Negative Sampling for Knowledge Graph Embedding Using Rule-Based Reasoning

Yu Anlan

March 2021 - December 2021

Abstract

Knowledge Graphs (KG) are multi-relational graphs that can be represented as triples. Knowledge Graphs are commonly used in AI for tasks such as question answering, link prediction, recommendation systems, etc. Knowledge Graph Embedding (KGE) models encode high-dimensional data into lower dimension vector representations. KGE models can solve problems such as scalability issues and make computing on KGs easier. The gold standard for training KGE models, contrastive learning, uses negative sampling to increase learning efficiency, usually, a negative sample corrupts either the head or tail of a positive triple. However, most negative sampling methods today do not consider the actual data and their logical connections, and it is difficult to find negative samples that result in a considerable gradient on the model.

In this thesis, a new KGE training pipeline is proposed. Under this new pipeline, a KG is materialized with false rules (created based on existing rule sets) using a rule engine, additional triples from such materialization are then used to form their own graph, effectively creating a set of facts and a set of lies. We proposed a procedure to automatically create such false rules and to generate lies. We then developed several new negative sampling methods to combine the facts and lies for the KGE training. We also studied the effects of corrupting the KG before materialization by the reasoning system.

Contents

1 Introduction

Knowledge Graph (KG) is a type of data representation that usually stores factual information as triples. Each triple consists of a head, relation, and tail, and a triple corresponds to a (directed) edge on the graph from one entity to another, and each edge represents a relation type. Graph embedding techniques are developed to encode high-dimensional knowledge graph data into lower-dimensional space. Knowledge graphs have scalability issues and operating on a large knowledge graph can be computationally expensive, this is one of the issues a graph embedding model can help to resolve. Many graph embedding models adopted the method of contrastive learning. Contrastive learning uses negative samples, which are created by corrupting the entities in the triples and are considered factually incorrect compared to the training samples from the original knowledge graph. Some negative sampling methods consist of probabilistically replacing the entity with other entities from the graph, intuitively, the randomly selected entity has relatively little resemblance to the original entity, which provides little information to the embedding model and makes distinguishing such negative samples less challenging. Numerous new negative sampling techniques have been studied.

Reasoning algorithms can be applied to knowledge graphs to infer new knowledge, many algorithms use rules and such processes are called rule-based reasoning. This paper mainly involves rule-based reasoning under the Resource Description Framework (RDF). A systematic way of automatically generating "false rules" in the reasoning system will be designed, a separate knowledge graph called the lies graph will be created using these rules, and combined with new negative sampling methods, will form a new KGE training pipeline.

We aim to answer these research questions in this work:

1. Is it possible to generate negative samples by using a rule-based reasoning system, and if so, how will the reasoning system be incorporated into the KG embedding training?

2. Can the aforementioned method improve the result of the embedding models, and what additional hyper-parameters have impact on such results?

3. In such proposed method, can we summarize the relation between various factors of the reasoning and the impact of the negative samples from such reasoning on the KGE performance?

This thesis will be structured as follows: in Section 2, we will first discuss the basis of knowledge graph embedding and relevant background information,

[0]The implementation for materializing KGs with false rules can be found in the Github repository at `https://github.com/PTSTS/MasterThesis`

The implementation for the new negative sampling methods can be found in the Github repository at `https://github.com/PTSTS/SANSoL`

The experiment records can results can be found in this archive on Google Drive: `https://drive.google.com/drive/folders/1VOZGrn2CTIIanH921iXyxYFw2XMZqRUu?usp=sharing` then a list of previous negative sampling methods will be presented, lastly we will also discuss related techniques used in training knowledge graph embedding with negative samples; in Section 3, the methodology and the rationale of the proposed negative sampling method will be presented, including different steps of the negative sampling process and variations of the method; in Section 4, we will present a detailed setup of experiments testing the proposed methods, including some information regarding the implementation and hyper parameter choices. We will follow up by presenting observed results from such experiments in Section 5, then give an analysis of the results, try to answer the aforementioned research questions, list the limitations of our approach, and outline future work in Section 6.

2 Related Work

In this section, a brief survey is conducted to cover previous works, which includes the theoretical background of this work, the methods and techniques that this work is based on, as well as the previous studies that influenced or inspired our methods.

2.1 Knowledge Graphs

A graph can be represented by $G(V, E)$ where V is the set of vertices and E is the set of edges. Knowledge graphs use graph structures to represent knowledge. In a knowledge graph, a vertex $v \in V$ is an entity, such entity can correspond to a real-world concept or an abstract one such as a class or a data type. An edge $e \in E$ is a relation from one entity to another, thus making knowledge graphs directed graphs[1] (in some cases the edges in KG can be undirected, if the relation is reflexive). These types of data are called "multi-relational data", the adjacency matrices of KGs contain categorical data rather than numerical ones. Formally define KGs, $G = \{V, E, R\}$, where $V = \{v_1, \cdots, v_n\}$ is a set of nodes each representing one entity, $R = \{r_1, \cdots, r_n\}$ is a set of relations denotes all possible relations in G, and $E \subseteq V \times R \times V$ is a set of edges, each edge is a mapping from a node to a relation, then to another node [2].

As previously discussed, KGs are stored as triples, and usually, triples are the only way for KGs to contain information, this means all types and attributes of a certain entity are stored as triples too. In contrast to other graph types, KGs do not associate numerical values with edges, as a result, there is no concrete definition of the distance between nodes, only the neighboring nodes and the type of relation will be considered. Embedding models on KG usually have their own methods to calculate the distance between nodes, which will be discussed later in this section.

2.2 Knowledge Graph Embedding

Computation on graphs has a high cost and scaling is difficult due to the high dimensionality. Graph embedding methods have been developed to solve this problem efficiently by encoding graphs into lower-dimensional spaces. The embedding process also needs to preserve the structure of the original graph, so that useful information can still be extracted from the embedded graph. Usually the number of dimensions of the embedding (d) satisfies $d \ll |V|$. Another reason to use graph embedding is to incorporate discrete and logical data into numerical machine learning tasks.

Two types of embedding problems will be discussed in this section. Node embedding maps every node in the KG to a vector, if certain nodes are close in the original graph then their embedding vectors should have shorter distances as well. The closeness of nodes on the graph can be defined in many ways, such as first-order proximity, etc.

Apart from node embedding, modeling relations on KGs can be challenging, mainly due to the variety of relations that can be present. For transitive relations, $(a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$, the embedding vectors for a, b and c should all be located in close proximity, but for non-transitive relations this does not hold true. Thus for an embedding model on multi-relational data, the model must also determine how to encode each relation to reflect their actual properties.

One way of approaching this challenge is translation models [3]. Here "translations" are the parameterization of relations. Relation embedding in translation models function in a very similar way but encodes relations to vectors. The embedded relation vector should reflect the difference between the embedded vectors of the head and tail entities. For example, given a triple $\langle h, r, t \rangle$, and let $\operatorname{emb}(x)$ be the embedding vector of entity or relation x, then given a perfect embedding model, $\operatorname{emb}(h) + \operatorname{emb}(r) = \operatorname{emb}(t)$.

We will have a closer look at some of the translation models.

2.2.1 TransE[3]

First, define the distance or dissimilarity function d to be the L_1 or L_2 distance between vectors. Then define margin ranking loss:

$$L = \sum_{(h,r,t) \in S} \sum_{(h',r,t') \in S'} \left[\lambda + d(\mathbf{h} + \mathbf{r}, \mathbf{t}) - d(\mathbf{h}' + \mathbf{r}, \mathbf{t}')\right]_+$$

In the loss function notation $[]_+$ makes the summation take only positive values, $\mathbf{h}', \mathbf{t}'$ denote the negative samples of h and t, λ is a hyperparameter for the margin. S' is the set of all negative samples, which will be discussed in the following sections.

This paragraph will describe the learning process of TransE. The model initializes with all entity and relation vectors taking some random uniform values normalized across k intended dimensions of the embedding. In each training loop, as a subset of the data is selected as a batch, and for each sample in the batch a negative sample is prepared and combined with the positive sample as a pair. Then the embedding model is updated by the following gradient:

$$\sum_{p \in P} \nabla [\lambda + d(\mathbf{h} + \mathbf{r}, \mathbf{t}) - d(\mathbf{h}' + \mathbf{r}, \mathbf{t}')]_+$$

in which P stands for the set of pairs for this particular loop, and the model is descended by the sum of gradients of margin ranking loss of all pairs of positive and negative triples.

TransE saw improvements compared to previous embedding methods, such as structured embedding [4] and semantic matching energy function [5]. More importantly, TransE needs fewer parameters and is more scalable, and is the baseline for other translation embedding models. However, TransE does have its shortcomings, such as difficulty to encode one-to-many or many-to-many relations [6]

2.2.2 TransH

Previously, TransE encodes relations in the same embedding space as all the entities. The idea of the Translation on Hyperplanes model (TransH) [7] is to use another representation of the relation embedding, which is to use a hyperplane with one less parameter than the embedding space to encode each relation. The trained entity embeddings of each triple are then projected onto its relation's hyperplane and the loss is calculated from the projected distance, $\mathbf{h}_\perp = \mathbf{h} - \mathbf{w}_r^\top \mathbf{h} \mathbf{w}_r$ and $\mathbf{t}_\perp = t - \mathbf{w}_r^\top \mathbf{t} \mathbf{w}_r$ ($\mathbf{w_r}$ is the hyperplane), the vector on the hyperplane for a specific relation r is defined as d_r.

The score function for a triple is $f_r(\mathbf{h}, \mathbf{t}) = \|\mathbf{h}_\perp + d_r - \mathbf{t}_\perp\|_2^2$, and the different learning process for TransH from TransE is that TransH tries to minimize all the projected distances of entity embeddings on relation hyperplanes. Similar to TransE, TransH also uses MRL as loss function.

2.2.3 TransR

Aiming to improve upon TransE and TransH, Translation in Relation Space (TransR) [8] encodes entities and relations on completely different spaces. Instead of projecting entities on the embedding space directly to calculate distance relative to the relation, TransR uses a projection matrix, the head and tail entities can be simply translated to the relation space as a product: $\mathbf{h}_r = \mathbf{h}\mathbf{M_r}$ and $\mathbf{t}_r = \mathbf{t}\mathbf{M_r}$.

The loss function of TransR is:

$$l = \sum_{(h,r,t) \in S} \sum_{(h',r,t') \in S'} \max(0, f_r(h,t) + \gamma - f_r(h',t'))$$

where f is the same score function as in TransH (Section 2.2.2).

2.2.4 TransD

Knowledge Graph Embedding via Dynamic Mapping Matrix (TransD) [6] functions similar to TransR but provides a system of two embedding vectors for each entity or relation. Differing from TransR, the second vector is called the projection vector and is responsible for projecting the entity or relations to a different relation vector space.

The transformation to embedding space of TransD is explained by the following formulas, note that $\mathbf{h}$, $\mathbf{r}$ and $\mathbf{t}$ are the original embedding vectors, $\mathbf{h}_p$, $\mathbf{r}_p$ and $\mathbf{t}_p$ are the projection vectors. $\mathbf{M}_{rh}$ and $\mathbf{M}_{rt}$ are the mapping matrices..

$$\begin{cases} \mathbf{M}_{rh} = \mathbf{r}_p\mathbf{h}_p^\top + \mathbf{I}^{m\times n} \\ \mathbf{h}_\perp = \mathbf{M}_{rh}\mathbf{h} \end{cases} \qquad \begin{cases} \mathbf{M}_{rt} = \mathbf{r}_p\mathbf{t}_p^\top + \mathbf{I}^{m\times n} \\ \mathbf{t}_\perp = \mathbf{M}_{rt}\mathbf{t} \end{cases}$$

After the transformation, $\mathbf{h}_\perp$ and $\mathbf{t}_\perp$ are the transformed vectors in the relation vector space and are in the same space as $\mathbf{r}$. TransD uses a similar score function: $f_r(\mathbf{h}, \mathbf{t}) = -\|\mathbf{h}_\perp + \mathbf{r} - \mathbf{t}_\perp\|_2^2$.

TransD's experiment results show that it outperforms all previous translation models, as well as a variant of TransR, CTransR, and it considers the diversity of both entities and relations.

2.2.5 ComplEx

Instead of using real value vectors to represent KGE, we can substitute the values with complex numbers. The embedding model ComplEx is based on this idea. Unlike some of the previously mentioned translation models, the entity embeddings are not symmetric, meaning that an entity would have a pair of different embeddings depending on whether it's the subject or object given a certain relation (in previous models, the entity would have the same embedding regardless of its position, hence the model is symmetrical). The entity's subject and object embeddings are complex conjugates of each other, so if we define $\overline{\mathbf{x}}$ as the complex conjugate of $\mathbf{x}$, and the real and imaginary parts of $\mathbf{x}$ are a and b respectively, then $\overline{\mathbf{x}} = a - b\mathrm{i}$.

Let $\mathbf{e}_h, \mathbf{w}_r, \mathbf{e}_t$ be the head, relation and tail embedding vectors respectively, every vector is complex: $\mathbf{e}_h, \mathbf{w}_r, \mathbf{e}_t \in \mathbb{C}$. The score function would be $\phi(h, r, t) = \mathbf{R}(\langle \mathbf{w}_r, \mathbf{e}_h, \overline{\mathbf{e}_t} \rangle)$ where $\mathbf{R}$ takes the real part of a complex number. The complex conjugation here transforms the same complex embedding vector into two distinct real value vectors on the embedding space, for differentiation between subject and object positions, instead of using two vectors for this purpose.

2.3 Resource Description Framework (RDF)

RDF is a standard of modeling data on the Web [9]. RDF was developed under the semantic web, which aims to organize real-world data in a machine-compatible way. The data used in the following experiments are expressed as RDF triples, the head, relation and tail in a triple are also expressed as subject, predicate and object in RDF. The entities and relations are stored as URI [10], specifically as IRI [11]. It is possible to use pre-defined domain names in RDF, such as the ones by W3C.

The advantage of using RDF includes its flexibility, schemaless property, scalability, and the ability to use rules and ontologies to infer additional information.

2.4 Training Assumptions

The information a KG can contain is limited, there are different ways to interpret the empty part of a KG (in other words, what to entail if a triple is not in the KG). These are called "assumptions".

2.4.1 Closed World Assumption (CWA)

CWA assumes that whatever is not known to be true is false[12]. This means that any triple not contained in the data is automatically false so that any triple that differs from the knowledge can be a negative sample.

CWA has two shortcomings. First, CWA should only be considered correct if the KG is complete, however, this is rarely the case in practice. No KG can claim to contain all correct knowledge in its domain, such an assumption disregards the complexity of world knowledge. Second, KGs do not have constraints like traditional databases do, applying CWA might cause inconsistency among existing tuples [13].

2.4.2 Open World Assumption (OWA)

In contrast to CWA, OWA considers the KGs to be incomplete [14]. In doing so, any triple that is not contained in the KG is no longer considered incorrect but "unknown". If the knowledge in the original graph is considered correct, there can still be an arbitrary number of new facts to be inferred from the KG, and the knowledge will never be complete.

Most relevant work in KG embedding negative sampling, especially in semantic web and RDF use OWA as their guideline [15].

2.4.3 Local Closed World Assumption (LCWA)

First proposed by Dong et al. [16], LCWA stems from the fact that no KG is complete. To summarize, LCWA only considers a triple to be incorrect if there exist other triples with the same relation and one of its entities.

Let us consider replacing the tail of a triple (h, r, t), we define a function that derive the set of triples that have the same head and relation in the KG as $\mathrm{T}(h, r)$, if $\mathrm{T}(h, r) = \varnothing$ then any negative triple (h, r, t') is not considered incorrect and thus remains unknown. The triples are only considered incorrect if $|\mathrm{T}(h, r)| > 0$ and $(h, r, t') \notin KG$.

2.4.4 Stochastic Local Closed World Assumption (sLCWA)

Instead of considering all potential triples that satisfy certain criteria (share one entity with an existing triple in the KG but different from all triples in the KG as in LCWA), sLCWA considers a randomly selected set of triples to be false while the rest of all potential triples are still unknown. Figure 1 shows a visualization of the two different assumptions with a simple example, we represent the relations between entities in an adjacency matrix (relation types are ignored). The red cells are existing triples in the KG, the dark blue region is always false, the light blue regions are sometimes false (if they are selected) and at other times unknown, and the yellow region is always unknown.

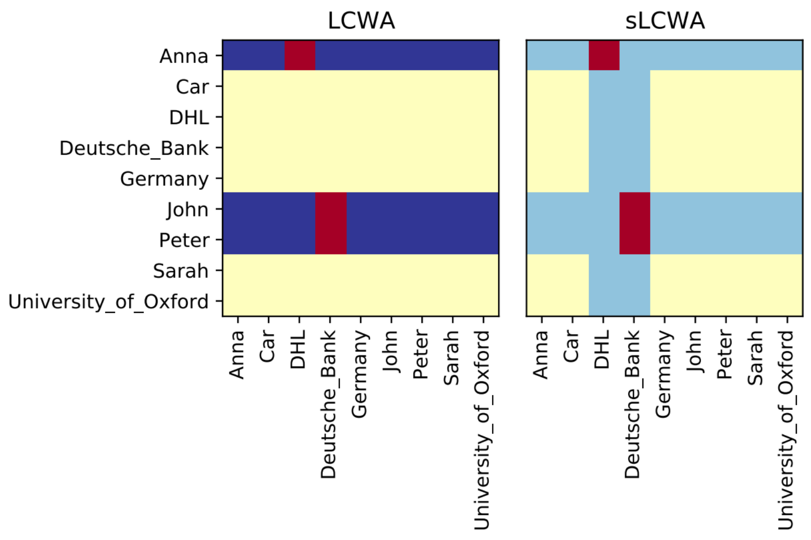

Figure 1: Example of LCWA and sLCWA comparison [17]

Stochastic local closed-world assumption enables us to define our own random process to select a set of triples to consider as false, which can be then used in contrastive learning. We can then sample negatively from the KG in a certain way, and assume that the samples are considered false instead of unknown, thus sLCWA is the basic underlying assumption for using negative sampling techniques. The advantage of using sLCWA is that the number of negative samples can be changed and the process of deriving these samples can be customized for our needs.

2.5 Negative Sampling

Contrastive learning has become the gold standard for training KG embeddings in recent years, especially for the translation models mentioned above. Negative samples are provided to the embedding model in contrast to the positive samples from the original graph, to speed up the training process and achieve higher scalability. This contrastive learning process is incorporated into the loss functions, such as MRL in TransE. For such a process to achieve its intended goal, the negative samples need to reflect actual incorrect knowledge, and such sampling is justified under the sLCWA (Section 2.4.4). The goal for contrastive learning usually includes increasing the distance of negative embedding entities and decrease the distance of positive ones [18].

The first approach of negative sampling was uniform sampling or other fixed random sampling methods [19]. However, such samples provide little information and likely have close to zero gradient for the loss function. Several methods have been proposed to generate "hard" negative samples.

2.5.1 GAN-Based Negative Sampling

Wang et al. [20] used Generative Adversarial Network (GAN) to create negative samples. The basic idea is to use a generator to create negative samples along with positive ones, and use a discriminator to provide feedback on the actual embedding model and update the generator in one loop. In this case, the discriminator is the embedding model itself such as translation models, with its own reward function $R = \tanh(L)$ where L is the loss for the embedding such as MRL, this reward is then returned to the generator for calculating gradient and updating parameters.

In their model, the generator has its entity, relation, and reversed relation embeddings. Either one of the head or tail entities is replaced (they can be replaced with either a discrete uniform distribution or a Bernoulli distribution), and the generator will calculate a probability distribution for all entities across the graph given the embeddings of the entity-relation pair as input:

$$p\left(e\,|(h,r,t)\,,z;\theta\,\right) = \begin{cases} p\left(e|t,r;\theta\right) & ,z=1 \\ p\left(e|h,r^{-1};\theta\right) & ,z=0 \end{cases}$$

Here, r^{-1} is the reversed relation, z is a binary flag set to 1 if the head is replaced and 0 if the tail is replaced, θ is the parameters, and e is the designated entity.

Inside each training loop, two steps are performed to train the generator and the discriminator separately, first the generator creates sample pairs to a static discriminator, the reward is calculated based on the output of the discriminator, and the generator is updated, second the generator remains static and provides negative samples to train the discriminator, the discriminator calculates loss and is updated with it. GAN-based model is shown to be effective with several graph embedding models.

2.5.2 Affinity Dependent Negative Sampling

Affinity Dependent Negative Sampling [21] also aims to solve the problem of finding hard negatives. In summary, this method tries to find entities that are closer to the replaced entity according to a certain measure to create negative samples.

Whether to replace the head or tail node is Bernoulli-distributed. Then, for each of the entities in the graph, a cosine similarity score [22] to the replaced entity is calculated. The similarity scores of all entities become the "affinity" vector $\mathbf{M}$, then the fitness score of each entity can be calculated as: $Fitness_i = \frac{\mathbf{M}_i}{\sum_j \mathbf{M}_j}$ and when creating negative samples the samples are drawn from the cumulative distribution of the fitness scores.

2.5.3 Self Adversarial Negative Sampling

In the embedding model RotatE, Sun et al. [23] proposed "self adversarial" method for negative sampling, with the following distribution:

$$p\left(h'_j, r, t'_j \middle| \{(h_i, r_i, t_i)\}\right) = \frac{\exp \alpha f_r\left(\mathbf{h}'_j, \mathbf{t}'_j\right)}{\sum_i \exp \alpha f_r\left(\mathbf{h}'_i, \mathbf{t}'_i\right)}$$

To put simply, the samples come from a distribution that is calculated based on the current embedding model and has a temperature value α, so that the further the two corrupted entities are located from each other in the learned embedding space compared to other entity pairs, the more likely they will be chosen as a negative sample.

However, computing the probability for all possible sample pairs requires square time and is quite costly on large graphs, so instead a weight factor is calculated for a uniformly sampled pair similar to the distribution formula and incorporated into the loss function:

$$L = l\left(\mathbf{h}, \mathbf{t}\right) - \sum_{i=1}^{n} p\left(h'_i, r, t'_i\right) l\left(\mathbf{h}'_i, \mathbf{t}'_i\right)$$

where l is the desired loss function term for a single sample.

Results show that this embedding method improves the performance of TransE compared to uniform sampling.

2.5.4 Structure-Aware Negative Sampling (SANS)

SANS was proposed by Ahrabian et al. [24] and was designed to represent the structure of KGs in negative sampling processes compared to other methods. For each node, SANS considers its k-hop neighborhood, $\mathbf{K} = S^{+}\left(A^k + A^{k-1}\right)$ where A is the adjacency matrix and S^{+} is the element-wise sign function. K-hop neighborhood can be simply explained as the set of nodes that are exactly k edges away from the root node, for example 1-hop neighbors are adjacent nodes. To create negative samples, SANS uniformly samples from a node's k-hop neighborhood ($k > 1$), these negative samples are also called "local negatives" since these corrupted entities can be reached from the root entity in k steps.

The authors of SANS argued that these closely located samples are harder for the model to learn and will improve the embedding.

There is another version of SANS which uses random walk algorithm (RW-SANS) to approximate the k-hop neighborhood instead of calculating the adjacency matrices with tensors. SANS can also be combined with the self-adversarial negative sampling method, this can be done simply by restricting the sampling to the k-hop neighborhood instead of the whole graph.

SANS and its variants have improvements on multiple embedding models including previously mentioned TransE and RotatE compared to previous methods, and it can be considered a state-of-the-art negative sampling method for KG embedding.

2.6 Rule-Based Reasoning on Knowledge Graphs

The process of reasoning can be summarized as deriving a series of conclusions given a set of premises and a set of rules. We define the following: a set of constants C, a set of predicates P, a set of variables V, a term t is either a constant or a variable, an atom, $p(t)$, is a predicate with a number of terms $|t| = \mathrm{ar}(p)$, where $\mathrm{ar}(p)$ represents the arity or the number of arguments of a predicate. A fact is an atom without variables. A rule r is comprised of one head atom and several body atoms, in the following form: $H \leftarrow B_1 \cdots B_n$. [25]

We can regard a KG as a collection of facts from the definition above, and since all facts are triples in a KG, in general, we can represent all triples on the facts graph, as defined in Section 3.2, as 2-arity atoms: $\mathrm{ar}(p) = 2$ for all $p \in G_F$, in the from of $r(h,t)$. The facts in a KG can also be represented with 3-arity "triple" predicates: $triple(h,r,t)$. Usually, the rules used in reasoning do not follow a unified format.

A reasoning algorithm applies rules on the facts to derive new facts, and the reasoning process is complete if no additional facts can be derived from any rule in the rule set. A simple example is shown here: suppose we have two triples (facts) in the KG: (teacher1, teaches_course, course1), (student1, takes_course, course1), and the rule set has the following rules:

$$\text{teaches\_course}(x,y) \leftarrow \text{triple}(x, \mathit{teaches\_course}, y)$$

$$\text{takes\_course}(x,y) \leftarrow \text{triple}(x, \mathit{takes\_course}, y)$$

$$\text{teaches\_student}(x,z) \leftarrow \text{teaches\_course}(x,y), \text{takes\_course}(z,y)$$

$$\text{triple}(x, \mathit{teaches\_student}, y) \leftarrow \text{teaches\_student}(x,y)$$

The first two rules convert triples to their respective 2-arity predicates, the third rule assumes the logic that a course's teacher teaches the students that take such course, and the forth rule converts the predicate into a triple. After inputting the rules and the KG into the reasoning algorithm (such process is called "materialization"), an additional triple will be created: (teacher1, teaches_student, student1).

3 Methodology

3.1 Overview

The previous methods mentioned in related work provided decent negative samples, however, since all negative samples are sourced from the existing triples, they lack the consideration for the logical connections within the data itself. Therefore, this thesis explores the possibility to improve the negative sampling even further by using reasoning algorithms. In this section, the new method will be proposed and discussed. Negative samples are drawn from the result of reasoning with false rules. Such a negative sampling method is then combined with some of the previous methods to generate negative samples for knowledge graph embedding models.

The novel method that we proposed breaks down into two major tasks, the first one is creating an additional KG that contains false information, and the second one is sampling negatives from such a KG during our KG embedding training.

Our proposed method can be further divided into the following steps: first, the existential rules acquired for a certain KG will be modified to create false rule sets; optionally, a duplicate of the original KG is corrupted; then, the KG is materialized on these false rule sets to generate lies (also in the form of KGs), these lie are also filtered by another KG that has been materialized on correct rules, which we define as "full graph"; lastly, a new method is designed to train the KGE on both the lies graph and the original graph.

3.2 Knowledge Graphs in Pre-Processing

In this part, we will define three different KGs that we will use for embedding training. These three KGs are all created and stored during the pre-processing, and two of them, the facts graph and the lies graph are used for the actual training.

Facts Graph As mentioned in the training assumptions for KG embedding models, we consider the data inside a KG to be facts, as the data is divided into train, test, and validation sets (the details of this separation will be further explained in the Section 4); we will only have access to the train set, thus this part of the graph contains sound but incomplete knowledge. We define the original train set (which contains all possible entities, all different relation types, and a part of the links between the entities) as **Facts**.

Fully-Materialized Graph A core component of this work is to integrate reasoning on graphs into the KG embedding training. Given a set of reasoning rules, more triples can be generated from the facts graph (see section for the materialization process). If we reason with a set of correct rules on the known facts, we can expect to obtain new knowledge (as additional links across the existing entities). The new KG after reasoning on correct rules is defined in this work as "**fully materialized graph**".

This new knowledge may or may not benefit our KGE model. When we obtain reasoning rules for any KG, the rules are usually created manually according to certain logic and intuition. Although these rules are considered correct, we cannot assume that they fit the data in the facts graph with zero incompatibility, nor can we assume, due to the nature of the data stored in KGs, that certain data used to entail more knowledge are the way the creators of the rules intended. That is why in this work, we do not use the fully materialized graph as training data (whether materializing the KG on correct rules can improve the embedding models is an interesting point of discussion, but it is not the focus of this work, a previous study [26] has shown that the materialization on KGs has a negative impact on embeddings, although the study does not use the exact framework as ours).

However, under the stochastic local close world assumption, we can trust the new knowledge with an arbitrary level of certainty, even though we might never verify whether the newly materialized graph has false information or not. Thus we can still use the fully materialized graph to filter negative samples, namely, any negative samples that belong to either the original facts graph or the fully materialized graph are disqualified.

The fully materialized graph (as well as the lies graph in the next part) may happen to contain triples in the test and validation set, but since these triples are derived independently from the training data, we do not consider this a leakage.

Lies Graph Similar to the fully materialized graph, the lies graph is defined as the facts graph materialized with incorrect rules. Similar to the argument presented before, materializing the facts graph with incorrect reasoning rules will not necessarily result in incorrect knowledge, both correct and incorrect triples can be generated in this process. Thus we need to use stochastic close world assumptions again and assume the triples we created with incorrect rules are factually incorrect, as long as they are not contained in the facts or fully materialized graph.

Alternatively, the lies graph can also be created from materializing the KG with inserted corrupted triples. We can use either correct rules or incorrect rules in this step. We assume that we can entail incorrect information from existing incorrect information using either correct or incorrect reasoning rules.

3.3 Pre-Reasoning Corruption on Facts Graph

A set of additional triples is generated before the graph is materialized on truthful rules, these triples are then combined with the triples in the train set and input into the reasoner to carry out correct materialization. After the materialization, triples already existing in the train set will be filtered out. The reason for combining the corrupted data and the training data is that we assume the reasoner can derive more false information.

We define additional triples ratio ρ, given the total number of triples in the original training set t, a total of ρt additional triples will be generated. In this work, we limit our pre-reasoning corruption method to uniform corruption, namely the corrupted entity can take any value from the set of entities with uniform probability.

The pseudo-code for pre-reasoning corruption can thus be described as follows:

Algorithm 1: Pre-Reasoning Corruption on Facts Graph

```
   Input: ratio $\rho$, facts graph $G_F$
   Output: corrupted graph $G_C$
1  number of entities $|E|$ from $G_F$; $n := \lceil \rho\, |E| \rceil$; initialize $G_C$ as a copy of
     $G_F$
2  while $|G_C| < n$ do
3      randomly select $(h, r, t)$ from $G_F$
4      if 0.5 probability then
5          uniformly sample $\overline{h}$ from $E$
6          if $(\overline{h}, r, t) \notin G_C$ then
7              $G_C = G_C \cap \{(\overline{h}, r, t)\}$
8          end
9      end
10     else
11         uniformly sample $\overline{t}$ from $E$
12         add $(h, r, \overline{t})$ to $G_C$ if $(h, r, \overline{t}) \notin G_C$ then
13             $G_C = G_C \cap \{(h, r, \overline{t})\}$
14         end
15     end
16 end
```

3.4 Rule-Based Reasoning for Generating Negative Samples

3.4.1 Rule-Based Reasoning on Knowledge Graphs

In section 2.6 we discussed rule-based reasoning. For practical purposes, we divide our rule set into three parts: existential triples to predicates, predicates to predicates, and predicates to intentional triples. In the first part, each triple in the original knowledge graph is mapped to a certain rule predicate and their head and tail entities become the corresponding variables. An example would be:

`RP0(h,t) := TE(h,r,t)`

In which case the relation "r" would be mapped to predicate RP0. The actual reasoning would be carried out in the second step, where abstract rule predicates would infer other predicates. The third step is simply a reversal of step 1, where the same predicate as in the example would be converted to:

`TI(h,r,t) := RP0(h,t)`

The intentional triple generated by this step would then form the new knowledge graph with additional information from the reasoner.

All predicate-to-predicate rules should follow the format:

`RPa(`$x_1, \ldots x_m$`) := RPb(`$y_1, \ldots y_n$`)`

where “RPa” and “RPb” can be any arbitrary predicate names and x, y are variables.

We also define the mathematical notations for graph materialization, let the function $m\left(G, R\right)$ be the materialization process, where G is a knowledge graph, and R is a set of rules (and R should also be a rule set that is tailored to reason on G, meaning at least some of the constant in the rules should match the relation types in G). We also define the rules that will be actually triggered in the rule set on G to be R_G. The outcome of $m\left(G, R\right)$ should also be a KG with different relation connections between the entities provided that $R_G \neq \varnothing$.

3.4.2 Random Predicate Name Swapping (RPNS)

The purpose of RPNS is to create a rule set which can be used by a reasoning system to generate the lies graphs, as defined in Section 3.2. Define the facts KG as G_F, and the lies graph as G_L. We also define $R = \{r_1, r_2, \ldots r_n\}$ as a rule set, where r is a rule and the materialization of graph G_F with the rule set R is $\mathrm{m}\left(G_F, R\right)$ and thus $G_F \subseteq \mathrm{m}\left(G_F, R\right)$ always holds true.

In this proposed method of applying false rules, a truthful rule set must be obtained from the knowledge graph source first. We assume all triples from the materialization with such rules are also facts. Here, we use $L = \mathrm{m}\left(G, R_F\right) - \mathrm{m}\left(G, R_T\right)$ as the set of lies, where R_T is the set of correct rules, the idea is to ensure the generated lies does not happen to be a fact by coincidence. However, the materialization with correct rules does not add to the positive samples. Instead, the correct materialization is used only to filter negative samples to prevent false negatives.

We create a number of new rules by swapping some of the predicate names in the original ruleset. The rule predicates that can be swapped must have the same arity number. We define predicate name swapping ratio ρ_{RPNS}. First, the rule predicates are separated into groups with the same arity which we define as P_i with i being the arity number, then for each predicate group P_i, a number of $2\left\lfloor \frac{\rho_{RPNS}|P_i|}{2} \right\rceil$ predicates are selected (we round up the ratio times the number of predicates in this group to its closest even number), afterwards, all selected predicates are subdivided into pairs and the predicate name of each pair will be swapped.

Note that in the RPNS process, all rules are still in place, only the names of predicates are changed, e.g., if we have the following rules:

`RP1(A,B) := RP0(A)`

`RP2(A,B) := RP3(A,B,C)`

and the predicates “RP1” and “RP2” are swapped, the resulting rules would be: `RP2(A,B) := RP0(A)`

`RP1(A,B) := RP3(A,B,C)`
the rest of the rules will still be in place.

The name swapping will also affect all rules that contain the swapped predicates in the rule set, so in the given example above, all predicate “RP1” in other parts of the rule set will appear as “RP2” and vise versa.

In the following sections, the detailed negative sampling method in contrastive learning will be discussed. Note that these methods are not exclusive to each other and can be combined.

3.5 Negative Sampling

In this section, we will describe three proposed negative sampling methods in detail. Each negative sampling method is independent of the overall training process, (some other negative sampling methods in Section 3.5, such as self-adversarial negative sampling, require their specialized infrastructure, however, this is not the case for our methods) the process takes in one positive sample and produces a number of negative samples regardless of the state of the embedding model or the training parameters.

Thus, the top-level negative sampling process can be described as follows:

Algorithm 2: negative sampling base framework

```
   Input: positive triple $(h, r, t)$, negative sample size $n$, sample head or
          tail $sample\_head$ or $sample\_tail$, facts graph $G_F$, lies graph $G_L$
   Output: negative triples set $\overline{S}$ = containing $n$ triples
 1 initialize empty negative sample set $\overline{S}$ of size $n$
 2 number of sampled negatives $i = 0$
 3 while $|\overline{S}| < n$ do
 4     if $sample\_head$ then
 5         get corrupted head $\overline{h} = \text{NS}((h,r,t), G_L, sample\_head, ...)$
           /* The function NS is the particular negative sampling
              process will be explained in the sub-sections    */
 6         negative triple $\overline{s} = (\overline{h}, r, t)$
 7     else
 8         get corrupted tail $\overline{t} = \text{NS}((h,r,t), G_L, sample\_tail, ...)$
 9         negative triple $\overline{s} = (h, r, \overline{t})$
10     if $\overline{s} \notin G_F$ then
11         add $\overline{s}$ to $\overline{S}$
```

The NS function represents the procedure to obtain a single corrupted triple with a particular method, in this work the methods include pseudo-type on lies, SANSoL, and SANSoLF. This procedure is then repeated until the size of the negative samples set reaches the defined value, any generated negative samples that are already in the facts graph as positive edges and will not be added. Whether the head or tail entity will be corrupted is already decided in the training loop, thus it is represented by a flag value in the algorithms, and the NS function will only return a single entity value for either the head or the tail.

3.5.1 Pseudo Type Negative Sampling on Lies

The original pseudo-type negative sampling was implemented by Pykeen, a pseudo-type negative sampler takes into account which entities co-occur with which relation. Our proposed pseudo-type negative sampling on lies uses the same underlying idea and samples from the lies graph.

Before negative sampling is conducted in the training, we construct sets of relation to entity mappings on the lies graph, two such mappings, the relation head mapping M_h and the relation tail mapping M_t will be created, given a graph G, $M_h = \{(r, h) \mid (h, r, t) \in G\}$, and $M_t = \{(r, t) \mid (h, r, t) \in G\}$.

During the training, we simply sample a negative head or tail by selecting a different entity in the mapping that does not also appear in the facts graph. The following pseudo-code describes this process:

Algorithm 3: Pseudo Type Negative Sampling on Lies (1 Triple)

```
1 Def NS($(h, r, t)$, $G_L$, $sample\_head$, $sample\_tail$, $M_{h,R\rightarrow E}$, $M_{t,R\rightarrow E}$):
2     if $sample\_head$ then
3         $\overline{h} = uniform\_sample\left(M_h\left[r\right]\right)$
4         return $\overline{h}$
5     if $sample\_tail$ then
6         $\overline{t} = uniform\_sample\left(M_t\left[r\right]\right)$
7         return $\overline{t}$
```

.

3.5.2 Structure-Aware Negative Sampling on Lies (SANSoL)

We base our method on SANS which samples uniformly from a designated head or tail entity's k-hop neighborhood. Our new idea is to find the entities that belong to the k-hop neighborhood of the same entity on lies, we would like to examine if such negative samples have better effect than SANS.

We aim to incorporate the lies graph into the KGE training similar to SANS by finding the neighborhood for each entity in the KG.

k-hop neighborhood on lies A k-hop neighborhood is constructed given the hyper-parameter k (number of hops). We an simply use the adjacency matrix of the lies graph to construct the neighborhood on lies:

$$\mathbf{K}_L = \mathbf{S}^{+}\left(\mathbf{A}_L{}^{k} + \mathbf{A}_L{}^{k-1}\right)$$

where $\mathbf{A}_L$ is the lies graph's adjacency matrix, $\mathbf{K}_L$ is the lies graph's k-hop neighborhood adjacency matrix, S^{+} is the element-wise sign function, which turns any value larger than 1 into 1, we can refer this formula to Section 2.5.4.

We use the sum of 2 different powers because ${\mathbf{A}_L}^k$ has its limitations, a node can be within this k radius but inaccessible by such calculation. The power operation calculates every node that can find a path with exact length k to the source node, depending on the graph structure, this calculation does not always cover the radius if a structure such as an even-length loop or chain is present, only alternating nodes will be included. Take a chain $a \to b \to c$ for example, b is in the 2-hop neighborhood of a, but we can never reach b from a with 2 hops, if we add $k-1$ hops, then the calculation does include every node in the radius.

Random Walk k-hop approximation We can also approximate the k-hop neighborhood by using the Random Walk algorithm [27]. Apart from k, define the random walk count ω, we assign the result of the random walk algorithm from a particular entity $E[i]$ to the ith position of the k-hop neighborhood matrix,

$$\mathbf{K}_L[i] = random\_walk(k, \omega, E[i])$$

Due to limitations in computation time, the random walk count value here (as well as in SANSoLF) is always set to 100.

Note that the "adjacency matrix" $\mathbf{A}$ in this work comes from the undirected KG ($\mathbf{A} = \mathbf{A}^T$), any relation type is considered an edge in this transformation, if $(h, r, t) \in G_F, E[i] = h, E[j] = t$, then $\mathbf{A}_{i,j} = \mathbf{A}_{j,i} = 1$.

Algorithm 4: Structure Aware Negative Sampling on Lies (1 Triple)

```
1  Def NS($(h,r,t), G_L, sample\_head, sample\_tail, \mathbf{K}_L, E$):
2      if $sample\_head$ then
3          if $|\mathbf{K}_L[h]| > 0$ then
4              $\overline{h} = uniform\_sample(\mathbf{K}_L[h])$
5          else
6              $\overline{h} = uniform\_sample(E)$
7          return $\overline{h}$
8      if $sample\_tail$ then
9          if $|\mathbf{K}_L[t]| > 0$ then
10             $\overline{h} = uniform\_sample(\mathbf{K}_L[t])$
11         else
12             $\overline{h} = uniform\_sample(E)$
13         return $\overline{t}$
14     return 0;
```

We can then use the k-hop neighborhood adjacency matrix to provide negative samples for the training. Given a dedicated entity to corrupt, the corrupted entity is uniformly sampled from its k-hop neighbors, or uniformly sampled from the entire entity set if the k-hop neighborhood is empty.

To conserve resources during the embedding training, the calculation of k-hop neighborhoods is carried out beforehand as a pre-processing step.

3.5.3 Structure Aware Negative Sampling on Lies and Facts (SANSoLF)

We propose a method similar to SANSoL, the basic idea is to combine the structural information of both the facts and lies graph by calculating two k-hop adjacency matrices.

The two neighborhoods are not combined equally, when given an entity to corrupt, the neighborhood on lies is prioritized, and the neighborhood on facts is only used when the lies neighborhood is empty. Since we made the assumption that the lies triples are incorrect under the sLCWA. Therefore the neighborhood on the lies graph may contain better quality negative samples than the neighborhood on the facts graph, however, the neighborhood on facts is still useful, compared to pure uniform sampling with structural information, as proven by SANS.

3.6 Evaluation Metrics

MRR MRR (mean reciprocal rank, also known as "inverse harmonic mean rank") [28] is an evaluation method that uses the ranks of possible outcomes. Let i be the index of an outcome in the prediction (starting with the highest likelihood) and $rank_i$ be the correct rank of the prediction, then we calculate MRR with:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i}$$

where Q is the set of predictions in the evaluation.

There are other types of metrics for evaluating KGE performance, such as Hits@k, however, only MRR is used for results comparison in this work.

4 Experimental Setup

4.1 Pre-Processing

Data Sets Split and Reduction To reduce the size of the original Claros dataset, we carry out 3-core reduction on the graph, so all remaining nodes have a degree of at least 3.

The reduced graph is then split into train - test - validation sets of 0.8, 0.1, 0.1 desired ratio, however in graph dataset separation the desired ratios cannot be guaranteed, we need to ensure that all entities and relation types in the test and validation sets must be present in the train set. Let $E_{train}, E_{test}, E_{val}$ be the sets of entities, $R_{train}, R_{test}, R_{val}$ be the sets of relations and $T_{train}, T_{test}, T_{val}$ be the sets of triples respectively, we must satisfy $E_{test} \subseteq E_{train}, E_{val} \subseteq E_{train}, R_{test} \subseteq R_{train}, R_{val} \subseteq R_{train}$ while approximate $|T_{train}| = 0.8\left(|T_{train}| + |T_{test}| + |T_{val}|\right)$ as much as possible.

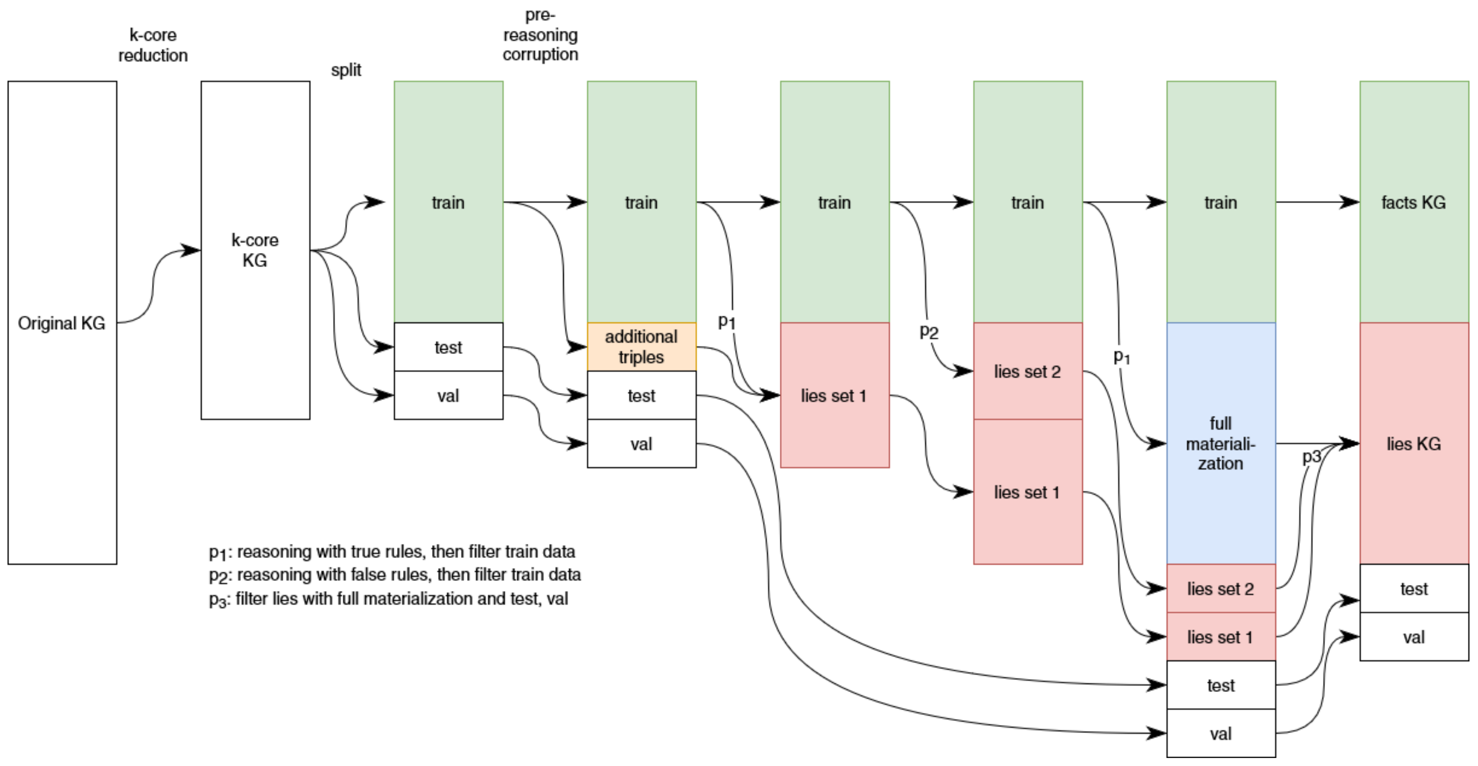

Figure 2: Data pre-processing pipeline. The image shows both the RPNS and the pre-reasoning corruption as part of the pipeline, we used a combination of either or both of these processes in our experiments.

Self-Loops Removal This removal is based on the rationale that a triple with the same head and tail would result in the relation embedding vector approaching 0 for some of the translation-based embedding models. Consider the loss in TransE if the head and tail are the same entity,

$$\begin{aligned}\ell &= \sum_{(h,r,h)\in S}\sum_{(h',r,t')\in S'_{(h,r,h)}} \left[\gamma + d\left(\mathbf{h}+\mathbf{r},\mathbf{h}\right) - d\left(\mathbf{h}'+\mathbf{r},\mathbf{t}'\right)\right]_+ \\ &= \sum_{(h,r,h)\in S}\sum_{(h',r,t')\in S'_{(h,r,h)}} \left[\gamma + \|\mathbf{r}\|_2^2 - d\left(\mathbf{h}'+\mathbf{r},\mathbf{t}'\right)\right]_+ ,\end{aligned}$$

this means minimizing the loss would also let $\|\mathbf{r}\|_2^2 \to 0$. If the self-loops occur for more than 1 relation (which is true for Claros KG in our experiments), it would cause undesired effects as multiple relation embedding vectors' length would converge to 0. To avoid these effects, we could either remove all self-loops from the embedding training process or use a model that could handle self-loops, in this work we opted for the former.

4.2 RPNS-Rates Sweeping

4.2.1 Generating Lies Graphs with Rules from RPNS

We aim to create multiple lies graphs from the same facts graph by using different rules, then study the differences these rule sets will result in. As described in Section 3.4.2, we will use different RPNS rates, and with each RPNS rate, multiple rule sets will be generated randomly. Lies graphs are then materialized with these rule sets and be used for KGE training.

When using corrupted rule sets to materialize the facts graph into lies, since the swapped rule predicates are no longer being used as intended, successful materialization in a finite time cannot be guaranteed. In our process of repeated random RPNS, a time limit is defined for the materialization, and the materialization will restart with different rules from RPNS until materialization can succeed in said time. We still cannot rule out the possibility that the materialization does not succeed after many attempts, in such case we will omit this particular RPNS rate. We will discuss whether any RPNS rates are omitted in the result section.

As mentioned before, we also generate the lies graphs using pre-reasoning corruption, this is also combined into the experiment at this step. A particular RPNS rule set can be used on the facts graph or the corrupted graph. To simplify the experiments, the pre-reasoning corruption is carried out with the same hyper-parameters for all RPNS rates and repetitions. Additionally, 1 lies graph with pre-reasoning corruption is materialized with true rules (without RPNS).

4.2.2 RPNS-Rate Sweeping Experiments

After the lies graphs are generated as above, the KGE model is trained on each lies graph and evaluated. The goal of this step is to compare the influence of the RPNS rates and the overall RPNS method over small scale experiments. The following negative sampling methods are used: SANSoL, SANSoLF, pseudo-type negative sampling on lies. With SANSoL and SANSoLF, different numbers of k-hop neighborhoods are also swept, as well as whether Random Walk is used to approximate the neighborhoods.

settings name	values
RPNS rates	{15%, 20%, 32%, 40%, 50%, 60%, 70%, 80%, 90%, 100%}
number of random rule sets for each RPNS rate	5
k-hops (for SANSoL and SANSoLF)	[2, 8]
use Random Walk (for SANSoL and SANSoLF	{True, False}
use pre-reasoning corruption	{True, False}

Table 1: RPNS-rate sweeping settings

The total number of desired experiments in this step will be:
$num\_RPNS\_rates \times num\_rule\_sets\_per\_RPNS\_rate$
$\times 2\,(num\_k\_hops\_per\_method \times 2 + 1) = 1500.$

The actual number of experiments also depends on the termination times of the reasoning algorithms, the product $num\_RPNS\_rates \times num\_rule\_sets\_per\_RPNS\_rate$ might have a smaller value than the calculation. The KGE training experiments have the same pre-defined hyper-parameters, since tuning the hyper-parameters for each of the lies graphs is too expensive, we assume that differences can be observed from these lies graphs with the same training settings. We use TransE model and the following hyper-parameters:

hyper-parameter	value
training length (steps)	2656
batch size	1024
negative samples size	256
embedding dimensions	1000
learning rate	0.0005
training steps	400
gamma	24

Table 2: RPNS-rate sweeping training hyper-parameters preset

We aim to answer the following additional questions after this part of the experiment, relating to the research question 1:

a. Is there a correlation between the RPNS rate and the KGE performance for each of the novel negative sampling methods?

b. Is the KGE performance from a particular rule set consistent across different negative sampling methods? Or does each negative sampling method has different lies graphs that suit it the best?

c. Does the number of k-hops (in SANSoL and SANSoLF) on the lies graphs affect KGE performance independently from the rule sets? Or does the lies graph from a particular rule set has its best-performing number of k-hops?

d. What is the influence of pre-reasoning corruption on the KGE? Does the pre-reasoning corruption affect the KGE performance independently from the RPNS rule sets?

The different rule sets are named as "RPNS-[RPNS rate (%)]-[RPNS ID]-[C (for pre-reasoning corruption]", e.g. the first rule set generated with RPNS rate 50%, without pre-reasoning corruption, would be named "RPNS-50-0", and the third rule set with RPNS rate with RPNS rate 80% with pre-reasoning corruption would be named "RPNS-80-2-C", additionally, one lies graph has been created with pre-reasoning corruption but without RPNS rules, this graph is named "C". This naming convention will be used for displaying and comparing results in later sections.

4.3 Training Setup

The training loop is divided into steps. Each step uses all the triples on the facts graph to train the model. We use the bidirectional training iterator (as originally implemented in SANS) to load the data in batches. The bidirectional training iterator alternates between head batches and tail batches, which will corrupt all the positive samples in the batch on either head or tail.

Two different learning rates can be optionally used as in simulated annealing, for simplicity, we divide the total number of steps equally with initial and final learning rates, the final learning rate is by default 0.1 times the initial learning rate.

4.4 Hyper-Parameter Tuning

After selecting some of the interesting RPNS lies graphs, we will carry out the next experiments. For each selected lies graph, hyper-parameter optimization will be carried out for every NS method, based on the MRR score on the validation set. We will also optimize hyper-parameters for existing NS methods (uniform NS, SANS, pseudo-type NS), but these methods are only used on the facts graph. After optimizing the hyper-parameters, long training will be performed with the best hyper-parameters, such experiments will be repeated a number of times and the performance will be recorded on the test set.

4.5 Implementation

The materialization of knowledge graphs uses the reasoning engine VLog [29], we chose this rule engine for its efficiency. The implementation of the KGE models and negative sampling methods are based on PyTorch, and are forked from the original SANS implementation [1].

5 Results

5.1 RPNS Rates Sweeping Results

Materialization with RPNS rules 191 lies graphs without pre-reasoning corruption and 144 lies graphs with pre-reasoning corruption were materialized. We observed that some of the rule sets with higher RPNS rates such as 100% cannot finish materializing on the pre-reasoning corrupted graphs, as a result, 100% RPNS rate will be excluded from the following RPNS rates sweeping experiments. All KGE training uses TransE model, and parameters are learned using Adam.

Considering the training time of the KGE, which is around 0.5 seconds per step, sweeping on every aforementioned lies graph is too costly, thus some of the lies graphs are skipped. In total, we experimented on 10 RPNS rates and 5 rule sets for each RPNS rate.

Observations from RPNS rates sweeping The absolute highest MRR score from the early RPNS rates sweeping is 0.383667, by rule set RPNS-20-2 and with negative sampling method RW-SANSoL.

To answer the questions from the experimental setup, we collected the MRR for each RPNS rule set and each NS method after the short training. The results under the same NS method, the same number of k-hops (but using lies graphs from different RPNS rule sets) are ranked. We observed that for methods without using random walk (either with or without pre-reasoning corruption), there is no significant consistency in performance with each rule set, meaning that which rule set results in the best or worst performance depends on which NS method is used and what is the number of k-hops.

On the other hand, we did observe strong correlations regarding the same question, under the NS methods with Random Walk, this includes RW-SANSoL and RW-SANSoLF. In other words, different rule sets have consistent relative performance under these methods.

Table 3 shows the comparison described above. Each region of the table shows the results from a particular method and k-hop, the ranking of the results under such a method is color-coded (rankings are within the same method, not across different methods). We can observe the relative performance of the different rule sets.

We can also observe that the lower RPNS rates perform better, to further demonstrate such finding, we can use the average performance of each RPNS rule set across all experimented methods, as in Table 4 (the color-coding is also based on ranking):

[1] Can be accessed at `https://github.com/kahrabian/SANS`

RPNS-rate	15%	20%	32%	40%	50%	60%	70%	80%	90%	100%
RPNS-ID	**RW-SANSoL k=2**									
0	0.356671	0.294691	0.315896	0.34236	0.24301	0.241821	0.259089	0.271688	0.190301	0.179872
1	0.358094	0.327963	0.366449	0.251615	0.267186	0.304125	0.202831	0.232039	0.233764	0.204824
2	0.376405	0.383666	0.296441	0.373426	0.377127	0.319912	0.362084	0.242217	0.24234	0.203348
3	0.33461	0.361644	0.253536	0.224199	0.312107	0.230911	0.236156	0.286871	0.272591	0.193958
4	0.366139	0.367601	0.295184	0.367874	0.290434	0.269054	0.283065	0.226121	0.248237	0.198533
	RW-SANSoL k=3									
0	0.356008	0.306125	0.286505	0.333978	0.164754	0.131267	0.182478	0.167242	0.128188	0.071991
1	0.373185	0.330883	0.326624	0.176389	0.17639	0.267894	0.132872	0.120144	0.119324	0.118087
2	0.377574	0.359138	0.217803	0.34086	0.345485	0.273149	0.353754	0.160241	0.15652	0.111715
3	0.318952	0.364991	0.182707	0.118457	0.197033	0.166428	0.17076	0.166802	0.167727	0.110997
4	0.366575	0.360692	0.22835	0.337724	0.181745	0.271696	0.187692	0.163193	0.12924	0.121114
	RW-SANSoL k=4									
0	0.356187	0.298733	0.31525	0.336113	0.249399	0.25135	0.258125	0.266189	0.190305	0.175525
1	0.369027	0.330244	0.364062	0.251942	0.277176	0.30399	0.195672	0.240333	0.224598	0.202364
2	0.377559	0.375584	0.298102	0.376274	0.373743	0.307296	0.3674	0.244635	0.231557	0.209178
3	0.324505	0.377132	0.253609	0.22679	0.302241	0.241404	0.23487	0.266259	0.272276	0.207113
4	0.364309	0.365972	0.290591	0.36503	0.298197	0.266383	0.284275	0.221073	0.236474	0.190251
	RW-SANSoLF k=2									
0	0.361485	0.296583	0.316935	0.342468	0.242655	0.244188	0.248522	0.26734	0.193821	0.170089
1	0.357372	0.327365	0.359527	0.25222	0.276301	0.308388	0.203498	0.225751	0.231579	0.203862
2	0.377813	0.377062	0.295184	0.377242	0.369279	0.320317	0.361898	0.243516	0.236633	0.202165
3	0.331355	0.352427	0.255188	0.228195	0.319104	0.23272	0.232507	0.282959	0.267617	0.18983
4	0.368292	0.366657	0.306683	0.368443	0.298687	0.272431	0.279471	0.220328	0.25167	0.200512

Table 3: RPNS rates relative comparison under selected methods

	Rule sets average, no pre-reasoning corruption									
RPNS ID/rate	15%	20%	32%	40%	50%	60%	70%	80%	90%	100%
0	0.189506	0.165171	0.132541	0.157694	0.115891	0.126903	0.120827	0.12456	0.100619	0.090955
1	0.202852	0.154965	0.153516	0.134814	0.152943	0.138323	0.095943	0.105501	0.120855	0.101839
2	0.187542	0.150979	0.146703	0.162605	0.151216	0.146609	0.148044	0.131068	0.112373	0.111428
3	0.165899	0.193768	0.166906	0.106342	0.141506	0.125122	0.115052	0.128379	0.135404	0.10653
4	0.203226	0.157565	0.138075	0.161005	0.142747	0.157429	0.149583	0.125619	0.123294	0.102046
mean	0.189805	0.16449	0.147548	0.144492	0.14086	0.138877	0.12589	0.123025	0.118509	0.102559

Table 4: RPNS rates average results without pre-reasoning corruption

We can further compare the influence of different numbers of k-hops on the result. According to table 5, the best number for k is 2, we see a decrease in model performance as the number of k-hops increases, this can be observed for both the rule sets with or without pre-reasoning corruption.

Comparing Table 4 and Table 6, limited similarities can be observed. Apart from the effects of different RPNS-rates as discussed before, we can also observe the correlations between the results shown in the two tables, for 70% RPNS rates, for example, rule sets RPNS-70-4 and RPNS-70-4-C both perform well. However, such similarity is limited and does not apply to all different RPNS rule sets in this part of the experiment.

The complete result tables can be found in the appendix.

5.2 Negative Sampling Methods Results

We complete the final experiments with longer training times on a few selected lies graphs (the exact lies graph may differ, depending on the NS method), as well as several existing methods for comparison. The training is repeated for every setting. We use 10000 steps for all of the experiments in this section.

The following existing negative sampling methods are studied: uniform,

RPNS-rate	15%	20%	32%	40%	50%	60%	70%	80%	90%	100%	mean
k-hop						**all methods**					
2	0.249803	0.234044	0.221253	0.216331	0.20307	0.202528	0.191369	0.192142	0.190593	0.167286	0.206842
3	0.199061	0.189826	0.129197	0.135967	0.148989	0.145052	0.137959	0.11729	0.110724	0.072781	0.138685
4	0.190992	0.166002	0.147238	0.129215	0.130066	0.142358	0.13283	0.124227	0.128903	0.136526	0.142836
5	0.199668	0.165121	0.134274	0.13827	0.118779	0.111247	0.104921	0.100422	0.106248	0.07204	0.125099
6	0.169515	0.0769	0.14718	0.161391	0.139378	0.152739	0.126365	0.12402	0.11182	0.107648	0.131695
7	0.142421	0.133581	0.112964	0.10039	0.087058	0.0832	0.095253	0.09647	0.082878	0.037119	0.097134
8	0.158449	0.149377	0.129291	0.114456	0.14802	0.11445	0.08256	0.099864	0.079847	0.090333	0.116665
mean	0.18713	0.159264	0.145914	0.142289	0.139337	0.135939	0.124465	0.122062	0.115859	0.097676	
						no pre-reasoning corruption					
2	0.24867	0.242585	0.222943	0.224059	0.218028	0.207586	0.204705	0.193279	0.189129	0.167286	0.211827
3	0.198239	0.192104	0.142975	0.15032	0.133751	0.16809	0.159789	0.13541	0.126479	0.072781	0.147994
4	0.197123	0.193045	0.170735	0.175077	0.169154	0.156212	0.151747	0.15072	0.157862	0.136526	0.16582
5	0.198912	0.19105	0.142536	0.147094	0.127226	0.12932	0.120631	0.094842	0.090031	0.07204	0.131368
6	0.184195	0.036209	0.170767	0.175044	0.167678	0.158842	0.152974	0.1423	0.134216	0.107648	0.142987
7	0.141072	0.151558	0.068806	0.106488	0.077308	0.05938	0.099487	0.088632	0.068388	0.037119	0.089824
8	0.146453	0.140945	0.123365	0.143816	0.137618	0.110225	0.101799	0.118215	0.088528	0.090333	0.12013
mean	0.187809	0.163928	0.148875	0.160271	0.147252	0.141379	0.14159	0.131914	0.122091	0.097676	
						with pre-reasoning corruption					
2	0.255472	0.223996	0.219375	0.206027	0.180057	0.196205	0.170852	0.190719	0.192315		0.203891
3	0.203176	0.187428	0.109514	0.115462	0.164227	0.116254	0.110672	0.091405	0.086486		0.131625
4	0.150116	0.127368	0.113672	0.037491	0.058997	0.123886	0.105806	0.080073	0.080637		0.097561
5	0.203446	0.128078	0.123257	0.126506	0.105784	0.083443	0.080753	0.107862	0.124266		0.120377
6	0.037398	0.108202	0.110893	0.144324	0.095838	0.145958	0.082015	0.095897	0.074492		0.099446
7	0.14917	0.111456	0.144135	0.093354	0.097558	0.100668	0.089716	0.105353	0.097367		0.109864
8	0.203435	0.157282	0.135217	0.077757	0.157266	0.118393	0.056326	0.075395	0.070443		0.116835
mean	0.171745	0.149116	0.136581	0.114417	0.122818	0.126401	0.099449	0.106672	0.103715		

Table 5: RPNS rates and number of k-hops

RPNS ID/rate	15%	20%	32%	40%	50%	60%	70%	80%	90%	100%
0	0.184782	0.169578	0.102344	0.101675	0.103898	0.1344	0.099003	0.100731	0.084439	
1	0.139314	0.11556	0.149011	0.166997	0.115396	0.050736	0.086678	0.124806		
2	0.10176	0.135827	0.108082	0.10169	0.113642	0.077488	0.127483	0.084795		
3	0.197459	0.201232	0.104137	0.135217	0.126206	0.093311	0.101314	0.122965		
4	0.138346	0.121542	0.131125	0.12717	0.149862	0.153704	0.126962	0.1183		

Table 6: RPNS rates average results with pre-reasoning corruption

SANS, pseudo-type. According to the previous section, 2-hop neighborhood works the best for any type of structural negative sampling, as a result, 2-hop sampling is set as the default value. The MRR score for uniform sampling and SANS can thus be used as the baseline.

We can observe significant differences across lies graphs. In Table 7 the repeated experiment results are shown, as well as their average. The repeated experiments (MRR#1 to MRR#5) are color-coded among each other and the MRR mean is color-coded on its own. Rule set RPNS-100-0 is the worse performing lies graph in RPNS sweeping, however, the result shows that with the correct method, such as RW-SANSoL, the performance exceeded the baseline (SANS, uniform). This implies that higher-performing lies graphs might exist outside the coverage of our experiments.

Pseudo-type negative sampling on lies did not achieve results comparable to the baseline in any of the experiments.

NS method	lies graph	k-hop	MRR#1	MRR#2	MRR#3	MRR#4	MRR#5	MRR mean
uniform			0.395732	0.404777	0.398518	0.395327	0.394267	0.3977242
SANS		2	0.411702	0.415189	0.413428	0.409151	0.407998	0.4114936
RW-SANS		2	0.41967	0.413173	0.424539	0.421396	0.410123	0.4177802
SANSoL	RPNS-15-2	2	0.416409	0.413079	0.401183	0.409667	0.410151	0.4100978
SANSoL	RPNS-20-3	2	0.417997	0.418447	0.411799	0.41189	0.406344	0.4132954
SANSoL	RPNS-70-2	2	0.419982	0.413698	0.409434	0.408386	0.409886	0.4122772
SANSoL	RPNS-70-2-C	2	0.408696	0.410809	0.403881	0.411122	0.410028	0.4089072
SANSoL	C	2	0.416907	0.408245	0.407472	0.41604	0.408245	0.4113818
SANSoLF	RPNS-20-2	2	0.43182	0.417117	0.409189	0.416732	0.416732	0.418318
SANSoLF	RPNS-70-2-C	2	0.41226	0.408973	0.425401	0.399966	0.417215	0.412763
RW-SANSoL	RPNS-20-3	2	0.404982	0.402831	0.403257	0.403707	0.405425	0.4040404
RW-SANSoL	RPNS-100-0	2	0.412629	0.420469	0.416307	0.415392	0.423031	0.4175656
RW-SANSoLF	RPNS-20-2	2	0.418865	0.420335	0.417855	0.4196	0.41943	0.419217
RW-SANSoLF	RPNS-100-0	2	0.421023	0.414389	0.414356	0.420725	0.417002	0.417499
RW-SANSoLF	RPNS-100-0	7	0.331273	0.301573	0.296138	0.269754	0.271174	0.2939824
pseudo lies	RPNS-60-1		0.169169	0.166981	0.169169	0.155839	0.200523	0.1723362
pseudo lies	RPNS-90-0		0.238904	0.243181	0.246777	0.242162	0.240969	0.2423986

Table 7: Final experiments results

6 Discussion

6.1 Critical Evaluation

RPNS rates sweeping results To be able to complete the large amount of data required in this experiment, the training only has 400 steps, this turns out to be insufficient for completing the KGE training. We can come to this conclusion since the training loss and validation MRR is still changing near the end of the 400 steps, the model still has the potential to improve. This is also evident when we compare the results to the follow-up experiments, some of the lies graphs perform significantly better with longer training, and when comparing different NS methods, such difference become more prominent. Therefore is it not advisable to discuss the differences between the methods using such results.

Our main goal at this point is to find out if the differences between the RPNS rates and other factors, such as the number of k-hops, can be summarized. This goal is partially achieved with the RPNS rates sweeping, but a more thorough round of experiments with longer training is needed to get a better overview of the RPNS outcome.

We can also conclude from the results that the RPNS procedure introduced in this work cannot be generalized, since we cannot predict if the lies graph resulting from a set of RPNS rules can train the KGE effectively. We only selected some of the promising lies graphs using the validation result, without such hindsight, we can assume that any improvement from SANS or even uniform sampling, can hardly be achieved.

Final results analysis Among the 3 baselines, SANS with Random Walk has the highest MRR score, followed by SANS, then uniform sampling, this result is consistent with Ahrabian et al. [24].

Some of the refined settings with our proposed methods did increase the

KGE performance but with little statistical significance. To confirm this finding, t-tests were performed on the results, we tested the MRR scores from the RW-SANS experiments with some of the best results from our method. The result from RW-SANSoLF with lies RPNS-20-2 (0.4192) is higher than the baseline (0.4178), but the t-test suggests that these values are not different ($p = 0.3268$). We can regard some of the best results as performing equally well to RW-SANS. However such performance only occurred under special settings, such as with manually selected lies-graphs. Some of the other settings performed worse than using uniform sampling, or between uniform and RW-SANS. There is very little consistency when using RPNS to generate lies for negative sampling.

The situation surrounding pre-reasoning corruption is similar: with pre-reasoning corruption, we also need to have the right RPNS rule sets to achieve desirable performance, with the variations in the outcome, it is not yet clear what exactly is the effect of pre-reasoning corruption.

6.2 Limitations

KG reduction The knowledge graph was simplified using 3-core reduction, the initial aim of this procedure is to significantly reduce the size of the KG while maintaining the most connected nodes. However, this reduction step had complicated effects, both with overall KGE performance and with our analysis in this work.

Performance-wise, some of the important information is lost during the reduction, both for the training and evaluation. We suspect that there is a correlation between the degrees of a certain entity and the most common types of relations occurring in such connections (e.g., certain relation types may only occur once for an entity, take the relation "mother/father of" as an example), reducing only the lower degrees nodes might overwhelmingly decrease the KGE's chance to encounter such relations. Additionally, the k-core reduction procedure we chose did not remove self-loops, the proportion of self-looping triples in the training set was significantly higher than the original graph, and thus further removal had to be performed, further reducing the information for the training.

We cannot be sure whether the k-core reduction decreased the KGE performance, but intuitively we believe there are unforeseen impacts.

Analysis-wise, the reduction changed the structure of the graph. With lower-degree nodes missing, the overall KG is much more inter-connected. Compared to the KG in previous studies, the neighborhoods in our KG is much more simple, thus higher k-hop numbers might increase the neighborhood to the whole graph (essentially turning the NS methods into uniform sampling), this explains why we see a decrease in model performance as the k-hop number becomes higher. This renders the analysis on the correlation between k-hops and RPNS rates impossible.

The flaws of RPNS RPNS is essential to generate negative samples in this work, however, this is still a gap between the achievements of RPNS and the goal we set at the beginning. We were looking for a way to come up with negative samples that are logical and realistic, but the resulting RPNS method on the other hand, even though it does incorporate logical information by using reasoning rules, is by no means generating realistic samples. The swapping process in RPNS is still stochastic, meaning any rule has a chance to be replaced, regardless of how important it is.

Hyper-parameters search The hyper-parameter search should ideally be carried out for each lies graph, but considering the training time (e.g. 1.17 seconds per step for SANSoL), a full hyper-parameter sweep is too costly. Furthermore, it is hard to predict the steps needed to reach the global minimum, lengthening the time needed for the search even more. As a result, only a limited number of the hyper-parameter search was completed, and not all of the potential hyper-parameters are covered. Some of the tuned hyper-parameters are shared across different experiments, and this is not the ideal approach to obtain the final result. The evaluation metrics on which we optimize the hyper-parameters also do not have to be limited to MRR, we can add Hits@k into the evaluation as well.

Negative sampling methods In this work, we have demonstrated that we can train a KGE using negative samples coming from the lies graph, and these negative samples sometimes increase the KGE performance. SANS is still the state-of-the-art NS method, so from our perspective, the structural information on the facts graph is still useful. In our methods SANSoL and SANSoLF, the facts graph is either not used or only used when the lies neighborhood is empty. This design does not take advantage of the SANS approach.

Shortcomings of SANS-based methods Structure-aware negative sampling and its modified versions in this work are proven to provide hard negative samples compared to methods such as uniform sampling, but with certain graph structures, they might not work as intended. As seen in the KG used for our experiments, there are a few highly connected nodes, that are related to most of the other nodes, when such nodes exist, the k-hop neighborhood is similar for all such related nodes because they can all find their paths to each other through the "central nodes". The more nodes these "central nodes" connect to, the more the k-hop neighborhoods for other nodes approach the entire graph. This is also the reason why in the experiments above, all k-hop greater than 2 for SANSoL and SANSoLF result in significantly poorer performance.

Restrictions with KG choices The RPNS methods in this work rely on the pre-existing reasoning rules provided by the KG authors, this effectively put a limitation on what KGs can these negative sampling methods be used. Ideally, we would compare our negative sampling methods on other KGs frequently used for KGE training, such as FB15K-237 [30] or WN18 [31], however, the reasoning rules on these KGs are less accessible, and a full survey on the negative sampling methods cannot be completed. This also puts a strain on the application of our methods, they can be used exclusively on KGs with proper reasoning rules.

6.3 Future Research

Rule-based reasoning beyond RPNS Rule-based reasoning for generating incorrect information has significant potential yet to be explored by our RPNS method. First of all, RPNS can be expanded to capture different weights of each rule predicate. Swapping different rule predicates might have different results in the final KGE training, we could, in theory, summarize the importance each rule played in the model performance. This information can then be used to change the swapping chance or other factors of RPNS. We can even design a method to directly link the rules or specific predicates to the loss and calculate the gradient.

Secondly, corrupting rule sets does not have to be limited in changing the rule predicates. Other parts of the reasoning rules and the relations between the rules can also be changed. Instead of swapping predicates, we can also swap the variable names, or add negation to the rules. The structures of the rules can be changed as well, such as adding or removing existential predicates from the body of the rule. We can also design a mechanism to stochastically perform the swapping, instead of swapping every instance of a predicate in the rule set.

Lastly, the idea of manually creating rules for reasoning was abandoned in this work, since it is difficult to obtain an insight into the actual structure of the KG and what is the logic between the relations. But if such insights can be obtained, it is still useful to explore the manual creation of false reasoning rules. This might lead to more realistic negative samples and interesting discoveries.

Evolutionary algorithm This work has shown that the reasoning process, including the lies graphs materialization, can be seen as a black box. Meaning that we hardly predict the outcome of a corrupted rule set, let alone the impact of such rule sets on the KGE model. One of the best ways to optimize this black box problem is by using evolutionary computing. A particular rule set can be encoded as a gene, while processes similar to RPNS or other methods mentioned above can be used as mutation mechanisms. The individuals can thus be evaluated by their lies graph's resulting KGE performance.

7 Conclusion

In this thesis, we investigated how to use rule-based reasoning and how to use it for negative sampling for knowledge graph embedding training. We introduced two methods (random predicate name swapping and pre-reasoning corruption) to generate an additional set of triples, regarded as an extra knowledge graph, the lies graph, in parallel to the facts graph. This work has demonstrated that it is possible to train a KGE with negative samples from two graphs (in this case, the two graphs have the same set of entities, but this should not be mandatory, as long as the entities on the two graphs can be mapped), this might have further implications other than using a pair of facts and lies graph. We then introduced three negative sampling methods, SANSoL, SANSoLF, and pseudo-type negative sampling on lies, to use the lies graph for generating negative samples. Our new knowledge graph embedding training pipeline did not make significant improvements.

References

[1] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. “A comprehensive survey of graph embedding: Problems, techniques, and applications”. In: *IEEE Transactions on Knowledge and Data Engineering* 30.9 (2018), pp. 1616–1637.

[2] Aidan Hogan et al. “Knowledge graphs”. In: *arXiv preprint arXiv:2003.02320* (2020).

[3] Antoine Bordes et al. “Translating embeddings for modeling multi-relational data”. In: *Neural Information Processing Systems (NIPS)*. 2013, pp. 1–9.

[4] Antoine Bordes et al. “Learning structured embeddings of knowledge bases”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 25. 1. 2011.

[5] Antoine Bordes et al. “A semantic matching energy function for learning with multi-relational data”. In: *Machine Learning* 94.2 (2014), pp. 233–259.

[6] Guoliang Ji et al. “Knowledge graph embedding via dynamic mapping matrix”. In: *Proceedings of the 53rd annual meeting of the association for computational linguistics and the 7th international joint conference on natural language processing (volume 1: Long papers)*. 2015, pp. 687–696.

[7] Zhen Wang et al. “Knowledge graph embedding by translating on hyperplanes”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 28. 1. 2014.

[8] Yankai Lin et al. “Learning entity and relation embeddings for knowledge graph completion”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 29. 1. 2015.

[9] Ora Lassila, Ralph R Swick, et al. “Resource description framework (RDF) model and syntax specification”. In: (1998).

[10] Tim Berners-Lee. *Universal resource identifiers in WWW*. 1994.

[11] Martin Dürst and Michel Suignard. *Internationalized resource identifiers (IRIs)*. Tech. rep. RFC 3987, January, 2005.

[12] Raymond Reiter. “Towards a logical reconstruction of relational database theory”. In: *Readings in Artificial Intelligence and Databases*. Elsevier, 1989, pp. 301–327.

[13] Alvaro Cortés-Calabuig et al. “On the local closed-world assumption of data-sources”. In: *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer. 2005, pp. 145–157.

[14] Todd Millstein, Alon Halevy, and Marc Friedman. “Query containment for data integration systems”. In: *Journal of Computer and System Sciences* 66.1 (2003), pp. 20–39.

[15] Mehdi Ali et al. “Bringing light into the dark: A large-scale evaluation of knowledge graph embedding models under a unified framework”. In: *arXiv preprint arXiv:2006.13365* (2020).

[16] Xin Dong et al. “Knowledge vault: A web-scale approach to probabilistic knowledge fusion”. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2014, pp. 601–610.

[17] Mehdi Ali et al. “Bringing light into the dark: A large-scale evaluation of knowledge graph embedding models under a unified framework”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).

[18] Xiaocui Li et al. “Semi-supervised clustering with deep metric learning and graph embedding”. In: *World Wide Web* 23.2 (2020), pp. 781–798.

[19] Tomas Mikolov et al. “Distributed representations of words and phrases and their compositionality”. In: *arXiv preprint arXiv:1310.4546* (2013).

[20] Peifeng Wang, Shuangyin Li, and Rong Pan. “Incorporating gan for negative sampling in knowledge representation learning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[21] Mirza Mohtashim Alam et al. “Affinity Dependent Negative Sampling for Knowledge Graph Embeddings.” In: *DL4KG@ ESWC*. 2020.

[22] *Wikipedia Cosine Similarity*. `https://en.wikipedia.org/wiki/Cosine_similarity`. Apr. 2021.

[23] Zhiqing Sun et al. “Rotate: Knowledge graph embedding by relational rotation in complex space”. In: *arXiv preprint arXiv:1902.10197* (2019).

[24] Kian Ahrabian et al. “Structure Aware Negative Sampling in Knowledge Graphs”. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2020). DOI: `10.18653/v1/2020.emnlp-main.492`.

[25] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Vol. 8. Addison-Wesley Reading, 1995.

[26] Andreea Iana and Heiko Paulheim. “More is not always better: The negative impact of A-box materialization on RDF2vec knowledge graph embeddings”. In: *arXiv preprint arXiv:2009.00318* (2020).

[27] Leo Grady. “Random walks for image segmentation”. In: *IEEE transactions on pattern analysis and machine intelligence* 28.11 (2006), pp. 1768–1783.

[28] Ellen M Voorhees et al. “The TREC-8 question answering track report”. In: *Trec*. Vol. 99. Citeseer. 1999, pp. 77–82.

[29] David Carral et al. “Vlog: A rule engine for knowledge graphs”. In: *International Semantic Web Conference*. Springer. 2019, pp. 19–35.

[30] Kurt Bollacker et al. “Freebase: a collaboratively created graph database for structuring human knowledge”. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, pp. 1247–1250.

[31] George A Miller. “WordNet: a lexical database for English”. In: *Communications of the ACM* 38.11 (1995), pp. 39–41.

Appendix A RPNS Rates Sweeping Results

[2]

k-hop	RPNS ID/Rate	15%	20%	32%	40%	50%	60%	70%	80%	90%	100%
	0	0.146879	0.135027	0.134013	0.13447	0.151169	0.147436	0.143168	0.130961	0.154214	0.134377
	1	0.140519	0.143178	0.136083	0.129111	0.137265	0.125905	0.143811	0.140394	0.132675	0.140954
2	2	0.140022	0.142378	0.128907	0.134718	0.136434	0.145365	0.137013	0.139728	0.146262	0.141701
	3	0.136959	0.135226	0.139682	0.141831	0.122261	0.134194	0.14375	0.133616	0.140263	0.138873
	4	0.139922	0.143977	0.153305	0.130082	0.13669	0.133177	0.154759	0.142076	0.150361	0.145917
	0	0.038717	0.035584	0.038868	0.040267	0.043419	0.042574	0.036312	0.041809	0.04002	0.110437
	1	0.037498	0.041742	0.03599	0.044046	0.044432	0.038539	0.042131	0.044302	0.0378	0.131486
3	2	0.038332	0.043251	0.039924	0.038869	0.034311	0.037103	0.041499	0.038782	0.039278	0.11004
	3	0.038221	0.042409	0.036781	0.03948	0.044139	0.041569	0.036391	0.043149	0.039455	0.127273
	4	0.041113	0.036658	0.037196	0.036927	0.040184	0.039004	0.036798	0.038157	0.041041	0.037311
	0	0.035523	0.034585	0.034876	0.039612	0.034477	0.033551	0.034119	0.037148	0.038827	0.04486
	1	0.032556	0.035159	0.037099	0.03815	0.042963	0.0396	0.0368	0.034071	0.036901	0.038606
4	2	0.032496	0.037599	0.030877	0.039188	0.03756	0.039004	0.040417	0.04366	0.037312	0.038902
	3	0.033011	0.037636	0.037018	0.039682	0.037659	0.040932	0.040869	0.038805	0.034943	0.035833
	4	0.036022	0.038934	0.042978	0.047904	0.043042	0.035191	0.03919	0.034632	0.036999	0.039681
	0	0.039648	0.039226	0.034242	0.035189	0.037276	0.035236	0.038706	0.03292	0.037419	0.040392
	1	0.038878	0.039417	0.039007	0.046907	0.043574	0.039821	0.036214	0.036124	0.038319	0.043905
5	2	0.034066	0.037126	0.038401	0.032835	0.03831	0.031425	0.035554	0.038705	0.045697	0.035283
	3	0.038488	0.03579	0.034522	0.037139	0.037088	0.038867	0.034051	0.034662	0.036821	0.033964
	4	0.041987	0.034432	0.044337	0.037749	0.037162	0.03991	0.035966	0.033614	0.03914	0.033235
	0	0.031891	0.034128	0.034404	0.046861	0.037845	0.040277	0.034302	0.039541	0.037666	0.034993
	1	0.045234	0.037614	0.037178	0.038877	0.037575	0.038396	0.040087	0.037733	0.034683	0.035105
6	2	0.034116	0.035093	0.038595	0.037213	0.038025	0.036269	0.03497	0.038314	0.035111	0.035705
	3	0.034391	0.034161	0.033744	0.037358	0.035597	0.035481	0.036259	0.036023	0.034912	0.034358
	4	0.034353	0.036844	0.041664	0.037821	0.036193	0.038503	0.034418	0.039131	0.039624	0.035429
	0	0.034222	0.039695	0.039005	0.037957	0.0368	0.040814	0.035624	0.041937	0.038415	0.037616
	1	0.035607	0.035412	0.035312	0.033533	0.036279	0.035986	0.035729	0.039595	0.035895	0.034897
7	2	0.031688	0.034502	0.037637	0.038372	0.036658	0.04327	0.036228	0.040192	0.036718	0.033119
	3	0.036371	0.036235	0.035177	0.034778	0.03329	0.038338	0.034429	0.037516	0.032198	0.040438
	4	0.032559	0.039323	0.036149	0.035865	0.038618	0.037104	0.038373	0.037165	0.03437	0.036981
	0	0.037654	0.036018	0.035358	0.03457	0.040033	0.039291	0.038426	0.033896	0.041184	0.033791
	1	0.039164	0.040657	0.033593	0.035913	0.041212	0.037413	0.038836	0.036227	0.032234	0.033624
8	2	0.039234	0.034405	0.032717	0.035491	0.041931	0.036968	0.034542	0.037583	0.036328	0.035448
	3	0.035042	0.037325	0.032881	0.033444	0.043939	0.033711	0.033335	0.034332	0.034353	0.041712
	4	0.037569	0.035305	0.036539	0.035159	0.03454	0.038353	0.037967	0.034076	0.039884	0.03219

Table 8: SANSoL RPNS rates sweeping

[2]the results in the following tables are ranked and color-coded together: Table 8, Table 9

k-hop	RPNS ID/Rate	15%	20%	32%	40%	50%	60%	70%	80%	90%	100%
	0	0.123777	0.147264	0.147883	0.125712	0.128047	0.131791	0.14925	0.14131	0.133819	0.136512
	1	0.129539	0.137838	0.137658	0.144242	0.150583	0.144463	0.15042	0.148294	0.143957	0.138523
2	2	0.141851	0.141742	0.133016	0.132264	0.129973	0.143267	0.136068	0.133734	0.141049	0.136362
	3	0.142588	0.137751	0.148769	0.134888	0.135974	0.143222	0.130764	0.13108	0.134193	0.138658
	4	0.143099	0.131654	0.138529	0.145829	0.136284	0.159035	0.13598	0.125564	0.137239	0.146856
	0	0.036147	0.039009	0.03538	0.041295	0.038916	0.031788	0.039122	0.045822	0.043577	0.11195
	1	0.037647	0.044778	0.03482	0.038402	0.040055	0.040434	0.040464	0.037369	0.036107	0.144553
3	2	0.04035	0.037505	0.037621	0.037875	0.038817	0.035304	0.039418	0.042612	0.042038	0.099202
	3	0.039581	0.035012	0.041852	0.042669	0.038359	0.043631	0.036423	0.040897	0.044407	0.03802
	4	0.035358	0.041842	0.040965	0.035932	0.041755	0.038111	0.047371	0.043739	0.038165	0.034445
	0	0.03696	0.044543	0.03514	0.034283	0.035704	0.037407	0.034957	0.043146	0.125514	0.111607
	1	0.035132	0.038505	0.035561	0.036619	0.037945	0.036125	0.032286	0.036058	0.127119	0.105901
4	2	0.038078	0.033973	0.036822	0.038381	0.036332	0.03647	0.034025	0.035099	0.171472	0.128079
	3	0.034639	0.036514	0.043744	0.038821	0.035584	0.034843	0.033168	0.12135	0.120886	0.120663
	4	0.034864	0.037092	0.03275	0.040173	0.033955	0.034753	0.034652	0.115753	0.118634	0.114745
	0	0.036457	0.039938	0.033739	0.036144	0.04226	0.033834	0.037082	0.039169	0.032629	0.03827
	1	0.032888	0.040332	0.039733	0.046919	0.03511	0.037045	0.037626	0.036801	0.045008	0.03744
5	2	0.042043	0.035124	0.034912	0.035839	0.03595	0.034321	0.041981	0.03506	0.030849	0.043038
	3	0.042043	0.037229	0.033182	0.039435	0.035194	0.037292	0.035974	0.033762	0.037755	0.035868
	4	0.042043	0.032942	0.044665	0.03431	0.045071	0.040902	0.033128	0.035916	0.035863	0.039038
	0	0.036094	0.036616	0.035389	0.034743	0.035159	0.036902	0.03885	0.036495	0.037069	0.03506
	1	0.042515	0.039104	0.037132	0.03528	0.034472	0.038298	0.038166	0.039927	0.033903	0.03179
6	2	0.041437	0.035765	0.038598	0.035464	0.033663	0.034142	0.038543	0.037784	0.035799	0.036434
	3	0.044185	0.034099	0.040678	0.037425	0.03447	0.040317	0.039059	0.034391	0.041422	0.036174
	4	0.03733	0.038661	0.037616	0.039028	0.03735	0.038145	0.035848	0.034589	0.041334	0.035481
	0	0.040522	0.037924	0.044854	0.034917	0.031766	0.037163	0.033323	0.037133	0.042435	0.035276
	1	0.039752	0.033858	0.039878	0.033228	0.033997	0.042846	0.036917	0.034986	0.036424	0.035313
7	2	0.041687	0.03673	0.033807	0.03662	0.039913	0.037587	0.039474	0.036161	0.03834	0.039997
	3	0.033183	0.035461	0.038504	0.034745	0.033607	0.035804	0.036874	0.042284	0.041202	0.037348
	4	0.033666	0.039582	0.036489	0.036838	0.034516	0.035188	0.035997	0.037048	0.033344	0.04021
	0	0.036394	0.033997	0.037469	0.03674	0.038996	0.032911	0.036071	0.0351	0.032652	0.039019
	1	0.034998	0.036053	0.037772	0.036967	0.035741	0.040217	0.035564	0.036689	0.035707	0.03336
8	2	0.040002	0.038551	0.038319	0.03864	0.038182	0.037185	0.039569	0.033822	0.038341	0.034182
	3	0.03728	0.034013	0.036939	0.033584	0.032697	0.040898	0.033071	0.035368	0.039182	0.033455
	4	0.036796	0.038106	0.035921	0.037127	0.040456	0.032453	0.038618	0.039382	0.034819	0.035734

Table 9: SANSoLF RPNS rates sweeping

k-hop	RPNS ID/Rate	15%	20%	32%	40%	50%	60%	70%	80%	90%	100%
	0	0.356671	0.294691	0.315896	0.34236	0.24301	0.241821	0.259089	0.271688	0.190301	0.179872
	1	0.358094	0.327963	0.366449	0.251615	0.267186	0.304125	0.202831	0.232039	0.233764	0.204824
2	2	0.376405	0.383666	0.296441	0.373426	0.377127	0.319912	0.362084	0.242217	0.24234	0.203348
	3	0.33461	0.361644	0.253536	0.224199	0.312107	0.230911	0.236156	0.286871	0.272591	0.193958
	4	0.366139	0.367601	0.295184	0.367874	0.290434	0.269054	0.283065	0.226121	0.248237	0.198533
	0	0.356008	0.306125	0.286505	0.333978	0.164754	0.131267	0.182478	0.167242	0.128188	0.071991
	1	0.373185	0.330883	0.326624	0.176389	0.17639	0.267894	0.132872	0.120144	0.119324	0.118087
3	2	0.377574	0.359138	0.217803	0.34086	0.345485	0.273149	0.353754	0.160241	0.15652	0.111715
	3	0.318952	0.364991	0.182707	0.118457	0.197033	0.166428	0.17076	0.166802	0.167727	0.110997
	4	0.366575	0.360692	0.22835	0.337724	0.181745	0.271696	0.187692	0.163193	0.12924	0.121114
	0	0.356187	0.298733	0.31525	0.336113	0.249399	0.25135	0.258125	0.266189	0.190305	0.175525
	1	0.369027	0.330244	0.364062	0.251942	0.277176	0.30399	0.195672	0.240333	0.224598	0.202364
4	2	0.377559	0.375584	0.298102	0.376274	0.373743	0.307296	0.3674	0.244635	0.231557	0.209178
	3	0.324505	0.377132	0.253609	0.22679	0.302241	0.241404	0.23487	0.266259	0.272276	0.207113
	4	0.364309	0.365972	0.290591	0.36503	0.298197	0.266383	0.284275	0.221073	0.236474	0.190251
	0	0.361225	0.293468	0.282594	0.323536	0.172042	0.131418	0.181153	0.163177	0.122026	0.076581
	1	0.361398	0.324176	0.333543	0.166713	0.174934	0.26825	0.139424	0.12154	0.129517	0.112814
5	2	0.371572	0.370888	0.214999	0.339223	0.346087	0.271929	0.352133	0.167009	0.163568	0.115519
	3	0.324291	0.368078	0.183381	0.120635	0.201	0.166952	0.164656	0.158307	0.161564	0.113009
	4	0.371837	0.362546	0.22059	0.329168	0.185237	0.265926	0.185298	0.160499	0.137459	0.115213
	0	0.367566		0.320726	0.337925	0.247169	0.243593	0.252222	0.271181	0.188479	0.195377
	1	0.368774		0.356088	0.251059	0.28643	0.309548	0.196689	0.228662	0.223813	0.200495
6	2	0.374407		0.303942	0.378051	0.372675	0.316447	0.366286	0.239633	0.230395	0.209036
	3	0.335251		0.251025	0.232163	0.297779	0.241705	0.242789	0.278501	0.275935	0.204538
	4	0.368949		0.291676	0.37185	0.294269	0.274641	0.275812	0.221132	0.238406	
	0	0.355868	0.302249		0.322802	0.170019		0.179833	0.163547	0.125904	
	1	0.367128						0.136252			
7	2		0.379885		0.352772					0.160907	
	3	0.316886	0.371288		0.121653			0.165875	0.15058		
	4			0.229039		0.185123		0.185559	0.163939		
	0		0.309317		0.338627				0.277836		
	1	0.36746				0.280118		0.192737	0.219184		
8	2		0.380097	0.30066	0.372517		0.314667		0.244953		0.202915
	3			0.256695		0.293801		0.232155	0.274097	0.277398	0.204596
	4	0.357901	0.362935		0.375258	0.29209	0.268606			0.240163	0.198317

Table 10: RW-SANSoL RPNS rates sweeping

k-hop	RPNS ID/Rate	15%	20%	32%	40%	50%	60%	70%	80%	90%	100%
	0	0.361485	0.296583	0.316935	0.342468	0.242655	0.244188	0.248522	0.26734	0.193821	0.170089
	1	0.357372	0.327365	0.359527	0.25222	0.276301	0.308388	0.203498	0.225751	0.231579	0.203862
2	2	0.377813	0.377062	0.295184	0.377242	0.369279	0.320317	0.361898	0.243516	0.236633	0.202165
	3	0.331355	0.352427	0.255188	0.228195	0.319104	0.23272	0.232507	0.282959	0.267617	0.18983
	4	0.368292	0.366657	0.306683	0.368443	0.298687	0.272431	0.279471	0.220328	0.25167	0.200512
	0	0.355439	0.302676	0.285507	0.326159	0.167523	0.134476	0.186088	0.171194	0.128841	0.067662
	1	0.371524	0.331918	0.328829	0.177342	0.176344	0.269292	0.133638	0.121236	0.124201	0.118491
3	2	0.374454	0.360511	0.219005	0.342883	0.347048	0.27265	0.356596	0.168101	0.164651	0.112635
	3	0.324498	0.367134	0.180556	0.122443	0.198482	0.166938	0.17189	0.160212	0.169477	0.113864
	4	0.363596	0.360228	0.224224	0.334408	0.181932	0.271567	0.185335	0.162075	0.129894	0.121395
	0	0.364079	0.29683	0.3121	0.339084	0.251638	0.246793	0.257794	0.268085	0.196913	0.184389
	1	0.362718	0.330627	0.36441	0.248568	0.278608	0.310246	0.19567	0.225608	0.21873	0.201633
4	2	0.382809	0.376532	0.299682	0.372194	0.373753	0.310766	0.362268	0.241713	0.231216	0.198328
	3	0.325533	0.366118	0.258747	0.222915	0.298185	0.244492	0.234801	0.27345	0.268231	0.196138
	4	0.366459	0.368591	0.291274	0.369813	0.304918	0.273645	0.283574	0.227334	0.238338	0.186727
	0	0.364079	0.29683	0.3121	0.339084	0.251638	0.246793	0.257794	0.268085	0.196913	0.184389
	1	0.362718	0.330627	0.36441	0.248568	0.278608	0.310246	0.19567	0.225608	0.21873	0.201633
5	2	0.382809	0.376532	0.299682	0.372194	0.373753	0.310766	0.362268	0.241713	0.231216	0.198328
	3	0.325533	0.366118	0.258747	0.222915	0.298185	0.244492	0.234801	0.27345	0.268231	0.196138
	4	0.366459	0.368591	0.291274	0.369813	0.304918	0.273645	0.283574	0.227334	0.238338	0.186727
	0	0.370261		0.314977	0.329881	0.247851	0.254121	0.253707	0.267855	0.189996	0.176017
	1			0.354915	0.25412	0.271019	0.314905	0.197978	0.226403	0.226252	0.194528
6	2	0.376603		0.305218	0.376527	0.37401	0.32386	0.364815	0.232806	0.226068	0.20086
	3			0.253109	0.220821	0.299557	0.244393	0.247386	0.282424	0.273453	0.206283
	4	0.372144		0.28866	0.368417	0.302452	0.276888	0.291298	0.223475	0.240005	
	0		0.30147		0.322918	0.17105			0.164993	0.131765	
	1	0.36255	0.327438					0.134858			
7	2			0.219815				0.360126	0.160519	0.169519	
	3		0.373881		0.120327	0.200676		0.165805	0.159938		
	4	0.354387					0.269076		0.159209		
	0	0.363916			0.338323			0.260414			
	1	0.371678	0.328501	0.353962		0.276431		0.195369			
8	2			0.28666	0.374879	0.378392	0.311383		0.242987		0.197792
	3		0.368894						0.275915	0.268618	
	4	0.361702		0.294994		0.293329	0.279101	0.280315			0.198868

Table 11: RW-SANSoLF RPNS rates sweeping

k-hop	RPNS ID/Rate	15%	20%	32%	40%	50%	60%	70%	80%	90%	100%
	0	0.144626	0.126406	0.14132	0.140783	0.140793	0.142168	0.134734	0.139704	0.135393	
	1		0.13648	0.146092	0.131017	0.143158	0.144636	0.127699	0.128578	0.134691	
2	2		0.139656	0.127922	0.134924	0.151273	0.126913	0.132511	0.130412	0.131988	
	3		0.137417	0.135311	0.141866	0.126624	0.144828	0.135279	0.140762	0.149034	
	4		0.132248	0.149742	0.13501	0.14509	0.12636	0.14723	0.125489	0.14656	
	0	0.038644	0.039555	0.038131	0.037589	0.039836	0.036405	0.036379	0.039881	0.035167	
	1		0.042105	0.042704	0.036869	0.035923	0.038308	0.037898	0.040293	0.034466	
3	2		0.040243	0.038178	0.044257	0.038938	0.037266	0.039581	0.037664	0.041247	
	3		0.037169	0.041092	0.035177	0.035986	0.035239	0.039953	0.04364	0.040414	
	4		0.041521	0.0383	0.040643	0.03397	0.042094	0.037197	0.033964	0.038862	
	0	0.036147	0.035412	0.036184	0.032805	0.0386	0.033435	0.040972	0.033867	0.036082	
	1		0.037673	0.038943	0.038359	0.034613	0.035342	0.036398	0.034686	0.043547	
4	2		0.033083	0.038789	0.040678	0.037759	0.037265	0.037653	0.037756		
	3		0.033975	0.037651	0.03892	0.032907	0.034706	0.035995	0.035772		
	4		0.031444	0.034985	0.03822	0.038312	0.032751	0.036484	0.038729		
	0	0.033881	0.03287	0.033244	0.034087	0.034649	0.036951	0.036466	0.034897	0.034286	
	1		0.036901	0.045792	0.03407	0.041369	0.038713	0.035634	0.037565	0.035344	
5	2		0.037559	0.042882	0.036475	0.033998	0.03378	0.031947	0.037222	0.039735	
	3		0.037966	0.037068	0.035918	0.036422	0.034156	0.039277	0.035682	0.038359	
	4		0.03344	0.043608	0.042091	0.032672	0.040863	0.040591	0.040487	0.03612	
	0	0.036246	0.037185	0.038255	0.038234	0.034407	0.038105	0.03672	0.036485	0.035418	
	1		0.036475	0.035952	0.038927	0.039672	0.03627	0.033187	0.033202	0.0363	
6	2		0.039127	0.036094	0.037625	0.032867	0.034451	0.041361	0.035223	0.037009	
	3		0.033599	0.036853	0.042332	0.034318	0.034822	0.04	0.033828	0.034984	
	4		0.034749	0.044892	0.040217	0.04028	0.037416	0.04074	0.035922	0.033906	
	0	0.035989	0.035673	0.035295	0.034486	0.037194	0.034818	0.036005	0.044429	0.040023	
	1		0.037107	0.038598	0.035175	0.03426	0.037215	0.030999	0.041635	0.042821	
7	2		0.037992	0.035424	0.032306	0.040382	0.042449	0.035005	0.038775	0.036099	
	3		0.033899	0.035556	0.036827	0.037484	0.037766	0.041295	0.036966	0.035991	
	4		0.036623	0.034367	0.03334	0.037111	0.040167	0.035947	0.038807	0.037469	
	0	0.03664	0.037585	0.034309	0.035367	0.036422	0.039363	0.034547	0.036326	0.037191	
	1		0.035402	0.038743	0.034756	0.033084	0.036191	0.035088	0.032507	0.039794	
8	2		0.038564	0.036582	0.039922	0.035458	0.035471	0.037536	0.044541	0.033832	
	3		0.036207	0.034155	0.037473	0.044158	0.035679	0.037287	0.038623	0.03273	
	4		0.036115	0.032989	0.04134	0.039546	0.035455	0.039261	0.036963	0.035124	

Table 12: SANSoL-C RPNS rates sweeping

k-hop	RPNS ID/Rate	15%	20%	32%	40%	50%	60%	70%	80%	90%	100%
	0	0.144626	0.126406	0.14132	0.140783	0.140793	0.142168	0.134734	0.139704	0.135393	
	1		0.13648	0.146092	0.131017	0.143158	0.144636	0.127699	0.128578	0.134691	
2	2		0.139656	0.127922	0.134924	0.151273	0.126913	0.132511	0.130412	0.131988	
	3		0.137417	0.135311	0.141866	0.126624	0.144828	0.135279	0.140762	0.149034	
	4		0.132248	0.149742	0.13501	0.14509	0.12636	0.14723	0.125489	0.14656	
	0	0.038644	0.039555	0.038131	0.037589	0.039836	0.036405	0.036379	0.039881	0.035167	
	1		0.042105	0.042704	0.036869	0.035923	0.038308	0.037898	0.040293	0.034466	
3	2		0.040243	0.038178	0.044257	0.038938	0.037266	0.039581	0.037664	0.041247	
	3		0.037169	0.041092	0.035177	0.035986	0.035239	0.039953	0.04364	0.040414	
	4		0.041521	0.0383	0.040643	0.03397	0.042094	0.037197	0.033964	0.038862	
	0	0.036147	0.035412	0.036184	0.032805	0.0386	0.033435	0.040972	0.033867	0.036082	
	1		0.037673	0.038943	0.038359	0.034613	0.035342	0.036398	0.034686	0.043547	
4	2		0.033083	0.038789	0.040678	0.037759	0.037265	0.037653	0.037756		
	3		0.033975	0.037651	0.03892	0.032907	0.034706	0.035995	0.035772		
	4		0.031444	0.034985	0.03822	0.038312	0.032751	0.036484	0.038729		
	0	0.037705	0.035335	0.03844	0.043794	0.035592	0.032566	0.036886	0.036823	0.034657	
	1		0.039872	0.035581	0.03759	0.037761	0.034034	0.042187	0.043508	0.041208	
5	2		0.036994	0.034817	0.032868	0.033759	0.035787	0.044434	0.038769	0.040847	
	3		0.037037	0.037855	0.032916	0.036406	0.035418	0.041524	0.03534	0.035587	
	4		0.03515	0.038906	0.037664	0.037691	0.035228	0.044452	0.03619	0.034261	
	0	0.036246	0.037185	0.038255	0.038234	0.034407	0.038105	0.03672	0.036485	0.035418	
	1		0.036475	0.035952	0.038927	0.039672	0.03627	0.033187	0.033202	0.0363	
6	2		0.039127	0.036094	0.037625	0.032867	0.034451	0.041361	0.035223	0.037009	
	3		0.033599	0.036853	0.042332	0.034318	0.034822	0.04	0.033828	0.034984	
	4		0.034749	0.044892	0.040217	0.04028	0.037416	0.04074	0.035922	0.033906	
	0	0.035989	0.035673	0.035295	0.034486	0.037194	0.034818	0.036005	0.044429	0.040023	
	1		0.037107	0.038598	0.035175	0.03426	0.037215	0.030999	0.041635	0.042821	
7	2		0.037992	0.035424	0.032306	0.040382	0.042449	0.035005	0.038775	0.036099	
	3		0.033899	0.035556	0.036827	0.037484	0.037766	0.041295	0.036966	0.035991	
	4		0.036623	0.034367	0.03334	0.037111	0.040167	0.035947	0.038807	0.037469	
	0	0.03664	0.037585	0.034309	0.035367	0.036422	0.039363	0.034547	0.036326	0.037191	
	1		0.035402	0.038743	0.034756	0.033084	0.036191	0.035088	0.032507	0.039794	
8	2		0.038564	0.036582	0.039922	0.035458	0.035471	0.037536	0.044541	0.033832	
	3		0.036207	0.034155	0.037473	0.044158	0.035679	0.037287	0.038623	0.03273	
	4		0.036115	0.032989	0.04134	0.039546	0.035455	0.039261	0.036963	0.035124	

Table 13: SANSoLF-C RPNS rates sweeping

k-hop	RPNS ID/Rate	15%	20%	32%	40%	50%	60%	70%	80%	90%	100%
	0	0.365731		0.263366	0.341813		0.302313		0.273151	0.257828	
	1		0.345219		0.358344					0.268208	
2	2		0.336451	0.339191					0.283468		
	3		0.369913	0.374527		0.292681	0.277949	0.272683		0.253424	
	4			0.302847	0.303865		0.306449	0.28884	0.270736		
	0	0.369592	0.362344			0.284706		0.217759			
	1		0.352026		0.340438	0.372768	0.203002		0.200671		
3	2		0.33714			0.222296			0.24179		
	3		0.366247	0.347541		0.273229	0.221246	0.188252		0.233071	
	4		0.33535	0.224225	0.271105	0.301732	0.302976	0.289526			
	0	0.376159	0.352544				0.306784				
	1			0.304894							
4	2							0.257002	0.320776		
	3		0.368341			0.289247				0.270235	
	4			0.27154			0.305279	0.297911			
	0	0.373571	0.367462	0.212347							
	1			0.278633	0.336856						
5	2										
	3		0.367572	0.343368							
	4										
	0				0.331363	0.278902	0.30005				
	1		0.334541	0.308132							
6	2						0.269171		0.317664	0.279317	
	3			0.374711	0.361754		0.272196				
	4				0.300124		0.297063	0.300987	0.277025		
	0		0.360812								
	1			0.280121	0.341815		0.238482				
7	2			0.331487		0.230413	0.216977				
	3		0.361669	0.351995							
	4				0.287292						
	0	0.372937	0.348387	0.24706			0.299001				
	1					0.366397	0.276454			0.252707	
8	2			0.336624	0.287062	0.276843		0.249829			
	3		0.373377	0.375171		0.280859	0.269662				
	4		0.352908			0.308226					

Table 14: RW-SANSoL-C RPNS rates sweeping

k-hop	RPNS ID/Rate	15%	20%	32%	40%	50%	60%	70%	80%	90%	100%
	0	0.370748	0.336614	0.275765					0.285379	0.264004	
	1		0.34913		0.366996	0.359387				0.236097	
2	2			0.338765			0.288571		0.292223		
	3		0.372737	0.371207	0.356038	0.291738	0.266414			0.262565	
	4		0.362613	0.296063			0.308398	0.296434	0.271918	0.325581	
	0	0.369538	0.362131			0.283401		0.210733			
	1				0.338626	0.369526	0.213778		0.200739		
3	2		0.338106			0.211873			0.254023		
	3		0.36395	0.343555		0.274208	0.221191	0.196574		0.215309	
	4		0.333569	0.227996	0.27848	0.304372	0.300263	0.28963		0.29315	
	0						0.306229	0.274313			
	1		0.333915				0.273465				
4	2										
	3		0.375911	0.373559					0.270382		
	4			0.280316			0.308004	0.290734		0.317298	
	0	0.368627	0.366599								
	1			0.279043	0.340358						
5	2										
	3			0.347275							
	4		0.328338								
	0				0.341096	0.298344	0.294654				
	1		0.339899							0.253855	
6	2				0.301849		0.264121				
	3		0.365732	0.381542			0.277816				
	4				0.296092	0.300956	0.299123	0.298152	0.284865		
	0	0.371918		0.221224							
	1					0.370117	0.227981				
7	2			0.326539	0.230199		0.19861				
	3		0.36672	0.351068			0.25481				
	4			0.227941		0.303055					
	0	0.365734	0.349138						0.272736		
	1					0.370984	0.266246			0.233975	
8	2			0.326489	0.282119	0.269676					
	3		0.372425	0.382444		0.289324			0.261742		
	4		0.355186			0.302134	0.302613				

s

Table 15: RW-SANSoLF-C RPNS rates sweeping

Appendix B Final Results Hypothesis

X_1: MRR scores from the baseline (RW-SANS, k=2).
X_2: MRR scores from the best performing method (RW-SANSoLF, lies: RPNS-20-2, k=2)

$H_0 : \overline{X_1} = \overline{X_2}$
$H_\alpha : \overline{X_1} < \overline{X_2}$

	X_1	X_2
Mean	0.41778	0.419217
Variance	3.55E-05	8.55E-07
Observations	5	5
Pearson Correlation	-0.69227	
Hypothesized Mean Difference	0	
df	4	
t Stat	-0.48415	
P(T<=t) one-tail	0.326797	
t Critical one-tail	2.131847	
P(T<=t) two-tail	0.653593	
t Critical two-tail	2.776445	

Table 16: t-test for best result and baseline comparison