation name r. After that, line 2 defines the initially empty set B_T which represents all different node tuples corresponding to the terms of $H(u, v)$. However, P contains many invalid groundings which need to be excluded. Consequently, lines 3 to 8 compute the node tuples, corresponding to $H(u, v)$, but only for valid edge permutations of P. To this end, line 4 checks whether a p is a valid grounding for $B(u, v)$ which means looking for equal variables in $B(u, v)$ and checking whether the corresponding nodes in p are equal, too. For example, consider $B(u, v) = r(x, y) \wedge r(y, z)$, then p_1 is a valid grounding while p_2 is not. This is because variable y of $B(u, v)$ implies that the target node of first edge has to equal the source node of the second edge. Finally, line 9 computes the number of different node tuples that correspond to $H(u, v)$ which is the body support.

Algorithm 6.1 Naive algorithm to compute the body support of a rule c measured in graph G

Require: Rule $c = B(u, v) \rightarrow H(u, v)$ while $B(u, v) = \bigwedge_{i=1}^{k} A_i$, $H(u, v) = A_{k+1}$ and $A_i = r_i(t_{i,1}, t_{i,2})$, and graph G.
Ensure: $\text{BodySupp}_G(c) \in \mathbb{N} \cup \{0\}$.
1: Let $P \leftarrow E_{r_1}(G) \times E_{r_2}(G) \times \cdots \times E_{r_k}(G)$ while $E_{r_i}(G)$ denotes the set of all edges (a, r_i, b) present in G for any nodes a, b.
2: Let $B_T \leftarrow \emptyset$.
3: **for all** $p \in P$ **do**
4: **if** p is valid grounding of $B(u, v)$ **then**
5: Let a, b be the two of nodes present in p that correspond to the terms in $H(u, v)$.
6: $B_T \leftarrow B_T \cup \{(a, b)\}$
7: **end if**
8: **end for**
9: **return** $|B_T|$

However, the size of P scales polynomially with respect to the number of c's atoms. For example, if c's body contains two atoms with relation names r_1, r_2 and $|E_{r_1}(G)| = |E_{r_2}(G)| = 10^5$, then $|P| = 10^{10}$. To put this into a perspective, $|P|$ already is bigger than the 32-bit integer range.

To address this issue, Algorithm 6.2 computes all valid edge permutations in log-linear time. Lines 1 and 2 introduce table T to store all edge permutations and initialize T with the edges corresponding to the first atom. T semantically holds edge permutations as P of the naive algorithm. However, instead of enumerating all permutations, T is built iteratively to exclude many edge permutations in advance. Then, line 3 starts the loop over each atom of $B(u, v)$ starting from the second. Each loop iteration (line 4 to 12) extends table T by the edges of the next atom A_i such that T reflects all valid groundings of $A_1 \wedge \cdots \wedge A_i$. To this end, line 4 computes the set of common variables between the previously added atoms and the next atom. They are used to exclude invalid groundings. In case there is no common variable, each combination between row of T could be extended by any edge in $E_{r_i}(G)$ (line 6) which results in the complexity of Algorithm 6.1. This case could be improved by addressing another atom $A_j, i < j \leq k$ first, i.e., reorder the given atoms of $B(u, v)$. However, this is not implemented. If there is at least one common variable, then lines 8 to 10 prepare the table T and edges $E_{r_i}(G)$ to find all valid combinations efficiently. Note that sorting of T can be done in log-linear time [Blo14] and line 10 utilizes the implemented graph data structure by choosing the already existing sorted edges S. Afterwards, line 11 calls Algorithm 6.3 that uses the prepared table T and edges S to compute all valid combinations (based on common variables) of $t \in T$ and $s \in S$ in linear time. Finally, line 14 yields the final table of all valid groundings of $B(u, v)$, which were computed in log-linear time.

Algorithm 6.2 Algorithm to compute all valid groundings of a rule body in a graph

Require: Rule $c = B(u, v) \rightarrow H(u, v)$ while $B(u, v) = \bigwedge_{i=1}^{k} A_i$, $H(u, v) = A_{k+1}$ and each $A_i = r_i(t_{i,1}, t_{i,2})$, and graph $G = (N, R, E)$.
Ensure: All valid $B(u, v)$ groundings = ValidBodyGroundings(c, G).

1: Let T be an empty table.
2: Add all edges of $E_{r_1}(G)$ in a new column, each $e \in E_{r_1}(G)$ in a new row.
3: **for all** $i \in \mathbb{N}, 2 \leq i \leq k$ **do**
4: Let X be the set of common variables which is contained in any atom $A_j, 1 \leq j < i$ and the current atom A_i.
5: **if** $X = \emptyset$ **then**
6: $T \leftarrow$ all combinations of T and $E_{r_i}(G)$.
7: **else**
8: First, choose main common variable $x \in X$ and then, second common variable $x' \neq x, x' \in X$ only if $|X| > 1$.
9: Sort rows of T based on common variables X in the way that $t < t'$ $(t, t' \in T)$ if and only if, $n_{x,1} < n_{x,2}$ for nodes $n_{x,1}, n_{x,2}$ corresponding to x. Furthermore, if $n_{x,1} = n_{x,2}$ and x' is present, $n_{x',1} < n_{x',2}$ for nodes $n_{x'2,1} < n_{x',2}$ corresponding to x' must hold, too.
10: Choose sorted set S of edges $E_{r_i}(G)$ which is sorted based on variable x.
11: $T \leftarrow$ ExtendTableWithEdgesOfAtom$(\{A_1, \cdots, A_i\}, T, S, x, x')$
12: **end if**
13: **end for**
14: **return** T

Algorithm 6.3 iterates over each entry in T, S once, i.e., has linear time complexity. i_T, i_S, which are introduced in line 2, reflect a pointer to the current element of the respective collection of elements. At each loop iteration, the current elements (row of T, edge of S) are compared based on their nodes at the position of common variables (lines 4 to 6). If the nodes are equal, then a match was found.

Now, the algorithm is able to leverage the sorting of T and S by continuing to find consecutive matches which have same replacements for the common variables (lines 7 and 8). Then, line 9 adds all combinations of the found matches to T' and lines 10 to 11 update the pointers i_T, i_S such that they point to the first element that does not match anymore. Afterwards, the loop continues with the next elements and does not consider any elements before i_T, i_S anymore. This still yields all possible combinations, because comparison of current elements with previous elements is not needed, as the sorting characteristic ensures that the nodes at the position of common variables cannot be equal. Lastly, if the current elements do not match, lines 13 and 14 update the pointers by incrementing the one which points to the "smaller" nodes. Only incrementing the pointer to the smaller node ensures that no match is skipped.

Algorithm 6.3 Algorithm to extend valid groundings for a conjunction of atoms by the edges of another atom

Require: Atoms $\{A_1, \cdots A_i\}$, Table T with all valid grounding of $A_1 \wedge \cdots \wedge A_{i-1}$, set of edges S for A_i, main common variable x and optional common variable x'.

Ensure: Table of edges T' that contains all valid groundings of $A_1 \wedge \cdots \wedge A_i$ which is referred by = ExtendTableWithEdgesOfAtom($\{A_1, \cdots A_i\}, T, S, x, x'$).

1: Let T' be an empty table.
2: Let $i_T, i_S \leftarrow 1$.
3: **while** $i_T \leq |T| \wedge i_S \leq |S|$ **do**
4: Let $n_{x,1}, n_{x,2}$ be the node for variable x in the i_T-th entry of T, i_S-th entry of S.
5: Let $n_{x',1}, n_{x',2}$ be the node for variable x' in the i_T-th entry of T, i_S-th entry of S, if x' is present.
6: **if** $n_{x,1} = n_{x,2} \wedge n_{x',1} = n_{x',2}$ (ignore second comparison, if x' is not present) **then**
7: Let T_x be all consecutive rows in T, beginning at i_T, for which the node for x equals $n_{x,1}$ and (if x' is present) the node for x' equals $n_{x',1}$.
8: Let S_x be all consecutive edges in S, beginning at i_S, for which the node for x equals $n_{x,1}$ and (if x' is present) the node for x' equals $n_{x',1}$.
9: Add all combinations of rows $\in T_x$ and edges $\in S_x$ to T'.
10: $i_T \leftarrow i_T + |T_x|$
11: $i_S \leftarrow i_S + |S_x|$
12: **else if** $n_{x,1} < n_{x,2} \vee (n_{x,1} = n_{x,2} \wedge n_{x',1} < n_{x',2})$ (ignore second comparison, if x' is not present) **then**
13: $i_T \leftarrow i_T + 1$
14: **else**
15: $i_S \leftarrow i_S + 1$
16: **end if**
17: **end while**
18: **return** T'

Example 6.1
For example, Algorithm 6.2 addresses the computation of the body support of the rule

$$c = \text{parent}(x, y) \wedge \text{sibling}(x, z) \rightarrow \text{parent}(z, y)$$

on graph G depicted in Figure 6.2 the following way.

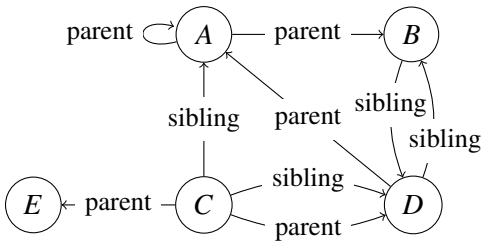


Figure 6.2: Example knowledge graph to illustrate how the body support is computed

First, initialize table T with c's first atom, namely parent(x, y)*, which looks like Table 6.1.*

parent(x, y)
(C, parent, D)
(C, parent, E)
(A, parent, B)
(D, parent, A)
(A, parent, A)

Table 6.1: Example initial table to illustrate how the body support of a rule is computed

Then, identify the set of common variables of the table (currently consisting of only parent(x, y)*) and the next atom* sibling(x, z)*. This set only contains variable x. After that, sort table T based on variable x and retrieve edges for the second atom's relation name* sibling *while choosing the sorting according to variable x. This is given by the knowledge graph datastructure, as already explained in Section 6.1. The last step for the first loop is applying Algorithm 6.3 to extend the initial T by the edges of* sibling*. Table 6.2a shows the content of the initial T sorted according to variable x. Next, Table 6.2b depicts the edges of relation name* sibling *which will be used to extend Table 6.2a. The result of the extension Algorithm 6.3 can be seen in Table 6.2c. As rule c contains two atoms, all body groundings were successfully computed and are reflected in Table 6.2c.*

parent(x, y)
(A, parent, A)
(A, parent, B)
(C, parent, D)
(C, parent, E)
(D, parent, A)

(a) Sorted initial table by common variable x

sibling(x, z)
$(B, \text{sibling}, D)$
$(C, \text{sibling}, A)$
$(C, \text{sibling}, D)$
$(D, \text{sibling}, B)$

(b) All edges for relation name sibling sorted by common variable x

parent(x, y)	sibling(x, z)
(C, parent, D)	$(C, \text{sibling}, A)$
(C, parent, E)	$(C, \text{sibling}, A)$
(C, parent, D)	$(C, \text{sibling}, D)$
(C, parent, E)	$(C, \text{sibling}, D)$
(D, parent, A)	$(D, \text{sibling}, B)$

(c) Extended table

Table 6.2: Example illustration of how Algorithm 6.3 extends a table of edges by a new set of edges

Lastly, the set of node tuples corresponding to the variables of c's head can be derived from (z, y) *entries in Table 6.2c which is*

$$B_T = \{(A, D), (A, E), (D, D), (D, E), (B, E)\}.$$

Thus, $\text{BodySupp}_G(c) = 5 = |B_T|$.

The support of any rule c measured on any graph $G = (N, R, E)$ can be derived by considering B_T. It can be calculated by

$$\text{Supp}_G(c) = |\{(a, b) \in B_T : (a, r_H, b) \in E\}|$$

while r_H is the relation name of c's head. Intuitively, check for each $(a, b) \in B_T$ whether the corresponding edge (a, r_H, b) is part of G. The number of $(a, b) \in B_T$ for which this holds is the support. Note that the support of the example rule is 0 for the graph depicted in Figure 6.2. As B_T is implemented via a hash table, checking containment can be done in constant time [Blo98b]. Therefore, computing the support (given B_T) has a linear runtime complexity. Lastly, utilizing the explained algorithms, all expressions of the loss function ℓ can be computed and thus, ℓ can be calculated for any rule c on any knowledge graph G.

6.3 Computing Rule Body Subsumption

For this thesis, it is assumed that rules are small and thus, neither contain many atoms nor many variables. Therefore, it is sufficient to address the subsumption computation of rule bodies B_1, B_2 with an algorithm that checks every way how B_2 can be transformed such that each atom of the transformed B_2 occurs in B_1. Transforming B_2 means replacing each variable of B_2 by a term of B_1 (either node or variable). Furthermore, the subsumption of B_1 to B_2 holds, if there exists a transformation of B_2 such that each atom of B_2 occurs in B_1. The pseudo code of the implemented (recursive) algorithm is depicted in Algorithm 6.4.

Algorithm 6.4 Recursive algorithm to check whether a rule body subsumes another rule body

Require: Rule bodies B_1, B_2.
Ensure: Whether B_1 subsumes B_2 or not = $\textsc{Subsumes}(B_1, B_2)$.

```
1:  Let V be the set of B2's remaining original variables.
2:  if V = ∅ then
3:      return Atoms(B2) ⊆ Atoms(B1)
4:  end if
5:  Let x be variable of V.
6:  Let T be all terms of B1.
7:  for all t ∈ T do
8:      Let B2' be a conjunction of atoms that represents B2 while each occurrence of variable x is
        replaced by term t.
9:      if Subsumes(B1, B2') then
10:         return true
11:     end if
12: end for
13: return false
```

Lines 1 to 4 represent the base case for the recursion, i.e., all variables of B_2 were set to a term of B_1. The subsumption holds for the base case, if each atom of B_2 occurs in B_1. If $\textsc{Atoms}(B_2)$ is not a subset of $\textsc{Atoms}(B_1)$, then B_1 could still subsume the original B_2, however, it could not be

determined by the particular transformation of B_2. After that, lines 7 to 12 transform an original variable of B_2 to a each term of B_1 and check whether this transformation leads to the possibility to proof the subsumption. If not the next term has to be tested.

Finally, Algorithm 6.4 has an exponential runtime complexity of $O(n^m)$ while n is the number of B_1's terms and m is the number of B_2's variables. Thus, increasing the size of rules eventually leads to a bad runtime.

7 Evaluation

This chapter addresses the research questions RQ1 and RQ2 introduced in Chapter 1. First of all, Section 7.1 explains the setting of the evaluation which consists of how generated graphs are compared and an algorithm that produces a random set of rules. Then, RQ1 is addressed in Section 7.2 by running each knowledge graph generation variant on multiple sets of rules. Apart from the generation algorithms presented in Chapter 5, there are three more variants to evaluate by modifying parts of the two algorithms. On the one hand, there are two additional variants for hill climbing characterized by a different initial graph. On the other hand, there is an additional variant for the greedy algorithm by ordering the rules randomly. Afterwards, Section 7.3.1 analyzes how hill climbing with greedy initial graph handles different kinds of rule sets. Here, rule sets are characterized by an overlapping degree of relation names which intuitively reflects how many relation names occur in how many rules. Then, Section 7.3.2 contains a case study which provides a more intuitive insight into how a generated graph by hill climbing with greedy initial graph (order based on subsumptions) looks like. Finally, Section 7.3.3 addresses RQ2 by applying *AMIE* [GTHS13] on generated knowledge graphs for all generation variants. On the one hand, it is evaluated whether AMIE finds the rules that the generation algorithms aimed to fulfil. On the other hand, it is evaluated whether AMIE finds more rules which were not intended to occur.

7.1 Setting

Generated knowledge graphs are evaluated by comparing their loss value with the function ℓ which was introduced in Chapter 4. This is because ℓ reflects a quality metric for the goal of this thesis by measuring how close the body support and confidence of each rule in the generated graph is compared to the expected values. In addition to that, graphs are compared by considering their body support, support and confidence (for each rule) in comparison to the corresponding expected values. This is done in an aggregated way, e.g., each difference of expected value and actual value is computed and aggregated by average and standard deviation. For example, consider any three rules with expected body supports (3, 2, 4) and corresponding actual body supports in a generated graph (3, 1, 7). Then the differences are (0, 1, 3), the average difference is 1.333 and standard deviation is 1.247. This is also written as 1.333 ± 1.247 or Δ BodySupp. Δ Supp and Δ Conf have an analogous semantic.

Sets of rules for this evaluation are generated by Algorithm 7.1. Line 2 starts the loop that generates n rules. Next, lines 3 and 4 choose the number of atoms for the current rule c and choose corresponding relation names out of the set of all relation names R_C. After that, lines 5 and 6 choose a random relation name for the rule's head while the remaining relation names are implicitly for the rule body. Then, lines 7 to 12 annotate each relation name of the body with two variables (x, y) in a way such that each atom has two different variables $(x \neq y)$ and each atom in the list introduces at most one new variable. The latter condition is needed such that the computation of c's

body support remains tractable, see Algorithm 6.2. After that, two different variables are added to the rule head while both variables already have to occur in the rule body (line 13). Finally, line 14 transforms all atoms into rule c and adds it to C.

Algorithm 7.1 Algorithm that generates a random set of rules

Require: Number of rules $n \in \mathbb{N}$, maximum number of atoms per rule $\#A_{\max} \in \mathbb{N}, \#A_{\max} \geq 2$ and non-empty set of relation names R_C.

Ensure: A randomly generated set of rules $C = \textsc{GenerateRuleSet}(n, \#A_{\max}, R_C)$.

1: Let C be an empty set.
2: **for all** $i \in \mathbb{N}, 1 \leq i \leq n$ **do**
3: Let $\#A_i$ be a random number from $\{2, 3, \cdots, \#A_{\max}\}$.
4: Let R_i be a list of $\#A_i$ randomly chosen relation names from R_C, possibly with duplicates.
5: Let r_H be a random relation name randomly chosen from R_i.
6: Remove single occurrence of r_H in R_i.
7: Let X be a set of $\#A_i$ unique variables.
8: Let B be an empty list.
9: **for all** $r \in R_i$ (taken in order) **do**
10: Choose $(x, y) \in X^2$ randomly such that $x \neq y$ and at least one of $\{x, y\}$ already occurs in B (latter condition not for first iteration).
11: Append $r(x, y)$ to B.
12: **end for**
13: Choose $(x, y) \in X^2$ randomly while $x \neq y$ and x, y occur in B.
14: $C \leftarrow C \cup \{B \rightarrow r_H(x, y)\}$
15: **end for**
16: **return** C

Lastly, each random choice during this evaluation is made based on uniform probabilities for each possibility.

7.2 Algorithm Comparison

This section compares all implemented knowledge graph generation variants. In addition to the two developed algorithms in Chapter 5, this comparison evaluates two different initial graphs for hill climbing and random ordering for the greedy algorithm. The three modifications of the original algorithms are considered as three additional variants. On the one hand, instead of using the initial graph generated by Algorithm 5.1, hill climbing is evaluated with a fully randomly generated initial graph. It is generated by randomly choosing two nodes and a relation name to create an edge for a fixed amount of times. The possible relation names are given by the input rule set C, the number of randomly generated edges is set to

$$T = \sum_{i=1}^{n} (\text{EBodySupp}(c_i) \cdot b_i + \text{ESupp}(c_i)) \quad (b_i \text{ is the number of atoms of } c_i\text{'s body})$$

and the available nodes are given by generating $\frac{T}{3}$ distinct nodes. On the other hand, hill climbing is evaluated with the initial graph set to the output of the greedy algorithm (with order based on logical subsumptions) presented in Section 5.2. The climbing variants are called “Climbing

SuppReach", "Climbing random" and "Climbing subsumption", respectively. Finally, instead of using the sophisticated order of rules computed by Algorithm 5.3 ("greedy subsumption"-variant), the greedy algorithm presented in Algorithm 5.4 is evaluated by fulfilling a random order of rules in a greedy way ("greedy random"-variant).

For the comparison, each variant runs on 97 sets of rules. These sets of rules are given by applying Algorithm 7.1 with GenerateRuleSet$(n, 7, R_C)$ 97 times while n and $|R_C|$ are chosen randomly from $[2, 20]$ for each set of rule. Note that R_C is given by $|R_C|$ as $\{r_1, \cdots, r_{|R_C|}\}$. Then, each rule is annotated with a random body support in $[100, 300]$ and random support between 0 and the previously chosen body support. Afterwards, each variant is applied on each set of rules. Several metric measurements of the resulting generations are depicted in Table 7.1.

Generation variant	Loss	Runtime (s)	Δ BodySupp	Δ Supp	Δ Conf
Greedy random	328.6 ± 1803.1	0.2 ± 0.4	235.2 ± 327.1	97 ± 134.4	0.121 ± 0.168
Greedy subsumption	437.6 ± 2468.6	0.1 ± 0.2	254 ± 310.8	146.5 ± 176.4	0.128 ± 0.17
Climbing SuppReach	3.5 ± 3.5	1383 ± 1829.4	53.8 ± 53.6	55.1 ± 44.1	0.201 ± 0.183
Climbing random	10.7 ± 5.7	1054.5 ± 1469.9	142 ± 90.9	96.9 ± 60.7	0.48 ± 0.27
Climbing subsumption	3.1 ± 3.4	1033.7 ± 1374.6	48.9 ± 47.5	41.8 ± 39.4	0.139 ± 0.151

Table 7.1: Aggregated performance metric measurements of running each knowledge graph generation variant on 97 sets of rules

Interestingly, the greedy algorithm with randomly ordered rules outperforms the greedy variant with rule order based on subsumptions. On the one hand, this is visible by the loss which is less and has a lower standard deviation. On the other hand, the difference between expected and actual value for body support, support and confidence is smaller. Also, the standard deviations are lower except for Δ Supp. Nevertheless, the standard deviations are high for both greedy variants in comparison to the corresponding averages. A standard deviation which is higher than the corresponding average means that some rules are fulfilled better while others aren't fulfilled at all. In contrast, a low standard deviation means that matching the expected and actual value is done equally well for all given rules. The high standard deviation happens due to the characteristic of the greedy algorithm that rules which occur first in the order are possibly affected by many further rules. Contrary, rules that occur late in the order will not be affected by many further rules. As the greedy algorithm fulfils each rule perfectly after processing it (assuming that the body support and support is not already exceeded) the fulfilled body support and support of rules which occur late in the order is less likely to be changed much. Lastly, the average runtime and its standard deviation of the greedy algorithm with random order is higher in comparison to the other greedy variant. This is surprising, because the random order is not as complex as the order based on subsumptions. Likely this is due to the fact that the runtime for both greedy variants is as low such that the execution environment, e.g., process scheduler, has a deciding influence.

Besides that, each hill climbing variant outperformed both variations of the greedy algorithm. This is visible by considering the loss and average difference between expected and actual value for body support, support and confidence. Also, the standard deviations for body support, support and confidence are lower than the corresponding averages which means hill climbing fulfils rules more equally well in comparison to the greedy variants. However, the runtime is over 10 thousand times longer than the greedy variants.

Considering only hill climbing variants, climbing with the initial graph set to the result of the greedy algorithm with subsumption order is the best. This is evaluated by loss and average difference between expected and actual value for body support, support and confidence. Second best is climbing with the SuppReach initial graph and worst climbing variant is starting with a random initial graph. Therefore, the initial graph has a deciding impact on the result.

Also, climbing with greedy graph (subsumption order) is the fastest climbing approach. The runtime of climbing scales with the size of the initial graph, i.e., number of edges, and the number of rules. This is because climbing runs for a fixed amount of steps computed at the beginning by multiplying the size of the initial graph with the number of rules. As each variant faces the same sets of rules, the size of the SuppReach initial graph is the biggest.

Lastly, the number of climbing steps during the executions of the three hill climbing variants can be seen in Table 7.2.

Initial graph	Total number of steps	Number of steps until no loss improvement
SuppReach	167771 ± 84870	119664 ± 74428
Random	170751 ± 89324	124031 ± 79842
Subsumption	147363 ± 93063	101208 ± 85592

Table 7.2: Number of climbing steps for hill climbing variants

The steps until termination were lower than the total number of steps. This suggests that the estimated number of steps are sufficient for hill climbing to reach a local optimum.

7.3 Follow-Up Analysis

This section provides more insights about the generation algorithms. First, Section 7.3.1 evaluates how hill climbing with greedy subsumption initial graph handles different kinds of rule sets. Rule sets C are distinguished by an overlapping degree $\mathrm{OD}(C)$ which represents how many relation names occur in how many rules. Afterwards, there is a case study in Section 7.3.2 which provides a more intuitive insight on hill climbing with greedy subsumption initial graph by applying it on a small set of rules with small expected body supports and small expected supports. Finally, Section 7.3.3 presents the results of applying the rule mining tool *AMIE* [GTHS13] on graphs generated by each variant.

7.3.1 Comparison of Rule Sets

Distinguishing a randomly generated set of rules C from another set of rules C' will be done by their overlapping degrees $\mathrm{OD}(C), \mathrm{OD}(C')$. Given a set of rules C and the set of all relation names that occur in C called $R_C = \{r_1, \cdots, r_{|R_C|}\}$, the overlapping degree of C is defined as

$$\mathrm{OD}(C) := \frac{\sum_{i=1}^{|R_C|} \frac{n_i}{|C|}}{|R_C|}$$

while n_i is the number of rules r_i occurs in. Note that occurring of relation name r in a rule c means that any atom of c states a relationship with r. Intuitively, $\mathrm{OD}(C)$ averages the percentage of rules that contain a relation name at least once. For example, if $\mathrm{OD}(C) = 1$, then every relation name occurring in any $c \in C$ would occur in every rule $c \in C$. In contrast, if $\mathrm{OD}(C) = \frac{1}{|C|}$, then each relation name occurring in any rule $c \in C$ would occur in a single rule only. Lastly, $\mathrm{OD}(C) = 0.5$ means that every relation name occurring in any rule $c \in C$ occurs in 50% of $c' \in C$ on average.

Controlling the overlapping degree of a produced set of rules with Algorithm 7.1 can be done with the parameter R_C. If R_C becomes bigger, then there are more relation names to sample from. Thus, it is less likely that the same relation name gets chosen several times and consequently the overlapping degree becomes lower. In contrast, if R_C becomes smaller, the resulting overlapping degree will become bigger.

Then, using Algorithm 7.1, 30 sets of rules are produced for each $|R_C| \in \{2, \cdots, 20\}$. Furthermore, $\#A_{\max}$ is 7 for any rule set generation and the number of rules will be determined by a random number out of $\{2, \cdots, 20\}$ (possibly different for each set of rules). Next, each rule is annotated with a random body support in 100 to 300 and support from 0 to the previous body support. Finally, hill climbing with greedy initial graph generated a knowledge graph for each rule set.

First, Figure 7.1 verifies the assumption that generating a set of rules by inserting a higher $|R_C|$ in Algorithm 7.1 results in a lower $\mathrm{OD}(C)$.

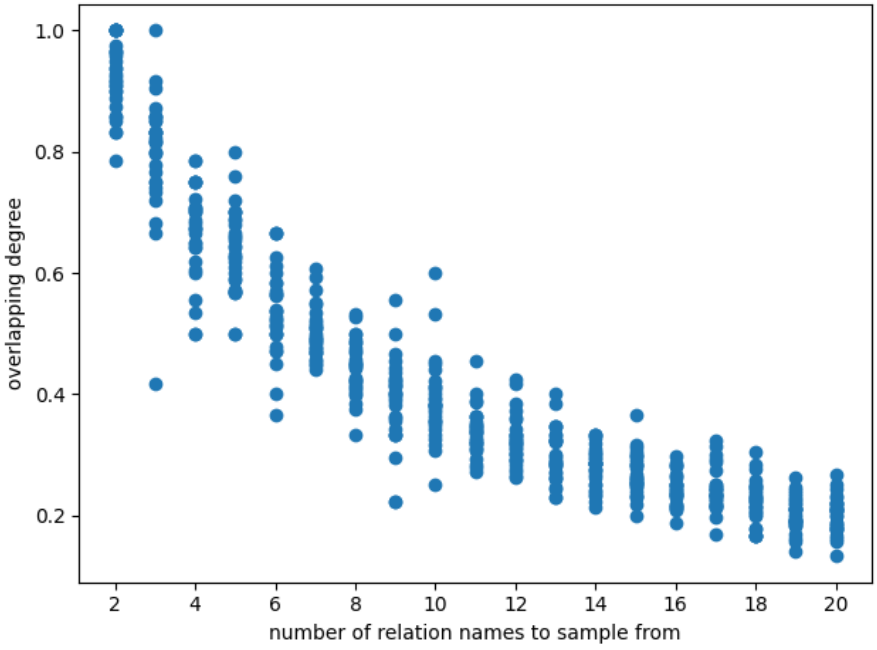


Figure 7.1: Visualization of number of relation names to sample from with resulting overlapping degree of generated set of rules

Furthermore, the overlapping degree of each set of rules and the loss function value of the corresponding generation is depicted in Figure 7.2.

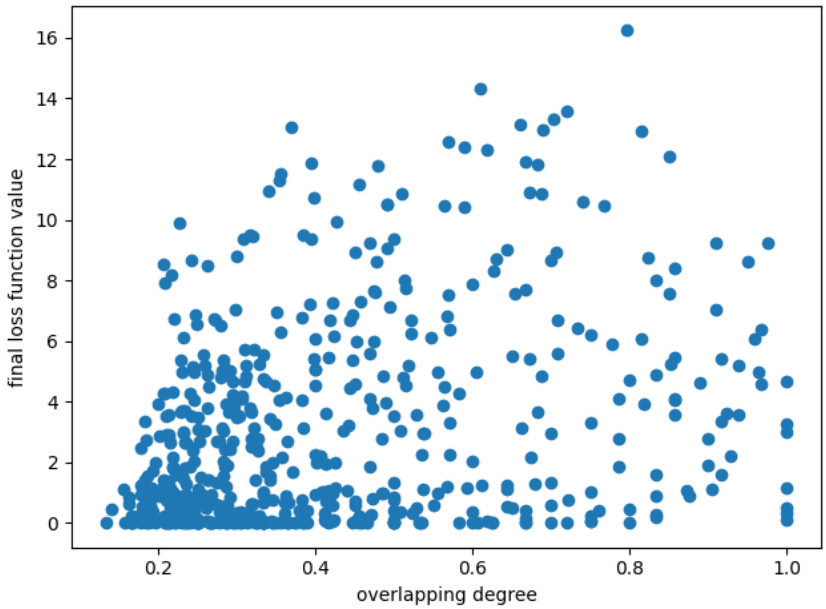


Figure 7.2: Plot of overlapping degree of any generated rule set with corresponding loss function of knowledge graph generation

Here, no correlation between overlapping degree and loss function value is visible, expect that low overlapping degrees < 0.2 always yield graphs with comparably low loss. Nevertheless, Figure 7.3 visualizes the size of the rule set and the corresponding loss function of the graph generation.

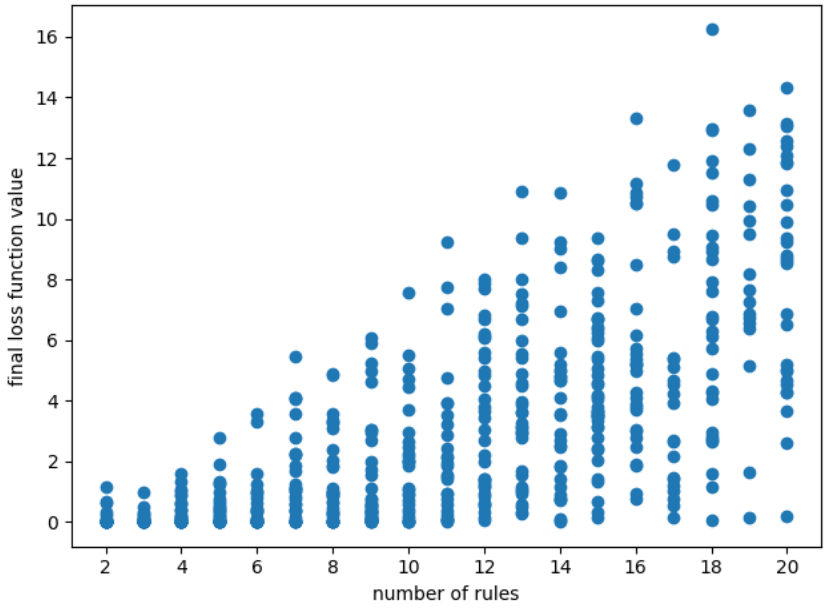


Figure 7.3: Visualization of number of rules in a rule set with loss function of resulting knowledge graph generation

Figure 7.3 shows that there is a correlation between the number of rules and loss function value, because the range of measured loss values increase as the size of rule sets increases. Therefore, visualizing the overlapping degree of a rule set with the loss function of the corresponding graph generation for fixed-sized sets of rules might reveal correlations. Indeed, if considering a number of rules $\in \{4, \cdots, 20\}$, each visualization for any fixed number of rules depicts an increasing loss for higher overlapping degrees. For example, consider Figure 7.4 which depicts the overlapping degree and corresponding loss function value of the generation for rule sets with size 18.

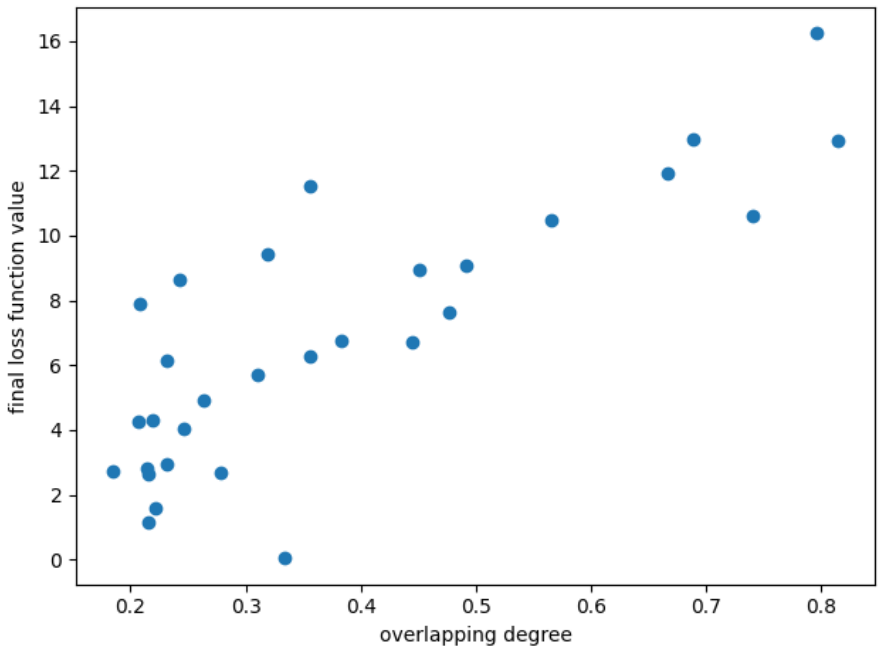


Figure 7.4: Visualization of number of overlapping degree and loss function of resulting knowledge graph generation, only considering sets of rules C with $|C| = 18$

This suggests that a higher overlapping degree of a set of rules makes it more difficult to generate a good graph using hill climbing with greedy (subsumption order) initial graph. Intuitively this makes sense, because in case a relation name occurs across several rules, generating a graph requires considering all relation name occurrences at the same time. As soon as a relation name only occurs in a few rules, adding edges during the graph generation with said relation name only affects the corresponding few rules. Thus, good or bad changes during the graph generation can be distinguished easier.

However, the correlation is not always linear, as seen in Figure 7.5.

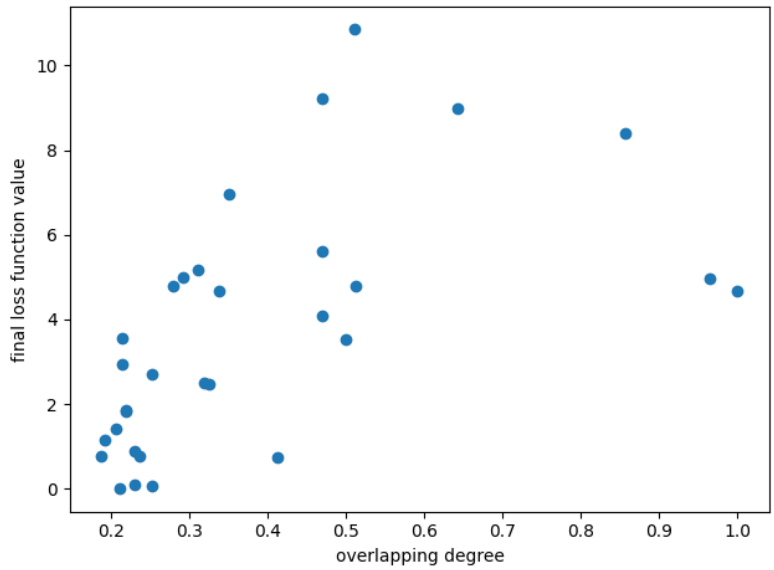


Figure 7.5: Visualization of number of overlapping degree and loss function of resulting knowledge graph generation, only considering sets of rules C with $|C| = 14$

In contrast, generations for sets of rules C with $|C| \in \{2, 3\}$ generations have a no correlation and a comparably low loss for any overlapping degree. However, this likely does not refute the previously made assumption that a higher overlapping degree of a rule set leads to a higher loss of the corresponding graph generation. Instead, this probably shows that generating graphs for any set of rules with a low size leads to a graph with low loss. For instance, consider Figure 7.6.

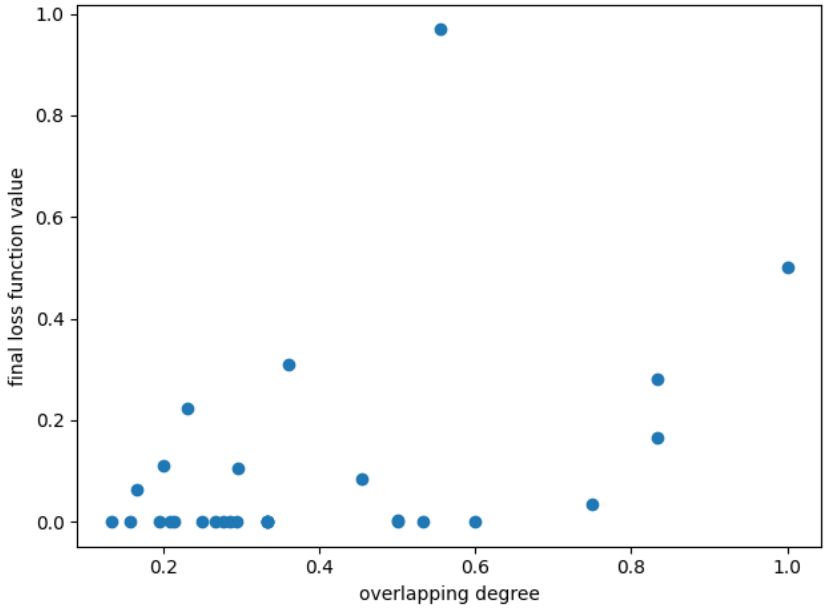


Figure 7.6: Visualization of number of overlapping degree and loss function of resulting knowledge graph generation, only considering sets of rules C with $|C| = 3$

7.3.2 Case Study

This section provides a more intuitive insight into the hill climbing variant with greedy initial graph (subsumption order) by applying it on the small set of rules depicted in Table 7.3.

Rule	EBodySupp	BodySupp	ESupp	Supp	EConf	Conf
$r_2(y,x) \rightarrow r_2(x,y)$	3	3	2	0	0.667	0
$r_1(z,x) \wedge r_3(y,x) \rightarrow r_3(z,x)$	5	5	0	1	0	0.2
$r_2(z,y) \wedge r_3(y,v) \wedge r_2(v,x) \rightarrow r_2(x,y)$	4	1	3	0	0.75	0
$r_3(x,y) \wedge r_1(x,z) \rightarrow r_1(x,y)$	2	2	2	1	1	0.5

Table 7.3: Rule set generated by GENERATERULESET$(4,3,4)$ (Algorithm 7.1) with expected body supports chosen randomly from $\{2,3,4,5\}$ and expected support chosen randomly from 0 between the previous body support

Applying hill climbing yields the graph visualized in Figure 7.7.

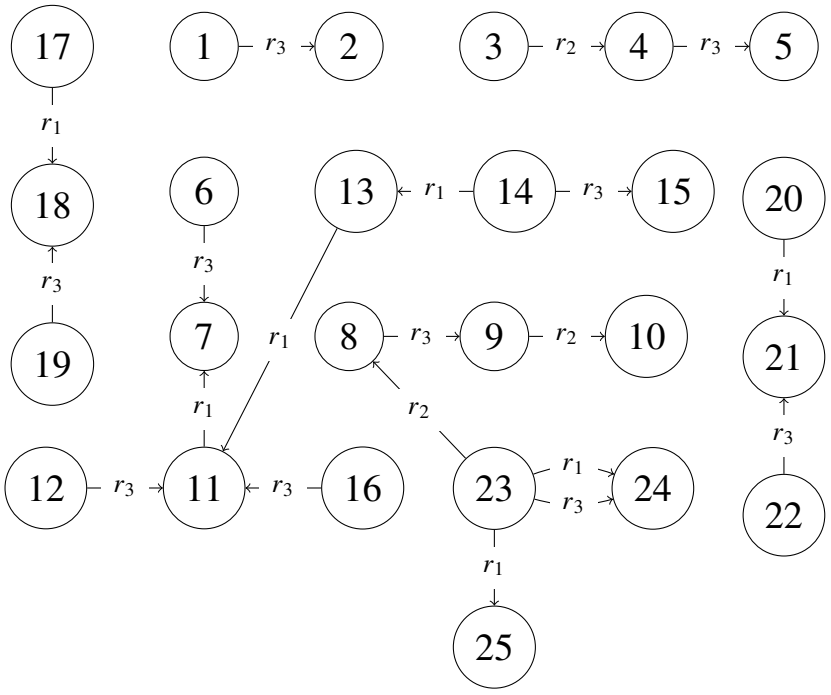


Figure 7.7: Knowledge graph generated by hill climbing algorithm with greedy initial graph (subsumption order) and the rules of Table 7.3 as input

Firstly, no rule is reflected perfectly, i.e., no rule's actual body support, support and confidence matches with their expected values, see Table 7.3. Indeed, the aggregated body support, support and confidence difference are respectively 0.75 ± 1.299, 1.75 ± 0.829 and 0.529 ± 0.21. Interestingly, the body supports are reflected better than supports and confidences. This is likely due to the fact that the loss function ℓ, introduced in Chapter 4, focuses on matching the body supports. On the one hand, the differences of support values are not considered in the loss function, only the difference between body supports (left term in summand of ℓ) and confidences (right term in summand of ℓ).

On the other hand, the confidence difference is always $\in [0, 1]$ while the first term, which measures the normalized difference between body supports, might be bigger than 1. Therefore, matching the body support might be favoured in comparison with matching the confidence.

7.3.3 AMIE

This sections presents the results of applying the state-of-the-art rule mining tool AMIE [GTHS13] to answer RQ2. To do that, a set of eight rules is produced by Algorithm 7.1 while the maximum number of atoms for a rule is set to 5 and the set of relation names is $\{r_1, r_2, \cdots, r_5\}$. Then, each rule got a body support by a random choice between 50 and 200, and support between 0 and the previous body support. The generated set of rules is depicted in Table 7.4.

Rule	EBodySupp	ESupp
$r_1(x_1, x_3) \wedge r_1(x_3, x_4) \wedge r_2(x_2, x_3) \rightarrow r_4(x_4, x_3)$	86	21
$r_3(x_3, x_1) \wedge r_5(x_1, x_3) \rightarrow r_1(x_1, x_3)$	170	53
$r_4(x_1, x_3) \wedge r_5(x_1, x_3) \rightarrow r_2(x_3, x_1)$	55	23
$r_1(x_4, x_3) \wedge r_4(x_4, x_1) \wedge r_2(x_4, x_3) \rightarrow r_5(x_3, x_4)$	142	70
$r_4(x_2, x_4) \wedge r_2(x_5, x_2) \wedge r_2(x_2, x_5) \wedge r_3(x_5, x_4) \rightarrow r_3(x_5, x_2)$	133	87
$r_3(x_1, x_2) \rightarrow r_3(x_1, x_2)$	63	18
$r_2(x_1, x_3) \wedge r_5(x_1, x_3) \rightarrow r_4(x_1, x_3)$	78	36
$r_3(x_2, x_1) \wedge r_2(x_3, x_2) \rightarrow r_2(x_3, x_2)$	62	41

Table 7.4: Set of rules randomly generated by GenerateRuleSet$(8, 5, \{r_1, r_2, \cdots, r_5\})$ (Algorithm 7.1) with their corresponding body support and support

Afterwards, each generation variant generated a graph for the input rules. Furthermore, hill climbing was repeated ten times to reduce the effect of its non-determinism. The loss function value of each generation is depicted in Table 7.5.

Variant	Loss
Greedy random	29.33
Greedy subsumption	29.33
Climbing SuppReach	2.407 ± 0.056
Climbing random	6.624 ± 0.068
Climbing subsumption	2.168 ± 0.068

Table 7.5: Loss function values of all generation variants on a set of eight rules while the loss of climbing variants is aggregated from ten climbing executions

Then, AMIE was configured to mine rules of up to five atoms, "-maxad 5", use the support metric for pruning rules, "-pm support" and no restrictions for support, as well as confidence, "-mins 0 -minc 0.0". Lastly, each generated graph is serialized into the format AMIE expects and AMIE is executed on it.

The results of comparing the mined rules with the intended rules are depicted in Table 7.6.

Variant	Total number of mined rules	Number of mined intended rules
Greedy random	619	0
Greedy subsumption	619	0
Climbing SuppReach	24418.8 ± 8664.982	3.5 ± 0.5
Climbing random	1238.5 ± 280.962	0 ± 0
Climbing subsumption	1458.6 ± 352.951	0.5 ± 0.5

Table 7.6: Number of mined rules for each generation variant while results of hill climbing variants are aggregated by average ± standard deviation across ten executions

First of all, AMIE was not able to identify any intended rules for both variants of the greedy algorithm. Likely, this is due to the high loss, as seen in Table 7.5, which means the rules are not reflected well in the graph. Nevertheless, the graphs are well formed in the way that AMIE mines over 600 unintended rules. Besides that, AMIE behaves similar when encountering the generations of the two hill climbing variants with random or greedy (subsumption order) initial graph. The only difference was that AMIE found one intended rule in some graphs generated by the hill climbing with greedy initial graph. As with the greedy algorithms, the loss is likely still too high such that the rules are reflected well enough to be mined.

However, AMIE finds half of intended rules in most graphs generated by hill climbing with SuppReach initial graph, although the loss is not the best. Additionally, AMIE mined many more unintended rules, in comparison with all other generation variants. The assumption is that this is caused by the way how AMIE mines rules. AMIE looks for relationship cycles to mine rules, e.g., node A is related to node B, then B is related to node C and C is related to A again. Likely, there are more such cycles in the graph if the number of edges is high while the number of nodes is low. This is because if a node has multiple outgoing edges, there are more possibilities to traverse through the graph. In this case, few nodes form many relationships from which AMIE finds many different cycles and thus, many rules. The number of nodes, number of edges and the fraction of both are depicted in Table 7.7.

Variant	#Edges	#Nodes	$\frac{\text{Number of Edges}}{\text{Number of Nodes}}$
Greedy random	1775	2112	1.19
Greedy subsumption	1775	2112	1.19
Climbing SuppReach	1600.5 ± 14.03	721.7 ± 3.348	2.218 ± 0.021
Climbing random	1891.4 ± 21.869	778 ± 2.324	2.431 ± 0.03
Climbing subsumption	1749.6 ± 7.723	1774.6 ± 0.49	0.986 ± 0.004

Table 7.7: Graph statistics for graphs generated by each generation variant while the values for hill climbing variants are aggregated by average ± standard deviation across ten executions

Both greedy variants and hill climbing with greedy initial graph have a small fraction of edges per node. The value suggests about 1 relationship per node and thus, AMIE does not find many cycles. In contrast, the fraction doubles for hill climbing with SuppReach or random initial graph. This could explain the many unintended rules find for the graphs of the hill climbing variant with SuppReach initial graph. However, the reason why AMIE does not find a similar amount of unintended rules for hill climbing with random graph is likely because this variant is the most

random out of all hill climbing variants. Hence, the graph lacks reoccurring patterns AMIE could consider. Lastly, the reason why AMIE finds intended rules for graph generated by hill climbing with SuppReach initial graph is not caused by the rules being reflected better. Instead, there are so many mined rules by AMIE such that the intended rules are “luckily” mined, too.

8 Conclusion

In this thesis, two algorithms that generate random knowledge graphs have been developed and implemented. Moreover, both algorithms allow modifications which led to the implementation of five knowledge graph generation variants. On the one hand, there are three variations of hill climbing which are distinguished by different initial graphs. On the other hand, there are two variations of a greedy algorithm, each characterized by a different rule-ordering strategy. For the evaluation, each generation variant has been applied on random sets of rules and consequently, generated a knowledge graphs. Comparing generated graphs has been done by utilizing a loss function and measuring body support, support and confidence for each rule.

Firstly, both greedy variants (random order and ordering based on rule body subsumptions) have not been able to generate graphs for which the loss function yielded a low value. Furthermore, it has been seen that the body support, support and confidence vary much in comparison to their expected values. Lastly, the evaluation showed that ordering the rules based on expected body supports and subsumptions leads to worse graphs in comparison to greedily fulfilling given rules in a random order.

Interestingly, all hill climbing variants (SuppReach initial graph, random initial graph and greedy initial graph with subsumption order) yielded better knowledge graphs than the graphs generated by both greedy variants while hill climbing with greedy initial graph was the overall best algorithm. Indeed, the greedy variants produced graphs with an average loss of greater than 300 and the graphs generated by hill climbing had average losses ranging from 3.1 to 10.1. Besides, the difference between expected and actual body support, support and confidence improved a lot, too. For example, the average difference between expected body support and actual body support of greedily generated graphs was around 250. In contrast, the average body support difference for hill climbing starting with the greedy (subsumption order) initial graph was 50. However, as the difference of the actual body support, support and confidence in comparison with their expected value remains high, users of the algorithms have to be careful. In general, using an implemented generation variant on a set of rules will not yield a graph that gets close to fulfilling the expected body support, support and confidence.

Another notable piece of information is that a higher overlapping degree of a given set of rules increases the difficulty to generate a graph. This has been found by applying the hill climbing variant with the greedy (subsumption order) initial graph on many rule sets with a variety of overlapping degrees. The overlapping degree and loss function of the resulting graph correlated for rule sets of a fixed size in the following way. If the overlapping degree becomes higher, the loss becomes higher, too, i.e., worse. However, this only holds for sets of rules which contain more than three rules. For lower rule set sizes, the loss was constantly low which suggests that generating a graph for a small-sized set of rules is easy, independent of overlapping degree.

Finally, AMIE found many unintended rules for the graphs generated by any variant. Moreover, it has been found that the amount of unintended rules varies significantly depending on the generation variant. There is the assumption that there is a correlation between the fraction $\frac{\text{number of edges}}{\text{number of nodes}}$ and the number of mined rules. A higher fraction suggests that the mining strategy of AMIE is able to find more patterns in the graph and thus, mines more rules. However, AMIE hardly found any intended rules except half of the rules for the graphs generated by hill climbing with SuppReach initial graph. Nevertheless, the intended rules for hill climbing with SuppReach initial graph were likely mined due to the big total number of mined rules, instead of the rules being represented well. Indeed, AMIE mines the most unintended rules for hill climbing with SuppReach initial graph.

Outlook

As the developed hill climbing adaptation is the applied general technique to solve discrete optimization problems, there are further parts where the approach can be modified. First, the initial graph, i.e., initial state, can be changed. Second, the set of neighbours can be changed, e.g., all graphs that are two changed edges away from the current graph, not just one. Indeed, a change of a single edge could be such a small change that the climbing stops at a local optimum quickly. Lastly, the loss function can be adjusted to benefit further properties, e.g., add a punishing term that punishes graphs that are not connected or do not follow an expected degree distribution.

Finally, the tool could be extended to support further graph types, e.g., temporal knowledge graphs which enhance each edge with the time $t \in \mathbb{N}$. The fourth edge component represents the time when the information stated by the edge holds. Thus, such graphs enable queries based on time constraints. Another example is typed graphs which assign a class to each node. Hence, constraints can be formulated to restrict which class of nodes can be part of which kind of relationships.

Bibliography

[Blo14] Bloch Josh, Gafter Neal. *List (Java Platform SE 8)*. https://docs.oracle.com/javase/8/docs/api/java/util/List.html#sort-java.util.Comparator-. Accessed: 2024-05-31. 2014 (cit. on p. 37).

[Blo98a] Bloch Josh. *Tree Set (Java Platform SE 8)*. https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html. Accessed: 2024-05-30. 1998 (cit. on p. 35).

[Blo98b] Bloch Josh, Gafter Neal. *HashSet (Java Platform SE 8)*. https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html. Accessed: 2024-06-03. 1998 (cit. on p. 40).

[CGT89] S. Ceri, G. Gottlob, L. Tanca. "What you Always Wanted to Know About Datalog (And Never Dared to Ask)". In: *IEEE Trans. Knowl. Data Eng.* 1.1 (1989), pp. 146–166. DOI: 10.1109/69.43410. URL: https://doi.org/10.1109/69.43410 (cit. on p. 13).

[CN10] S. Cook, P. Nguyen. *Logical Foundations of Proof Complexity*. Cambridge, United Kingdom: Cambridge University Press, Apr. 2010. ISBN: 9780511686146 (cit. on p. 13).

[Coo23] S. Cooper. "Sturgeon-GRAPH: Constrained Graph Generation from Examples". In: *Proceedings of the 18th International Conference on the Foundations of Digital Games*. FDG '23. Lisbon, Portugal: Association for Computing Machinery, 2023. ISBN: 9781450398558. DOI: 10.1145/3582437.3582465. URL: https://doi.org/10.1145/3582437.3582465 (cit. on pp. 15, 24).

[Don21] Dongsheng Wang and Hongyin Zhu. *5 Reasons Knowledge Graph will never bloom*. https://dongshengwang.medium.com/5-reasons-knowledge-graph-will-never-bloom-418601957f33. Accessed: 2024-06-02. 2021 (cit. on p. 17).

[DRW98] R. P. Dick, D. L. Rhodes, W. H. Wolf. "TGFF: task graphs for free". In: *Proceedings of the Sixth International Workshop on Hardware/Software Codesign, CODES 1998, Seattle, Washington, USA, March 15-18, 1998*. Ed. by G. Borriello, A. A. Jerraya, L. Lavagno. IEEE Computer Society, 1998, pp. 97–101. DOI: 10.1145/278241.278309. URL: https://doi.org/10.1145/278241.278309 (cit. on pp. 14, 15, 23).

[ER59] P. Erdös, A. Rényi. "On Random Graphs I". In: *Publicationes Mathematicae Debrecen* 6 (1959), pp. 290–297 (cit. on pp. 14, 23).

[FŞA+20] D. Fensel, U. Şimşek, K. Angele, E. Huaman, E. Kärle, O. Panasiuk, I. Toma, J. Umbrich, A. Wahler, D. Fensel, et al. "Introduction: what is a knowledge graph?" In: *Knowledge graphs: Methodology, tools and selected use cases* (2020), pp. 1–10 (cit. on p. 24).

[GKHT11] F. Gandon, R. Krummenacher, S.-K. Han, I. Toma. “The resource description framework and its schema”. In: *Handbook of Semantic Web Technologies* (2011) (cit. on p. 25).

[Gla24] G. Glaser. *Knowledge Graph Generator*. Version v1. 2024. DOI: 10.18419/darus-4436. URL: https://doi.org/10.18419/darus-4436 (cit. on p. 35).

[GNHS23] C. Gregucci, M. Nayyeri, D. Hernández, S. Staab. “Link Prediction with Attention Applied on Multiple Knowledge Graph Embedding Models”. In: *Proceedings of the ACM Web Conference 2023, WWW 2023, Austin, TX, USA, 30 April 2023 - 4 May 2023*. ACM, 2023, pp. 2600–2610. DOI: 10.1145/3543507.3583358. URL: https://doi.org/10.1145/3543507.3583358 (cit. on p. 13).

[GTHS13] L. A. Galárraga, C. Teflioudi, K. Hose, F. Suchanek. “AMIE: Association Rule Mining under Incomplete Evidence in Ontological Knowledge Bases”. In: *Proceedings of the 22nd International Conference on World Wide Web*. WWW ’13. Rio de Janeiro, Brazil: Association for Computing Machinery, 2013, pp. 413–422. ISBN: 9781450320351. DOI: 10.1145/2488388.2488425. URL: https://doi.org/10.1145/2488388.2488425 (cit. on pp. 13, 43, 46, 52).

[HBC+21] A. Hogan, E. Blomqvist, M. Cochez, C. d’Amato, G. D. Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier, et al. “Knowledge graphs”. In: *ACM Computing Surveys (Csur)* 54.4 (2021), pp. 1–37 (cit. on p. 13).

[HMd+24] N. Hubert, P. Monnin, M. d’Aquin, D. Monticolo, A. Brun. *PyGraft: Configurable Generation of Synthetic Schemas and Knowledge Graphs at Your Fingertips*. 2024. arXiv: 2309.03685 [cs.AI] (cit. on pp. 15, 25).

[JLR17] P. Jonsson, V. Lagerkvist, B. Roy. “Time Complexity of Constraint Satisfaction via Universal Algebra”. In: *CoRR* abs/1706.05902 (2017). arXiv: 1706.05902. URL: http://arxiv.org/abs/1706.05902 (cit. on p. 24).

[KKS+23] J. Kunegis, P. Kumar, J. Sun, A. Samoilenko, G. Pirr’o. “SynGraphy: Succinct Summarisation of Large Networks via Small Synthetic Representative Graphs”. In: *ArXiv* abs/2302.07755 (2023). URL: https://api.semanticscholar.org/CorpusID:256868451 (cit. on pp. 14, 15, 23).

[KSY23] J. Kunegis, J. Sun, E. Yoneki. “Guided Graph Generation: Evaluation of Graph Generators in Terms of Network Statistics, and a New Algorithm”. In: *CoRR* abs/2303.00635 (2023). DOI: 10.48550/ARXIV.2303.00635. arXiv: 2303.00635. URL: https://doi.org/10.48550/arXiv.2303.00635 (cit. on pp. 14, 15, 23).

[Lib04] L. Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN: 3-540-21202-7. DOI: 10.1007/978-3-662-07003-1. URL: http://www.cs.toronto.edu/%5C%7Elibkin/fmt (cit. on p. 18).

[LLP+15] S.-H. Lim, S. M. Lee, S. Powers, M. Shankar, N. Imam. “Survey of approaches to generate realistic synthetic graphs”. In: *Oak Ridge National Laboratory* (2015) (cit. on p. 13).

[ON16] M. A. Omrani, W. Naanaa. “A Constrained Molecular Graph Generation with Imposed and Forbidden Fragments”. In: *Proceedings of the 9th Hellenic Conference on Artificial Intelligence*. SETN ’16. Thessaloniki, Greece: Association for Computing Machinery, 2016. ISBN: 9781450337342. DOI: 10.1145/2903220.2903234. URL: https://doi.org/10.1145/2903220.2903234 (cit. on pp. 15, 24).

[RRN20] S. Russell, S. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson series in artificial intelligence. Pearson, 2020. ISBN: 9780134610993. URL: https://books.google.de/books?id=koFptAEACAAJ (cit. on pp. 19, 20).

[TK02] T. Tobita, H. Kasahara. "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms". In: *Journal of Scheduling* 5.5 (2002), pp. 379–394 (cit. on pp. 14, 23).

[TKBG23] T. Thanapalasingam, E. van Krieken, P. Bloem, P. Groth. "IntelliGraphs: Datasets for Benchmarking Knowledge Graph Generation". In: *arXiv preprint arXiv:2307.06698* (2023) (cit. on pp. 13, 15, 24, 25).

[Tur37] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* 1 (Jan. 1937), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230 (cit. on p. 13).

[VJ24] M. Vecovska, M. Jovanovik. *Synthetic RDF graph generator based on SHACL constraints*. https://github.com/mveco/RDF_generator. Last accessed at 2024-04-08, commit 8dfa7d3. 2024 (cit. on pp. 15, 25).

[VM04] F. Van Harmelen, D. L. McGuinness. "OWL web ontology language overview". In: *World Wide Web Consortium (W3C) Recommendation* 69 (2004), p. 70 (cit. on p. 25).

[WMWG17] Q. Wang, Z. Mao, B. Wang, L. Guo. "Knowledge Graph Embedding: A Survey of Approaches and Applications". In: *IEEE Transactions on Knowledge and Data Engineering* 29.12 (2017), pp. 2724–2743. DOI: 10.1109/TKDE.2017.2754499 (cit. on p. 13).

[YW09] X. Ying, X. Wu. "Graph generation with prescribed feature constraints". In: *Proceedings of the 2009 SIAM International Conference on Data Mining*. SIAM. 2009, pp. 966–977 (cit. on pp. 14, 15, 23).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature