MSc thesis in Computer Science

Exploiting modularity during program synthesis

Alexander Freeman

August 2023

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

The work in this thesis was carried out in the:

Algorithmics group
Delft University of Technology

Supervisors:	Dr. Neil Yorke-Smith
	Dr. Sebastijan Dumančić
Co-reader:	Dr. Sebastian Proksch

Abstract

Inductive logic programming is a technique that generates logic programs which keep to a given specification using a background knowledge. We propose a new task in the field of program synthesis called Time-gated Partition-selection Inductive Logic Programming, consisting of splitting the background knowledge into partitions and selecting only the relevant partitions to a given set of examples. In order to show an initial direction of research and demonstrate the effectiveness of the approach, we have constructed a set of partitioning functions and a selection function. These were implemented using existing graph clustering and community detection algorithms applied to static call graphs of existing programs in the target language and using a linear time evaluation selection function. By comparing the inductive logic programming approach Popper to a version of Popper with its search space reduced using this technique, we show that these partitioning- and selection functions can improve the generated programs on three out of four different domains. Finally we show that there is a difference in partition quality by comparing the results to a random partitioning function. This work establishes background knowledge partitioning- and selection as a useful tool in program synthesis research.

Preface

The completion of this thesis is the final step in achieving my masters degree in Computer Science at the Delft University of Technology. It marks the end of eight months of work, spanning from January to August of 2023. Finishing this thesis has been incredibly rewarding and I am immensely grateful for having been able to accomplish this feat.

First, I would like to thank my supervisor Sebastijan Dumančić for his continued support, insights, feedback and guidance throughout this entire thesis. I would also like to thank my friends for enduring the days I spent talking about my thesis and lending an ear in the moments where the work was not easy.

Finally, but most importantly, a massive thanks to my parents for your constant and never ending support. I could not have done it without you.

Alexander Freeman
Delft, August 2023

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

Program Synthesis is a form of machine learning in which the aim is to generate or find a program given a specification. If this specification is a complete specification (meaning it fully describes all the expected behaviors and functionalities of the target program, leaving no ambiguity) the approaches are called *deductive* and otherwise it is called *Inductive* Program Synthesis (IPS). In the case of Inductive Program Synthesis, the specification is is typically a domain specific language and examples of inputs and outputs (Gulwani, Polozov, & Singh, 2017). IPS has successfully been used in various domains, like mining library specifications (Sankaranarayanan, Ivancic, & Gupta, 2008) and guiding reinforcement learning algorithms (Yang et al., 2021).

Generating these programs is usually done in the form of a search over all possible programs in the given programming language (Kitzelmann, 2009). This search is a hard combinatorial problem and presents one of the main challenges in inductive program synthesis: keeping the search procedure tractable (i.e. keeping the search space small enough), while having enough syntax to allow for complex programs (Barke, Peleg, & Polikarpova, 2020; Cropper & Dumančić, 2022; Gulwani et al., 2017).

Over the years various approaches to limit the search space have been proposed, such as predicting which functions are likely to be part of the program using neural networks (Balog et al., 2017), using an example-dependent loss function to guide the search (Cropper & Morel, 2020) or pruning the search tree via types (Diaconu, 2020). Other approaches (for example) encode the specification as a SAT problem and leverage existing constraint solving technologies to speed up the search (Gulwani et al., 2017).

One of the reasons for the fact that the search space can get large, is because the given language is too loosely constraining (Cropper & Dumančić, 2022). That is, a non-trivial subset of the provided language is not needed to solve the given task(s) while it is still extending the search tree. We propose that we could improve the performance of current synthesis approaches by exploiting the modularity of a given grammar. By partitioning the grammar/language and only providing the necessary part, akin to a programmer using only modules of a software library, the overall search tree will be reduced while allowing the synthesis of sufficiently complex programs. Since the search itself is costly, finding these modules would have to rely on structures in the grammar and/or language itself.

In this thesis we will explore methods of extracting modules and applying them during inductive program synthesis with the goal of improving the programs found in equal amounts of time. The resulting main research question that this paper tries to answer is therefore:

Can inductive program synthesis approaches exploit modularity in grammars in order to improve the synthesis of programs within a given time limit?

In order to answer this question, the following research questions will be answered:

1. In what ways can we extract partitions from grammars that optimize the search of programs?
2. How do we leverage grammar partitions during the search of programs effectively?
3. Does varying the partitioning algorithm cause a difference in performance of the final programs?
4. Does the effect of applying modularity differ between types of problems?

To be able to understand the rest of the thesis, a background and prior work on program synthesis, program comprehension and graph algorithms will be covered in chapter 2. After that, chapter 3 will contain a more detailed problem statement and chapter 4 will detail the evaluation of the proposed methods. The results of these will be described in chapter 5. Finally chapter 6 will be the future work, limitations and conclusion.

2. Prior work and background

Several fields of research will be leveraged in this thesis: software modularization, program synthesis and graph algorithms like community detection. In this chapter we will provide the necessary background to understand the rest of the thesis.

2.1. Inductive Program Synthesis

As stated in the introduction, Inductive Program Synthesis is a machine learning technique that aims to synthesize a program in a given programming language that adheres to a given specification that is incomplete by definition. This specification is usually in the form of tasks, each containing a list of examples $(x_1, y_1), (x_2, y_2)...(x_n, y_n)$, where y_n is the expected output of the program. The output of this procedure is a program that (hopefully) covers all the cases in the specification (Gulwani et al., 2017). Depending on the implementation, programs can even be reused during synthesis, leading to shorter programs. This is also called *Program* or *Function re-use* (Diaconu, 2020).

The form in which the programming language is encoded varies a lot across the field. It it important to note that the choice of the language is itself a form of *inductive bias*. For example, a language without loops would require a larger depth of search if the specification requires a lot of repetitive actions and the search space induced by language based off of mathematical functions might not contain a string manipulation program. Examples of encodings are grammars (Cohen, 1994), the 'BNF' syntax used by Diaconu (2020) and mode declarations (Muggleton, 1995). Since these are equivalent, we will just refer to this encoding as the *grammar*. The available units are also often called the *Background Knowledge* (BK).

⟨decl ⟩ ::= 'val' ⟨ident⟩ '=' ⟨expr ⟩ – non recursive definition
— 'rec' ⟨ident⟩ '=' ⟨expr ⟩ – recursive definition
— 'Pex' ⟨expr ⟩ =¿ ⟨expr ⟩ – a way to specify positive examples
— 'Nex' ⟨expr ⟩ =¿ ⟨expr ⟩– a way to specify negative examples
⟨expr ⟩ ::= 'Num' n
— 'Char' c
— 'True'
— 'False'
— 'Variable' ⟨ident⟩
— 'Lambda' [⟨ident⟩] ⟨expr ⟩
— ⟨expr ⟩ ⟨expr ⟩
— 'If' ⟨expr ⟩ 'then' ⟨expr ⟩ 'else' ⟨expr ⟩'
— [i] – we need to represent holes in the syntax

Figure 2.1.: Example syntax from the 'BNF' syntax used in (Diaconu, 2020)

In case the language is a logic programming language, it enters the field called *Inductive Logic Programming* (ILP). Inductive logic programming is a field which is widely researched (Cropper & Dumančić, 2022). The goal in ILP is to generate a target predicate that evaluates to true for the positive examples and to false for the negative examples, based on predicates that are defined in the background knowledge. Examples of ILP solvers are Popper (Cropper & Morel, 2020), Aleph (Srinivasan, 2001) and Metagol (Cropper & Muggleton, 2016).

In this thesis we primarily use Popper. Popper specifically is an inductive logic programming approach that Cropper and Morel (2020) call *learning from failures*, which is currently considered state-of-the-art. They use a three-staged approach of *generate*, *test* and *constrain*. During the generation step, Popper generates a logic program, which they call a hypothesis. These are tested against the examples and succeeds if it does not entail all positive or a single negative example. Then they use this information to constrain the hypothesis space to prune the search space. Negative and positive examples are used differently in this process, so a mix of both is useful for Popper to work efficiently.

Popper solely generates Prolog programs. A Prolog program consists of *functions*. Each consists of *rules*, each of which have a *literal* as the head of the rule, terms along with *predicates* as the body. The rule evaluates to true if all the predicates in the body evaluate to true. If a predicate has multiple rules, it evaluates to true if *any* of the rules evaluates to true. An example of a logic program by can be seen in listing 2.2. In this case, we define a single function, 'last', which has two rules (Cropper & Dumančić, 2022).

```
last(A,B):- tail(A,C),empty(C),head(A,B).
last(A,B):- tail(A,C),last(C,B).
```

Figure 2.2.: An example snippet of Prolog code by Cropper and Dumančić (2022)

By using these encodings and/or templates, the problem of program synthesis is reduced to a first-order search over all possible programs induced by this template. The size of this search tree, however, is the main problem in program synthesis. Since it is unknown how long the program is or which parts of the syntax are important, any of the next tokens could be considered. This leads to a combinatorial explosion, based on the size of the grammar.

Many techniques to tackle the size of the search space have been used over the years, under which genetic algorithms, stochastic search and constraint solving are just a few (Gulwani et al., 2017). We will highlight certain directions that have been proposed and evaluated.

2.1.1. Search based synthesis

The most straightforward method of searching is through enumerative search, either bottom-up or top-down, using algorithms like A*, best first search or Iterative Budgeted Exponential Search. Guiding such a search is necessary to keep it tractable. Examples of this are Probe (Barke et al., 2020) and Brute (Cropper & Morel, 2020). Probe is an ILP system that learns a probabilistic model during bottom-up search, that is used to generate the most likely programs first. This is done by using a probabilistic context-free grammar in which the probabilities are learned based on *entailment*, the percentage of examples of the specification that is being satisfied. On the other side, Brute is an ILP system that uses domain-specific loss functions as a heuristic in a best-first search, showing an increase in performance compared to systems that

only use entailment as their target. Another technique that can be used during search, is the pruning of subtrees. Typing systems can, for example, be used to rule out possible branches (Diaconu, 2020) as can program equivalence (Smith & Albarghouthi, 2019).

2.1.2. Neural approaches

Another branch of research is the integration of neural networks with program synthesis. The majority of these approaches still utilize search based techniques in their implementation, with neural networks being used to learn heuristics (Balog et al., 2017; S. Zhang et al., 2023) or probability distributions (Ellis et al., 2020) that guide the search like earlier examples. DeepCoder for example learns a probability of each part of the syntax appearing in the final program (as can be seen in figure 2.3. This is used to prioritize elements during depth first search (Balog et al., 2017). Large language models have recently had a large surge in popularity and they work remarkably well. However they, along with the the other neural approaches mentioned, require a large amount of samples and training times (Austin et al., 2021; Cropper & Morel, 2020).

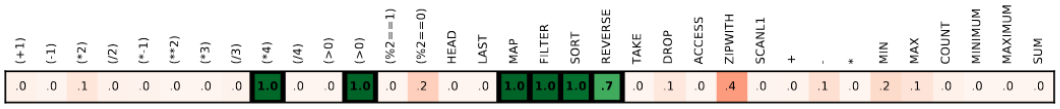


Figure 2.3.: An example of the output of the network from Balog et al. (2017)

.

2.1.3. Dynamic background knowledge

Finally, while all prior techniques have been pruning and prioritizing *during* search, there has been little research into pruning parts of the syntax *before* the search. If the set of functions of a language is larger than needed, the search tree will also be larger than needed. An analysis of source code written in the programming language Python showed that human programmers do not use all subsets at all times while coding. Furthermore, they showed partitioning the background knowledge into subsets can significantly reduce the search tree (McDaid & McDaid, 2023).

Another piece of research added functions corresponding to different domains (e.g. dates, emails, phone-numbers and timestamps) to the same language in order to analyze performance gains by inferring the right domain in an effort to shrink the search space. Inference was approached via machine learning techniques like neural networks and random forests using extracted features from the specification. Results showed that this reduced search times while keeping the same accuracy as without domain inference (Contreras-Ochando et al., 2020).

It is important to note that, while domain splitting- and inference showed performance gains, the domains were handcrafted along with the corresponding functions. No research has yet to dynamically determine domains in a given language and infer the dynamic domains needed for a given problem in program synthesis. To find these domains, the program must first be decomposed into modules that can be used separately. This requires understanding the software on a more fundamental level than just the grammar.

Grammars used in program synthesis are often simpler than the generic languages used by programmers, largely because of the combinatorial explosion. Logic programming languages like Prolog for example often have no classes or namespaces, just like the domain specific languages like the one in figure 2.1. However it is still software and existing methods of analysing the structure and breaking down the software into parts could prove useful in the research.

2.2. Program comprehension

The method of finding modules in a unsorted set of software units falls in a field called 'program comprehension'. Generally the program comprehension techniques use a way of expressing relations between entities in the code. Entities can come in various forms, like software modules, functions or subsystems (Mitchell & Mancoridis, 2006). The relationships between them form the structure of the software that is being examined and can be generated in various ways. In 2015 a survey concluded that symbol level dependencies, i.e. invocation of one entity by another, can lead to more accurate techniques for recovering architectures in comparison to file dependencies only, suggesting that coarser dependency graphs produce better results. They also showed that more detailed dependencies improve the final result as well (Lutellier et al., 2015).

One type of these views is a *call graph* and an example can be seen in figure 2.4. Call graphs have been widely utilized to understand the structure of software (Alanazi, Gharibi, & Lee, 2021; Bhattacharjee, Roy, & Schneider, 2022; Gharibi, Alanazi, & Lee, 2018) and can, in turn, be processed in order to extract clusters of program units, like functions or methods. There are two types of call graphs: static and dynamic. Static callgraphs are defined by the program itself and dynamic call graphs are execution paths that happen in a single run of the program. We only focus on static callgraphs in this research and at any time we refer to a callgraph it is a static call graph.

2.2.1. Hierarchical clustering of execution paths

One method of extracting information from call graphs is through execution paths (Alanazi et al., 2021; Bhattacharjee et al., 2021, 2022; Walunj, Gharibi, Ho, & Lee, 2019). An example of a processing pipeline is the following by Alanazi et al. (2021): first a subset of the possible execution path (like a $\rightarrow$ b $\rightarrow$ c $\rightarrow$d in figure 2.4) are converted into a feature matrix which uses a bit indicating membership for each entity in the graph (1 if it is in the path and 0 otherwise). Then the distance between each path is calculated using a similarity measure, which is used in a hierarchical clustering algorithm. The resulting clusters of paths are then converted to their

```
def a():
    b()
    c()

def b():
    c()
    e()

def c():
    d()

def d():
    pass

def e():
    pass
```

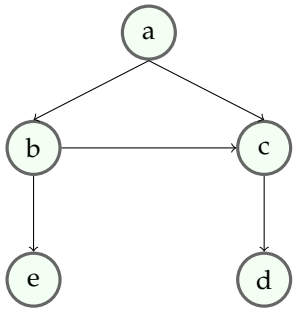


Figure 2.4.: A snippet of Python code with its corresponding call graph

own graphs by reconnecting all nodes in all paths for each cluster, which they use to visualise the clusters as part of the original graph. Bhattacharjee et al. (2022) improved on this approach by including a flattening stage in order to remove redundant nodes.

Converting the clusters of paths to clusters of *nodes* would require designing a custom algorithm, as this has not been explored in the research. Since nodes can be part of multiple clusters, overlapping clusters could be produced, which may or may not be useful in the scope of this research: since it duplicates work when checking whether clusters are useful but it also allows heavily used nodes to be present in more domains. It is also important to note that the similarity measure and linking type can heavily influence the results and while Alanazi et al. (2021) suggest that the Jaccard measure works best together with a complete linkage, experimentation is required in order to evaluate it properly.

2.2.2. Graph metrics, graph clustering and community detection

Another way of finding clusters in a call graph (or in fact any graph) is through the use of *graph clustering*. Graph clustering is the process of grouping nodes of a graph into disjoint groups. These types of algorithms have shown to produce good results at program comprehension tasks: Mitchell and Mancoridis (2006) evaluated hill climbing and simulated annealing algorithms using custom metrics on software, packaged in a tool called *Bunch*. They showed that their results are close to the reference decomposition, which they take to be the optimal solution. These algorithms take a graph of file dependencies as input and as such can also be modified to take any call graph. While Mitchell and Mancoridis (2006) do not specifically name these algorithms 'graph clustering' algorithms, they do fall under graph clustering.

One specific type of graph clustering is *community detection*. Communities are substructures in graphs that have a high concentration of edges inside the group and low amounts of edges between different groups. They are inherently useful for identifying clusters in real networks

(Fortunato, 2010) and software dependency networks have been shown to have significant community structures (Subelj & Bajec, 2011). Community detection is a heavily researched field and many approaches exist, such as spectral based methods (M. E. J. Newman, 2013) and using graph neural networks (Tsitsulin, Palowitch, Perozzi, & Müller, 2020).

Many of these algorithm use a concept called *modularity*. Modularity is defined by M. Newman (2010) as the following:

Modularity is a similarity measure, calculated by the formula

$$Q = \frac{1}{2m} \sum_{i,j} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta(c_i, c_j)$$

where:

Q is the modularity of the graph.
m is the total number of edges in the graph.
A_{ij} represents the element in the adjacency matrix of G between nodes i and j.
k_i is the degree of node i, i.e., the number of edges connected to node i.
c_i is the community to which node i belongs.
$\delta(c_i, c_j)$ is the Kronecker delta function, $\delta(c_i, c_j) = \begin{cases} 1, & \text{if } c_i = c_j \\ 0, & \text{otherwise} \end{cases}$

In other words, modularity is a measure of how closely-knit nodes are within their own community compared to how they interact with nodes outside of their community. It helps us understand the strength of connections inside a community versus those that reach beyond its boundaries. An example of a highly modular graph can be seen in figure 2.5, where the high modularity comes from the many edges *inside* the communities versus the few edges *between* the communities.

This research will use algorithms from both community detection and graph clustering. Specifically, we apply Paris (Bonald, Charpentier, Galland, & Hollocou, 2018), METIS (Karypis & Kumar, 1999), Greedy modularity maximization (Clauset, Newman, & Moore, 2004), Label propagation (Raghavan, Albert, & Kumara, 2007) and the Louvain algorithm (in two forms) (Karypis & Kumar, 1999). Label propagation, Louvain and Greedy Nodularity Maximization are all available via the python package NetworkX (Hagberg, Swart, & S Chult, 2008) and have a common interface, making experimentation easy. METIS and Paris have accompanying implementations that allow for rapid application as well. An overview of the algorithms with parameters, types and outputs can be found in table 2.1. In the next sections we will provide a brief explanation of each algorithm.

Louvain

The Louvain algorithm is a widely used community detection method in graph clustering. The algorithm's objective is to optimize the modularity of the graph as explained above. In

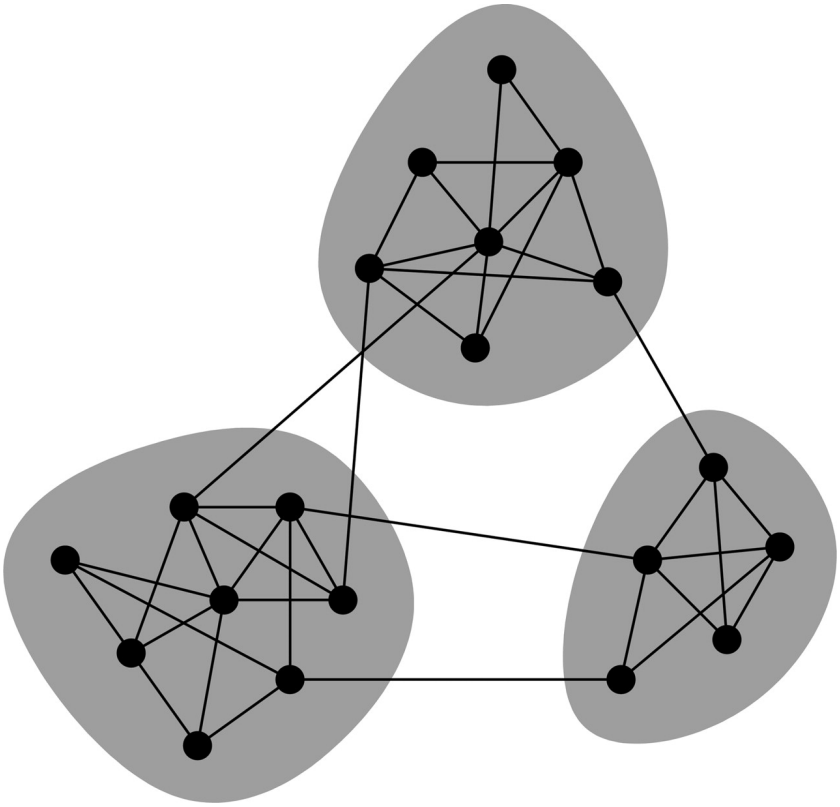


Figure 2.5.: An example of a highly modular network by M. E. J. Newman (2006)

.

2017 Mothe, Mkhitaryan, and Haroutunian (2017) evaluated various community detection algorithms and determined that the Louvain algorithm (Blondel, Guillaume, Lambiotte, & Lefebvre, 2008) gives the best results for identifying communities with high modularity. The Louvain algorithm also has the added benefit of not needing a predetermined amount of modules as well as being able to use weights on edges to guide the clustering. It must be noted however, that Mothe et al. (2017) did not evaluate any neural approaches and as such no comparisons between them can be made.

The Louvain algorithm operates in two phases iteratively. First, it greedily optimizes modularity by moving nodes between communities. In the second phase, it constructs a new graph where communities found in the first phase are treated as individual nodes. The process is then repeated. This approach uncovers community structures by iteratively refining the graph's partition efficiently. The NetworkX implementation by default then outputs the version with the highest amount of modularity, however, the intermediate results can be seen as a dendrogram, giving partitions with increasing amounts of modularity.

A dendrogram is a hierarchical representation of a clustering process in graphs or data sets, akin to the hierarchical approach employed by the Paris algorithm for community detection. It does not directly involve modularity optimization but captures the hierarchical relationships among clusters.

The dendrogram is constructed through a bottom-up merging approach, starting with individual data points as individual clusters and iteratively combining them based on their similarity or distance until all data points are grouped into a single overarching cluster. Each level of the dendrogram represents a different level of granularity in the clustering, and branches in the dendrogram illustrate the merging process. The structure of the dendrogram allows for a visual understanding of the hierarchy of clusters and allows for the identification of nested and overlapping clusters within the data.

An example can be seen in figure 2.6. In this dendrogram we have five singular predicates (A to E), seen as nodes as the bottom. The y-axis is a distance metric, which as an example is set to a range between 0 and 10. Each point where two lines meet is a new cluster. For example, the cluster of node A and the cluster of node B get combined when the distance threshold is set at 3.

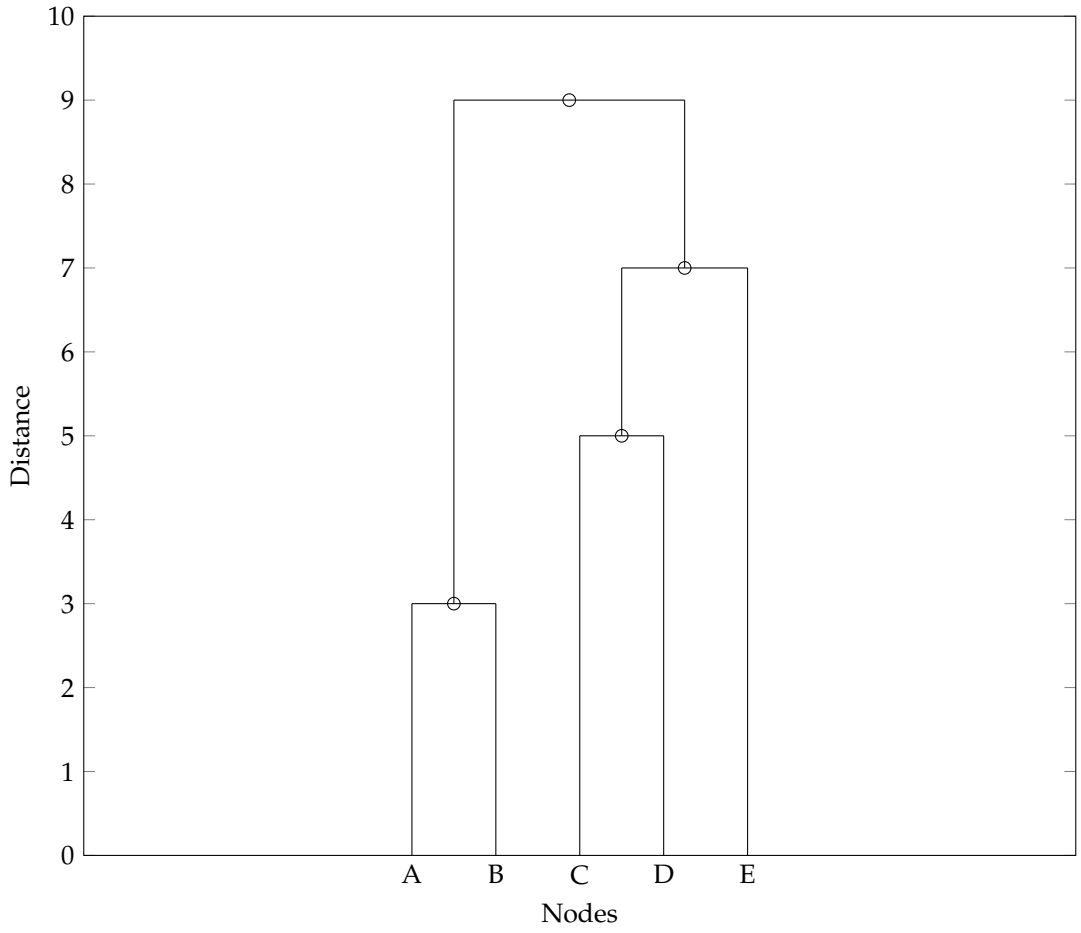


Figure 2.6.: An example of a dendrogram

Scikit-network (Bonald, de Lara, Lutz, & Charpentier, 2020) has a implementation of iterative louvain returning this dendrogram, both bottom-up, (combining clusters), and a top-down (breaking clusters apart) view.

Finally, the Louvain algorithm has an optional parameter called the *resolution*. This parameter impacts the size of the partitions. The closer to zero it gets, the larger the partitions. By default this value is set to 1.

Greedy modularity maximization

Greedy modularity maximization, similar to the Louvain algorithm, is a popular graph clustering approach employed for community detection. It shares the common goal of optimizing modularity, however the specific technique differs from the Louvain algorithm. Greedy modularity maximization follows a simpler and more direct strategy. The algorithm places each node in the community that yields the highest increase in modularity, and this process is repeated until no further improvements can be made. Despite its straightforward nature, greedy modularity maximization demonstrates efficiency, particularly in handling large-scale graphs.

Paris

Paris is also a graph clustering algorithm utilized for community detection in graphs. Similar to the previous two algorithms, Paris focuses on optimizing modularity. However, Paris differentiates itself by using a hierarchical approach to uncover community structures. It uses a custom distance metric based on probabilistic measures, which capture the probability of sampling node pairs. They also show that this has links to modularity. In the end the result is a dendrogram, like iterative louvain.

Label propagation

Label propagation, in contrast to modularity-based algorithms like Louvain and greedy modularity maximization, does not directly optimize modularity as its objective for community detection. Instead, this method adopts a distinct approach centered around label updating and propagation. The algorithm assigns unique labels to each node in the graph and iteratively updates these labels based on the labels of neighboring nodes.

Through this iterative process, labels propagate across the graph, converging nodes with similar connectivity patterns into the same community. The primary goal of label propagation is to achieve label convergence.

METIS

Finally, METIS is a graph partitioning algorithm widely utilized for clustering and partitioning large graphs. Like label propagation and unlike Louvain, Paris and greedy modularity maximization, METIS does not directly optimize modularity. Instead it operates on the principle of multilevel graph partitioning: it aims to minimize a objective function, such as the edge-cut (the amount of edges that cross partitions regardless of weight) or the total communication volume (the amount of edges that cross partitions along with their weights), which measures

the quality of the graph partitioning. The algorithm uses a process with multiple phases, starting with a coarse-grained representation of the graph and then refining the partition to finer levels recursively.

As opposed to all other named algorithms, METIS has a single non-optional parameter. This is k, the amount of clusters it should output. METIS will try to balance the size of clusters to be approximately equal, while keeping the amount of clusters set to k.

Name	**Type**	**Considered parameters**	**Output**
Louvain	Community detection	Resolution	Set of nodes
Louvain (Iterative)	Hierarchical graph clustering	Resolution	Dendrogram
Label propagation	Community detection	n.a.	Set of nodes
Greedy	Community detection	n.a.	Set of nodes
Paris	Hierarchical graph clustering	n.a.	Dendrogram
METIS	Graph clustering	The amount of clusters k	Set of nodes

Table 2.1.: An overview of the partitioning algorithms

The algorithms mentioned here all allow us to partition graphs into lists of nodes. Louvain, Label Propagation, Greedy Modularity Maximization and METIS do this directly, but Paris and Iterative Louvain return dendrograms. Dendrograms still have to be processed further.

Since these algorithms work directly on a graph, the graph does not necessarily only have to contain symbolic dependencies. While Lutellier et al. (2015) determine that symbolic dependencies work better than file dependencies only, the comparison did not contain any other information. Information like naming could be used to enhance the graph by connecting nodes with similar names. Edges could also be annotated with the amount of times an edge is part of an execution path to produce weighted graphs. An example of this is the application of changes to methods in the same version using analysis of version control by Mattis (2018).

One aspect all these methods have in common is the need for existing source code in the target language. This is not necessarily the case in program synthesis. Evaluating these methods would require finding datasets that fit this requirement or generating our own dataset entirely.

3. Problem statement

In the previous chapter we identified a gap in current research related to identifying domains in the background knowledge and providing a program synthesis solver solely with the ones needed to find an optimal program. In this chapter we will formulate the problem more formally using the prior knowledge as a background.

ILP In this research we will focus on **inductive logic programming**. As described previously, inductive logic programming is a problem, where a program is synthesized that is accordance with a set of examples, positive and/or negative, using a background knowledge (or BK) using a grammar. Since we focus on ILP the grammars under consideration are logic programs. By default this problem has a binary solution: either the problem entails all examples or it does not. We propose that any ILP instance can be reformulated as a problem we call **Partition-selection Inductive Logic Programming** . This problem consists of three steps: partitioning, selection and searching.

Partitioning The first step is to split the background knowledge up into partitions using a *partitioning function* that takes the background knowledge and optional extra information as input and returns sets of functions from the BK, so that the union is the background knowledge. This definition does not exclude overlapping clusters, it only assumes that all functions of the original are covered. Each of these sets is a group of functions that belong together in some sense. We call this the **modularized background knowledge**. We also call each of these sets a **partition**.

Selection In order to use the modularized background knowledge, we need to choose the partitions that are *relevant*. A partition is relevant if a program that entails all examples contains a function in the partition. This selection is based off of the examples of the specific test instance we are considering. We view this as a a selection function that returns a subset of the modularized background knowledge.

Search Because we want to synthesize a program, the union of the selected partitions is then supplied to the synthesizer as background knowledge. The synthesizer is an arbitrary synthesizer, since we have no made any assumptions about its functionality. We merely assume that it takes examples and a background knowledge, which is a common definition (Cropper & Dumančić, 2022). Therefore the reformulation theoretically works with any ILP solver.

Time constraints Finally, we define a more constrained version of the problem where we impose a timeout T that applies to the partition and search simultaneously, but not the partitioning. That is, $Time(Select) + Time(Search) <= T$. We call this **Time-gated Partition-selection Inductive Logic Programming**.

The proposed task is a very general task that is underexplored in the current state of the literature, and in this thesis we develop an initial direction. Based on the prior work on program comprehension, we propose that one particular partitioning function that could prove useful is the use of graph clustering and/or graph community detection algorithms. We use these on a callgraph of existing programs in the same language. In order to keep our approach separate from the problem, we call this method of finding partitions and then selecting the partitions to combine in a search the **Partition-search procedure** throughout the thesis. This partition-search procedure is a proposed *approach* for Partition-selection Inductive Logic Programming and also for Time-gated Partition-selection Inductive Logic Programming if the selection and search adhere to the timeout. We focus on the time-gated variant.

We can now redefine our research questions using the established context. Our main research question can be reformulated as

Can inductive program synthesis approaches exploit partitions found through applying graph clustering on callgraphs from existing programs by applying a partition-search procedure in order to improve the synthesis of programs within a given time limit?

The subquestions can similarly be restated as the following:

1. In what ways can we vary the construction of a static callgraph from Prolog programs to use graph clustering algorithms on and does it effect performance of the partition-search procedure?
2. Does varying the graph clustering algorithm in the partition-search procedure cause a difference in performance of the final programs?
3. Does the effect of using a partition-search procedure differ between types of problems?

4. Methodology

In this chapter we establish the framework that we need in order to perform our experiments. To do this, it is essential to identify the variables that will need to be evaluated and define the algorithms that we are going to implement. In order to enumerate the possible variables, let us first consider the pipeline.

The datasets that we are going to need are in the form of logic programs. As stated previously, logic programs each contain rules containing sets of predicates and each predicate uses a set of child predicates. We can recursively find the child predicates up until a set of optional predefined *base* predicates, which are predefined as part of the background knowledge. These connections will be used to generate the callgraph, where the nodes are the predicates. Different topologies of connecting nodes based on these connections have been identified and are elaborated further upon in chapter 4.1.

This callgraph will then be processed using different graph algorithms to produce partitions of predicates. Each algorithm will have a different partition and therefore potentially a different performance. Which algorithms and how they are applied is elaborated upon in chapter 4.2. The resulting partitions are evaluated using a custom selection algorithm which is described in chapter 4.3.

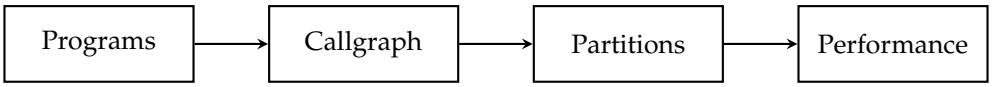


Figure 4.1.: A high level overview of the pipeline

4.1. Topologies

The next aspect is the construction of the static call graph. Each predicate with child predicates in the dataset contributes a few connections, but it is not immediately clear which way of contributing is the most optimal. Take the following predicate:

```
example(A, B) :- child1(A,B),
                 child2(A,B),
                 child3(A,B).
```

Three methods have been identified, each having an underlying viewpoint of connection behind it. The first is **co-occurrence**. Co-occurrence is when a predicate is used in the same parent as the other predicate. In our example predicate, this is the case for child1, child2 and child3 and the method is analogous to connecting only the children of the parent predicate together. The second method is based off of **usage**. For usage, two predicates get connected if

one predicated uses another. In the example that would mean the predicate *example* gets connected to each to the child predicates, but the child predicates would not be directly connected. Finally, there is the method of connecting for both usage and co-occurrence. This would mean the nodes form a **clique**. The three topologies are visualised in figure 4.2.

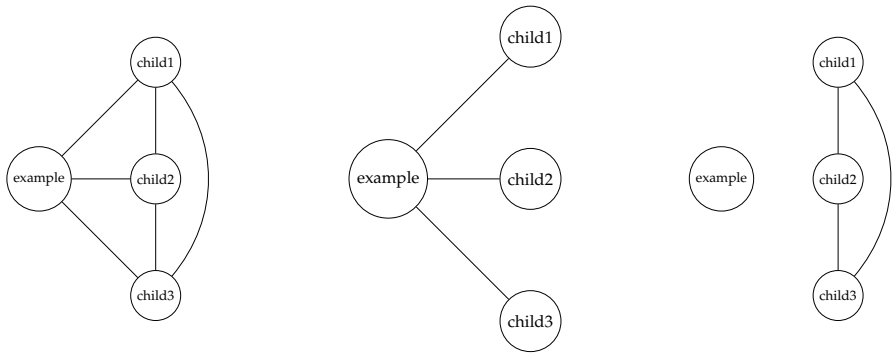


Figure 4.2.: Topologies for contributing nodes from a single predicate.
From left to right: clique, usage, co-occurrence

4.2. Graph clustering algorithms

Taking the list of algorithms found in the prior work, there are five algorithms to consider. From those we select nine total versions, based on parameters of resolution for Louvain and the method of cutting dendrograms from Paris and Iterative Louvain. For the Louvain resolution parameter we use a version where the resolution is 1 and one where it is 1.5. We also want to cut the dendrograms returned by the graph clustering algorithms into balanced size clusters. How large these clusters should be, depends on initial results.

Dendrograms As previously stated, dendrograms are a representation of multilevel clusters, showing which clusters get combined at certain distance threshold. By cutting at a particular distance threshold, partitions can be extracted. It is also possible to extract clusters with a maximum size by keeping track of the merges and the corresponding partition sizes. The custom algorithm used for this purpose can be found in listing 1 and it works generally like the following.

A dendrogram can be seen as a sequence of merges. We iterate through the merges going up in y-axis. Along the iteration, we keep track of how big each cluster gets after each merge and as long as it is below the allowed size, we continue. If the merge would create a cluster that is too large we ignore the merge, as it does not exist from our perspective. Any cluster that is merged with an ignored cluster *also* gets ignored because it too can not exist. This way we get all maximum possible clusters from the dendrogram for a given maximum size.

Algorithm 1: Merge clusters until maximum size algorithm

predicate merge_clusters(*linkage_matrix*, *nodes*, *max_cluster_size*)**:**

Input :

- *linkage_matrix*: A matrix representing the linkage information between clusters.
- *nodes*: A list of node identifiers.
- *max_cluster_size*: The maximum allowed size for a cluster.

Output: A list of clusters, where each cluster is represented as a set of node identifiers.

clusters ← empty dictionary;
current_cluster_i ← length of nodes;
ignored_clusters ← empty set;

for *merge* ***in*** *linkage_matrix* **do**

// A 'merge' object represents the merging of two clusters in the linkage matrix.
// It contains information about the indices of the two clusters being merged.

first_cluster_idx, second_cluster_idx ← convert merge to indexes of clusters to merge;

if *first_cluster_idx Graph* ***in*** *ignored_clusters* ***or*** *second_cluster_idx* ***in*** *ignored_clusters* **then**

add current_cluster_i to ignored_clusters;
current_cluster_i ← current_cluster_i + 1;
continue;

end

cluster_1 ← clusters[first_cluster_idx];
cluster_2 ← clusters[second_cluster_idx];

merged_set ← union of cluster_1 and cluster_2;

if *size of merged_set* $\leq$ *max_cluster_size* **then**

add merged_set to clusters with key current_cluster_i;
remove first_cluster_idx from clusters;
remove second_cluster_idx from clusters;

end
else

add current_cluster_i, first_cluster_idx, and second_cluster_idx to ignored_clusters;

end

current_cluster_i ← current_cluster_i + 1;

end

return list of values in clusters;

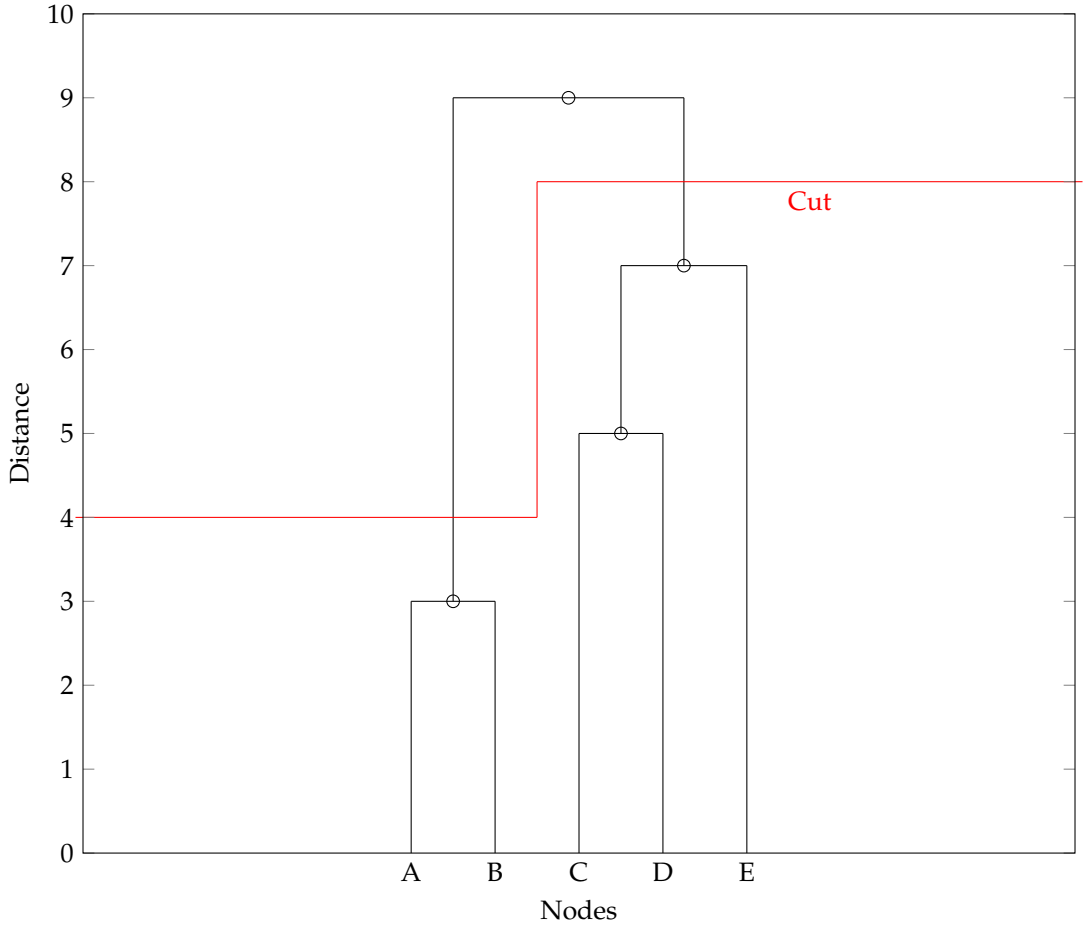


Figure 4.3.: The result of algorithm 1 applied on figure 2.6

In figure 4.3 we show an example result of the algorithm when applied to figure 2.6 with a maximum size of 3. In the figure the red line is the line where we cut our clusters. In this case, we do not allow the merge of the clusters after the initial merges, because the cluster would have a size of 5.

Edge weights All of the algorithms we consider can also take advantage of edge weights. If this is utilized, it is not merely the connections that determine the clustering, but also a specific variable on the edges between nodes. We will make use of this feature by setting the weight of the edge to be the amount of times that connection in contributed by the programs. Edges that rarely occur can otherwise cause the graph to be overly connected, and edge weights make sure that if that is the case, those connections are less important.

4.3. Selecting relevant partitions

Since this research is the first to apply partitions of background knowledge, there is no answer to how they are best selected. One readily apparent solution would be to apply the partition on its own in a search. According to prior research, useful predicates will give a signal in the form of a program that covers at least one example (Barke et al., 2020; Shi, Steinhardt, & Liang, 2019). However, it is not clear how this relates to predicates that need related predicates to provide a signal.

This is where our partitioning could be important: if we group related predicates into one partition and they are evaluated simultaneously, they should provide a signal because the required predicates are in the same group. If the result of a search with one single partition is a program covering even a single example, the partition is included in the combined search, otherwise it is not. We can utilize a small trick here as well: the search procedure for a single partition does not have to continue after finding a single example, making the most of the time provided for the remaining cluster.

The individual timeout imposed upon the selection search is still left unclear as well. We propose evaluating each partition with a timeout proportional to the ratio between the length of the partition to the remaining amount of predicates left to be evaluated. Without any change this would have the nice property of having a fixed upper bound. However, to avoid having small partitions being deprived of the time they need to actually run and because scheduling small time intervals is difficult on a busy supercomputer, we maintain a lower bound on the time scheduled for each partition of 5 seconds.

The side effect is that the selection process may go over its own time limit, as the individual partitions may take more time than the linear schedule would have allowed. This is not necessarily a problem as long as it does not go over the total time limit of the whole partition-search procedure. If this happens, we we will reevaluate the lower bound. Practically, the only reason this would happen is in the case where there the composition of the partition sizes is such that many of the scheduled times would be below the lower bound in a row. This would not be a very practical partitioning in the first place, since it involves a lot of tiny clusters. Regardless, there is no 'right' answer here, so we choose to favor the runtime of smaller partitions and more stable scheduling versus having more search time for the combined result.

Finally, we set the time limit of the selection process to half that of the timeout for the whole partition-search. We set that to 30 minutes, so the selection process will have 15 minutes as a base time limit, which it can exceed if there are a lot of small clusters.

Based on this information, we construct the algorithm in listing 2. We call this algorithm the **linear time evaluation selection function**.

Algorithm 2: The linear time evaluation selection function

Input : List of partitions *partitions*
Output: List of useful partitions

$total\_length \leftarrow 0$;
$useful\_partitions \leftarrow$ empty list;
foreach *cluster in partitions* **do**
  $total\_length \leftarrow total\_length + cluster.length$;
end
$time\_left \leftarrow TIMEOUT$
$functions\_left \leftarrow total\_length$
foreach *partition in reversed(sort_by_length(partitions))* **do**
  $timeout\_for\_partition \leftarrow max(\frac{cluster.length}{functions\_left} \times time\_left, 5)$;
  $time\_before\_search \leftarrow$ now();
  $contains\_useful\_predicate \leftarrow$ ApplySearch($partition$, $timeout\_for\_partition$);
  $time\_after\_search \leftarrow$ now();
  $time\_left \leftarrow time\_left - (time\_after\_search - time\_before\_search)$;
  $functions\_left \leftarrow (time\_after\_search - len(partition)$
  if *contains_useful_predicate* **then**
    $useful\_partitions$.append($partition$);
  end
  $total\_length \leftarrow total\_length - cluster.length$
end
return $useful\_partitions$;

Obviously this is not the only way to construct a similar algorithm. The timeout can be based on a heuristic as well or even just have a exponential relationship with cluster size instead of a linear relationship. It can also use information from the problem under consideration. The goal of this thesis is not to optimize every step, so we will consider this single case and evaluate this method, keeping time constraints in mind as well.

4.4. Synthesizers

Finally, we consider the synthesizer that serves as the search function in the partition-search procedure. As stated in the methodology, the partition-search procedure should in theory work with any synthezier. In this thesis we will focus on a singular solver, Popper, since it is state-of-the-art and focusing on a single solver limits the amount of options. Popper uses Prolog and therefore we will use Prolog exclusively as well. Popper also returns programs that are not optimal (i.e. they only cover a subset of the examples), which is useful for finding the aforementioned signal. Using this, we will make two modifications to Popper in order to allow for more extensive logging and to allow for breaking out of the search on the first found program that satisfies any positive number of examples like mentioned in the previous chapter.

5. Results

This chapter contains the resources we obtained in order to perform the experiment described in chapter 4 and the corresponding results. The experiments have been performed in discrete blocks and will be described chronologically. As a reminder, the research questions are

Can inductive program synthesis approaches exploit partitions found through applying graph clustering on callgraphs from existing programs by applying a partition-search procedure in order to improve the synthesis of programs within a given time limit?

With the subquestions:

1. In what ways can we vary the construction of a static callgraph from Prolog programs to use graph clustering algorithms on and does it effect performance of the partition-search procedure?
2. Does varying the graph clustering algorithm in the partition-search procedure cause a difference in performance of the final programs?
3. Does the effect of using a partition-search procedure differ between types of problems?

5.1. Datasets

Since the clustering techniques we are focusing on rely on call graphs existing, appropriate datasets are needed. These datasets should either contain programs that solve the same task being addressed or provide a means to generate them manually. Two potential sources for these datasets have been identified: one that is readily available and another that requires manual generation.

5.1.1. Playgol

The one dataset that is readily available is the Playgol dataset (Dumancic & Cropper, 2020). This dataset was generated during the testing of a program synthesis method and consists on programs synthesised with predicate re-use on two problem types: string manipulation and a lego structure building problem. These are Prolog programs and as such consist of functions, complete with the definitions and including child functions, that are needed to generate a static call graph. Test cases are also part of the dataset, making it an ideal candidate for evaluation. An example of the syntax can be seen in figure 5.1.

Each problem type within the Playgol dataset consists of 10 variations. Snapshots of these variations were taken during the generation process at intervals of 200 added tasks (which they call 'play tasks') that they generate programs for, ranging from sets of 200 to 4000 tasks.

```
...
p142(A,B):-skip1(A,C),mk_uppercase(C,B).
p154(A,B):-not_empty(A),copy1(A,B).
p158(A,B):-not_empty(A),skip1(A,B).
p159(A,B):-copy1(A,C),mk_uppercase(C,B).
p163(A,B):-not_empty(A),copy1(A,B).
p165(A,B):-not_empty(A),skip1(A,B).
p168(A,B):-not_empty(A),mk_lowercase(A,B).
p170(A,B):-not_empty(A),copy1(A,B).
p184(A,B):-mk_uppercase(A,C),mk_lowercase(C,B).
p185(A,B):-not_empty(A),copy1(A,B).
p195(A,B):-not_empty(A),mk_lowercase(A,B).
p196(A,B):-mk_uppercase(A,C),copy1(C,B).
p1(A,B):-mk_lowercase(A,C),p34(C,B).
p2(A,B):-copy1(A,C),p2_1(C,B).
p2_1(A,B):-p43(A,C),p34(C,B).
p3(A,B):-copy1(A,C),p34(C,B).
p13(A,B):-p34(A,C),p13_1(C,B).
p13_1(A,B):-p133(A,C),copy1(C,B).
p15(A,B):-p34(A,C),copy1(C,B).
...
```

Figure 5.1.: Programs sampled from the first problem type of the Playgol dataset: string manipulation

The included test set is known to be solvable (since it was generated using a known process) but has not been solved in its entirety by the original paper. In order to make evaluation easier (because there at least should be a known solution), we will sample from the solved test instances as our own testing set.

Task description

The task for these datasets are exactly the same as in the original paper: use the programs in a given datasets to construct a *new* program that entails the examples of a test instance. We do not have to do any modification to the data to make that possible.

Limitations and improvements

In addition to the data volume, there are other noteworthy characteristics to consider. Alongside the reused functions, each problem type has a set of base functions that are specific to that problem type. Whether it is preferable to always supply these base functions to the solver is not clear: they are necessary in case the specific combination of base functions is not available, but it could cause all clusters to be marked as relevant.

Furthermore, the way the programs in the dataset are generated is of concern. Because of the use of predicate re-use, the functions can only use functions that have been generated

beforehand. This is important to note, because it impacts the way callgraphs are generated. In particular, the final functions of a given variation have little to no chance to be used, which can cause disconnected components and connections with limited weight in the callgraph.

Finally, there is a high amount of duplicate functions within the datasets. To optimize the datasets, a specific step of *pruning* is employed at the beginning.

Pruning The method we call pruning involves removing duplicate functions, by matching the names of the child nodes and replacing them with a single prototype. This algorithm consists of iterations, because after finding functions that match and replacing them, new duplicates can become clear. We only use a single iteration, because it already cuts the amount of functions significantly and we want enough functions to be able to find a potential difference in performance.

This effectively creates two datasets from a single one, as the removal of first order duplicates is a significant change in characteristics of the dataset and the amount of data decreases substantially. In consideration of time constraints we only consider the datasets *after* they are pruned. The pruning algorithm can be found in listing 3.

Algorithm 3: The pruning algorithm for Playgol

```
Result: node_to_definition, node_to_line
Input  : dataset, settings
Output: node_to_definition, solved_test_fns, node_to_line
Initialize node_name_to_definition, solved_test_fns, node_name_to_line, used_by,
 to_delete;
foreach line in dataset do
    result ← parse_line_to_node_name_and_children(line);
    if result is not None then
        (node_name, children) ← result;
        Add node_name and children to node_to_definition;
        foreach child in children do
            Add node_name to used_by[child];
        end
    end
end
Initialize invariant_to_node_name;
foreach node_name in keys of node_to_definition do
    if node_name is in BASE_FN then
        Continue to next iteration;
    end
    children ← node_name_to_definition[node];
    invariant ← join children with "-";
    if invariant is in invariant_to_node_name then
        parent ← invariant_to_node_name[invariant];
        foreach usage in used_by[node_name] do
            Update node_name_to_definition[usage];
        end
        Add node_name to to_delete;
    end
    else
        Add node_name to invariant_to_node;
    end
end
foreach node_name in to_delete do
    Remove node_name from node_name_to_definition;
end
Return node_name_to_definition, node_name_to_line
```

Constructed datasets

As stated previously, we do not have to do any modification to the datasets to make them useful in the context of our task. However, altogether these datasets form a large volume of data. A majority of the data from Playgol is not very informative: the variations do not provide a significant amount of value as Dumancic and Cropper (2020) used the same underlying examples for each variation. We will only use two variations per domain during evaluation:

one for the gridsearch and one for the larger run. Relative data sizes can be seen in figure 5.2. We will evaluate the dataset at T=600 and only after pruning.

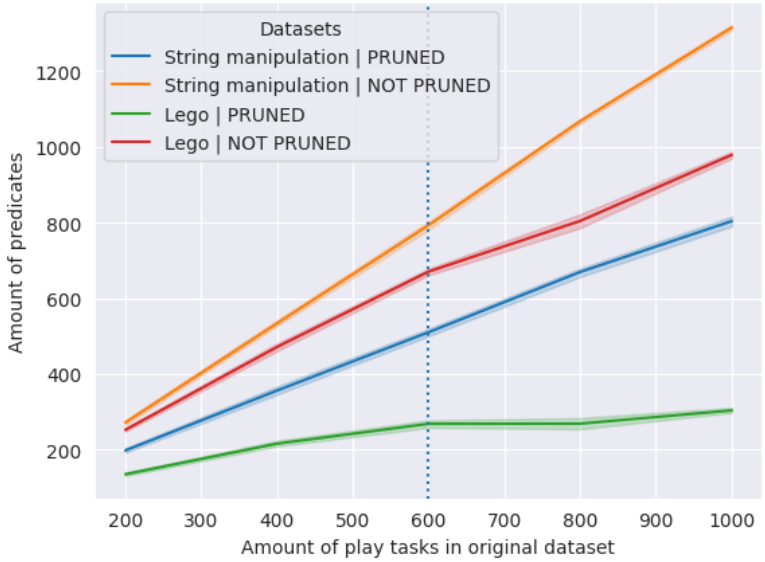


Figure 5.2.: Sizes in terms of functions for the Playgol dataset for the first variations with and without pruning. Error bars indicate 95% confidence interval over all variations constructed using bootstrapping. Dotted line indicates point where data was sampled.

5.1.2. Knowledge graphs

Knowledge graphs are directed graphs where the nodes are entities and the edges are relationships between those entities. For example, the NATION dataset (Dettmers, Minervini, Stenetorp, & Riedel, 2017) contains nodes for countries like the Netherlands, the UK, the USA and Indonesia, and edges for relationships indicating, for example, whether a country provides economic aid or has an embassy to the other country. These are encoded as a list of triples, consisting of an entity, a relation type and a target node to connect.

Task description

Knowledge graphs have been used for a task called *rule learning*, an example of which is an approach called AnyBURL (Meilicke, Chekol, Fink, & Stuckenschmidt, 2020). In essence, this process aims to 'explain' a target relationship using the other relationships present in the graph. AnyBURL generates a set of rules that covers one relation using the others in a language similar to Prolog. These rules can also be used to construct our needed callgraph, illustrating connections between relations.

Furthermore, ILP solvers can be set up to perform the same task. Initially a small subset of relations is extracted from the graph: the entities connected by these relations form a set of positive examples. Each relationship in the graph can be considered as a predicate in

the background knowledge, evaluating to true if the two entities are connected through the corresponding relationship and false otherwise. The target predicate should exhibit the same behavior: evaluating to true when provided with the entities in the positive examples and false for the negative.

Limitation and improvements

In order to accurately evaluate performance gains using the partition-search procedure, it would be useful to actually be able to generate some programs in the baseline. Initial results indicated that Popper needs more information in order to efficiently generate programs from knowledge graph triples. One way to make Popper more efficient, is to use negative examples. Popper uses these examples to constrain its internal hypothesis space (Cropper & Morel, 2020). Negative samples are not part of the triples themselves, but they can be generated. This is called *negative sampling* (Y. Zhang, Yao, & Chen, 2021).

Negative sampling We have implemented a negative sampling scheme using the following algorithm. First, we inventarise all entities from the triples in the original dataset. Then we sample a random pair from these entities for each relationship and if they are not a positive relationship, we add it as a negative example. Otherwise we just keep sampling. There is no correct choice as to how many negative samples are needed, however we chose to generate 10 times as many negative examples as there are positive examples per relationship type. That means that if the aforementioned 'embassy' relationship has 5 positive triples, we generate 50 negative triples as supplementary examples.

Constructed datasets

In order to generate a dataset fit for use in evaluation of the partition-search procedure, we will draw a percentage specific to the dataset of the total amount of relations as the test set of the knowledge graph. These relations will be removed prior to the rule learning using AnyBURL in order to avoid overfitting on structures present in the rules as an extra precaution. Afterwards we have generated a dataset of these rules using AnyBURL for 500 seconds.

Two useful knowledge graphs have been identified, with the main differences being in entity and relationship count. Since the amount of relationships is also the amount of functions, this is the count that is most important for the size of the search tree. References to the knowledge graph datasets along with their corresponding counts of relationships and entities can be found in table 5.1. Based on these knowledge graphs, we have constructed three datasets, one from UMLS and two from FB15K-237 with 5, 5 and 15% of the relationships as test sets respectively. We create them independently, with splitting different random samples for the test examples before rule learning.

Dataset	Entities	Relationships	% Taken for datasets
FB15K-237 (Toutanova & Chen, 2015)	14,505	237	5% (11 functions), 15% (33 functions)
UMLS (Dettmers et al., 2017)	134	48	5% (7 functions)

Table 5.1.: Summary of the datasets that were constructed from the knowledge graphs

5.2. General experimental setup

In this chapter, we will summarize and outline the general experimental setup that we re-use throughout our experiments, based on the techniques we have described in prior chapters. All of this is evaluated on the DelftBlue supercomputer (Delft High Performance Computing Centre (DHPC), 2022).

5.2.1. Data processing and partitioning

For each dataset we have two pieces we want to acquire: test instances in the form of positive and negative example and a modularized background knowledge. We will describe the processing pipeline per dataset.

Knowledge graphs For knowledge graphs we do not have predefined test instances in a format that is useful for our current task. Instead, they are pre-split into test-, validation and testsets where the relations overlap. We instead want sets in such a way that relations are separated. In order to achieve tha we need to split the original datasets ourselves. We concatenate all triples and select a percentage specific to the dataset as test instances (defined in table 5.1). For those instances, we select the entities that were bound by that relationship and keep them as the tests. Finally we apply negative sampling (chapter 5.1.2) to generate negative examples.

To generate the modularized background knowledge, we generate rules using AnyBURL on the split training set. Then we construct a callgraph from these rules using a predefined topology and generate partitions using a clustering algorithm, both based on the experiment at hand.

Playgol For Playgol the process is less involved. We already have a sufficient testset as part of the original dataset. but we still filter based on whether the tests were solved in the original paper as described in chapter 5.2.1. We also have a set of programs. On these programs we apply the prior defined pruning (chapter 5.1.1). Then we construct a callgraph with a given topology and generate partitions using the predefined clustering algorithm just like with the knowledge graphs.

5.2.2. Selection

After this, we apply the selection function. This returns a subset of the partitions that we want to search through along with a remaining time for the final search. If this is less than or equal to zero (e.g. it took exactly all or more than the specified time for selection) we will count this as a failed test instance. In case we use the linear time evaluation selection function we also find the duration of running and reasons for stopping the evaluation (timing out or finding a program covering one or more examples).

5.2.3. Search

With the union of the selected partitions as the background knowledge, we run a search using Popper for the specified amount of time. Popper returns the best program found during its runtime, along with the amount of true positives, false positives, true negatives and false negatives of the examples. Obviously it can also fail to find a program, in which case treat it as an empty program, covering no examples.

5.2.4. Evaluation

The evaluation will we done based on the recall of the program. The recall is defined as

$$\frac{TP}{TP + FN}$$

where TP is the number of true positive and FN is the number of false negative and it specifies how many of the initial positive examples were 'recalled' by the algorithm. We only report the gain in recall, because Popper always makes sure that the precision is 1, i.e. it never returns solutions with false positives.

In order to account for stochasticity, we take the test set of the corresponding datasets and run the entire procedure with a random seed five times. From these five times, we select the program with the *maximum recall*. An example can be seen in figure 5.3.

We then compare this single value to a baseline using a standard search with the complete background knowledge for the entire timeout, where we also select the maximum recall over five repetitions. We get the difference between the two values by subtracting the maximum baseline from the maximum of our experiment and use these values as the basis for our further calculations described in the experiments.

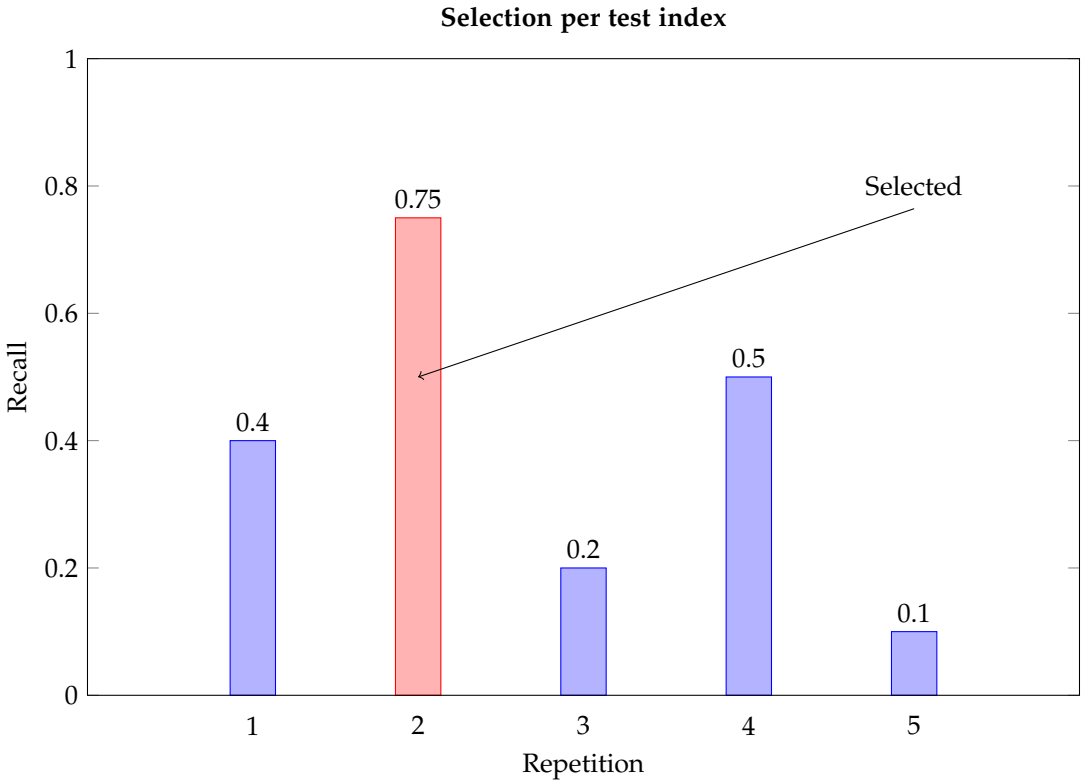


Figure 5.3.: Selection for a single test index.

5.3. Experiments

In this chapter we will discuss the experiments that we have performed in order to ask our research question. We have performed three experiments in total: parameter discovery on UMLS, a gridsearch on FB15k-237, Playgol-string and Playgol-lego and finally a larger evaluation on more points to get an indication of generalization.

5.3.1. Parameter discovery

The first experiment is solely on UMLS and has two goals: we want to discover useful values for the sizes of the graph clustering algorithms and we want to know how the partition-search procedure performs on UMLS.

Experiment setup Since the parts of the partition-search procedure can not be evaluated independently, the variables will be evaluated in a grid search. Initially we have run only the community detection algorithms, (Louvain, Label propagation and Greedy modularity communities). We will use the data about performance to inform the choice on how large the graph clustering partitions should be.

Results The results for the clique topology of the run can be seen in figure 5.4. The others can be seen in appendix A, but they are not significantly different to the clique topology. As can be seen, the results are either equal to or worse than baseline. The sole exception here is the label propagation algorithm, which is never worse. This is, however, to be expected, since the resulting clusters for all topologies with label propagation are a single cluster with all nodes. While this is not the intended result of a graph clustering algorithm, it apparently does perform the best, so it could be construed as being correct.

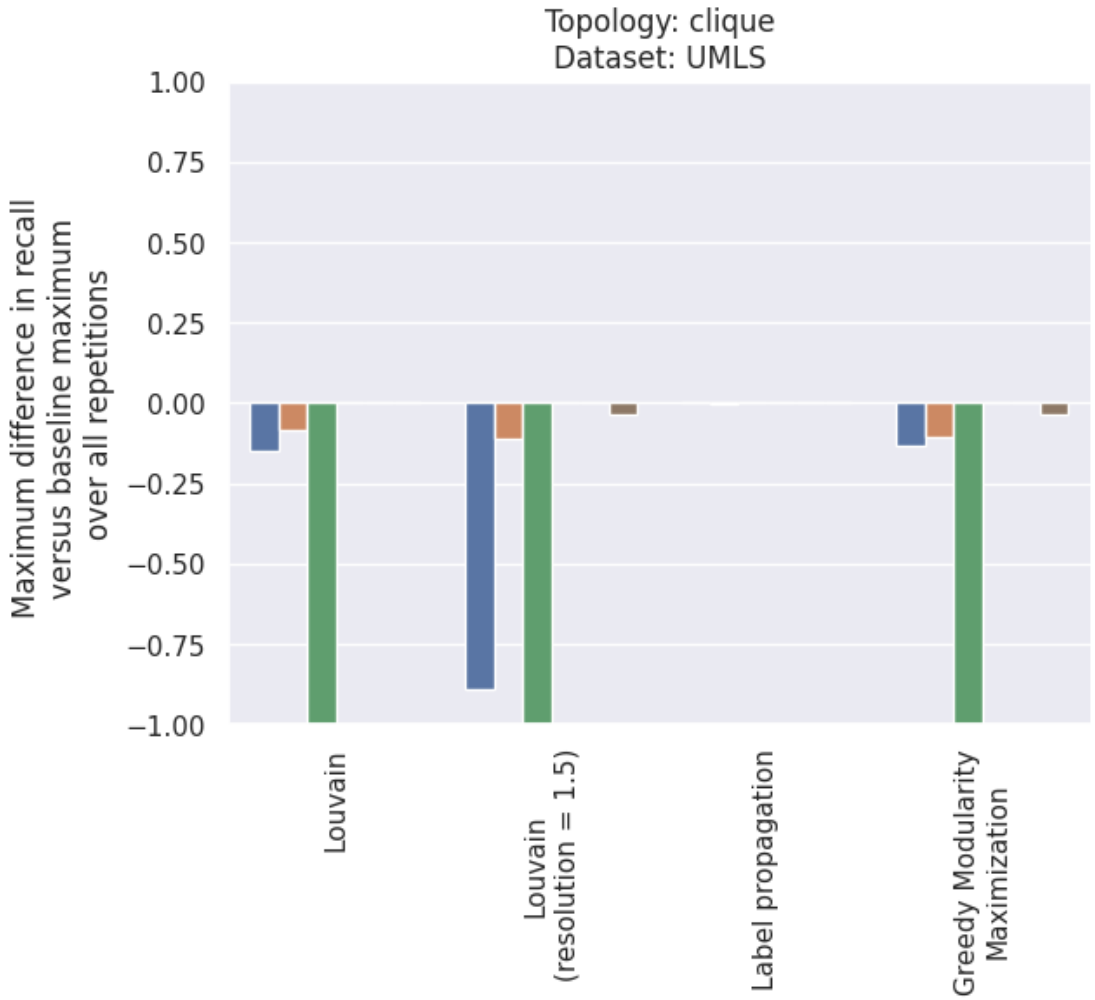


Figure 5.4.: Results of the evaluation for UMLS with community detection algorithms. Higher is better. Results show a lack of recall gain and major decreases on several test instances.

The fact that the others primarily perform worse than baseline is also not expected. In general, there are two reasons why the search process would result in a worse background knowledge: a partition that contains a function used for the baseline solution either timed out while searching for a signal or it failed to find a signal in its search space. In the first case the search for the partition would not have enough time to generate a signal. In the second the partition

misses functions that it needs in order to generate the signal using the available functions in that particular partition: it is a failure of structure.

By analyzing the selections that do not produce the best programs, we can find partitions that contain functions of the better program, but were not selected. We can also find the reason why it was not selected. After grouping them by partition length, we find the distribution in figure 5.5. When the partition is larger than 8 functions, generally the function will time out, with the exception of length 12, which unexpectedly break this pattern. This is unfortunate, because it limits the differences between possible structures that we can evaluate.

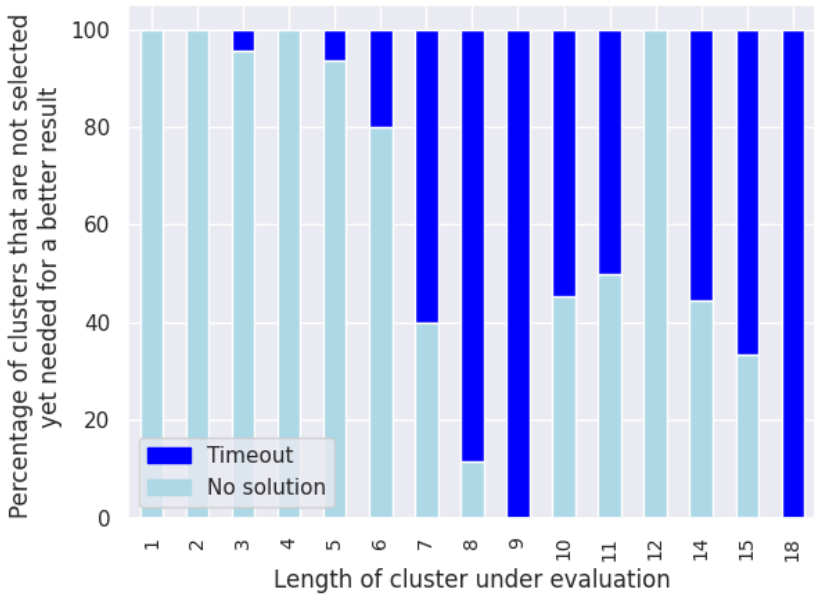


Figure 5.5.: Reasons for not finding crucial partitions during selection for UMLS. Results show timeouts as a majority for partition lengths as short as 9 functions.

Based of these results, we chose to limit the size of graph clustering approaches to two variations with a maximum of 5 and 10 functions respectively. METIS also has been limited to clusters of size 5. The results for this run can be seen in figure 5.6. They perform relatively worse to the community detection algorithms for all topologies in terms of recall gain, but some test instances actually show a tiny improvement. Furthermore, since the partitions are smaller, the amount of timeouts is lower overall. In order to be able to compare to these results, we use the same configuration for graph clustering for the remaining datasets, as has been stated in chapter 4.

In conclusion, for UMLS the data suggests that the problem itself is not a good fit for using community detection and/or graph clustering as a heuristic for partition-search within the current configuration of the problem (e.g. a timeout of 30 minutes and the linear timeout in the selection process).

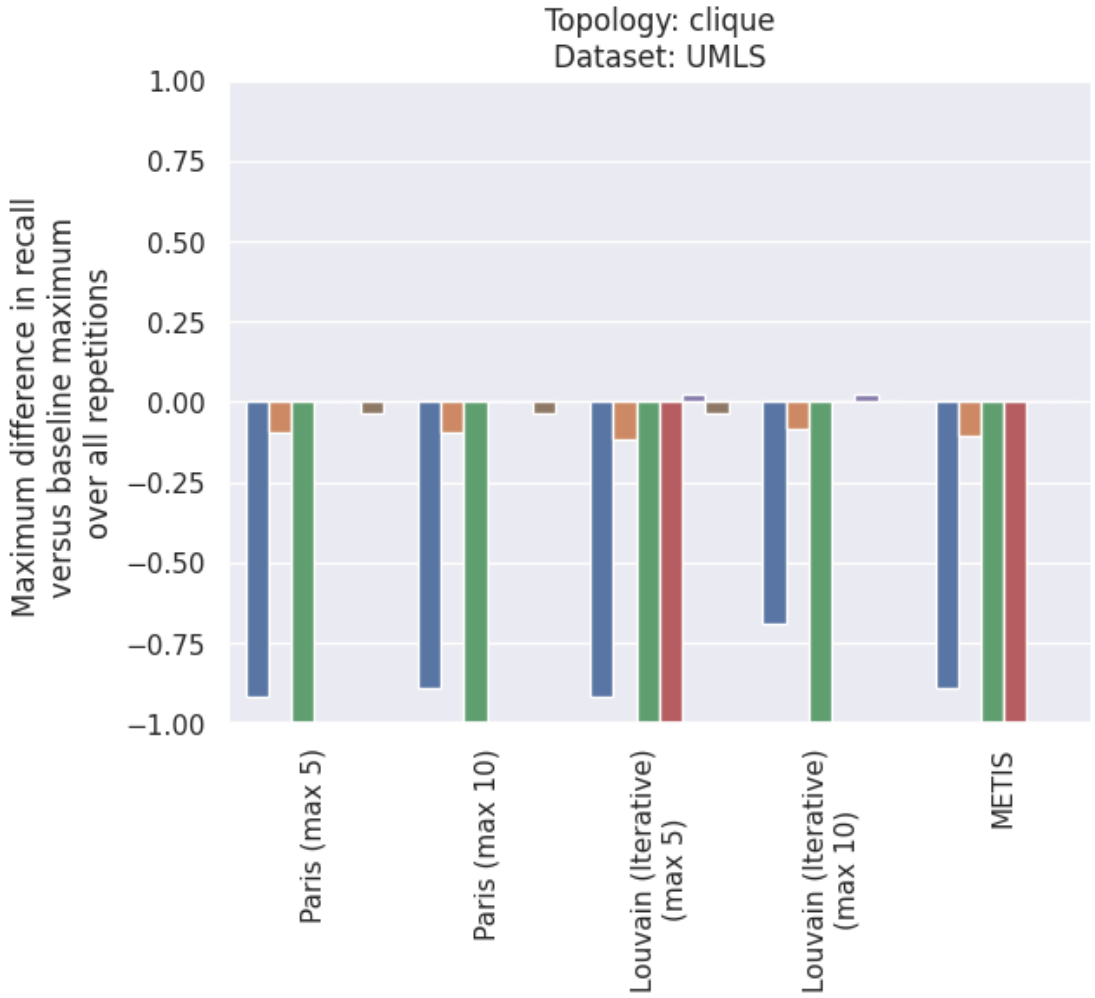


Figure 5.6.: Results of the evaluation on UMLS for graph clustering algorithms. Higher is better. Results show a lack of recall gain and major decreases on several test instances.

5.3.2. Grid search

The next experiment aims to investigate how the partition-search procedure performs on our larger datasets.

Experiment setup In this experiment we run our general setup described in chapter 5.2.4 on the rest of the datasets (FB15k-237, Playgol-lego and Playgol-string). We also use the same sizes for the graph clustering algorithms found in the first experiments.

In order to aggregate the results, we report the results averaged in table 5.2. Since we are only interested in the gain and not necessarily any drops in performance, we drop any datapoint where the difference between the experiment and the baseline (calculated from the maximum

per repetition as described in chapter 5.2.4) is positive. We take the mean of the test instances that we did not drop for each approach. We also note the amount of test instances that the partition-search procedure impacts positively. The combination of these values can be used to gauge improvements.

Using a combination of values also makes interpreting the results tricky, since it is not clear what 'improve' means in this context. One approach can increases the recall for a single test instance, while another can increase it over multiple test instances for a lower amount.

FB15-237

Running the grid search on FB15-237, we obtain the results in table 5.2. To get an impression of the performance per test instance, we also show the clique topology in figure 5.7. The other figures showing the results per test instance can be found in appendix A. The partition-search procedure here actually has an higher performance in recall for a significant number of test instances. Besides that, only a few instances have a drop in recall. This is in stark contrast to the previous dataset, where there was not a test instance with any significant increase.

If we use either the mean recall gain and amount of improved test instances as objective values, the best performing approach is the Louvain algorithm with a clique topology. This is encouraging for the viewpoint that modularity makes for a useful heuristic when applying the partition-search procedure. METIS and both versions of Paris also seem to be severely under performing. The reason for this is not readily apparent, but one hypothesis is that the partition sizes are not optimal for this specific context since both are bound to relatively small clusters.

Luckily, even without the larger evaluation we can already draw a few conclusions. With the results of the gridsearch we prove that it is possible to improve the performance of search-based program synthesis using the partition-search procedure. These results also show it can differ per type of problem, as it improves the FB15-237 rule learning problem, but not for UMLS. Finally, we show that, at least on a small scale, the partitioning algorithm makes a measurable difference on the improved performance.

For the larger run we select **(Clique, Louvain)**, **(Clique, Greedy Modularity Maximization)** and **(Usage, Greedy Modularity Maximization)** as the stronger three and **(Co-occurrence, METIS)**, **(Co-occurrence, Paris (max 5))** and **(Usage, Louvain (iterative) (max 5))** for the weaker approaches. The selection criteria are based on the extremes of the recall gain, spreading out over the three topologies in order be able to see changes in the hierarchy better.

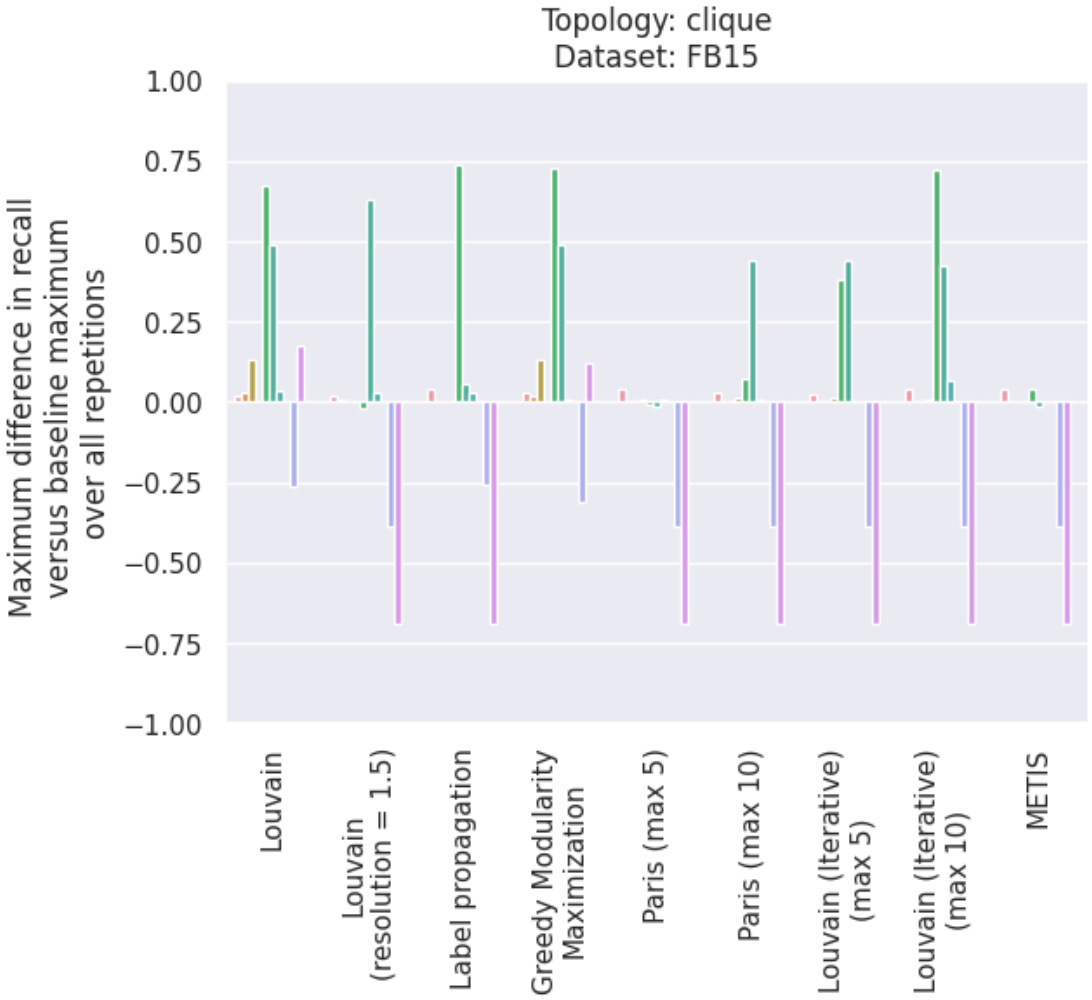


Figure 5.7.: Results of the gridsearch on FB15k-237 grouped by algorithm; each bar is a test relation and the color consistently matches a specific test instance over each group. Higher is better. Results show a high potential in recall gains which is not consistent over all test instances.

Topology		Clustering algorithm	Mean of best improvement	# of test instances improved out of 10
Clique	+	Greedy Modularity Maximization	0.191 (+= 0.27)	**8**
	+	Louvain	**0.195** (+= 0.25)	**8**
		Louvain (resolution = 1.5)	0.114 (+= 0.25)	6
		Label propagation	0.124 (+= 0.27)	7
		Louvain (Iterative) (max 5)	0.144 (+= 0.21)	6
		Louvain (Iterative) (max 10)	0.180 (+= 0.28)	7
		METIS	0.016 (+= 0.02)	5
		Paris (max 5)	0.013 (+= 0.02)	4
		Paris (max 10)	0.081 (+= 0.16)	7
Usage	+	Greedy Modularity Maximization	**0.182** (+= 0.30)	**7**
		Louvain	0.180 (+= 0.28)	**7**
		Louvain (resolution = 1.5)	0.154 (+= 0.25)	**7**
		Label propagation	0.047 (+= 0.10)	6
	−	Louvain (Iterative) (max 5)	0.009 (+= 0.01)	6
		Louvain (Iterative) (max 10)	0.145 (+= 0.22)	**7**
		METIS	0.081 (+= 0.18)	6
		Paris (max 5)	0.013 (+= 0.02)	4
		Paris (max 10)	0.164 (+= 0.27)	**7**
Co-occurrence		Greedy Modularity Maximization	0.140 (+= 0.25)	8
		Louvain	0.156 (+= 0.23)	**8**
		Louvain (resolution = 1.5)	**0.178** (+= 0.29)	7
		Label propagation	0.062 (+= 0.08)	**8**
		Louvain (Iterative) (max 5)	0.154 (+= 0.22)	6
		Louvain (Iterative) (max 10)	0.171 (+= 0.28)	7
	−	METIS	0.080 (+= 0.16)	5
	−	Paris (max 5)	0.010 (+= 0.01)	6
		Paris (max 10)	0.124 (+= 0.21)	**8**

Table 5.2.: Results of the gridsearch on FB15-237 for values where the difference in recall in relation to the maximum of the baselines for that test instance is positive. Values are calculated over the set of maximum values over 5 repetitions. Bold means best value for topology in column. + means selected as top 3, − means selected as bottom 3.

Playgol: lego

The first problem type of the Playgol dataset we evaluate is the lego structure building task, where the functions define movement and placing blocks.

For both Playgol datasets we have chosen to not include the base functions during the selection process search. Initial tests show that many more partitions are selected as useful, since the base functions are useful very often. After all, the other functions are build from them. We only include the base functions in the final combined search.

The results of the gridsearch can be found in table 5.3. Like in table 5.2, we limit the values to test instances where there is a recall *gain*. Since that only happens for a small subset of the clustering algorithms, we only report those. The results for this dataset is quite different from the results in the knowledge graphs. This can most likely be attributed to the fact that the data generating process is well defined (as they were generated by software) causing a program to be either correct or false without any in-between. This is opposed to the relatively messy real life connections in the knowledge graphs.

Label propagation with a clique topology stands out as solving the most test instances. This is interesting, since it does not directly maximize modularity and with that deviates from the FB15-237k results, hinting towards the idea that modularity based approaches do not always provide the best results.

Finally a key point to mention is that the baseline actually did not find a program that covers a single example across all test instances. The relative gains are also absolute gains in table 5.3.

Topology		Clustering algorithm	Mean of best improvement	# of test instances improved out of 10
Clique	+	Greedy Modularity Maximization	**1.000** (+= 0.00)	2
	+	Label propagation	**1.000** (+= 0.00)	**3**
		METIS	**1.000** (+= 0.00)	1
Usage	−	Greedy Modularity Maximization	**1.000** (+= 0.00)	1
	+	Louvain	**1.000** (+= 0.00)	**2**
		Louvain (resolution = 1.5)	**1.000** (+= 0.00)	1
		Label propagation	**1.000** (+= 0.00)	1
	−	Louvain (Iterative) (max 5)	**1.000** (+= 0.00)	1
		Louvain (Iterative) (max 10)	**1.000** (+= 0.00)	1
Co-occurrence	−	Louvain	**1.000** (+= 0.00)	**1**
		METIS	**1.000** (+= 0.00)	**1**

Table 5.3.: Results of the gridsearch on Playgol-lego after pruning for values where the difference in recall in relation to the maximum of the baselines for that test instance is positive. Values are calculated over the set of maximum values over 5 repetitions. Bold means best value for topology in column. + means selected as top 3, − means selected as bottom 3.

With all these approaches performing similarly in terms of mean recall gain, we'll select the top three based on the average number of improved test instances: **(Clique, Greedy modularity maximization)**, **(Clique, Label propagation)** and **(Usage, Louvain)**. The set of weaker

approaches is more arbitrary in this context, but we'll use **(Usage, Greedy modularity maximization)**, **(Usage, Louvain iterative (max 5))** and **(Co-occurrence, Louvain)**. We have also marked these in the table with $+$ for the top three and $-$ for the bottom three.

Playgol: string manipulation

Finally we have the results of the string manipulation problem type. Like with the previous dataset, we see sizable increases in performance. The *Co-occurrence* topology is performing considerably worse to the other two topologies. In hindsight this is not entirely unexpected, since the definition of a predicate only happens once for Playgol and so the head needs to be used in order to be well integrated in the callgraph. This does not always happen and functions that are defined later have little to no chance to be used. This is different for the knowledge graphs, because AnyBURL finds many different rules explaining the same functions. The same is true for the lego problem type, but there it is not as apparent. Another point of note is that the maximum increase of all these approaches, Iterative Louvain with size 10, actually solves all test instances perfectly.

The top three we pick from these datasets are **(Usage, Louvain Iterative (max 10))**, **(Clique, Louvain iterative (max 10))** and **(Usage, Louvain))**. For the the bottom three we ignore the *Co-occurrence* topology, because of the problems discussed prior are not based on the test instances, but a fundamental issue with the data generation process. From the remaining approaches we choose **(Clique, METIS)**, **(Clique, Greedy modularity maximization)** and **(Usage, Paris (max 5))**.

Topology		Clustering algorithm	Mean of best improvement	# of test instances improved out of 10
Clique	−	Greedy Modularity Maximization	0.400 (+= 0.52)	3
		Louvain	0.400 (+= 0.52)	3
		Louvain (resolution = 1.5)	0.400 (+= 0.52)	3
		Label propagation	0.400 (+= 0.52)	3
	+	Louvain (Iterative) (max 5)	**0.800** (+= 0.17)	3
		Louvain (Iterative) (max 10)	**0.800** (+= 0.17)	3
	−	METIS	0.325 (+= 0.45)	**4**
		Paris (max 5)	0.400 (+= 0.52)	3
		Paris (max 10)	0.400 (+= 0.52)	3
Usage		Greedy Modularity Maximization	0.550 (+= 0.52)	**4**
	+	Louvain	0.550 (+= 0.52)	**4**
		Louvain (resolution = 1.5)	0.550 (+= 0.52)	**4**
		Label propagation	0.400 (+= 0.52)	3
		Louvain (Iterative) (max 5)	0.400 (+= 0.52)	3
	+	Louvain (Iterative) (max 10)	**0.850** (+= 0.17)	**4**
		METIS	0.550 (+= 0.64)	2
	−	Paris (max 5)	0.400 (+= 0.52)	3
		Paris (max 10)	0.550 (+= 0.52)	**4**
Co-occurrence		Greedy Modularity Maximization	**0.100** (+= 0.00)	**2**
		Louvain	**0.100** (+= 0.00)	**2**
		Louvain (resolution = 1.5)	**0.100** (+= 0.00)	**2**
		Label propagation	**0.100** (+= 0.00)	**2**
		Louvain (Iterative) (max 5)	**0.100** (+= 0.00)	**2**
		Louvain (Iterative) (max 10)	**0.100** (+= 0.00)	**2**
		METIS	**0.100** (+= 0.00)	**2**
		Paris (max 5)	**0.100** (+= 0.00)	**2**
		Paris (max 10)	**0.100** (+= 0.00)	**2**

Table 5.4.: Results of the gridsearch on Playgol-string-manipulation after pruning for values where the difference in recall in relation to the maximum of the baselines for that test instance is positive. Values are calculated over the set of maximum values over 5 repetitions. Bold means best value for topology in column. + means selected as top 3, − means selected as bottom 3.

Timeouts and scheduling

One method to explain the differences in results between the datasets in the gridsearch, is by using the same type of plots as in figure 5.5. For UMLS we showed that, of the missed useful partitions, larger partitions more often lead to timeout. It was also based on this that we chose to limit graph clustering techniques to 5 and 10 functions per group. In retrospect, with new data and visualisations, this was not the best choice for all datasets. Note that these graphs have been generated using the data of the grid search and not the final evaluation runs.

To motivate this statement, we produce figures 5.8, 5.9, 5.10 and 5.11. These show the distribution of the reasons of ending a partition specific search over the lengths of the partitions.

Comparing figure 5.8 to the others shows that, in comparison to UMLS, the other datasets show a weaker timeout/no-solution ratio for larger partitions. Furthermore, the distribution over all partitions for UMLS (figure 5.8) shows that larger partitions can be evaluated efficiently, but the missed partitions have a larger tendency to time out.

The two Playgol datasets show both extremes: for string manipulation all sizes of partitions do not time out on average, while for lego every size tends to time out. These are fundamental differences in the problem type and this influences the effectiveness of the partition-search procedure. In reality, the Playgol-string-manipulation dataset could most likely have handled graph clustering approaches with a larger maximum size.

We propose two potential solutions that are not majorly different compared to the current methodology. First of all, the scheduling consists of evaluating the biggest clusters first. Since the distributions for FB15 and string manipulation show that the smaller clusters predominantly return a no solution result, they often use less time than their designated timeout. In this case, evaluating the smallest partitions first could perhaps be beneficial to the time spent on larger partitions, since the remaining time can be split linearly amount less remaining functions.

The other method is by increasing the time spent on the selection process, either by increasing the percentage of the total time spent on the selection or the timeout altogether. This could be the case, since in a sense the timeouts reflect that there is too little time to properly search through the partitions.

These results show that he scheduling of timeouts for the selection process is a crucial component to optimize and it is not clear what the best approach is or if there is a universal best approach for all datasets. Furthermore, the graph clustering approaches could potentially perform better with bigger maximum sizes for other problems.

Conclusion

In conclusion, we have found that there are differences between the various approaches and we have chosen subsets to further evaluate on bigger datasets for generalization. We have also seen that there are clear differences in how the search performs on different datasets, which causes differences in the effect of the partition-search procedure.

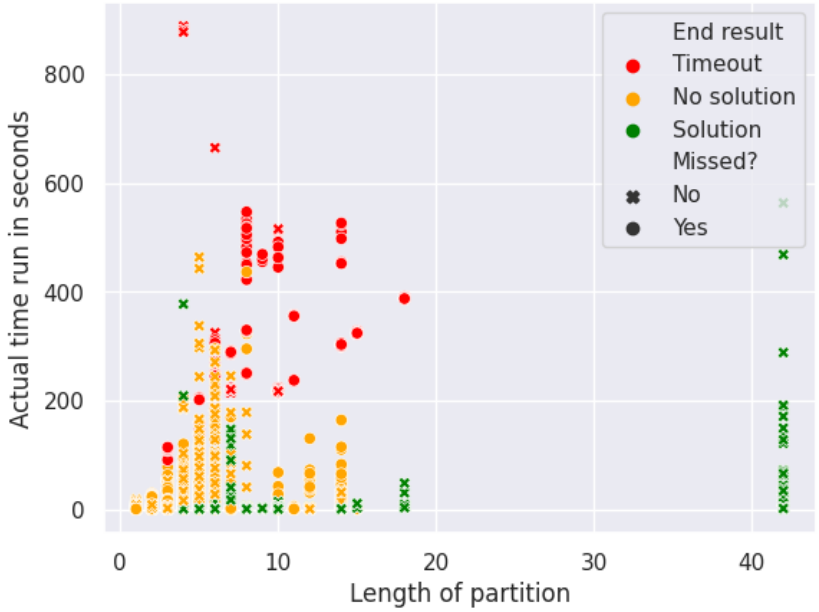
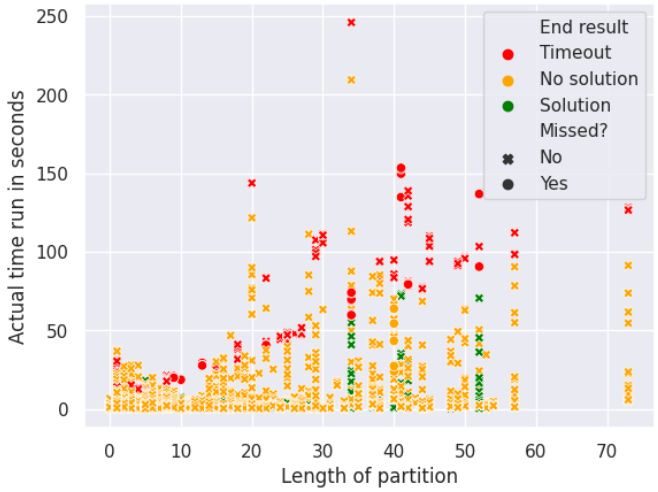
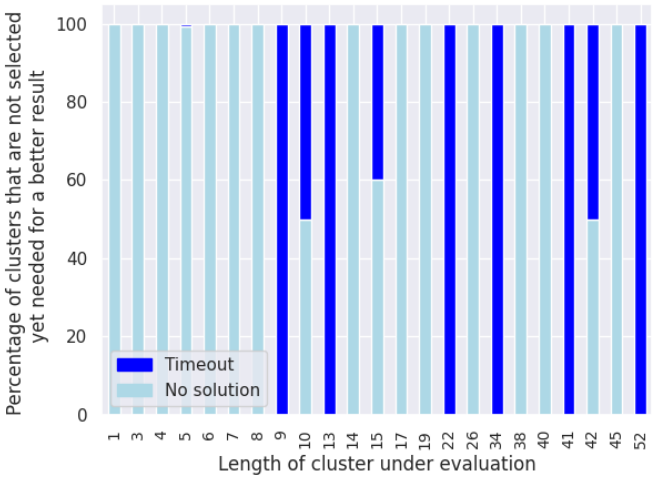


Figure 5.8.: Distribution of time run in seconds; color means reason of ending and marker indicates if it was a missed crucial partition or not for UMLS. Results show that unfair scheduling is not the cause of the timeouts under crucial partitions, but that larger partitions simply time out more often.

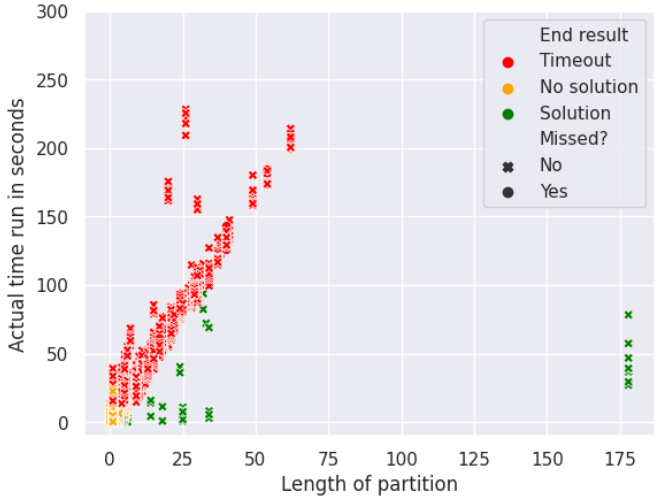


(a) Distribution of time run in seconds; color means reason of ending and marker indicates if it was a missed crucial partition or not.

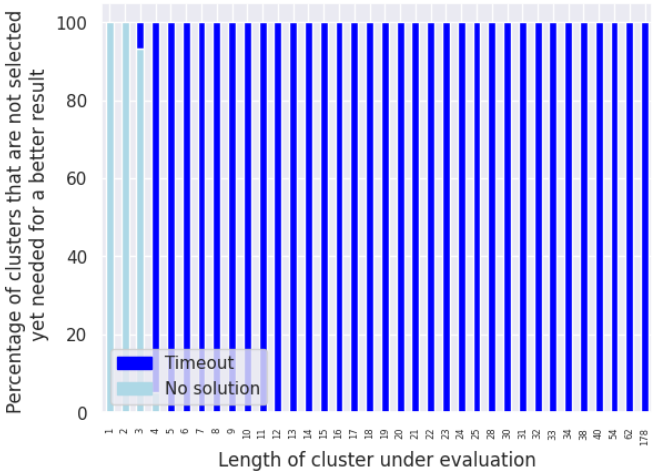


(b) Distribution of reasons of missing crucial partitions per partition length

Figure 5.9.: Figures showing reasons of ending the search for a given partition over lengths for the Playgol: string manipulation dataset. Results show failures in structure as the majority reason of not selecting a crucial partition.

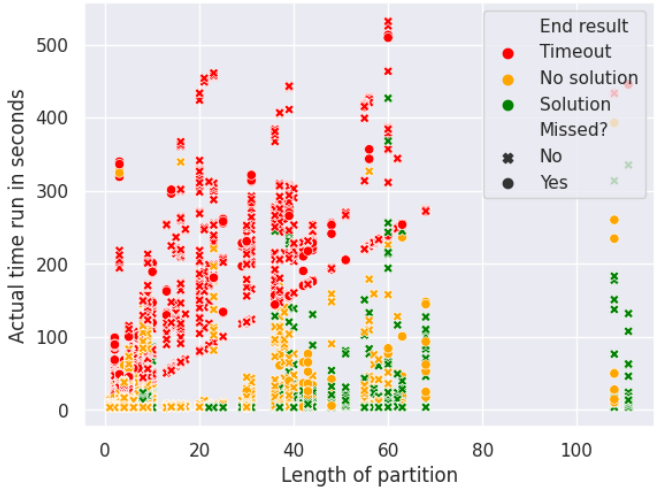


(a) Distribution of time run in seconds; color means reason of ending and marker indicates if it was a missed crucial partition or not.

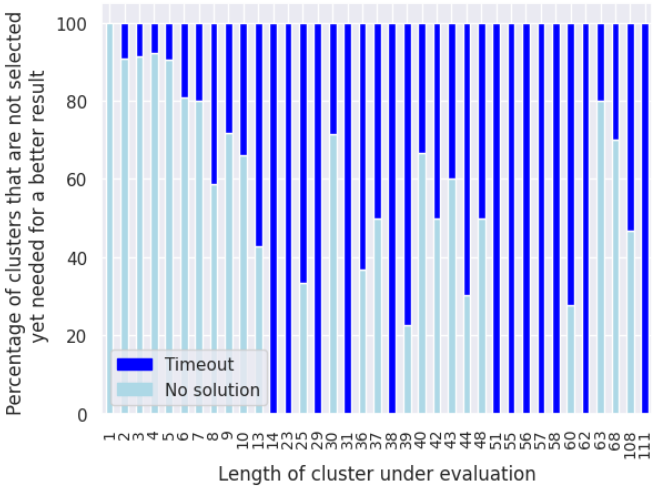


(b) Distribution of reasons of missing crucial partitions per partition length

Figure 5.10.: Figures showing reasons of ending the search for a given partition over lengths for the Playgol: lego dataset. Results show timeouts as the majority reason of not selecting a crucial partition and a mix in probability of missing crucial elements over all lengths.



(a) Distribution of time run in seconds; color means reason of ending and marker indicates if it was a missed crucial partition or not.



(b) Distribution of reasons of missing crucial partitions per partition length

Figure 5.11.: Figures showing reasons of ending the search for a given partition over lengths for the FB15k-237 dataset. Results show a varying ratio between timeouts and failures in structure and a mix in probability of missing crucial elements over all lengths.

5.3.3. Generalization

Our next experiment is meant to determine whether the results of the previous experiment are stable when more data is added and whether the graph clustering algorithms actually import any information in the structure of the partitionings.

Experiment setup Just like the previous experiments, we re-use our general setup described in chapter 5.2.4. Because of the long runtime, we can not run more datasets on all combinations when increasing the sample size. Instead, we focus on the top 3 and bottom 3 combinations we identified in the previous experiment. We also add two random partitionings. The first is generated by shuffling the list of all functions in the background knowledge and iteratively sampling a number between 1 and 20 or the length of the remaining functions if there are less than 20, taking that number of functions as a separate partition and continuing until there are no functions left. The second one is generated by first shuffling the background knowledge and splitting it up by consecutive chunks of size 5, mimicking the behaviour of METIS but randomized.

In order to obtain more data, we use the independently generated set of from FB15k-237 with 15% of relations selected and use a different variations for Playgol, variation 9 for Playgol-string and variation 3 for Playgol-lego, chosen because they maximize the amount of solved test instances.

Finally, the question is whether the partitions actually encode any information or if the act of using partitioning and selection *itself* is what provides performance gains. We included the random partitions especially for that reason. In order to make this more concrete, we would normally employ the use of a statistical test. Depending on the characteristics of the resulting data, we will use an applicable test.

Interpreting the results The results of the experiment can be found in tables 5.5, 5.6 and 5.7 for Playgol-string, Playgol-lego and FB15k-237 respectively. From these results we gather three conclusions.

First of all, for FB15K-237 the hierarchy between the top and bottom 3 for each seems to stay relatively intact. This means that the smaller test set was a relatively good representation of the bigger distribution of problems. Obviously we have no results for the other approaches, so we cannot draw any conclusions as of their performance, but there is not enough evidence that the original performance was not indicative of the performance with more data.

Second, for Playgol the difference seems to have shrunk: for example, the iterative Louvain method in the Playgol-lego dataset now performs as well as the top 3. Another good example is the (Clique, Iterative Louvain (max 10)) approach for Playgol-string. This was the only approach that solved all instances perfectly before and now it performs worse than an approach in the bottom 3. From these results we conclude that they would imply that the performance on the original set of test instances was not representative of the performance with more test instances added. The conclusion that the partition-search procedure *can* improve performance stands, but which approach performs better is likely to change with more data.

Finally we conclude that the recall gains for Playgol-lego are not attributable to the information provided by our clustering algorithms. The random partitioning performs better than all

others, showing that even a clustering without heuristics is more useful than those build *with* heuristics.

Group	Topology	Clustering algorithm	Mean of best improvement	# of test instances improved out of 19
Top 3	Clique	Louvain (Iterative) (max 10)	0.517 (+= 0.31)	6
	Usage	Louvain	0.650 (+= 0.32)	6
	Usage	Louvain (Iterative) (max 10)	0.550 (+= 0.34)	6
Other		Random	0.500 (+= 0.33)	4
		Random (GC)	0.460 (+= 0.30)	5
Bottom 3	Clique	METIS	0.500 (+= 0.33)	4
	Clique	Greedy Modularity Maximization	0.443 (+= 0.37)	7
	Usage	Paris (max 5)	0.600 (+= 0.30)	6

Table 5.5.: Results of the evaluation run on Playgol-string-manipulation after pruning for values where the difference in recall in relation to the maximum of the baselines for that test instance is positive. Differences are calculated over the set of maximum recall over 5 repetitions.

Group	Topology	Clustering algorithm	Mean of best improvement	# of test instances improved out of 20
Top 3	Clique	Greedy Modularity Maximization	1.000 (+= 0.00)	4
	Clique	Label propagation	1.000 (+= 0.00)	4
	Usage	Louvain	1.000 (+= 0.00)	3
Other		Random	1.000 (+= 0.00)	5
		Random (GC)	1.000 (+= 0.00)	2
Bottom 3	Usage	Greedy Modularity Maximization	1.000 (+= 0.00)	4
	Usage	Louvain (Iterative) (max 5)	1.000 (+= 0.00)	2
	Co-occurrence	Louvain	1.000 (+= 0.00)	1

Table 5.6.: Results of the evaluation run on Playgol-lego after pruning for values where the difference in recall in relation to the maximum of the baselines for that test instance is positive. Values are calculated over the set of maximum values over 5 repetitions.

Significance Finally, like described in the setup for this experiment, we would have ideally provided a statistical test for significance. However, the characteristics of the results makes this difficult. First of all, the data is non-normal, which rules out any tests based on normality. Furthermore, many of the non-parametric tests then require independent samples, such as the Mann Whitney U Test (Mann & Whitney, 1947). If we want to compare between two algorithms that will entail comparing across the same test instances, making the two samples dependent. Paired tests solve this solution, but most of those have additional requirements the data does not meet. For example, the commonly used paired non-parametric Wilcoxon Signed Rank Test (Conover, 1999) assumes a symmetric distribution of the distances $x - y$ of the two samples under the null hypothesis. This means that if the differences are not symmetrically distributed it can cause the test to give significant results for differences other than a difference

Group	Topology	Clustering algorithm	Mean of best improvement	# of test instances improved out of 33
Top 3	Clique	Louvain	0.203 (+= 0.25)	27
	Clique	Greedy Modularity Maximization	0.176 (+= 0.26)	29
	Usage	Greedy Modularity Maximization	0.175 (+= 0.25)	29
Other		Random	0.168 (+= 0.21)	22
		Random (GC)	0.183 (+= 0.23)	17
Bottom 3	Usage	Louvain (Iterative) (max 5)	0.176 (+= 0.19)	22
	Usage	Paris (max 5)	0.164 (+= 0.17)	23
	Co-occurrence	METIS	0.111 (+= 0.16)	21

Table 5.7.: Results of the evaluation run on FB15k-237 after pruning for values where the difference in recall in relation to the maximum of the baselines for that test instance is positive. Values are calculated over the set of maximum values over 5 repetitions.

in medians.

Finally there is the sign test (Sprent, 2011), which is a less powerful test than the ones mentioned before. Its null hypothesis is that the differences between two samples has a median of 0 and the alternative hypothesis is that the median is *different* (not greater) than 0. This test uses a binomial distribution and ranks the significance based on the signs of the differences between the points only, while the Signed Rank Test also uses the magnitude. This test is very general and has very little assumptions.

1. The dependent variable (i.e. the recall) should be at least ordinal. The call is a real number, so this holds.
2. The independent variable (i.e. the test instance) should be categorical and matched between the pairs. This holds too.
3. The paired observations need to be independent, meaning they cannot influence each other. This holds as well.
4. and finally the differences between the two scores must be continuous. This is true, because the recalls are both real numbers and so the difference is as well.

Assuming this holds, we ran a two-sided test on the complete dataset without filtering for positive recall changes for the top and bottom 3 against both random partitioning results. For Playgol (both lego and string manipulation) no statistically significant changes were found. For FB15k-237 the top 3 results have significant (P-value smaller than 0.05) differences over both of the random approaches. The Paris approach with size 5 and the iterative louvain approach with size 5 has a significant difference against random graph clustering, not against the other random partitioning. All P-values can be found in table 5.8.

These are not necessarily improvements (because that was not the alternative hypothesis) and in order to show the differences, we show the graphs in figure 5.12. Based on these results, the differences in distribution that the tests respond to are most likely improvements over the random partitionings. For the two bottom 3 approaches with P-values smaller than 0.05, the results are less clear. Both approaches show improvements over the random graph clustering variant, which is most likely the reason for the low p-value.

Group	Topology	Clustering algorithm	P-value	Random variant
Top 3	Clique	Louvain	**0.007000**	Random
		Louvain	**0.000535**	Random (GC)
		Greedy Modularity Maximization	**0.035082**	Random
		Greedy Modularity Maximization	**0.000324**	Random (GC)
	Usage	Greedy Modularity Maximization	**0.004551**	Random
		Greedy Modularity Maximization	**0.000535**	Random (GC)
Bottom 3	Co-occurrence	METIS	0.458258	Random
		METIS	1.000000	Random (GC)
	Usage	Paris (max 5)	0.442068	Random
		Paris (max 5)	**0.001544**	Random (GC)
		Louvain (Iterative) (max 5)	0.701108	Random
		Louvain (Iterative) (max 5)	**0.004077**	Random (GC)

Table 5.8.: Results of the sign test on the data of the evaluation of FB15. Bold means the P-value is smaller than 0.05

From these tests we conclude that the top 3 approaches of FB15 most likely include more information than the random partitionings. In other words: these partitionings most likely capture a form of useful information for inductive logic programming.

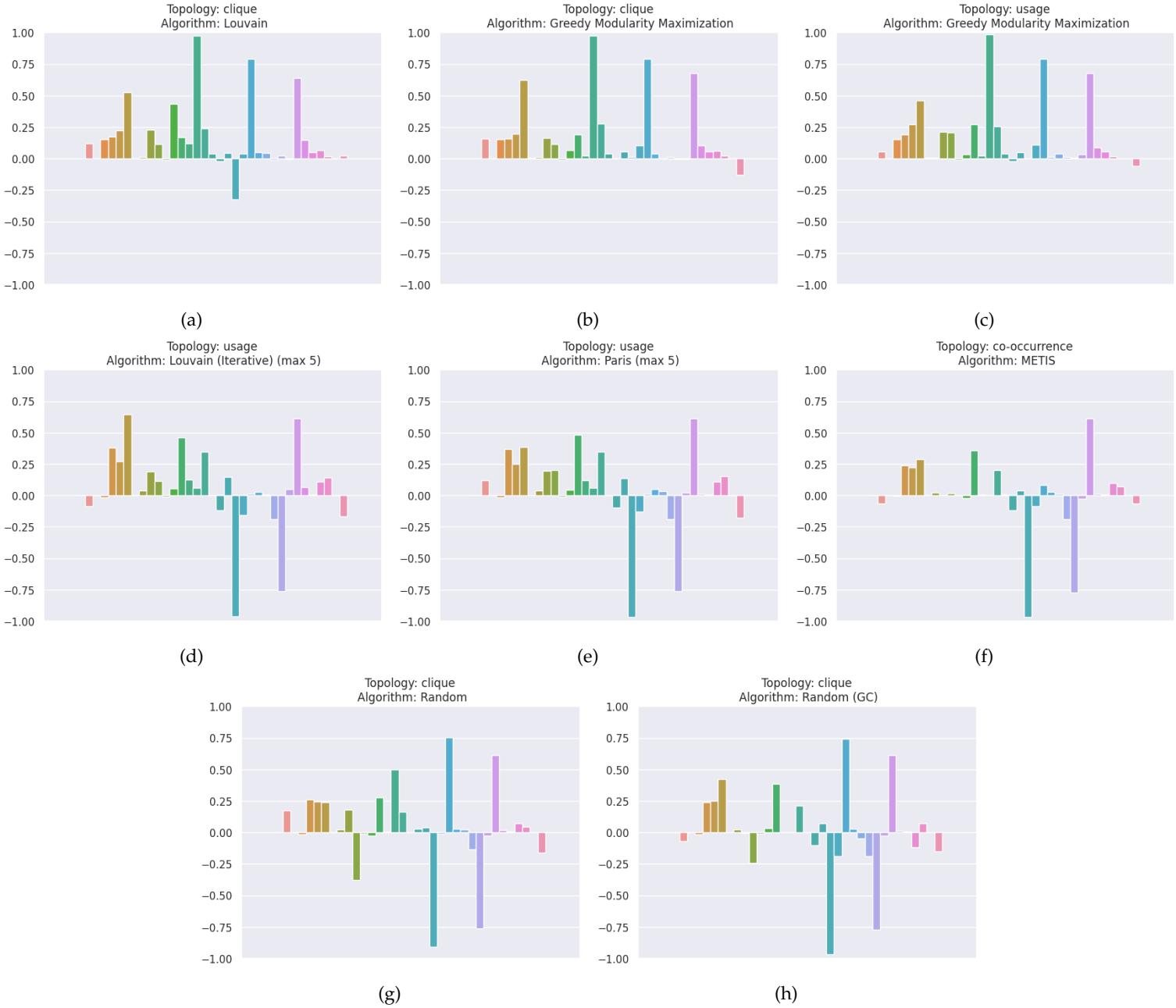


Figure 5.12.: Graphs showing the results of the evaluation run of FB15. A to C show top 3. D to F show bottom 3. G and F show random and random graph clustering respectively. Results show that the top 3 visually perform better than the bottom 3 or random partitionings.

5.4. Conclusion and answers to research questions

At this point we can start answering our research questions using the results in this chapter.

In what ways can we vary the construction of a static callgraph from Prolog programs to use graph clustering algorithms on and does it effect performance of the partition-search procedure?

We can answer this question using the results of our grid search: clearly the topology of the graphs matters in terms of performance increase of the resulting programs. This is especially clear for the Playgol datasets, where the *Co-occurrence* topology causes a breakdown in performance.

However, this is the only clear example of the effect of the topology. There is obviously is an effect, because a graph clustering algorithm can have a change in recall gain when only varying the topology. Unfortunately, it is not consistent which topology functions better across graph clustering algorithms. Furthermore, with a sample size of 10 significance tests are not useful, especially with non-normal data. While we show that the *combination* of graph clustering algorithm and the topology has an effect, we cannot conclude there is a best performing topology.

Does varying the graph clustering algorithm in the partition-search procedure cause a difference in performance of the final programs?

Just like with the topologies, there are clear differences in the effectiveness of the partitions returned by the varying algorithms. However the same caveat as with the topologies holds for the graph clustering algorithms in that the combination of the topology and algorithms is more important than the algorithm alone. The difference in performance between algorithms is not consistent over topologies for any of the datasets. We find that there exists an effect, but we cannot attribute it directly to the algorithm.

Topology and algorithms combined While we cannot deal with the previous two variables on their own, we can reason about them as a combination. The current partition-search procedure has the potential to perform significantly better with engineered partitions from graph clustering algorithms (which includes community detection algorithms) in comparison to a random partitioning. This is based on the significance tests during the generalization run, where we show a significant difference between particular algorithm- and topology combinations compared to random partitionings, but not for all.

This indicates that there is an optimal partitioning at least for a given set of test instances. Note that even if the random partitioning is better than community detection or graph clustering algorithms, it is a positive result, since it realistically means better performance on a subset of the test set for 'free'.

Does the effect of using a partition-search procedure differ between types of problems?

Finally, the effectiveness of the partition-search procedure heavily differs between problem type and even domain within a given problem type. For UMLS the current setup does primarily yield decreases in performance, but for FB15k-237 there are major increases. Both are knowledge graphs and for both the rules are generated by the same tool. The only difference is the structure of the knowledge graph itself and yet the usefulness of the proposed method differs severely. Similar differences exist between Playgol and the knowledge graphs. These results answer the subquestion directly.

The main research question

And with these answers, we can answer our main research question, **'Can inductive program synthesis approaches exploit partitions found through applying graph clustering on callgraphs from existing programs by applying a partition-search procedure in order to improve the synthesis of programs within a given time limit?'**. The answer is that the resulting programs from the partition-search procedure have the potential to be an significantly improvement over basic inductive logic programming using a state of the art inductive logic programming approach, even using a naive selection function and a random partitioning. Using partitions build using heuristics like modularity can push the performance even further.

6. Conclusion and future work

In this thesis we have proposed and evaluated a new task, meant for improving search-based inductive logic programming. By splitting the background knowledge into partitions with related functions, we can select and provide only relevant segments of the background knowledge, increasing the time the solver can spend on finding programs with useful functions. We have applied graph-based partitioning algorithms as an example approach as an initial direction and have shown that even basic community detection and graph clustering algorithms can find partitions that improve resulting programs for some test instances. Furthermore, this work does not rely on specific functionality for its internal solver. Because of this, our work allows for more efficient use of larger grammars in existing inductive logic programming approaches.

6.1. Contributions

This work introduced a new task in the field of inductive program synthesis: partitioning and selecting parts from the background knowledge that are relevant to the problem instance at hand prior to synthesis. We have proposed an initial direction by implementing a partitioning function based on graph clustering algorithms and a linear time evaluation selection function. Using these, we have shown that it can lead to better performing programs, even with a naive implementation of the selection process. Based on the results, we can conclude that there could be partitioning functions that are better for a specific selection function than random partitionings. Most importantly, we show that reframing an inductive logic programming problem into a Time-gated Partition-selection Inductive Logic Programming problem is a useful tool in the toolkit of an inductive logic programming practitioner.

6.2. Limitations

The limitations of this work are majorly based on the fact that there is very little data available in terms of potential domains to test. The effects vary dramatically between the domains and as such more domains could have given us more interesting results. The datasets that we have utilized have sample sizes of N smaller than 35, because of time constraints and the current selection function is also not completely tuned for the problem sizes that we evaluated, judging from the fact that we have a lot of timeouts when evaluating. The results could therefore have been better if the problems were smaller in terms of total amount of functions.

Secondly, the graph partitioning algorithms could potentially perform better for FB15k-237 and Playgol if the maximum size is increased. We have shown that there is a difference in behaviour related to timeouts and since we tuned the sizes on UMLS, this could have impacted their recall gains.

Furthermore, while the results show improvement for three out of four domains, it remains a difficult problem to estimate what partitions are useful *prior* to evaluating them. While we can conclude that the majority of our domains work decently well with modularity based graph clustering algorithms, this does not need to be the case for all domains and it is also not *completely* consistent. Playgol-lego for example works best with label propagation (which does not use modularity) and even random partitionings work well on that specific domain. The majority of the algorithms evaluated in this research also use modularity, so the fact that most of the algorithms that work well use modularity is also to be expected. Determining the best clustering algorithm for a given selection functions is therefore still trial-and-error.

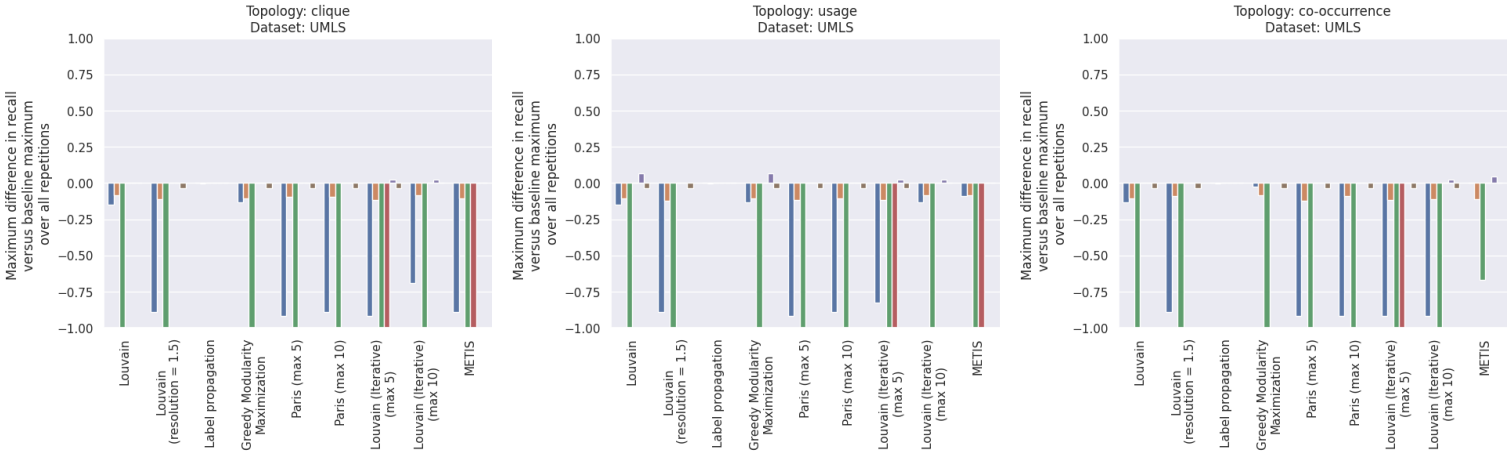
Finally, we have not evaluated any other selection function to compare to linear time evaluation. The results could vary wildly based on the choice of the selection function alone. We will discuss specific research avenues related to this problem in the next section.

6.3. Future work

Several directions of future work have been highlighted in this thesis. The current implementation of the partition-search procedure can most certainly be improved: especially the selection function has a lot of potential to be researched further. Possible methods are the integration of problem type specific information and/or using embeddings of the problem instances as input for selection. This can already be achieved using the current datasets. Selecting partitions for the knowledge graphs could for example leverage the connections between entities and functions in order to extract partitions that are likely useful to a given problem instance. Even without adding more information, our results show that scheduling the time spent selecting partitions could be spent more effectively, since the majority of combinations of topologies, algorithms and datasets only use a small portion of the time set out for the selection process, yet they still have timeouts.

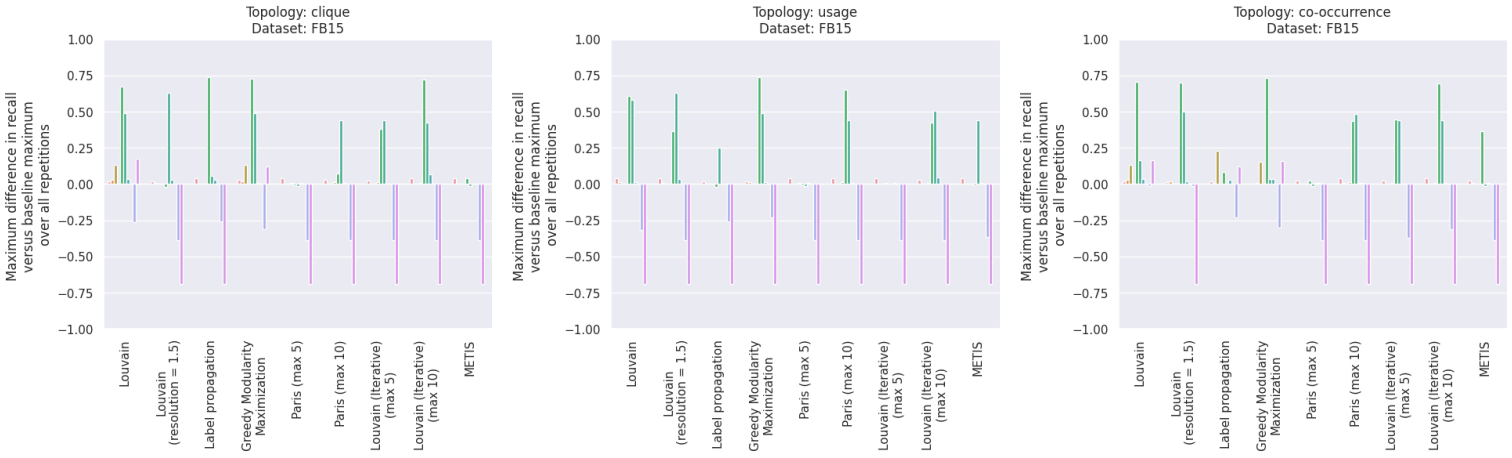
Furthermore, as stated previously, the current lack of datasets containing the resulting programs makes evaluation over various problem types difficult. Creating new datasets with a larger background knowledge and evaluating methods on a larger scale could potentially yield interesting results. Extending this work outside of the field of inductive logic programming by using a grammar that is not a logic programming language is also a potential avenue of future improvement. This would require solving problems similar to the problem with the base functions that we encountered, since if functions are in multiple partitions, that function can cause more than one partition to be included. If a given language consists of more than just functions, this could have the same effect. Following this thread we also have the hierarchical clustering of execution paths, which is used multiple times in the field of program comprehension, but that we have ignored because of this issue. Finding a solution for this problem would be an useful direction of research.

A. Extra figures



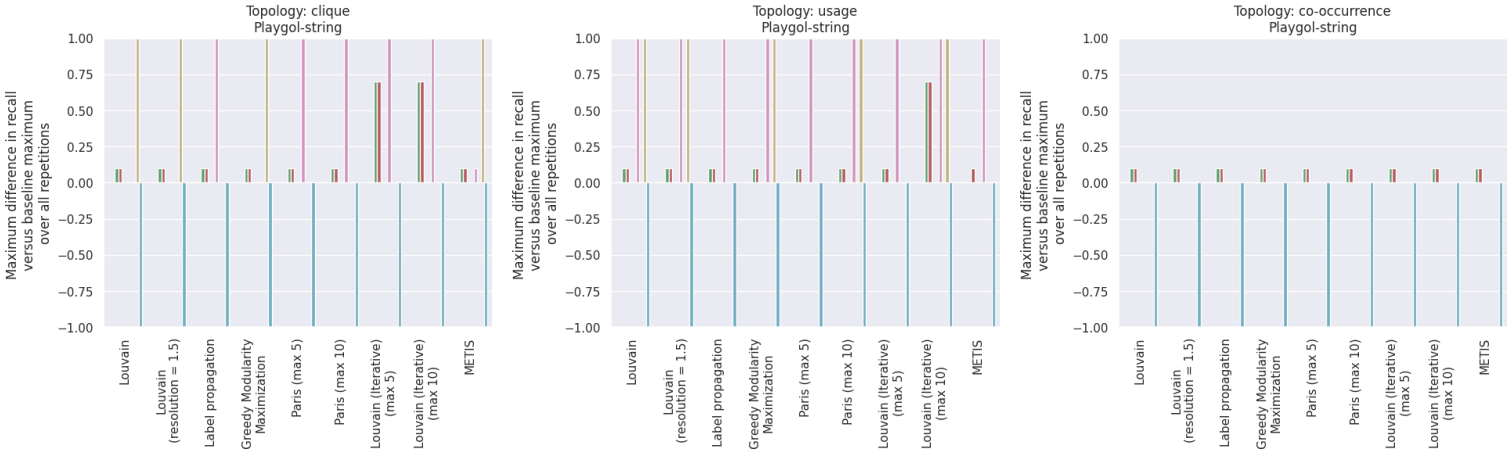
(a) (b) (c)

Figure A.1.: All graphs from the grid search of UMLS



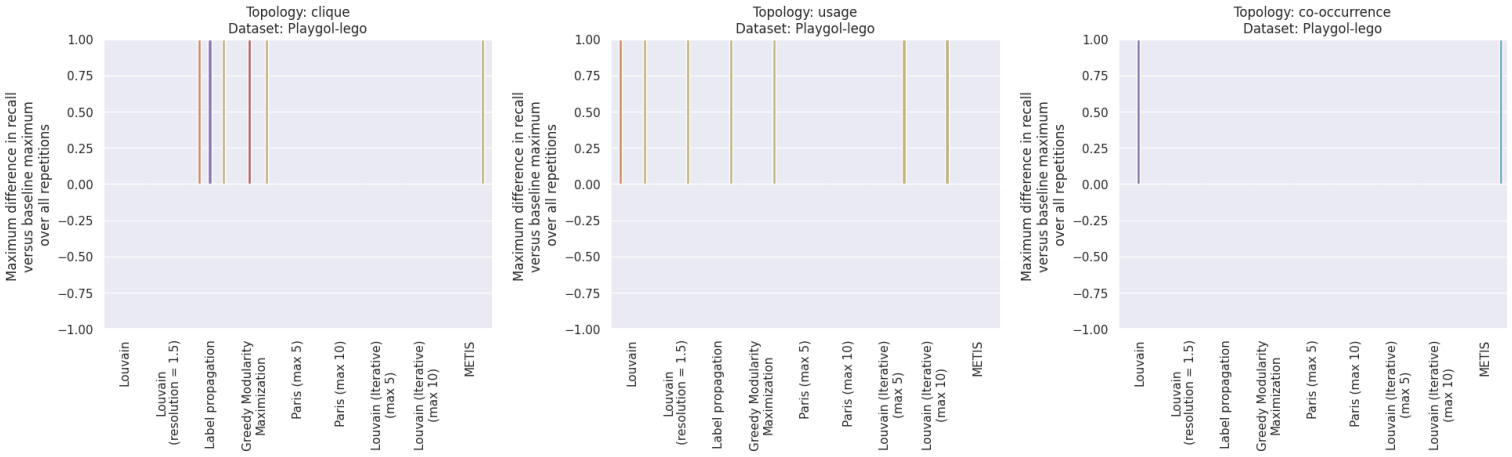
(a) (b) (c)

Figure A.2.: All graphs from the grid search of FB15



(a) (b) (c)

Figure A.3.: All graphs from the grid search of Playgol-string



(a) (b) (c)

Figure A.4.: All graphs from the grid search of Playgol-lego

Bibliography

Alanazi, R., Gharibi, G., & Lee, Y. (2021). Facilitating program comprehension with call graph multilevel hierarchical abstractions. *Journal of Systems and Software*, *176*, 110945. Retrieved from https://www.sciencedirect.com/science/article/pii/S016412122100042X doi: https://doi.org/10.1016/j.jss.2021.110945

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., ... Sutton, C. (2021). *Program synthesis with large language models.*

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2017). *Deepcoder: Learning to write programs.*

Barke, S., Peleg, H., & Polikarpova, N. (2020). Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.*. doi: 10.1145/3428295

Bhattacharjee, A., et al. (2021). *Hcpc: Human centric program comprehension by grouping static execution scenarios* (Unpublished doctoral dissertation). University of Saskatchewan.

Bhattacharjee, A., Roy, B., & Schneider, K. A. (2022). Supporting program comprehension by generating abstract code summary tree. In *2022 ieee/acm 44th international conference on software engineering: New ideas and emerging results (icse-nier)* (p. 81-85). doi: 10.1145/3510455.3512793

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., & Lefebvre, E. (2008, oct). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, *2008*(10), P10008. Retrieved from https://doi.org/10.1088%2F1742-5468%2F2008%2F10%2Fp10008 doi: 10.1088/1742-5468/2008/10/p10008

Bonald, T., Charpentier, B., Galland, A., & Hollocou, A. (2018). *Hierarchical graph clustering using node pair sampling.*

Bonald, T., de Lara, N., Lutz, Q., & Charpentier, B. (2020). Scikit-network: Graph analysis in python. *Journal of Machine Learning Research*, *21*(185), 1-6. Retrieved from http://jmlr.org/papers/v21/20-412.html

Clauset, A., Newman, M. E. J., & Moore, C. (2004, Dec). Finding community structure in very large networks. *Phys. Rev. E*, *70*, 066111. Retrieved from https://link.aps.org/doi/10.1103/PhysRevE.70.066111 doi: 10.1103/PhysRevE.70.066111

Cohen, W. W. (1994). Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, *68*(2), 303-366. Retrieved from https://www.sciencedirect.com/science/article/pii/0004370294900701 doi: https://doi.org/10.1016/0004-3702(94)90070-1

Conover, W. (1999). *Practical nonparametric statistics*. Wiley. Retrieved from https://books.google.nl/books?id=n_39DwAAQBAJ

Contreras-Ochando, L., Ferri, C., Hernández-Orallo, J., Martínez-Plumed, F., Ramírez-Quintana, M. J., & Katayama, S. (2020). Bk-adapt: Dynamic background knowledge for automating data transformation. In U. Brefeld, E. Fromont, A. Hotho, A. Knobbe, M. Maathuis, & C. Robardet (Eds.), *Machine learning and knowledge discovery in databases* (pp. 755–759). Cham: Springer International Publishing.

Cropper, A., & Dumančić, S. (2022). Inductive logic programming at 30: a new introduction. *Journal of Artificial Intelligence Research*, *74*, 765–850.

Cropper, A., & Morel, R. (2020). Learning programs by learning from failures. *arXiv: Artificial Intelligence*. doi: 10.1007/s10994-020-05934-z

Cropper, A., & Muggleton, S. H. (2016). *Metagol system.* https://github.com/metagol/metagol. Retrieved from https://github.com/metagol/metagol

Delft High Performance Computing Centre (DHPC). (2022). *DelftBlue Supercomputer (Phase 1).* https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1.

Dettmers, T., Minervini, P., Stenetorp, P., & Riedel, S. (2017). Convolutional 2d knowledge graph embeddings. *CoRR*, *abs/1707.01476*. Retrieved from http://arxiv.org/abs/1707.01476

Diaconu, A. (2020). Learning functional programs with function invention and reuse. *arXiv: Programming Languages*. doi: null

Dumancic, S., & Cropper, A. (2020). Knowledge refactoring for program induction. *CoRR*, *abs/2004.09931*. Retrieved from https://arxiv.org/abs/2004.09931

Ellis, K., Wong, C., Nye, M., Sable-Meyer, M., Cary, L., Morales, L., ... Tenenbaum, J. B. (2020). *Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning.*

Fortunato, S. (2010). Community detection in graphs. *Physics Reports*. doi: 10.1016/j.physrep.2009.11.002

Gharibi, G., Alanazi, R., & Lee, Y. (2018). Automatic hierarchical clustering of static call graphs for program comprehension. In *2018 ieee international conference on big data (big data)* (p. 4016-4025). doi: 10.1109/BigData.2018.8622426

Gulwani, S., Polozov, A., & Singh, R. (2017). *Program synthesis* (Vol. 4). NOW. Retrieved from https://www.microsoft.com/en-us/research/publication/program-synthesis/

Hagberg, A., Swart, P., & S Chult, D. (2008). *Exploring network structure, dynamics, and function using networkx* (Tech. Rep.). Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Karypis, G., & Kumar, V. (1999). Multilevel k-way hypergraph partitioning. In *Proceedings 1999 design automation conference (cat. no. 99ch36361)* (p. 343-348). doi: 10.1109/DAC.1999.781339

Kitzelmann, E. (2009). Inductive programming: A survey of program synthesis techniques. *Approaches and Applications of Inductive Programming*. doi: 10.1007/978-3-642-11931-6_3

Lutellier, T., Chollak, D., Garcia, J., Tan, L., Rayside, D., Medvidovic, N., & Kroeger, R. (2015). Comparing software architecture recovery techniques using accurate dependencies. In *2015 ieee/acm 37th ieee international conference on software engineering* (Vol. 2, p. 69-78). doi: 10.1109/ICSE.2015.136

Mann, H. B., & Whitney, D. R. (1947). On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, *18*(1), 50 – 60. Retrieved from https://doi.org/10.1214/aoms/1177730491 doi: 10.1214/aoms/1177730491

Mattis, T. (2018, April). Mining concepts from code using community detection in co-occurrence graphs. In *Conference companion of the 2nd international conference on art, science, and engineering of programming.* ACM. Retrieved from https://doi.org/10.1145/3191697.3213797 doi: 10.1145/3191697.3213797

McDaid, E., & McDaid, S. (2023). *Shrinking the inductive programming search space with instruction subsets.*

Meilicke, C., Chekol, M. W., Fink, M., & Stuckenschmidt, H. (2020). Reinforced anytime bottom up rule learning for knowledge graph completion. *CoRR*, *abs/2004.04412*. Retrieved from https://arxiv.org/abs/2004.04412

Mitchell, B. S., & Mancoridis, S. (2006). On the automatic modularization of software systems

using the bunch tool. *IEEE Transactions on Software Engineering*. doi: 10.1109/tse.2006.31

Mothe, J., Mkhitaryan, K., & Haroutunian, M. (2017). Community detection: Comparison of state of the art algorithms. In *2017 computer science and information technologies (csit)* (p. 125-129). doi: 10.1109/CSITechnol.2017.8312155

Muggleton, S. (1995). Inverse entailment and progol. *New Generation Computing*. doi: 10.1007/bf03037227

Newman, M. (2010). *Networks: An Introduction*. Oxford University Press. Retrieved from https://doi.org/10.1093/acprof:oso/9780199206650.001.0001 doi: 10.1093/acprof:oso/9780199206650.001.0001

Newman, M. E. J. (2006). Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, *103*(23), 8577-8582. Retrieved from https://www.pnas.org/doi/abs/10.1073/pnas.0601602103 doi: 10.1073/pnas.0601602103

Newman, M. E. J. (2013, oct). Spectral methods for community detection and graph partitioning. *Physical Review E*, *88*(4). Retrieved from https://doi.org/10.1103%2Fphysreve.88.042822 doi: 10.1103/physreve.88.042822

Raghavan, U. N., Albert, R., & Kumara, S. (2007, sep). Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, *76*(3). Retrieved from https://doi.org/10.1103%2Fphysreve.76.036106 doi: 10.1103/physreve.76.036106

Sankaranarayanan, S., Ivancic, F., & Gupta, A. (2008). Mining library specifications using inductive logic programming. *2008 ACM/IEEE 30th International Conference on Software Engineering*. doi: 10.1145/1368088.1368107

Shi, K., Steinhardt, J., & Liang, P. (2019, jan). Frangel: Component-based synthesis with control structures. *Proc. ACM Program. Lang.*, *3*(POPL). Retrieved from https://doi.org/10.1145/3290386 doi: 10.1145/3290386

Smith, C., & Albarghouthi, A. (2019). Program synthesis with equivalence reduction. In *International conference on verification, model checking and abstract interpretation*.

Sprent, P. (2011). Sign test. In M. Lovric (Ed.), *International encyclopedia of statistical science* (pp. 1316–1317). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from https://doi.org/10.1007/978-3-642-04898-2_515 doi: 10.1007/978-3-642-04898-2_515

Srinivasan, A. (2001). *The aleph manual*.

Subelj, L., & Bajec, M. (2011). Community structure of complex software systems: Analysis and applications. *CoRR*, *abs/1105.4276*. Retrieved from http://arxiv.org/abs/1105.4276

Toutanova, K., & Chen, D. (2015, July). Observed versus latent features for knowledge base and text inference. In *Proceedings of the 3rd workshop on continuous vector space models and their compositionality* (pp. 57–66). Beijing, China: Association for Computational Linguistics. Retrieved from https://aclanthology.org/W15-4007 doi: 10.18653/v1/W15-4007

Tsitsulin, A., Palowitch, J., Perozzi, B., & Müller, E. (2020). Graph clustering with graph neural networks. *arXiv preprint arXiv:2006.16904*.

Walunj, V., Gharibi, G., Ho, D. H., & Lee, Y. (2019). Graphevo: Characterizing and understanding software evolution using call graphs. In *2019 ieee international conference on big data (big data)* (p. 4799-4807). doi: 10.1109/BigData47090.2019.9005560

Yang, Y., Inala, J. P., Bastani, O., Pu, Y., Solar-Lezama, A., & Rinard, M. (2021). Program synthesis guided reinforcement learning. *arXiv: Artificial Intelligence*. doi: null

Zhang, S., Chen, Z., Shen, Y., Ding, M., Tenenbaum, J. B., & Gan, C. (2023). Planning with large language models for code generation. In *The eleventh international conference on learning representations*. Retrieved from https://openreview.net/forum?id=Lr8cOOtYbfL

Zhang, Y., Yao, Q., & Chen, L. (2021). *Efficient, simple and automated negative sampling for knowledge graph embedding*.