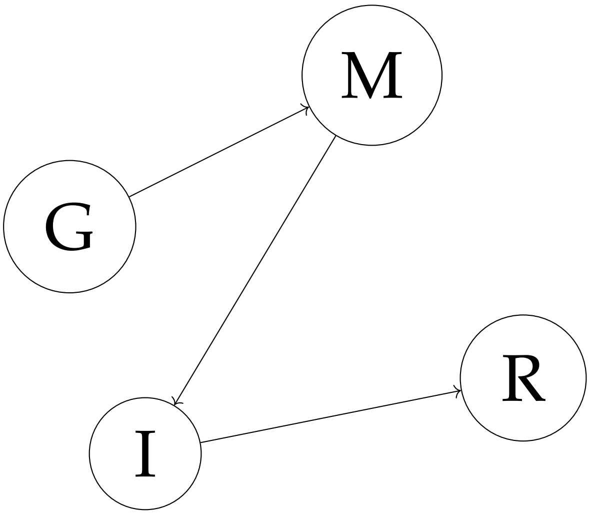
Domain-Specific Abstractions for Algorithmic Graph Processing

Master's Thesis



Johannes Hendrik (Jochem) Broekhoff

Domain-Specific Abstractions for Algorithmic Graph Processing

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER AND EMBEDDED SYSTEMS ENGINEERING

by

Johannes Hendrik (Jochem) Broekhoff
born in Krimpen aan den IJssel, the Netherlands

TU Delft
Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: a directed graph with letters as a play on the GMIR abbreviation.

Domain-Specific Abstractions for Algorithmic Graph Processing

Author: Johannes Hendrik (Jochem) Broekhoff
Student id: 5076218

Abstract

Graphs and richer property graphs are common models for real-world data. We typically run algorithms on such data to extract meaningful information. Using domain-specific programming languages (DSLs) is a common approach to expressing such algorithms, contrasting to general-purpose programming languages and declarative graph query languages. On one hand, algorithms in general-purpose languages are verbose and conceptually far removed from from the algorithm theory, as is the case for some community detection algorithms in DSLs. On the other hand, the DSLs that are available are insufficient to express all common graph analysis algorithms. The Green-Marl Intermediate Representation (GMIR) is such a graph algorithm DSL. As it has been built from the ground up, it only provides a minimal feature to support the algorithms it initially needed to support, similar to how other DSLs are developed. This specifically prevents frontier exploration algorithms and community detection algorithms to be expressed, such as Dijkstra's shortest path and the Louvain clustering method. We use GMIR as a vehicle to introduce new domain-specific abstractions for algorithmic graph processing, targeting those algorithms. We evaluate our abstractions by implementing them in the commercial GMIR compiler, which we then use to compile various new algorithms to existing commercial graph processing platforms. This shows that we have successfully enabled more graph algorithms to be expressed in GMIR, even though there are still many algorithms that remain inexpressible.

Thesis Committee:

Chair:	dr. J.G.H. Cockx, Faculty EEMCS, TU Delft
Committee Member:	dr. E. Demirović, Faculty EEMCS, TU Delft
University Supervisor:	dr. C. Bach, Faculty EEMCS, TU Delft
External Advisor:	G.H. Wachsmuth, Oracle Labs

Preface

After having worked at Oracle Labs in Zürich for 8 months, this thesis concludes my journey through the master's programme of Computer & Embedded Systems Engineering (CESE). I started this degree as the first experimental shift in 2023, as the Embedded Systems and Computer Engineering programmes were fused into CESE.

While programming languages are not necessarily central to either the embedded systems or computer engineering disciplines, I am glad I have pursued my thesis in the field. Starting all the way in the final years of high school, I started my fascination for compilers. Being somewhat bored during computer science class, I got permission from the teacher to search for some alternative projects to challenge myself. This started my fascination for compilers. I wrote a small but working compiler using ANTLR, compiling my own high-level language into Minecraft game commands.

During my bachelor's in Computer Science, I participated in the Honours Programme. I reached out to Eelco Visser, proposing to further develop my high school project into a more mature solution. With that project I slowly got to know the Spoofax Language Workbench and its inner workings. Due to Eelco's unexpected passing, Jesper took over the supervision of that project. For my bachelor's thesis, I also worked under his supervision to develop an LLVM backend for the Agda compiler [16].

The hands-on experience with Spoofax proved invaluable when I moved on to my master's. I was planning on doing a thesis in PL, but was unsure what to specialize in and was debating between different many different projects. As a prerequisite, I participated in the PL seminar course where we had to pick a paper to do a small project on. I happened to choose the 2022 SLE paper on Green-Marl IR [11], supervised by Guido Wachsmuth. A few months later, I found myself in Zürich where he became my manager.

I especially want to thank Guido and Vasileios, who provided invaluable feedback on my ideas, guided me through the patent writing process and always sparked my mind. Special thanks go to Arnaud, Riccardo and Calin for their general involvement, suggestions and enjoyable cooperation. I would also like to thank Jesper, Casper, Arno and Rutger for providing critical feedback on my text. Being at the office almost daily, I cannot have imagined what it would have been like without my excellent peers Hugo, Alexander, all Martins, Vlad, Antonio, Enrico and all other interns. I wish the graph teams at Oracle the best of luck.

Johannes Hendrik (Jochem) Broekhoff
Rotterdam, the Netherlands
March 4, 2025

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

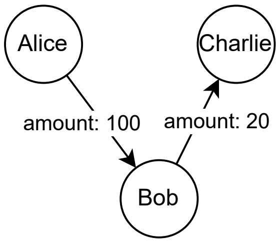


Figure 1.1: A small property graph with edge-bound properties.

Graphs are a fundamental mathematical structure which is well-suited for modeling information about the real world. For example, a road network can be modeled as a graph, where roads are represented by edges and traffic junctions by vertices. In practice, we need to store more detailed facts to make the graph interesting. Enriching a graph is done using properties, key-value pairs attached to vertices and/or edges. Figure 1.1 shows this by modeling people as vertices and monetary exchange directed edges, with an edge property representing the amount transferred.

Like any data, graphs that merely exist are not useful, especially not when they get larger. We want to run algorithms on these graphs to extract meaningful information.

Graphs can also be stored in databases and queried similar to relational data with SQL-like dialects such as PGQL [62]. Such graph query languages are only intended to be used for relatively simple queries. Although it is theoretically possible to construct convoluted queries that implement some algorithm, this is not commonly done.

Queries are insufficient, but we still want graph algorithms to be executed close to the data source. This comes from the the key principle of SQL, letting database engines handle the data-intensive tasks they are intended for as they sit directly on top of the data. For this reason, some vendors provide imperative extensions to their SQL dialects or define functions that abstract away some much more complex functionalities [5, 6, 21, 30].

Domain-specific languages (DSLs) are a different approach to enhancing database engines with more imperative analytical capabilities. They can still build on the same principles as SQL, but are more imperative than declarative instead. This aligns more with how algorithms are typically thought of and implemented. Gremlin [4] is most well-known and is used extensively in industry [7, 31, 71]. However, it focuses more on graph traversal patterns and less on supporting arbitrary algorithms. Academically, graph analytics DSLs have also gained some attention [18, 35, 82]. Green-Marl [35] in particular, one of the earliest in the field, has been used extensively by Oracle to express graph algorithms.

Green-Marl has a monolithic design and is limited in modularity. This poses difficulties in portability and compatibility with multiple targets. Recently, Oracle has introduced two new languages that address these flaws: PGX Algorithm [58] as a high-level language and a highly modular intermediate representation (IR) called Green-Marl IR (GMIR) [11]. Both can be and are actively being used to implement useful algorithms. An example of the PageRank algorithm in GMIR can be found in Listing 1.1. However, both have drawbacks that currently make it infeasible for neither non-expert customers nor experts to write maintainable and efficient implementations. The issues are as follows.

```
procedure void pageRank(
  in graph G, in int maxIter, in double epsilon,
  out property<vertex(G), double> rank
) {
  long N;
  double x;
  double diff;
  property<vertex(G), double> newRank;
  int i;

  N = numNodes(G);
  foreach ((v): vertices(G)) (true) {
    v.rank = 1d / (double)N;
  }
  x = 0.15d / (double)N;

  i = 0;
  do {
    diff = 0.0;
    foreach ((v): vertices(G)) (true) {
      double inSum =
        sum (() <- (w): neighbors(G, v)) (true) {
          let long outDeg = outDegree(G, w);
          in w.rank / (double)outDeg
        };
      double newRankVal = x + 0.85d * inSum;
      double oldRankVal = v.rank;
      diff += |newRankVal - oldRankVal|;
      v.newRank = newRankVal;
    }
    swapVertexDoubleProperties(rank, newRank);
    i = i + 1;
  } while (diff > epsilon && i < maxIter);
}
```

Listing 1.1: PageRank in classic GMIR.

- **PGX Algorithm** is a Java-based DSL, aiming to provide a Java-native way of expressing graph algorithms. The fundamental problem is that it remains too generic and not graph specific enough. It supports most of Java's flow control and provides some readily importable application programming interfaces (APIs) for graph traversal. Additionally, it provides implementations of some of Java's collections, such as lists, maps, queues. Graph-specific concepts, such as properties, are also only modeled through APIs and are thus second-class citizens. There are only few restrictions on the usage of general-purpose data structures, opening the door for unintended inefficient and longwinded algorithm implementations. This, in combination with an overall lack of graph-specific abstract data structures leads to cumbersome implementations that are hard to read.

- **GMIR**, being developed more recently, has been designed from first principles, resulting in a small and modular multi-target IR. Its lower-level design relieves some issues that PGX Algorithm has in being too general purpose. However, several classes of algorithms simply cannot be expressed in GMIR at all. This is due to the lack of higher-level abstractions and an effect of the intentional removal of general purpose features.

None of the other graph algorithm DSLs we cited earlier offer feature sets comparable to GMIR. Moreover, none actively receive contributions and most appear to be one-off works that have not been iterated on in many years. Considering this and the fact that PGX Algorithm is effectively superseded by GMIR, we use GMIR as our target language. To summarize, in this thesis we address *the lack of domain-specific abstractions for algorithmic graph processing*.

1.1 Contributions

The primary goal of this thesis is to close the feature gap that arises from the lack of domain-specific abstractions for algorithmic graph processing. We achieve this by providing new language constructs for the GMIR DSL in order to be able to express more algorithms. We arrive at this by providing the following contributions:

- A domain analysis to determine which fundamental concepts are missing and worth abstracting (Chapter 2). For more information about why we did not abstract certain concepts, please refer to the future work (Section 10.2).

- Syntax and semantics for abstractions in three subdomains. First summarized in Chapter 3 for a global overview and further refined in individual deep dive chapters (Chapters 5 to 7) in the context of the current state of GMIR (Chapter 4). These are established as a result from the domain exploration. During our employment at Oracle, we submitted the latter two abstractions as U.S. patent applications.

  - **Fixed-point iteration**: a fundamental building block for controlled repetition in many algorithms. We provide a single new control flow statement that repeats its body until a predefined number of iterations passed, some arbitrary boolean condition holds or until none of some configured variables changes anymore.
  - **Frontier exploration**: the core of algorithms that gradually but systematically explore part of a graph. We model such algorithms by a new frontier data structure and instructions to expand it and process elements from it. We further allow these algorithms to concisely extract information from the frontier by observing which elements pass through it. Example: Dijkstra's shortest path algorithm [22].
  - **Community detection**: a broad class of algorithms that try to reveal some structure in unstructured graphs. We specifically focus on algorithms that gradually group vertices together. We provide means to construct these structured communities in the form of community graphs, a special type of graph that represents relationships between vertices and their groupings. Example: Leiden algorithm [47].

- An evaluation of the designs on the design dimensions by Voelter [77] and a demonstration that more algorithms can be expressed in GMIR, partially by lowering to existing constructs (Chapter 8).

1.2 Design Methodology

DSL design needs to be done systematically. The entire domain of all algorithms that could ever be expressed is infinite and it is unpredictable what will be relevant in the future. Therefore, the main deciding factor for including a certain feature is whether or not there exists an algorithm that uses it.

Other than this, we rely on the seven design dimensions by Voelter [77]. In the evaluation (Chapter 8), we elaborate on what exactly they are and how they impact the design. Most importantly, they provide a framework for important areas to consider when designing a DSL based on a domain exploration.

Voelter also defines many other guidelines for systematic DSL design. Most applicable to our scenario are ‘iterative development’ and the ‘co-evolution of concepts and language’. Iterative development refers to the process where the language is gradually refined from some initial rough proposal. The co-evolution of concepts and language means that we start the iterative language design process during the domain analysis.

In the remaining chapters, we discuss related work (Chapter 9) and finally draw the overall conclusion in Chapter 10 that our proposed extensions to GMIR do indeed result in more algorithms being expressible.

Chapter 2

Initial Domain Exploration

In this chapter, we start from the basics and take a look at a handful of practical algorithms and analyze their essential behavior. From this, we distill the three primary aspects that we are interested in abstracting. In that context, we take a look at existing DSLs and determine their level of support. Seeing that there are no satisfactory solutions out there, we start to build our own new abstractions. We do not yet cover the intricate details and all trade-offs of different approaches, continue to Chapters 5 to 7 for that.

2.1 Graphs and Property Graphs

Before we can analyze graph algorithms, we need to establish the definitions of graphs and property graphs. As terminology varies slightly across literature, we provide our own definitions as follows which we adhere to throughout the thesis.

A *graph* $G = (V, E)$ consists of *vertices* V and *edges* E, which are both sets. Without any edges, the graph is a *null graph*. Edges connect two *endpoints*, which are vertices. If the endpoints are the same vertex, the edge is a *self-edge*. If the edges indicate a direction, the graph is considered *directed* and *undirected* otherwise. The number of edges connected (*incident*) to a vertex is its *degree*, which can be further refined in an *in-degree* and *out-degree* in the context of directed graphs.

A *path* describes a sequence of edges joining a sequence of vertices. It is *simple* if no vertex is included more than once. If it is possible to construct a path in a graph such that the start and end vertex are the same, the graph is *cyclic* and *acyclic* otherwise.

Graphs can have various connectivity properties. If any path can constructed to and from any other vertex it is *strongly connected*, or *strong*. If this is only possible when directionality is ignored, it is *weakly connected* or just *connected*. If all vertices share edges with all other vertices, the graph is *fully connected* or *complete*.

A graph consists of one or more *components*, sets of vertices that do not share any edges (i.e., are disjoint). The connectivity properties from the previous paragraph are by extension also applicable to these components. For example, a graph as a whole may not be connected, but each of its components is (by definition of a component).

Trees and forests are useful specializations of graphs; both are graphs but with additional constraints. *Trees* are connected acyclic graphs. A *forest* is a set of disjoint trees.

Most algorithms we will be working with operate on *property graphs*. A *property* is a key-value pair of a label and some arbitrary value that can be associated to vertices or edges. In practice, labels are typically strings and values are scalar values or references to other graph elements. In Figure 1.1 an example of a small property graph can be seen. Here, the property amount is attached to the edges and associates a numeric value.

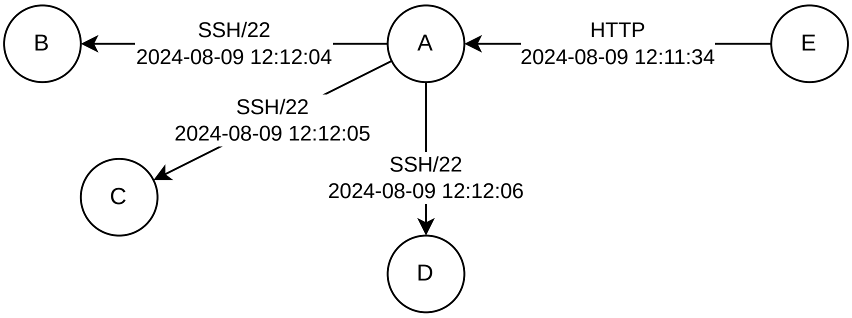


Figure 2.1: A small computer network interaction graph.

2.2 Concrete Practical Use-Cases

To give an impression of the broad applications of graph algorithms, we first take a look at how they are used in practice. Typically, just a single graph algorithm is not a product by itself. The combination of various algorithms with the necessary data wiring and further processing is what makes a product. In the following paragraphs, we describe some high-level use-cases from various disciplines to show that it is a broad landscape.

Route Planning Perhaps the graph application that people are most familiar with: navigating from A to B. A natural way of representing information about the street map is by modeling streets as edges and intersections, crossroads, roundabouts and any other infrastructure key points as vertices. The roads or edges bear properties, such as distance and typical throughput across the day to account for differences between peak and off-peak times. Addresses and in general any place of interest, can be modeled as vertices too. A request for a route from point A to B can be answered by traversing the graph starting at the vertex corresponding to point A and following the path that leads to B.

Cybersecurity Threat Detection Most software keeps activity logs, often processed in plain text format. Gathering such information from a system as a whole and representing the information as a graph allows for easier correlation of events. Consider the example drawn in Figure 2.1, where vertices represent network-connected machines and edges represent communication events between those. Properties on vertices contain meta-data such as the IP address and risk flags. Properties on edges contain information such as data, time and the communication protocol used. If the average vertex out-degree is 2, the out-degree of 3 from vertex A might signal suspicious behavior, especially because the connections were established very closely together in time. Products such as the Oracle SaaS Security Advanced Detection Platform are capable of performing such analysis.

Money Laundering Detection Banks are also highly interested in revealing patterns in graphs. By law, most banks are required to perform money laundering detection and prevention. In products such as Oracle Financial Crime and Compliance Studio, bank accounts are modeled by vertices and transactions as edges. Consider the case from Figure 2.2, where we want to detect a potentially suspicious relationship between Alice and Dave. Alice transfers some money to Bob, who transfers the same amount to Charlie, who in turn transfers it to Dave. This may indicate that Alice and Dave are controlled by the same person, while Bob and Charlie are 'mules' who may have fallen victim to becoming intermediaries in the process.

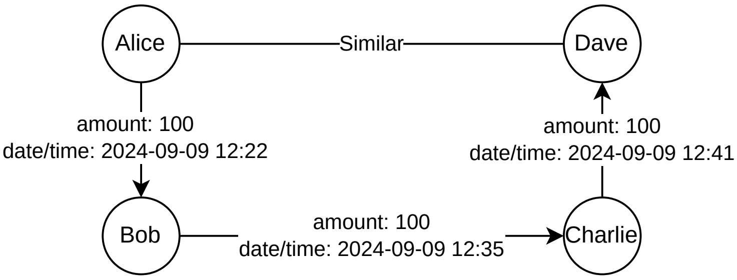
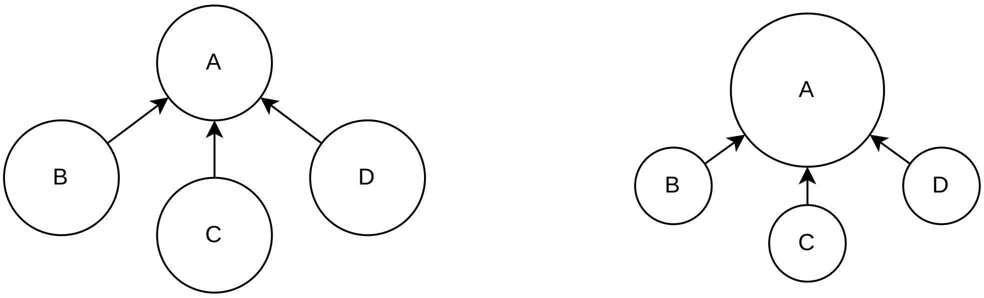

Alice
Similar
Dave
amount: 100
date/time: 2024-09-09 12:22
amount: 100
date/time: 2024-09-09 12:41
Bob
amount: 100
date/time: 2024-09-09 12:35
Charlie

Figure 2.2: Suspicious money laundering activity.



(a) Before.

(b) After convergence.

Figure 2.3: PageRank in action.

2.3 Analyzing Algorithms

In this section, we analyze a select few algorithms to get a general understanding of the variety of graph algorithms and what distinguishes one from the other. We intentionally stay at surface level here and do not go into more details. There are also many more algorithms each with their unique behavior. In the next chapter (Chapter 2), we will spend more on this and go into more detail.

2.3.1 PageRank

One of the most well-known graph algorithms is Google's PageRank [14, 55]. The algorithm basically considers the entirety of the web as a graph where vertices are the pages and edges the hyperlinks between those. The goal is to associate each vertex (web page) with an appropriate rank value that corresponds to its global popularity.

Initially, all vertices get a uniform rank value (Figure 2.3a). Then, each page linking to another gives up a small fraction of its rank value to the link destination. This process needs to be repeated several times to propagate the values until pages converge on their rank values (Figure 2.3b). While theoretically the rank values become exactly stable in the limit, we usually cut off the process once the values converge below some predetermined threshold. Even though this results in the ranking being an approximation, it is sufficient in practice. Additionally, to prevent edge cases where convergence does not occur fast enough, we may limit the amount of times the rank value is recomputed.

We can characterize this algorithm by its iterative *approximate* convergence: the algorithm modifies vertex properties and is reapplied to the same graph, but terminates when a sufficiently approximate convergence is established. Another key property of this algorithm is that in each iteration only the previous rank value is used to calculate the next, which is more generally known as a 'memoryless' or Markov property, found in many other algorithms. Keeping track of Normally, this takes some additional bookkeeping logic: at the end of one round of computation, the previous values have to be discarded, the new values become the next previous values and room has to be made for the new values.

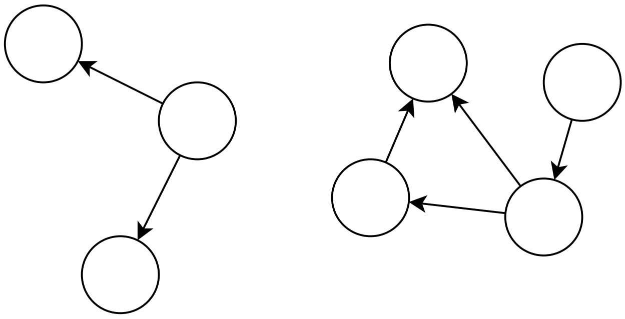
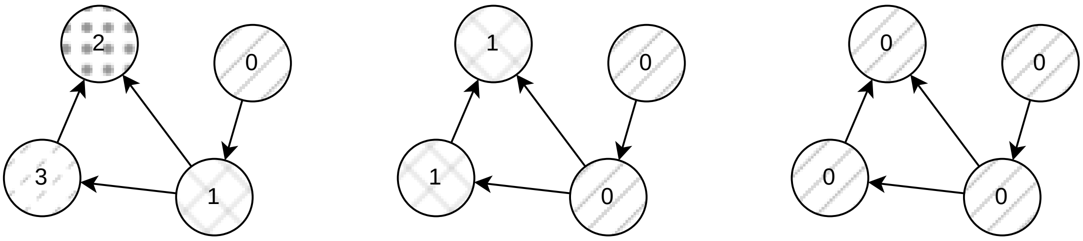


Figure 2.4: Two weakly connected components.



(a) Initial assignment.

(b) After first iteration.

(c) Convergence.

Figure 2.5: Propagation of the WCC component ID in action.

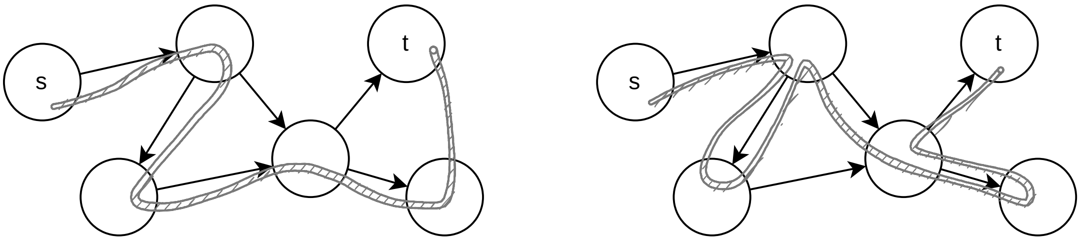
In Listing 1.1, we showed what the PageRank algorithm implementation looks like in classic GMIR, adapted from Boukham et al. [11]. Core to the implementation is the **`do-while`** loop, guarded by two conditions: the convergence condition and an iteration safeguard limit. Throughout the algorithm a temporary property `newRank` is used, which is only internal to the algorithm. It is used in combination with the builtin function `swapVertexDoubleProperties` which is assumed to be implemented by the runtime platform. The two key inefficiencies in this code are the relative verbosity of the temporary property and the suboptimal choice of the swapping builtin. Ideally when swapping, we want to forget about `newRank` as soon as it is swapped into `rank`, but that cannot be expressed, causing the memory to remain allocated.

2.3.2 Weakly Connected Components – Deterministic Variant

Another conceptually much simpler algorithm is the calculation of weakly connected components. This algorithm assigns common identifiers to vertices that can reach each other, ignoring edge directions. In Figure 2.4 a graph with two weakly connected components is drawn. While visually it is obvious that there are two components, algorithmically it is not.

One approach to solving this problem is by initializing all vertices with an arbitrary but unique value (Figure 2.5a). Each vertex then iteratively propagates the minimum value from its neighborhood, again, ignoring edge directions (Figure 2.5b). Since everybody always takes the minimum, all vertices belonging to the same weakly connected components recognize the same value at some point. This may take some time, because worst-case the value has to propagate along the entire span of the graph. As soon as no vertex finds any neighbor with a lower value, the algorithm terminates (Figure 2.5c).

This algorithm also reaches converges at some point similar to PageRank, but with integral values instead. Instead of an approximation, it reaches exact convergence on the condition that no property value changes anymore. So, we can characterize this algorithm by its *exactly stable* convergence, contrasting to PageRank.



(a) Breadth-first search.

(b) Depth-first search.

Figure 2.6: Different approaches to solving s–t-connectivity.

2.3.3 General Search in Breadth-First and Depth-First Order

Breadth-first search (BFS) and depth-first search (DFS) may very well be among the most well-known graph algorithms. Both are essentially approaches to the general search or s–t-connectivity problem. We primarily care about finding *some* vertex, an additionally may care about how. This is where BFS and DFS differentiate. Starting from an arbitrary vertex, BFS discovers neighbors and only visits them when all other earlier pending vertices have been visited. DFS on the other hand visits neighbors eagerly. Both also keep track of which vertices have already been visited. This is necessary since we are dealing with arbitrary graphs that may contain cycles. By keeping track we can simply ignore vertices that would lead to a cycle when we are about to add them.

Either way, these building blocks cover the concept of exploring a graph from a single starting node in different ways. The different iteration orders are demonstrated in Figure 2.6. The key takeaway here is that for such a general problem, the iteration order does in fact not matter. Ideally, if we were to write this down as an algorithm, we would not have to make a choice. While in practice a choice is always made, it is typically done by the programmer and not left up to the compiler.

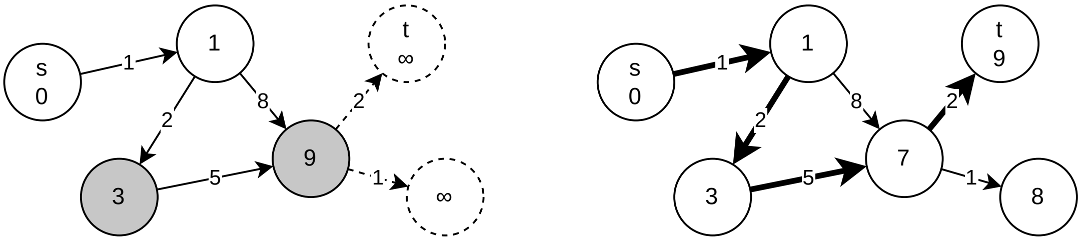
2.3.4 Directed Search with Dijkstra

Knowing the answer to the s–t-connectivity of vertices s and t is often not a satisfactory answer to practical queries. Often we have some information available about weights associated with edges. Based on this information, we then may want to optimize taking the path from s to t such that the sum of weights associated with the traversed edges is minimized. We refer to this as the general single-source shortest path (SSSP) problem.

The most well-known solution to this problem is Dijkstra's algorithm [22]. Instead of using DFS or BFS to traverse the graph, we intelligently choose which vertex to proceed with based on a heuristic. Specifically, the vertex with the shortest distance so far is selected. When visiting a vertex, we consider the direct outgoing neighborhood and ensure it is up-to-date with the fact that there may be a shorter path through the current vertex. This process effectively starts with an underestimation and slowly *relaxes* the solution until it becomes exact.

Key to Dijkstra's way of efficiently finding the shortest path is that the order occurs based on dynamic weights associated to vertices. Generalized, this is a priority-based selection scheme that functions as a generic algorithm building block just as DFS and BFS do. In fact, if the distances between vertices are all equal, Dijkstra's order is equivalent to BFS. In Figure 2.7, an intermediate state and the final found shortest path are drawn. Notice that the tentative distance of the vertex having distance 9 in Figure 2.7a is relaxed to 7 in Figure 2.7b.

Both Dijkstra's algorithm and the earlier discussed s–t-connectivity problem gradually explore the graph. While they differ in the order in which this occurs, they share two common patterns. Primarily, all have a notion of vertices pending to be processed, to which we



(a) Second iteration. Tentative distances are greyed, undiscovered vertices dashed.

(b) Shortest path found with distance 9 to t. Fat edges indicate the shortest path.

Figure 2.7: Dijkstra's algorithm in action.

refer as the *frontier*. In BFS and DFS vertices are put into the frontier and retrieved by a temporal order, whereas in Dijkstra's algorithm they are retrieved based on an associated 'best' value. More similarity can be seen in how all prevent a visited vertex from being visited again. This is achieved by rejecting such vertices from being placed into the frontier.

```
procedure boolean dijkstra(
  in graph G,
  in property<edge(G), double> weight,
  in vertex(G) root,
  in vertex(G) dest,
  out property<vertex(G), vertex(G)> parent,
  out property<vertex(G), edge(G)> parentEdge
) {
  property<vertex(G), boolean> reached;
  map<vertex(G), double> reachable;
  bool found;

  foreach ((v): vertices(G)) (true) {
    v.parent = none;
    v.parentEdge = none;
    v.reached = false;
  }

  reachable[root] = 0.0;
  found = false;
  while (!found && reachable.size() > 0) {
    vertex(G) next = reachable.getKeyForMinValue();
    if (next == dest) {
      found = true;
    } else {
      v.reached = true;
      double dist = reachable[v];
      reachable.remove(v);
      foreach (() -[e]-> (w): neighbors(G, v)) (!v.reached) {
        if (w !in reachable || reachable[w] > dist + e.weight) {
          reachable[w] = dist + e.weight;
          w.parent = v;
          w.parentEdge = e;
        }
      }
    }
  }
}
```

Listing 2.1: Dijkstra's algorithm in classic pseudo-GMIR

In Listing 2.1 we show what an implementation of Dijkstra's algorithm could look like in classic GMIR. The data structures and access patterns that are used do not exist, they are merely copied from a similar implementation in Green-Marl. This implementation uses the

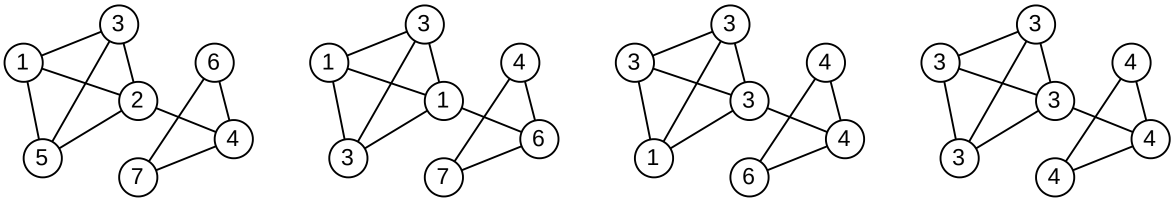


Figure 2.8: Label propagation in action.

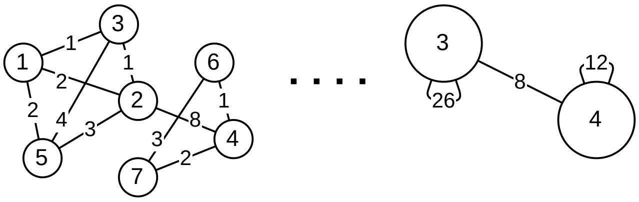


Figure 2.9: Result of running the Louvain algorithm.

general-purpose map data structure to emulate a priority queue. A vertex property is insufficient because we need to retrieve the vertex for which the value is the smallest. There exists no operator for that.

2.3.5 Detecting Community Structures in Unstructured Graphs

Another different but common algorithmic task is detecting structure in unstructured data. Doing this on graphs may reveal hidden structures of interest. While for small graphs this is a more or less trivial task for a human, it is nontrivial to automate this and scale it to graphs with millions of vertices and billions of edges. Earlier we have seen a weakly-connected components (WCC) algorithm that slowly propagated information about membership to a particular graph component. While this reveals the structure of weakly-connected components, that structure is not terribly useful. We are interested in finding structure on a much more fine-grained level, dependent on much more information hidden in the graph.

With a seemingly minor modification we can extend WCC to do so. Instead of selecting the minimal value of all neighbors, we instead select the *most frequent* value. This is better known as the label propagation algorithm (LPA) [60], which asynchronously propagates labels, or communities, and terminates when the assignment stabilizes. In Figure 2.8 a potential iteration of such an algorithm can be seen. Note that due to the asynchrony the result is not deterministic and the number of iterations required for convergence is neither.

From now on, we will refer to any form of vertex and/or edge grouping based on some similarity, relatedness or membership metric as *communities*. There exists no universally agreed upon definition of what communities are as most literature relies on the intuitive definition of what a community within a graph is.

While LPA was an innovative algorithm at its time in 2007, it only took about a year for Blondel et al. [10] to publish their highly influential Louvain algorithm. Although this algorithm attempts to solve the same problem, it uses a significantly different approach. Instead of continuous and asynchronously propagating some information, the Louvain method gradually builds communities by joining previous ones together, starting off with each vertex being a unique community. It does this by using an earlier proposed metric called *modularity* [54], a single number in range $[-1, 1]$ that describes how strongly communities are separated relative to an average random assignment. This metric considers relative importance of relations between edges too, based on weight values on the edges between.

Optimizing for modularity is not as simple as selecting the neighbor with the best value. Instead, selection is done by checking all neighbors and determining whether joining their community would result in a positive modularity increase and then picking the best one. If for all vertices none does, the algorithm terminates.

We characterize these community detection algorithms by their need to modify the graph. This is done with the intention of superimposing structure on an underlying unstructured graph. Some do this by gradually converging to a stable assignment, while others follow a strict hierarchical approach. The hierarchical algorithms specifically can be subdivided further into agglomerative and divisive algorithms. The Louvain method is agglomerative, in the sense that each in each iteration it only causes agglomeration of communities. Divisive algorithms, those which work in the opposite way, cutting global community into smaller and smaller ones, exist too but are not covered in this thesis.

Chapter 3

Design Overview

This chapter serves as a summary of all our work without elaborate justifications for why certain choices were made the way they were. Before proposing the solutions, we first explain our general design procedure that we applied across all areas. Following this, we reiterate and elaborate on key insights from the domain exploration from the previous chapter, revolving around a few important algorithms. Finally, we demonstrate our proposed solution with some concrete code snippets and accompanying explanation.

In the following chapters, we go into more detail. Starting with Chapter 4, we describe the current state of GMIR. Then in Chapters 5 to 7 we cover each of the three key domain areas in more detail.

As mentioned briefly in the introduction, we chose to extend GMIR primarily due to its modularity and relative maturity. Alternatives we considered are classic Green-Marl [35] and Java-based PGX Algorithm [58]. Both are more mature and used every day in production systems, while GMIR is more experimental. Green-Marl has not received any academic nor much commercial attention besides regular maintenance and gradual introduction of minor features. In fact, it has effectively been phased out in favor of PGX Algorithm which caters towards programmers more familiar with Java's syntax. GMIR on the other hand was only introduced recently (2022) and is actively maintained, experimented with and has been extended with new backends, adapting to changing business needs.

3.1 Design Approach & Guidelines

Designing a DSL needs to be done systematically. Our goals are ambitious and try to cover different large independent domain areas. At first glance, the sheer amount of algorithms in existence and their subtle variations seems overwhelming. There can be a gut feeling that certain topics are related, but this needs to be formalized and most importantly communicated clearly. For this reason we adhere to principles set forth by Voelter [77], as we mentioned in the introduction.

The work we present in this thesis is the final version, even though many iterations have gone by before we reached this point. We achieved this by doing it in the following way. Before all, we started with the overall domain analysis, understanding the high-level differences of various algorithms. Having discussed and agreed upon this with the team, we started a design cycle for each of the identified aspects. First, we start analyzing the domain by discovering many algorithms that we intend to cover. Based on the original publications, public and internal implementations we inform about the common patterns. Then, we make an initial attempt at constructing DSL syntax to express these patterns concisely and construct the corresponding semantics. Some aspects rely on the conceptual presence of domain specific data structures. While these may be lowered to primitive data structures in practice, we simultaneously have to develop the interface of the conceptual data structures for some aspects. From this point, we gather feedback from the team and iterate. Each iteration we express several algorithms in the proposed design to verify its expressiveness.

While Voelter's book covers DSL design in general, we are working with a commercial solution that has some additional requirements. In summary, this project was also under the following additional constraints, in no particular order:

- Although we should focus on increasing expressiveness, we should not necessarily be discouraged from using more general-purpose constructs in internal representations. However, we should prevent or certainly discourage users from using this in algorithms directly. That is more the responsibility of a higher-level language.
- One exception where general-purpose collection data structures do not have to be discouraged is in the algorithm parameters. This is more efficient for integration reasons with other products. It can also be used to intentionally hide certain implementation details of an algorithm. However, it is usually preferred to use more generic input and more specific output parameters.
- GMIR is an IR and is hence not meant to be user-facing. Although user-friendliness is not the first priority, we should not needlessly sacrifice readability. Writability however may be compromised for faster parsing.
- Existing algorithms cannot be broken. Backward compatibility must be guaranteed.
- New constructs should aim to conform to the style of existing constructs.
- Newly introduced normal forms must be checked for compatibility with existing normal forms and vice-versa.
- New features must retain the composable nature of the language, this relates to Voelter's language modularity dimension.

3.2 Introducing New Abstractions in GMIR

Having seen and analyzed various algorithms in more detail, we can go back to the drawing board. In summary, there are four major areas that we want to abstract into new language constructs: fixed-point iteration, frontier exploration, community detection and stochasticity. In the following subsections we discuss each aspect in more detail. We stay at an informal level and do not yet cover the full domain analysis, design considerations, edge cases and potential further improvements. For the full details of each aspect, refer to Chapters 5 to 7.

3.2.1 Fixed-Point Iteration

```
double sumValue;
fix (rank) [int[0..maxIter) i] {
  foreach ((v): vertices(G)) (true) {
    double inSum = sum (() <- (w): neighbors(G, v)) (true) {
      let long outDeg = outDegree(G, w);
      in current(w.rank) / (double)outDeg
    };
    deferred(v.rank) = x + 0.85d * inSum;
  }
  sumValue =
    sum((v): vertices(G)) (true) { |current(v.rank) - deferred(v.rank)| };
} until (sumValue < epsilon)
```

Listing 3.1: Main PageRank loop using the `fix` operator.

In Listing 3.1 we introduce a new `fix` operator by example. This is a snippet from the PageRank algorithm earlier seen in Listing 1.1. The main body of the algorithm is rewritten here into a new construct, but expresses the same meaning. In fact, it is possible to construct a transformation the produces the same code as the original.

Bounds on the number of iterations are specified as `[int[0..maxIter) i]`, where all parts are user-defined. The type must be integral, i.e. `int` or `long` and the interval be left-bounded, i.e., either $[\cdot, \infty)$ or $[\cdot, \cdot)$. While it is possible to hard code the iteration limit by providing a literal value instead of `maxIter`, it is best practice to make at least the upper limit an algorithm parameter.

Note that the convergence condition is expressed significantly differently. The intention is to converge when the rank value becomes more-or-less stable, which we now express as a fully separate expression. For the purposes of introducing new abstractions, we prefer clear expressive algorithms and assume optimizations are performed by the compiler.

Notice that we calculate the variables necessary for the convergence condition in the body already. Since `fix` is a core block, so has to be the expression in `until`. While this may write slightly unnaturally, this is fine because GMIR is not intended to be written by hand.

For the full details, refer to Chapter 5.

3.2.2 Frontier Exploration

```
found = false;
min frontier<vertex(G), double> distance;
init distance (src, priority: 0.0);
visit distance ((v)) [priority: double vDistance] terminate if (v == dst) {
  found = true;
} process {
  foreach (() -[e]-> (w): neighbors(G, v)) {
    double eWeight = e.weight;
    expand distance (w, priority: vDistance + eWeight);
  }
}
```

Listing 3.2: Dijkstra's algorithm using the `visit` block.

To express algorithms that perform a frontier exploration, we introduce a new block statement `visit` as demonstrated in Listing 3.2. This block takes the frontier to be visited, a pattern by which to extract elements from it, context variables, an optional termination condition and a mandatory body from which the frontier can be further expanded.

We initialize the exploration by first declaring which type of elements we want to process and by what policy. In this example, a min-weighted vertex frontier is selected. Other variations are possible to account for different usage patterns.

Once an initial set of elements has been placed into the frontier, the `visit` block and its content will be executed for each element leaving the frontier. First, if provided, the termination condition is checked. If it holds, no further action is taken and the optional block associated with it gets executed. In this Dijkstra implementation, we use that to record the fact that we terminated early due to finding the destination. Next, if present, a block of arbitrary processing is executed. Finally, expansion happens similar to the initialization.

For more in-depth details, refer to Chapter 6.

3.2.3 Community Detection

To express algorithms that perform community detection, we introduce a new block statement `agglomerate`, as extensively demonstrated based on the Louvain method in Listing 3.3. This block captures all the operations related to grouping vertices into communities. We need to step through this code snippet to understand what is going on.

```
 1  procedure void louvain(
 2    in directed graph G, in property<edge(G), double> weight, in int maxIter, in int nbrPass, in double tol,
 3    out property<vertex(G), long> com
 4  ) {
 5    community graph(G) CG;
 6    community property<vertex(CG), double> degIn;
 7    community property<vertex(CG), double> degOut;
 8    community property<edge(CG), double> cgWeight;
 9    double m; double modularity; double overall_mod_delta;
10
11    m = sum (() -[e]-> (): edges(G)) (true) { let double eWeight = e.weight; in eWeight };
12    project communities of CG into com;
13
14    build initial (CG as G => Aggr, weight -> cgWeight) {
15      foreach ((v): vertices(G)) (true) {
16        vertex(Aggr) c = community of v in Aggr;
17        // calculate selfSum, inSum and outSum by aggregation over neighbors
18        c.containedWeight = /* selfSum */;
19        c.degIn = /* inSum */;
20        c.degOut = /* outSum */;
21      }
22    }
23
24    modularity = globalModularity(G, weight, m); // implemented in a helper procedure
25    fix (modularity) [int[0..nbrPass) p] {
26      agglomerate (CG as Base => Aggr) arrange {
27        fix (inplace Aggr) [int[0..maxIter) i] {
28          for ((baseVertex): vertices(Base)) (true) { // must be sequential
29            vertex(Aggr) currentCommunity;
30            vertex(Aggr) bestCommunity;
31            double bestModularityDelta;
32            // (snip: more declarations)
33            foreach (() - (neighbor): neighbors(Base, baseVertex)) (true)
34              group by (vertex(Aggr) candidateCommunity = community of neighbor in Aggr;) (
35                let bool candidateIsCurrent = verticesEqual(Aggr, candidateCommunity, currentCommunity);
36                in !candidateIsCurrent
37              ) {
38              // (snip: local helper declarations)
39              simulate (Base => Aggr, move baseVertex from currentCommunity to candidateCommunity) {
40                foreach (() -[eb]- (): neighbors(Base, baseVertex)) (true) case { /* (snip: update locals) */ }
41              }
42              bestModularityDelta <bestCommunity> max=
43                (/* snip: modularity delta formula */) <candidateCommunity>;
44            }
45
46            deferred(modularity) = deferred(modularity) + bestModularityDelta;
47
48            execute (Base => Aggr, move baseVertex from currentCommunity to bestCommunity) {
49              // (snip: side-effects)
50            }
51          }
52        }
53      } aggregate (() -[baseInterEdge]-> (): edges(Base)) into (() -[newCommunityEdge]-> (): edges(Aggr)) {
54        double baseCgWeight = baseInterEdge.cgWeight;
55        newCommunityEdge.cgWeight += baseCgWeight;
56      }
57
58      overall_mod_delta = |current(modularity) - deferred(modularity)|;
59    } until (overall_mod_delta < tol)
60  }
```

Listing 3.3: The Louvain method using the `agglomerate` block.

Looking at the signature (1–4), we receive as in-params (2) a `weight` vertex property and several configuration parameters to constrain the algorithm runtime and accuracy. As out-param (3), we see a single `out` vertex property, intended to be populated such that vertices with equal values belong to the same community.

Within the algorithm, we first declare the community data structures (5–8) and set up some helper variables (9, 11). The takeaway here is that we can locally declare a new instance of a community graph and let it be bound to the underlying graph `G` (5). On this community graph we declare properties (6, 7) which are only visible to elements of the community graph and do not back-propagate to the underlying graph.

Before we can start arranging vertices into communities, the community graph needs to be initialized once to create vertices, edges and populate the community properties (14–22). As we are performing agglomerative community detection, each vertex will initially be assigned to a unique singleton community. This initial community is represented as a vertex in the graph that we name `Aggr` and is retrieved using the `community of` expression. On this representative community vertex, we initialize the relevant vertex-bound community properties (18–20). The edge-bound community property `cgWeight` is initialized by a shorthand notation (`weight -> cgWeight`) which indicates a one-to-one copy of values.

Now we arrive at the main algorithm body (24–59) which is wrapped in a `fix` block (25) which repeats until the `modularity` variable, Louvain's main objective, converges. The `agglomerate` block itself (26–56) is split into two parts: `arrange`ment and aggregation (`aggregate`).

In the arrangement block (27–52), the algorithm attempts to find a vertex-to-community association that is optimal and stable. This is what the inner `fix` block signifies: `inplace Aggr` means to observe the `Aggr` graph and terminate the repetition if none of its vertices–which represent communities–are modified.

Mildly nested, we finally arrive at the logic that actually optimizes the community assignment (28–50). Summarizing from Section 2.3.5, Louvain sequentially considers each vertex (28) and determines which neighbor's community (33–37) is most optimal to be associated with. The quality of such a move is determined by a `simulate`d or dry-run move, while observing side effects (39–41). After all neighbors are considered, we actually move the vertex according to what is best (48) and apply the relevant side-effects to the community vertices (49, omitted).

Finally, once a stable arrangement has been reached, we enter the `aggregate` block (53–55). This creates the new community graph which represents the aggregated form of the vertex-to-community assignment created during the arrangement phase. Vertices belonging to the same community are represented by a single vertex, which are effectively the same as the vertices from the graph we named `Aggr`. Edges between communities, however, are only constructed during the aggregation. The pattern (53) and body (54, 55) of the `aggregate` block essentially define the rule to reduce one or more edges from the `Base` graph into a single edge in the `Aggregate` graph.

Once the `agglomerate` block ends, the community graph `CG` is ready to be refined in a second pass. This happens whenever the `modularity` has not converged enough. For further details and more elaborate reasoning for the design choices, please refer to Chapter 7.

Chapter 4

Current State of GMIR

In this chapter, we provide the necessary background information about the GMIR language [11] in order to contextualize the content of the following chapters. We first cover a more in-depth history of GMIR, followed by the existing syntax and static semantics, including conventions used in other chapters. This prepares for Chapters 5 to 7, where we gradually build up the language concepts for the three key domain areas.

4.1 Brief History

More than a decade before GMIR was even conceptualized, Hong et al. [35] published the Green-Marl DSL which had already been in development for several years at that time. Meanwhile, the language continued to evolve as it is part of a commercial product [12, 37]. Only in 2022, Boukham et al. [11] introduced GMIR to simplify and increase reusability of the compilation pipeline. The classic Green-Marl language maps to it, as does the more recently introduced PGX Algorithm [58], a Java-based DSL. Effectively, GMIR serves as the common ground for these languages, but with support for other front-end languages in the future too. Most importantly, it enables compilation of all of these languages to all the backends that the GMIR compiler supports.

What sets GMIR apart from Green-Marl is not only that Green-Marl is intended to be a general-purpose language and GMIR an IR. Modularity and composability are its primary selling points, providing a solid basis which can be adapted to various needs. This allows the coexistence of multiple lean backends that pick and choose from optimizations, normal forms and lowerings for their need. Similarly, this leaves room for flexibility in the front-ends: new language features can be added without breaking backwards compatibility, provided a lowering transformation is implemented. However, as soon as a new backend is added or an existing backend gains first-class support for such a new feature, the lowering can be opted out of and compiled natively.

4.2 Syntax

In Figure 4.1, the full GMIR syntax is summarized in Backus–Naur form (BNF). For complete explanation of all constructs, refer to the original paper [11]. It sets forth some conventions and guidelines by the way it is designed. As stated in the design principles above, we aim to adhere to these. The following is a selection of these ideas:

- Separation between core and graph operations. Generally, graph operations are syntactically invalid to use as core operations and vice versa. Consider the **`foreach`** loop for example, this is one of the statements that is necessary to use to be able to enter a

$$
\begin{array}{rcl}
p & = & td\ td^{*} \\
td & = & \textbf{procedure}\ rt\ id\ (\ pd^{*}\ )\ b \\
pd & = & pt\ id \\
pt & = & a\ t \\
a & = & \textbf{in}\ \mid \textbf{out}\ \mid \textbf{in} - \textbf{out} \\
t & = & \textbf{int} \mid \textbf{long} \mid \textbf{float} \mid \textbf{double} \mid \textbf{number} \\
 & \mid & \textbf{bool} \mid \textbf{string} \\
m & = & \textbf{mutable} \\
rt & = & \textbf{void} \mid t \\
ft & = & pt^{*}\ \rightarrow\ rt \\
b & = & \{\ vd^{*}\ s^{*}\ r\ \} \\
vd & = & t\ id\ \textbf{;} \\
s & = & \textbf{if}\ (\ e\ )\ b\ \textbf{else}\ b \\
 & \mid & \textbf{while}\ (\ e\ )\ b \mid \textbf{do}\ b\ \textbf{while}\ (\ e\ )\ \textbf{;} \\
 & \mid & lr\ =\ e\ \textbf{;} \\
 & \mid & lr\ =\ tr\ (\ e^{*}\ )\ \textbf{;} \mid tr\ (\ e^{*}\ )\ \textbf{;} \\
r & = & \textbf{return}\ e\ \textbf{;} \mid \textbf{return}; \mid \varepsilon \\
e & = & i \mid l \mid d \mid s \\
 & \mid & e\ +\ e \mid e\ -\ e \mid e\ *\ e \mid e\ /\ e \mid e\ \%\ e \mid e\ **\ e \mid\ -\ e\ \mid |e| \\
 & \mid & (\ t\ )\ e \\
 & \mid & e\ <\ e \mid e\ \leqslant\ e \mid e\ >\ e \mid e\ \geqslant\ e \mid e\ =\ e \mid e\ \neq\ e \\
 & \mid & \textbf{true} \mid \textbf{false} \mid e\ \wedge\ e \mid e\ \vee\ e \mid e \mid \neg\ e \\
 & \mid & \textbf{exp}\ (\ e\ ) \mid \textbf{log}_{\textbf{2}}\ (\ e\ ) \mid \textbf{ln}\ (\ e\ ) \\
 & \mid & lr \\
lr & = & [\ a\ \textbf{param}\ t\ ]\ id \mid [\ m\ \textbf{local}\ t\ ]\ id \mid id \\
tr & = & [\ \textbf{procedure}\ ft\ ]\ id \mid [\ \textbf{builtin}\ ft\ ]\ id \mid id \\
id & & \text{names} \\
i & & \text{integer literals} \\
l & & \text{long integer literals} \\
d & & \text{double literals} \\
s & & \text{string literals}
\end{array}
$$

(a) Imperative core constructs.

$$
\begin{array}{rcl}
t & = & \ldots \\
 & \mid & \textbf{graph} \mid \textbf{directed graph} \mid \textbf{undirected graph} \\
 & \mid & et \\
 & \mid & \textbf{property}\ <\ et\ \textbf{,}\ t\ > \\
 & \mid & \textbf{set}\ <\ et\ > \mid \textbf{sequence}\ <\ et\ > \\
 & \mid & \textbf{order}\ <\ et\ > \mid \textbf{stack}\ <\ et\ > \\
 & \mid & \textbf{queue}\ <\ et\ > \\
et & = & \textbf{vertex}\ (\ id\ ) \mid \textbf{edge}\ (\ id\ ) \\
m & = & \ldots \mid \textbf{immutable} \\
b & = & \ldots \mid \{\ vd^{*}\ am^{*}\ s^{*}\ fm^{*}\ r\ \} \\
am & = & \textbf{allocate}\ lr\ \textbf{;} \\
fm & = & \textbf{free}\ lr\ \textbf{;} \\
s & = & \ldots \mid pr\ =\ e\ \textbf{;} \mid lr\ =\ ge\ \textbf{;} \mid gi \\
pr & = & [\ a\ \textbf{param}\ t\ ]\ id.id \mid [\ m\ \textbf{local}\ t\ ]\ id.id \mid id\textbf{.}id \\
ge & = & pr \\
 & \mid & [\ t\ ]\ op\ ce\ \{\ le\ \} \mid op\ ce\ \{\ le\ \} \\
op & = & \textbf{sum} \mid \textbf{product} \mid \textbf{max} \mid \textbf{min} \mid \textbf{average} \mid \textbf{any} \mid \textbf{all} \\
ce & = & (\ ee\ )\ (\ le\ ) \\
 & \mid & (\ ee\ )\ (\ le\ )\ \textbf{group by}\ (\ ld\ )\ (\ le\ ) \\
ee & = & id\ \textbf{:}\ \textbf{vertices}\ (\ id\ ) \mid id\ \xrightarrow{id}\ id\ \textbf{:}\ \textbf{edges}\ (\ id\ ) \\
 & \mid & \_\ \xrightarrow{id}\ id\ \textbf{:}\ \textbf{neighbors}\ (\ id\ \textbf{,}\ id\ ) \\
 & \mid & \_\ \xleftarrow{id}\ id\ \textbf{:}\ \textbf{neighbors}\ (\ id\ \textbf{,}\ id\ ) \\
le & = & \textbf{let}\ ld^{*}\ \textbf{in}\ e \\
ld & = & t\ id\ =\ e\ \textbf{;} \mid t\ id\ =\ tr\ (\ e^{*}\ )\textbf{;} \mid t\ id\ =\ ge\ \textbf{;} \\
gi & = & \textbf{foreach}\ ce\ \{\ ld^{*}\ is^{*}\ \} \\
 & \mid & \textbf{inBFS}\ ce\ \{\ ld^{*}\ is^{*}\ \}\ \textbf{inReverse}\ (\ le\ )\ \{\ ld^{*}\ is^{*}\ \} \\
is & = & pr\ =\ e\ \textbf{;} \mid pr\ aop\ e\ \textbf{;} \mid lr\ aop\ e\ \textbf{;} \mid gi \\
 & \mid & \textbf{if}\ (\ le\ )\ \{\ ld^{*}\ is^{*}\ \}\ \textbf{else}\ \{\ ld^{*}\ is^{*}\ \} \\
aop & = & + = \mid * = \mid \textbf{min} = \mid \textbf{max} = \mid \wedge = \mid \vee = \\
td & = & \ldots \\
 & \mid & \textbf{iterator}\ id\ (\ pd^{*}\ )\ \{\ gi\ \} \\
 & \mid & \textbf{aggregator}\ rt\ id\ (\ pd^{*}\ )\ \{\ \textbf{return}\ ge\textbf{;}\ \} \\
tr & = & \ldots \mid [\ \textbf{iterator}\ ft\ ]\ id\ \mid [\ \textbf{aggregator}\ ft\ ]\ id
\end{array}
$$

(b) Graph constructs.

Figure 4.1: Modernized syntax of GMIR. Adapted from [11].

graph context. This is partially a side effect of the modularity of the language, but it is intentionally achieved by using distinct syntax which is lexically similar.

- Syntactic enforcement of the placement of declarations at the start of a block. While in general-purpose languages (GPLs) this may be considered a code smell, it significantly simplifies compilation. The meaning of a program does not change, it is just constrained to one specific form of expressing declarations.

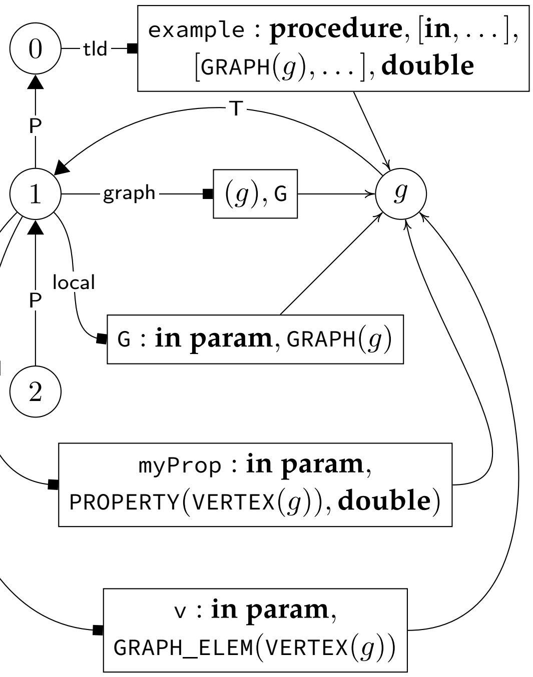
- Syntactic enforcement of memory consistency models. Core constructs are by default accessed in sequential memory consistency, but the body of a `foreach` loop is parallel in nature, as it is not a core construct, but graph-specific. It is syntactically impossible to accidentally mix sequential with parallel memory consistency.

4.3 Static Semantics

The GMIR paper does not provide any rules for static semantics, so we use the commercial implementation as the reference. The implementation is done in Statix [3, 68]. which is part of Spoofax. The basic Statix specification is about 1,200 source lines of code (SLOC), even

```
procedure double example(
  in graph G,
  in vertex(G) v,
  in property<vertex(G), double> myProp
) {
  double value;
  value = v.myProp;
  return value;
}
```

(a) Some arbitrary GMIR code


0
tld
example : procedure, [in, . . .], [GRAPH(g), . . .], double
P
T
1
graph
(g), G
g
local
P
G : in param, GRAPH(g)
local
2
local
myProp : in param, PROPERTY(VERTEX(g)), double)
v : in param, GRAPH_ELEM(VERTEX(g))

(b) Corresponding scope graph

Figure 4.2: Some code and its corresponding scope graph

though GMIR is not a complex language. The features we introduce in this thesis have also been implemented and consist of almost 1,300 SLOC, more than doubling the implementation size. Unfortunately, this is too much to cover in any paper, thesis or appendix. That means we have to select and condense the semantics to some key aspects and use concise notation. In the remainder of this section, we briefly describe some key concepts of Statix specifications, the notation we use and list some key rules that we refer to in the other topic sections.

4.3.1 Statix Specifications

Fundamentally, Statix is a constraint solver for graph representations of programs. It has a few convenience features, but remains a relatively low-level and verbose language. How exactly this graph is constructed and queried and how the constraints are composed is what a Statix specification describes.

In Figure 4.2, a small program with its corresponding scope graph is shown. Circles denote scopes, in this case 0 is the global scope, 1 is the scope for the procedure as a whole and 2 the scope of the statements within the procedure. A special scope g represents a graph instance for the `G` parameter. Referred to in text, they are prefixed with #.

Filled arrowhead edges, $\xrightarrow{\mathsf{P}}$ and $\xrightarrow{\mathsf{T}}$, represent hierarchical relationships between scopes. In our context, P indicates the parent scope and T the scope in which a type was declared (used for g). Notice that we use scopes for a unifying representation of two different concepts: lexical scope and type information. Because of that, we need to distinguish edges by labels.

Box-head edges, $\xrightarrow{\mathsf{graph}}$■ and $\xrightarrow{\mathsf{local}}$■, represent a datum declared on some scope. For example, #1 $\xrightarrow{\mathsf{local}}$■ $\boxed{\texttt{G} : \textbf{in param}, \texttt{GRAPH}(g)}$ states that a local declaration with key `G` is associated to scope #1 with additional information about the kind (**in param**) and computed type (`GRAPH`(g)).

Finally, harpoon-shaped edges ($\longrightarrow$) indicate scope indirection, essentially the resolution of a scope 'pointer'. In the example, these edges are inserted to represent the fact that all other datums reference scope #g, either directly or somewhere nested in the data.

```
signature
  sorts TYPE
  constructors DOUBLE : TYPE

typeOfExpr : scope * GMIR-Expr -> TYPE
typeOfExpr(_, DoubleLit(_)) = DOUBLE().
typeOfExpr(s, Mul(e1, e2)) = DOUBLE() :-
  typeOfExpr(s, e1) == DOUBLE(),
  typeOfExpr(s, e2) == DOUBLE().

stmtOk : scope * GMIR-Stmt
stmtOk(s, GMIR-Assign(ref, e)) :-
  {refK refT} infoOfLocalRef(s, ref)
    == LOCAL_INFO(refK, refT),
  try { accessOk(refK, WRITE()) },
  {eT} eT == typeOfExpr(s, e),
  try { eT == refT }.
```

(a) Statix specification, without syntax signatures

Expressions $\boxed{\Gamma \vdash e : t}$

$$\frac{}{\vdash d : \mathbf{double}}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{double} \qquad \Gamma \vdash e_2 : \mathbf{double}}{\Gamma \vdash e_1 * e_2 : \mathbf{double}}$$

Statements $\boxed{\Gamma \vdash s \ \textsc{ok}}$

$$\frac{\Gamma \vdash lr : \mathsf{T} \qquad \Gamma \vdash e : \mathsf{T}}{\Gamma \vdash lr = e;\ \textsc{ok}}$$

(b) Rewritten as inference rules

Figure 4.3: Statix rules and the corresponding inference rules

4.3.2 Inference Rule Notation

Since the native Statix notation is too verbose for this thesis, we use some shorthand notation partially inspired by the original paper [3], modified to account for some GMIR-specific convenience features. An introductory example is provided in Figure 4.3, where some Statix rules are compared to their counterparts in the notation of inference rules. We briefly describe each element seen in the example and describe additional constructs that are not demonstrated.

The inference rule notation is less specific than the actual Statix implementation. The actual implementation has many additional constraints that are much stricter. For example, enforcement of not writing to read-only (**in**) variables. We describe only a minimal set of rules that are primarily intended to support understanding the syntax.

Additionally, the inference rule notation does not use scope graphs. We only use a simple context Γ to store name-type pairs.

- **Expressions** indicates that the rules following are related to expressions and that the conclusions are shaped like $\Gamma \vdash e : t$. This reads "in the context Γ, the expression e is typed as t. Similarly, so does **Statements** for statements. Since statements do not have an associated result type, the inference rules only describe which statements are accepted (i.e., OK). This corresponds tho the Statix rule signatures, such as `stmtOk : scope * GMIR-Stmt`.

- *Italic* text style style refers to a symbol from the BNF grammar. An optional subscript may be used to distinguish different instances. For example, e_1 and e_2 refer to two distinct instances of an expression (e) symbol. Similarly, **bold** text refers to keywords from the grammar.

- BNF symbols with a Kleene star (such as ld^*) are not written as such, but with an overline instead: $\overline{lr}$. Premises containing ranged symbols without a corresponding conclusion that supports a ranged match are matched element-by-element. In other words, $\Gamma \vdash \overline{e} : t \equiv \underset{e \in \overline{e}}{\forall} e : t$. More specific sequence matches are written in cons notation, such as $[x|\overline{x}]$ to match x on the head element and $\overline{x}$ as the tail. $[x]$ is used for singleton sequences, a shorthand for $[x|[]]$, where $[]$ is the empty sequence.

Classifications $\boxed{t\ \textsc{numeric}}\ \boxed{t\ \textsc{core}}$

$$\frac{}{\textbf{int}\ \textsc{numeric}} \qquad \frac{}{\textbf{double}\ \textsc{numeric}}$$

$$\frac{t\ \textsc{numeric}}{t\ \textsc{core}} \qquad \frac{}{\textbf{bool}\ \textsc{core}} \qquad \frac{}{\textbf{string}\ \textsc{core}}$$

Expressions $\boxed{\Gamma \vdash e : t}$

$$\frac{}{\vdash i : \textbf{int}} \qquad \frac{}{\vdash l : \textbf{long}} \qquad \frac{}{\vdash d : \textbf{double}} \qquad \frac{}{\vdash s : \textbf{string}} \qquad \frac{}{\vdash \textbf{true} : \textbf{bool}}$$

$$\frac{}{\vdash \textbf{false} : \textbf{bool}} \qquad \frac{\Gamma \vdash e_1 : \mathrm{t} \qquad \Gamma \vdash e_2 : \mathrm{t} \qquad \mathrm{t}\ \textsc{numeric}}{\Gamma \vdash e_1 + e_2 : \mathrm{t}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{t} \qquad \Gamma \vdash e_2 : \mathrm{t} \qquad \mathrm{t}\ \textsc{numeric}}{\Gamma \vdash e_1 - e_2 : \mathrm{t}} \qquad \frac{(id : \mathrm{t}) \in \Gamma}{\Gamma \vdash id : \mathrm{t}}$$

Blocks $\boxed{\Gamma, t \vdash b\ \textsc{ok}}$

$$\frac{\Gamma \cup \{\overline{(id : t)}\}, t \vdash \overline{s}\ \textsc{ok}}{\Gamma, t \vdash \{\overline{t\ id\ ;}\ \overline{s}\}\ \textsc{ok}} \qquad \frac{\Gamma \cup \{\overline{(id : t)}\}, t \vdash \overline{s}\ \textsc{ok}}{\Gamma, \textbf{void} \vdash \{\overline{t\ id\ ;}\ \overline{s}\ \textbf{return;}\}\ \textsc{ok}}$$

$$\frac{\Delta = \Gamma \cup \{\overline{(id : t)}\} \qquad \Delta, t \vdash \overline{s}\ \textsc{ok} \qquad \Delta \vdash e : \mathrm{t}}{\Gamma, \mathrm{t} \vdash \{\overline{t\ id\ ;}\ \overline{s}\ \textbf{return}\ e;\}\ \textsc{ok}}$$

Statements $\boxed{\Gamma, t \vdash s\ \textsc{ok}}$

$$\frac{\Gamma \vdash e : \textbf{bool} \qquad \Gamma, t \vdash b_{\mathrm{T}}\ \textsc{ok} \qquad \Gamma, t \vdash b_{\mathrm{F}}\ \textsc{ok}}{\Gamma, t \vdash \textbf{if}\ (e)\ b_{\mathrm{T}}\ \textbf{else}\ b_{\mathrm{F}}\ \textsc{ok}} \qquad \frac{\Gamma \vdash e : \textbf{bool} \qquad \Gamma, t \vdash b\ \textsc{ok}}{\Gamma, t \vdash \textbf{do}\ b\ \textbf{while}\ (e);\ \textsc{ok}}$$

$$\frac{\Gamma \vdash lr : \mathrm{T} \qquad \Gamma \vdash e : \mathrm{T}}{\Gamma, \_ \vdash lr = e;\ \textsc{ok}}$$

Figure 4.4: Key static semantics rules for core constructs

4.3.3 Key Static Semantics Rules

We describe various important static semantics of GMIR necessary to understand the context in which we will introduce new features later. For brevity sake, we only state and explain some essential rules.

Core Constructs

Figure 4.4 defines some rules related to the core constructs. Numeric literals, strings and booleans are typed directly. Binary arithmetic operations such as $+$ and $-$ must have left (e_1) and right (e_2) expressions typed equivalently to a numeric type. Local references (lr) are typed as the type associated to the declaration that is resolved in the current scope.

Blocks are typed in three different ways. Without a **return** statement or with a void-

Expressions $\boxed{\Gamma \vdash ge : t}\ \boxed{\Gamma \to \Delta \vdash ee \text{ OK}}\ \boxed{op \vdash t : t}$

$$\frac{\Gamma \to \Delta \vdash ee \text{ OK} \qquad \Delta \vdash le_{\text{filter}} : \mathbf{bool} \qquad \Delta \vdash le_{\text{body}} : \mathrm{t}_{\text{body}} \qquad op \vdash \mathrm{t}_{\text{body}} : \mathrm{t}_{\text{aggr}}}{\Gamma \vdash op\ (ee)\ (le_{\text{filter}})\ \{le_{\text{body}}\} : \mathrm{t}_{\text{aggr}}}$$

$$\frac{\Gamma \vdash id_{\mathsf{G}} : \mathbf{graph}}{\Gamma \to \Gamma \cup \{(id_{\mathsf{v}} : \mathbf{vertex}(id_{\mathsf{G}}))\} \vdash id_{\mathsf{v}} :\ \mathbf{vertices}(id_{\mathsf{G}}) \text{ OK}} \qquad \frac{\mathrm{t} \text{ NUMERIC}}{\mathbf{sum} \vdash \mathrm{t} : \mathrm{t}}$$

$$\frac{\mathrm{t} \text{ NUMERIC}}{\mathbf{average} \vdash \mathrm{t} : \mathbf{double}} \qquad \frac{}{\mathbf{any} \vdash \mathbf{bool} : \mathbf{bool}}$$

Let Expressions $\boxed{\Gamma \vdash le : t}\ \boxed{\Gamma \to \Delta \vdash \overline{ld} \text{ OK}}$

$$\frac{\Gamma \to \Delta \vdash \overline{ld} \text{ OK} \qquad \Delta \vdash e : \mathrm{t}}{\Gamma \vdash \mathbf{let}\ \overline{ld}\ \mathbf{in}\ e : \mathrm{t}} \qquad \frac{\Gamma \vdash ge : t \qquad \Gamma \cup \{(id : t)\} \to \Delta \vdash \overline{ld} \text{ OK}}{\Gamma \to \Delta \vdash [t\ id = ge;\ |\ \overline{ld}] \text{ OK}} \qquad \frac{}{\Gamma \to \Gamma \vdash [] \text{ OK}}$$

Statements and Iterators $\boxed{\Gamma, t \vdash s \text{ OK}}\ \boxed{\Gamma \vdash gi \text{ OK}}\ \boxed{\Gamma \vdash is \text{ OK}}$

$$\frac{\Gamma \vdash gi \text{ OK}}{\Gamma, \_ \vdash gi \text{ OK}} \qquad \frac{\Gamma \vdash pr : \mathrm{t} \qquad \Gamma \vdash e : \mathrm{t}}{\Gamma \vdash pr = e; \text{ OK}}$$

$$\frac{\Gamma \to \Gamma_i \vdash ee \text{ OK} \qquad \Gamma_i \vdash le_{\text{filter}} : \mathbf{bool} \qquad \Gamma_i \to \Gamma_b \vdash \overline{ld} \text{ OK} \qquad \Gamma_b \vdash \overline{is} \text{ OK}}{\Gamma \vdash \mathbf{foreach}\ (ee)\ (le_{\text{filter}})\ \{\overline{ld}\ \overline{is}\}}$$

Figure 4.5: Key static semantics rules for graph constructs

returning statement, all contained statements must be OK. If an expression is returned, the expression must be typed according to the expected return type. These conditions guarantee that returns happen according to the type, but they do not enforce that a return happens at all if the type is not **void**.

The notation $\Gamma \cup \{(id : t), \dots\}$ means that a new type context is constructed, based on Γ, with the fact that identifier *id* has type *t*. If *id* already exists in Γ as a pair key, the premise does not hold (no lexical shadowing).

Statements are typed plainly by verifying that the involved subexpressions are typed consistently. The control flow statements propagate the context type to their nested blocks for the purposes of return checking.

Graph Constructs

Figure 4.5 defines more rules specific to graph constructs. Expressions from graph constructs (*ge*) are syntactically distinct from core expressions (*e*). There are two graph expressions, property references (*pr*) and aggregations (*op ce* {*le*}). Property references are typed like local references from the core (*lr*). Aggregations organize the elements expression (*ee*) and filter (le_{filter}) such that the pattern variables are visible only in the body (le_{body}). Depending on the chosen aggregation operator (*op*), there are different constraints on the type of the aggregation body which dictates the resulting graph expression type. For example, sums can be performed on any numeric type, preserving its variant, while averages always generate a **double**. Element expressions exist in many variations, we only show the **vertices** iterator here, which verifies the existence of the referenced graph and declares the name of the variable for the matched vertices in the output environment Δ, the union of Γ and the declaration of id_{v}.

Let expressions combine other named graph expressions sequentially. A let expression (le) is typed to the type of the core expression (e) it encapsulates, which is checked under $\textsc{ok}$ of all let declarations $(\overline{ld})$. The let declarations are chained sequentially, such that earlier let declarations cannot see later declarations.

Finally, graph-specific statements and iterators. The first rule states that graph iterators (gi) are valid core statements (s), but that they ignore the context type. These iterators do not have core body blocks, but iterator-specific ones $(\{ld^{*}\ is^{*}\})$. These blocks reuse the rules for the let declarations from earlier, but specify new rules for iterator statements. Note that according to the grammar of Figure 4.1, any graph iterator (gi) is also a valid iterator statement (is). The property assignment statement checks similar to the local reference assignment, but uses the property reference info query instead. The **foreach** graph iterator behaves similarly to the aggregators explained above, but has an iterator body block instead. Note that the let declarations in the iterator body are also declared sequentially.

Chapter 5

Deep Dive: Fixed-Point Iteration

Based on our initial exploration in Chapter 2, we aim to further understand the domain of fixed-point iteration. We consider some additional algorithms and use the result of that domain exploration to propose a new operator. This is subsequently implemented in GMIR, supported by various auxiliary constructs.

This chapter and the two following deep dive chapters are structured similarly. First, we introduce the particular aspect by a more in-depth domain analysis than in Chapter 2. Based on this information, we propose the new abstractions and describe them in detail, in terms of abstract data structures and instructions. The remainder of the feature section is dedicated to the realized implementation, in terms of both syntax and static semantics. We wrap the chapter up with some notes about backward compatibility and integration with existing normal forms.

5.1 Algorithm Overview

PageRank [14, 55] One of the classical and most well-known graph algorithms. The goal is to associate each vertex with a rank value that represents its relative importance. Each vertex is initialized with a uniform value. Iteratively, the score of each vertex is updated by summing and weighting the rank values of neighboring vertices along incoming edges.

In the limit, the ranks associated to each vertex converge to a stable value, but for practical purposes an approximation is sufficient. This is typical for digital floating point operations. However, in case the structure of the graph is suboptimal and the ranks only converge just faster than the cut-off rate, we would still want the algorithm to terminate within some reasonable time. For this purpose, implementations typically limit the number of convergence attempts to a fixed number. If no convergence is reached by that point, the algorithm will be terminated early with inaccurate rank values. Decreasing the maximum allowed number of iterations can also be done for other reasons, such as performance.

Personalized PageRank (PPR) [34] Whereas classical PageRank calculates rank values from an initial uniform distribution, PPR initializes the ranks differently. Due to classical PageRank's uniform initialization, all vertices can be compared among each other to create a global ranking. Personalized PageRank (PPR) enables the usage of PageRank for *personalized* recommendations. By only initializing select vertices related to a certain topic to nonzero rank values, all rank values become relative to this topic. The intuition is that due to zero-initialization of most vertices, most stay at or close to zero if they are unrelated.

Hyperlink-induced topic search (HITS) [45] This algorithm performs a task similar to PageRank, as it was originally developed around the same time to help scoring web search results. Each vertex in the graph receives two values: an authority score and a hub score. The intuition is that some web pages (vertices) contain authoritative content on a certain topic, whereas others do less or not, but function as a hub leading to authoritative content. The higher the score, the more authoritative a source or the better of an index a hub is. Iteratively, authority scores are updated based on scores of hubs linking to it and vice-versa. Scores in HITS do not necessarily converge, hence an iteration cutoff is always used.

Weakly-connected components (WCC) One approach to determining weakly connected components in a graph is by an iterative approach. Initially, each vertex is assigned a unique value, signifying a component identifier. Repeatedly, each vertex checks the values of itself and its direct neighbors and assigns to itself the minimum out of these. At some point, repeating this step another time does not affect any values. In other words, then we have reached a fixed point, fixed on the component identifier. This also means there is no need for any safeguard on the maximum number of iterations as convergence is guaranteed to be reached in at most $|V|$ iterations.

Other Algorithms In the next feature aspect sections, it will become clear that other aspects also benefit from the ideas of fixed-point iteration for concise notation. We will refine the meaning of what it means to 'fix' on a value where necessary. The algorithms mentioned so far are inherently related to this repetitive operation and only define few other statements to achieve their goal. Other algorithms use these ideas just as a building block.

5.2 Conceptual Design

We can essentially capture any bounded iterative operation under fixed-point iteration. What exactly it means to reach convergence in a fixed point is up to the algorithm, but it must be known in advance that this point will be reached. Infinite repetitions are therefore not intentionally considered.

Limiting iterations Explicit bound on a fixed-point iteration have two clear use-case. First, preventing algorithms from running too long, due to implementation errors or unexpected or suboptimal input data. Second, if there is knowledge that a certain amount of iterations is sufficient, there is no point in refining the results even further.

We expose this iteration index as a variable to the iteration body for convenience. If the algorithm does not need it, it simply does not reference it or alternatively marks it explicitly as ignored. Exposing this value may raise expectations from algorithms to be able to start the iteration index at a custom offset, for example when a loop is manually split.

This common pattern can concisely be modeled by placing an interval constraint on an integral number that is designated for the limit.

Expressing convergence conditions The algorithm needs to express when to terminate. While we have observed various different ways that can indicate termination, we limit ourselves to only two cases. We could abstract convergence conditions further into separate domain abstractions, but this can also be done easily in a revision of the language if this appears beneficial. Needlessly abstracting this in advance without a clear reason only makes it harder to adapt later.

The most generic case is by requiring the algorithm to express this as a boolean value, leaving full control to the algorithm. Alternatively, if the algorithm does not provide such a boolean value, the assumption is that the fixed-point is reached when all variables in a given list do not change their values across one iteration. This is particularly useful for discrete values, which we see in the WCC algorithm for example.

Note that in a manually optimized implementation of PageRank the Δ value is aggregated during iteration. If we specify the convergence condition separately through a `sum` aggregator, it appears less efficient. Even though some target platforms would not be able to handle the hand-optimized version natively, for those who do, the compiler should be able to optimize this as a loop fusion.

Split Versioning and Subjects We refer to the list of variables that would be given if the algorithm uses stable convergence as *subjects*. These mark variables that are to be managed by the fixed-point operator, as conceptually both the value at the start of the iteration and at the end of the iteration need to be tracked to be able to determine whether there was any change.

Considering PageRank again, we see that the rank values that are being calculated in one iteration are derived from the rank values that were either assigned initially or in the previous iteration. Exposing the internal state of the subject variables used for stable convergence reveals a similar configuration where the 'current' and 'next' (or 'old' and 'new') value are both stored. Therefore, we see opportunities to combine this notation for usage under the more generic boolean value convergence.

In our design, any variable provided as a subject is hence made accessible in both its current and next form. We refer to this means of accessing such variables as done through either the current or next *accessor*. Being a subject variable disables the conventional variable access to prevent accidental confusion.

This expressive notation for what we refer to as 'split versioning' also makes some compiler optimizations more feasible. In the original PageRank example, one iteration of the algorithm concludes with swapping the rank and next-rank properties, but this does not clean up the existing rank value. Some runtimes may use this to their advantage to prevent memory reallocation, while others can optimize better by opting to use move semantics instead and actually discard the previous values including memory altogether.

Inplace Versioning There are situations where it is not necessary, inefficient or simply undesired for subject values to be versioned. Concrete use-cases are presented under the community detection concepts, where we are only interested in observing a derived value with is not modified directly by the algorithm.

As long as the memory consistency model permit this, it is possible to both perform change detection for stability and simultaneously not need to use the versioned accessors. Another perspective on this is that it is a shorthand notation for always accessing the deferred values, under the assumption that deferred values are both read- and writable. This holds some truth because the runtime still needs to maintain some state and/or insert additional logic to perform the change detection successfully.

There is one strict requirement for inplace subjects to be accepted: there must be no boolean convergence condition. In other words, inplace subjects can only be used under stable convergence. This is necessary to prevent scenarios where all subjects are configured inplace, meaning that the until clause would be pointless as the boolean value provided could not possibly have been calculated referencing the old value to compare with, unless kept track of intentionally through a local copy or other tracking means.

Subject Modification under Propagation to Lifted Iterators We place no constraints on the body of the fixed-point iteration. This means that it may be subject to GMIR's "Top-Level Iterations" normalization [11]. If the fixed-point construct is erased by lowering before the Top-Level Iterations normalization is applied, no conflict arises. In the opposite situation, it implies that lifted subject variable access needs to happen under the same conditions as it would before being lifted. Therefore, this must be considered carefully and the lifted subject

```
procedure void pageRank(
  in graph G, in int maxIter, in double epsilon,
  out property<vertex(G), double> rank
) {
  long Ndl;
  double Nd;
  double x;
  property<vertex(G), string> strProp;
  double sumValue;

  Ndl = numNodes(G);
  Nd = (double) Ndl;
  foreach ((v): vertices(G)) (true) {
    v.rank = 1d/Nd;
  }
  x = 0.15d / Nd;

  fix (rank) [int[0..maxIter) i] {
    foreach ((v): vertices(G)) (true) {
      double inSum = sum (() <- (w): neighbors(G, v)) (true) {
        let long outDeg = outDegree(G, w);
            double wRank = current(w.rank);
        in wRank / (double)outDeg
      };
      deferred(v.rank) = x + 0.85d * inSum;
    }
    sumValue = sum((v): vertices(G)) (true) {
      let double currentVRank = current(v.rank);
          double deferredVRank = deferred(v.rank);
      in |currentVRank-deferredVRank|
    };
  } until (sumValue < epsilon)
}
```

Listing 5.1: PageRank using the `fix` operator

variables should receive an appropriate annotation such that both callee and caller agree on the contract.

Initialization Guarantees Subject variables must have **in-out** access to guarantee that initial values are never undefined. This also simplifies memory management and removes the need for complex data flow analysis.

Nesting Fixed-point iterations can be nested as long as a nested iteration does not use the same variable as a subject. By extension, it is also not possible to fix on a variable accessed through either the current or deferred accessor.

5.3 Syntax Design

See Listing 5.1 for a full example of the operator in action. The keywords highlighted **`blue`** relate to the new features. In this example, the variable `rank` is provided as the only subject. The iteration is bound to a maximum of `maxIter` iterations (exclusive upper bound), made configurable by exposing this through a procedure parameter.

See Figure 5.1 for the extension GMIR's imperative core and graph processing constructs, extending the base grammar from Figure 4.1.

(1,2) We extend integral types with support to suffix an interval. Integers and longs are the only discrete numeric types, so other numeric types have no relevancy to the fixed-point

$$\begin{array}{rrcl}
(1) & t & = & \ldots \mid \textbf{int}\ iv \mid \textbf{long}\ iv \\
(2) & iv & = & [\ e\ ..\ e\ ) \mid [\ e\ ..\ e\ ] \mid [\ e\ ..\ \infty\ ) \\
(3) & a & = & \ldots \mid \textbf{versioned} \\
 & s & = & \ldots \\
(4) & & \mid & \textbf{fix}\ (\ fs^*\ )\ [\ t\ id\ ]\ b\ \textbf{until}\ (\ e\ ) \\
(5) & & \mid & \textbf{fix}\ (\ fs^*\ )\ [\ t\ id\ ]\ b \\
(6) & fs & = & lr \mid \textbf{inplace}\ lr \\
(7) & td & = & \ldots \mid \textbf{updater}\ rt\ id\ (\ pd^*\ )\ \{\ gi\ \} \\
(8) & va & = & \textbf{current} \mid \textbf{deferred} \\
(9) & lr & = & \ldots \mid [\ a\ \textbf{param}\ t\ ]\ va\ (\ id\ ) \mid [\ m\ \textbf{local}\ t\ ]\ va\ (\ id\ ) \mid va\ (\ id\ ) \\
(10) & pr & = & \ldots \mid [\ a\ \textbf{param}\ t\ ]\ va\ (\ id.id\ ) \mid [\ m\ \textbf{local}\ t\ ]\ va\ (\ id.id\ ) \mid va\ (\ id.id\ ) \\
(11) & tr & = & \ldots \mid [\ \textbf{updater}\ ft\ ]\ id
\end{array}$$

Figure 5.1: Syntax of fixed-point extension to GMIR.

operator. We also only support a select three intervals: offset with exclusive limit, offset and inclusive limit and lastly offset-only. Other variations, while mathematically valid, are not applicable to iteration indices.

(3) A new access type for versioned parameters to be used in conjunction with the Top-Level Iterators normal form (NF). This means that it replaces other access types, such as `in-out`. We do this intentionally as the semantics of `in-out` are incompatible with versioned variables and it would syntactically be inconsistent to use a modifier instead. This forces separation of concerns related to compatibility and integration with the Top-Level Iterators NF. See also (7, 11).

(4, 5, 6) We add a new statement which comes in two variations, with a boolean convergence condition (4) and without one for stable convergence (5). The `fix` statement mimics the shape of other control flow blocks. The list of local references declares the subjects, which can each optionally be decorated with a keyword to indicate 'inplace' behavior.

Even though at least one subject should be specified, we do not enforce this in syntax. This intentional relaxation of the syntax is compensated for by stricter static semantics. Even if semantically invalid code is supplied, it is still parsed properly and can thus still be analyzed to some degree. If we would catch these errors while parsing, any subsequent analysis would be completely blocked.

The fixed-form declaration and initialization of the iteration index variable in square brackets is mandatory. As GMIR is intended to be generated, it is trivial for a code generator to provide default values. Since arbitrary expressions (e) are allowed for both the offset and limit, constant values can be provided. By default, the offset should be `0` and the limit `+INF`.

Additionally, we also allow the index variable type to be customized. By allowing integral numeric types to be specified, this can lead to a reduced number of cast operations in the body. Since any integral number has well-defined increment and comparison behavior, any such type suffices.

Since `fix` itself is not inherently graph-related, the body block is a core block (b).

The placement of the convergence condition is intentionally at the end. Although this notation is unconventional, it is the most natural way to express the condition. Inside the body new deferred values are assigned and the condition is only meant to be evaluated afterwards. This is comparable to a `do-while` loop.

If no condition is supplied, stable convergence is used.

(7, 11) A top-level `updater`, a specialization of an `iterator` with a additional return type, is necessary for integrating iterators present within a versioned scope with existing lowerings. This is intended to be used in conjunction with the `versioned` access type from (3).

Statements $\boxed{\Gamma, t \vdash s \text{ OK}}$

$$\frac{\Gamma \rightarrow \Delta \vdash \overline{fs} \text{ OK} \qquad t \text{ NUMERIC} \qquad \Delta \cup \{(id : t)\}, \mathrm{t} \vdash b \text{ OK}}{\Gamma, \mathrm{t} \vdash \mathbf{fix}(\overline{fs})\ [t\ id]\ b \text{ OK}}$$

$$\frac{\Gamma \rightarrow \Delta \vdash \overline{fs} \text{ OK} \qquad t \text{ NUMERIC} \qquad \Delta \cup \{(id : t)\}, \mathrm{t} \vdash b \text{ OK} \qquad \Gamma \vdash e : \mathbf{bool}}{\Gamma, \mathrm{t} \vdash \mathbf{fix}(\overline{fs})\ [t\ id]\ b\ \mathbf{until}(e) \text{ OK}}$$

Subjects and Versioning $\boxed{\Gamma \rightarrow \Delta \vdash fs \text{ OK}}$ $\boxed{t \text{ VERSIONABLE}}$ $\boxed{\#s \vdash id \text{ VERSIONED}}$

$$\frac{(lr, \_) \in \Gamma}{\Gamma \vdash \mathbf{inplace}\ lr \text{ OK}} \qquad \frac{\mathrm{t} \text{ VERSIONABLE} \qquad \Gamma \overset{\text{versioned}}{\longrightarrow\!\blacksquare} \Delta}{\Gamma \rightarrow \Delta \vdash id \text{ OK}} \qquad \frac{t \text{ CORE}}{t \text{ VERSIONABLE}}$$

$$\frac{t \text{ VERSIONABLE}}{\mathbf{property} < \_, t > \text{ VERSIONABLE}} \qquad \frac{\Gamma \vdash id \xmapsto{\text{versioned}} \_}{\Gamma \vdash id \text{ VERSIONED}}$$

Expressions $\boxed{\Gamma \vdash e : t}$ $\boxed{\Gamma \vdash ge : t}$

$$\frac{\Gamma \vdash id : \mathrm{t} \qquad \Gamma \vdash id \text{ VERSIONED}}{\Gamma \vdash va(id) : \mathrm{t}} \qquad \frac{\begin{array}{c}\Gamma \vdash id_{\mathrm{elem}} : \mathrm{t}_{\mathrm{elem}} \\ \Gamma \vdash id_{\mathrm{prop}} : \mathbf{property} < \mathrm{t}_{\mathrm{elem}}, \mathrm{t}_{\mathrm{val}} > \\ \Gamma \vdash id_{\mathrm{prop}} \text{ VERSIONED}\end{array}}{\Gamma \vdash va(id_{\mathrm{elem}}.id_{\mathrm{prop}}) : \mathrm{t}_{\mathrm{var}}}$$

Figure 5.2: Static semantics of the fixed-point iteration construct

(8, 9, 10) Somehow we need to express which version of a subject we refer to. As terminology varies between perspectives, we decide to refer using the `current` and `deferred` accessors. Alternative names are `old` and `new`, but especially `new` is a highly overloaded term.

This deferred assignment has some similarities with classic Green-Marl (GM)'s deferred assignment operator (`<=`). However, in GM it is not possible to perform a read of the deferred assignment. Moreover, it is only defined for use in parallel memory consistency mode. Assigned values only become visible when sequential memory consistency is reached. However, within the fixed-point operator, we are not necessarily dealing with concurrency: `fix` itself is sequential. All of this is especially problematic for the convergence condition: both the current and deferred values must be accessible, independent of memory consistency.

5.4 Static Semantics

Figure 5.2 defines the static semantics rules for the fixed-point operator. Due to the two different ways of convergence, both variations of the statement have to be checked separately. The subjects are checked in a nested scope, which is nested one level deeper for the body block.

Subjects that are not defined **inplace** are marked by the versioned relation on the subject scope. This indicates by name of the variable that it is only accessible through accessors. Missing from the figure is a rejection rule for local references and property references to locally-versioned subject variables without the presence of accessors.

All core types are VERSIONABLE, as are properties with VERSIONABLE value types. This effectively rejects higher-level and more complex types from being versioned, such as general-

```
procedure void example(
  in graph G,
  in-out property<vertex(G), int> myProp
) {
  fix (myProp) [int[0..+INF) i] {
    foreach ((v): vertices(G)) (true) {
      int neighMax =
        max (()-(w): neighbors) (true)
        { let int neighV = current(w.myProp);
          in neighV };
      deferred(v.myProp) = neighMax;
    }
  }
}
```

(a) Pre-normalization

```
procedure void example(
  in graph G,
  in-out property<vertex(G), int> myProp
) {
  bool lifted_0_res;
  lifted_0_res = false;
  fix (myProp) [int[0..+INF) i] {
    lifted_0_res = lifted_0(G, myProp);
  } until (!(lifted_0_res))
}
updater bool lifted_0(
  in graph G,
  versioned property<vertex(G), int> myProp,
) {
  foreach ((v): vertices(G)) (true) {
    int neighMax =
      max (()-(w): neighbors) (true)
      { let int neighV = current(w.myProp);
          in neighV };
    deferred(v.myProp) = neighMax;
  }
}
```

(b) Post-normalization

Figure 5.3: Fixed-point stable convergence under Top-Level Iterators NF

purpose data structures or graphs themselves.

Not explicitly defined in the figure, but **inplace** *lr* OK has the additional restriction that it only appears in the first **fix** variant.

The accessor reference style (*va*) effectively only adds an additional constraint that the base identifier must be VERSIONED. This holds for both normal local references and property references.

5.5 Integration with Top-Level Iterators Normal Form

Iterators are a valid statement to be used within the `fix` body. One of the NFs of GMIR lifts iterators to the top-level. Therefore, we need to consider what happens in relation to versioning and convergence.

With the introduction of the current and deferred accessors, these may now occur in the lifted iterators too. In scope of a `fix` block it is obvious that a subject variable or property must be accessed through such an accessor, but the scope graph structure does not share this information to callees. To ensure the lifted iterator correctly uses the versioned accessors if and only if it is legal, we communicate this fact with the `versioned` access modifier.

Transformations get more complex when we combine lifted iterators with stable convergence. We need to communicate with the lifted iterator that any update that occurs invalidates a stop condition. To accommodate for this generically, we introduced the new `updater` declaration. By lowering any iterator that occurs within the `fix` body to an updater instead, we have only have to combine their return values to see if any update did occur.

Figure 5.3 provides an example where the `versioned` access type is used to communicate the fact that one of the parameters should be treated as if it were a `fix` subject. The lifted `updater` returns a `bool`, which indicates whether any change was detected in one of the `versioned` parameters. This is then used at the call site and fed into the `until` clause of the rewritten `fix` block. If there would be multiple iterators, each would return a boolean value and all of them would be combined, causing termination only to occur when none of the lifted iterators returns **`true`**.

Chapter 6

Deep Dive: Frontier Exploration

Gradual systematic traversal of graphs is a common task in graph analytics. Perhaps it is what is most often associated with the field of graph analytics. Algorithms in this subdomain typically extract information from a graph without necessarily considering the entire graph at once. For example, search algorithms start from one or several starting vertices and gradually explore the graph along edges until a destination vertex is encountered. Pathfinding algorithms extend this by recording the path along which the graph was explored in order to reach a particular destination vertex. Other algorithms generalize this even further and construct trees which satisfy some specific property, such as being minimum-spanning. Such algorithms are directly defined in terms of a traversal of a graph, from which the result follows directly or with minimal post-processing. Other richer and more informed algorithms, such as the Ford–Fulkerson algorithm (FFA) for the maximum flow problem, can solve one of their parts by performing a graph traversal to answer a reachability question under some additional constraints on feasible paths.

6.1 Algorithm Overview

Exploring a graph using a frontier is a very broad concept. In Sections 2.3.3 and 2.3.4, we already briefly covered some of the major algorithms. In the following paragraphs, we restate them in a more systematic way and elaborate with minimum spanning tree (MST) construction algorithms.

S–T connectivity The problem of S–T connectivity entails the question of whether a vertex t can be reached starting at a vertex s. An algorithm for this problem is very basic and only cares about whether it is possible, independent of the path taken. This means that any traversal order that considers each possible path is sufficient.

DFS and BFS The classical search patterns DFS and BFS exhibit more typical frontier-oriented behavior. Given a starting vertex the graph is gradually traversed, but in a more systematic way compared to S–T connectivity. Typically, the goal is to find some other vertex or to collect information about the graph. Depending on the needs of the algorithm and the expected structure of the graph, a DFS or BFS traversal pattern would be chosen.

Key to understanding these systematic traversal patterns is that they cause a three-way separation of elements in the graph. Some elements are already discovered but still pending to be processed, in the order determined by the choice of DFS or BFS. Other elements have been processed already and should not be considered again, while another fraction of the graph has not even been explored yet. We visualize this in Figure 6.1 and later discuss it more in-depth.

Pathfinding The basic traversal patterns are insufficient for more advanced applications such as pathfinding with Dijkstra's algorithm or A*. Contrasting to DFS and BFS, these pathfinding algorithms are more dynamic and need the traversal order to adapt to information discovered during the exploration. For example, Dijkstra's algorithm prioritizes checking vertices that have the shortest partial path so far. In other words, the graph exploration gradually discovers more and more vertices, but the order of this discovery is disconnected from the order in which they are consequently checked.

Minimum Spanning Tree Construction A completely different problem is the construction of MSTs. There are various approaches to building them, some of which use patterns similar to other frontier exploration algorithms.

Prim's algorithm can be modeled using a frontier of edges, instead of vertices as we saw until now. From a random starting vertex, all incident edges are initially added to the frontier. Edges prioritized by the longest one first form the MST. Edges that would break the tree structure are ignored.

Kruskal's algorithm can also be modeled onto a frontier of edges, but differently. Instead of gradually exploring the graph, the frontier is initialized with all edges. The algorithm logic that runs for each selected edge still considers the graph gradually, but no new edges are ever discovered after initialization. Similar to Prim's algorithm, the selected edges form the MST. The difference is that Kruskal's algorithm builds partial trees all over the graph at the same time, effectively creating a forest. By selecting more and more edges, the trees in the forest grow and may need to merge. Eventually, this results in a single MST if the graph is connected and in a forest of MSTs otherwise.

Guaranteeing this 'treeness' of the resulting MST can be done by an auxiliary property on the endpoints of vertices to mark if they are in the tree or not. This also generalizes to forests present in Kruskal's algorithm. We believe this is a key characteristic of these algorithms that lends itself for abstraction as we discuss further on.

6.2 Essential Summary

The frontier exploration abstraction revolves around a single abstract data type, the frontier. This encapsulates all information, such as the part of the graph which is already explored, which elements are pending and any data potentially associated to this. In the remainder of this section, we gradually refine the features of a frontier in terms of the data type itself and operations on it.

Starting from the basics, the question of reachability from one vertex to another is the simplest question that cannot be answered by a plain vertex iteration. In this problem ('s-t connectivity'), we are given two vertices, a source and a destination. The question can be answered by considering the source vertex and its neighbors. If none of these are the destination, the neighbors of the neighbors are considered and so on. Effectively, the entire graph that is connected to the source vertex can potentially be considered. When the destination is not found after all reachable vertices have been considered, the conclusion must be drawn that the destination vertex is not reachable from the source. Every time that we consider a vertex and mark all its neighboring vertices as to-be-considered, we are operating on a data structure that keeps track of which vertices specifically are to be considered next, i.e. what is still pending. Representing this collection of elements is the primary responsibility of the frontier and it is what literature typically defines to be a frontier.

Kicking off the exploration, we place the source vertex in the frontier. Generalizing this, we can place any number of vertices into the frontier. For example, we may want to test whether a destination vertex is reachable from at least one of various starting vertices without explicitly computing this for each vertex. From the frontier, we repeatedly take one el-

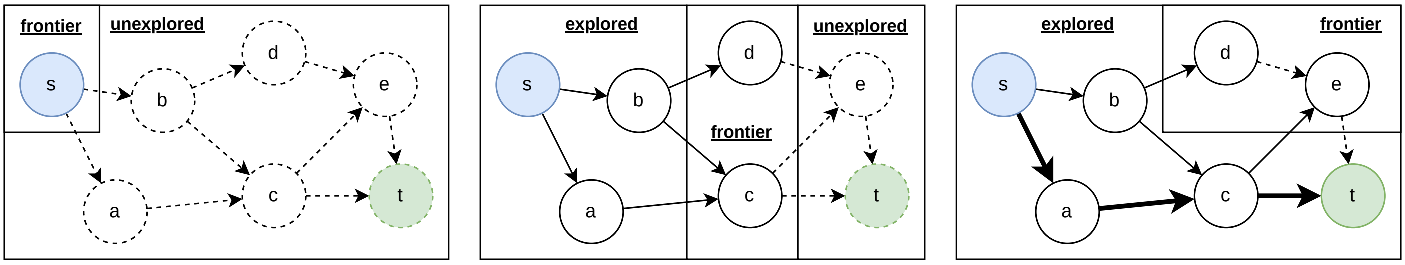


Figure 6.1: Gradual exploration of a graph through a frontier with an arbitrary ordering policy. Elements were visited in order $[s, b, a, c, t]$. A potential source-destination path is highlighted. Not all intermediate states are drawn.

ement to process further (visiting). This may happen arbitrarily if the algorithm does not care about the order in which the graph is explored. If the algorithm desires a specific order, this is specified by the ordering policy, which dictates the input-output relationship of graph elements. Upon visiting, the algorithm is free to perform arbitrary operations on the element (current element) from the frontier. We specifically provide two operations to interact with the frontier. First, the algorithm can specify a termination condition which causes the repeated visit to stop. Second, to make progress with a frontier that was not initialized by all elements of the graph, the algorithm also needs to specify which elements should be added to the frontier once the visit completes (expansion). Important is that we constrain algorithms to only expand the by elements in the direct neighborhood of the visited element. Allowing multiple-hop expansions would be redundant. This basic process is summarized in Figure 6.1. The frontier initially only contains the source (blue vertex 's'), but gradually moves towards the destination (green vertex 't'). The dotted edges and vertices indicate unexplored elements in the graph. Notice that this frontier only contains vertices: edges between vertices in the frontier are dotted, which means that they cannot have been discovered yet.

Frontiers can contain either vertices or edges, but not both. There are some subtle differences between the two, which become more important later on. Both vertex and edge frontiers can be initialized with a single, some or all vertices or edges, respectively. When visiting a vertex from a vertex frontier, that frontier can only be expanded with vertices that are neighbors of the current vertex, independent of the edge direction. When visiting an edge from an edge frontier, the situation is slightly different but captures the same idea adapted to edges: an edge frontier can only be expanded by edges incident to either endpoint of the current edge. Both vertex and edge frontiers can be used to search for a particular vertex or edge in the graph, but one pattern may lend itself better to the problem at hand.

6.3 Conceptual Design

Before we provide a concrete instantiation of our abstractions in terms of syntax, we first need to reiterate and clarify our interpretation, models, views and ideas of frontier exploration. Frontier exploration is an umbrella term for various strategies to gradually explore a graph. The goal of algorithms that perform a frontier exploration is to extract information from a graph, contextualized by the information gathered from the explored fraction of the graph. This section elaborates on the essentials described earlier.

6.3.1 Ordering Policies and Default Filters

Ordering Policy	**Accepted input** (from initialization or expansion)	**Produced output per visit**	**Remark**
Unordered/none	A single element.	An arbitrary element from the frontier.	
Last-In-First-Out (LIFO)		The element that was most recently added to the frontier.	Also known as pre-orderdepth-first search (DFS).
First-In-First-Out (FIFO)		The element that was least recently added to the frontier.	Also known as breadth-first search (BFS).
Max-weighted	A single element with an associated priority (or weight).	The element in the frontier with the highest associated priority, ties broken arbitrarily.	If the element already existed with a worse priority, it is reinserted with the better priority (relaxation).
Min-weighted		The element in the frontier with the lowest associated priority, ties broken arbitrarily.	

Table 6.2: Provided frontier ordering policies

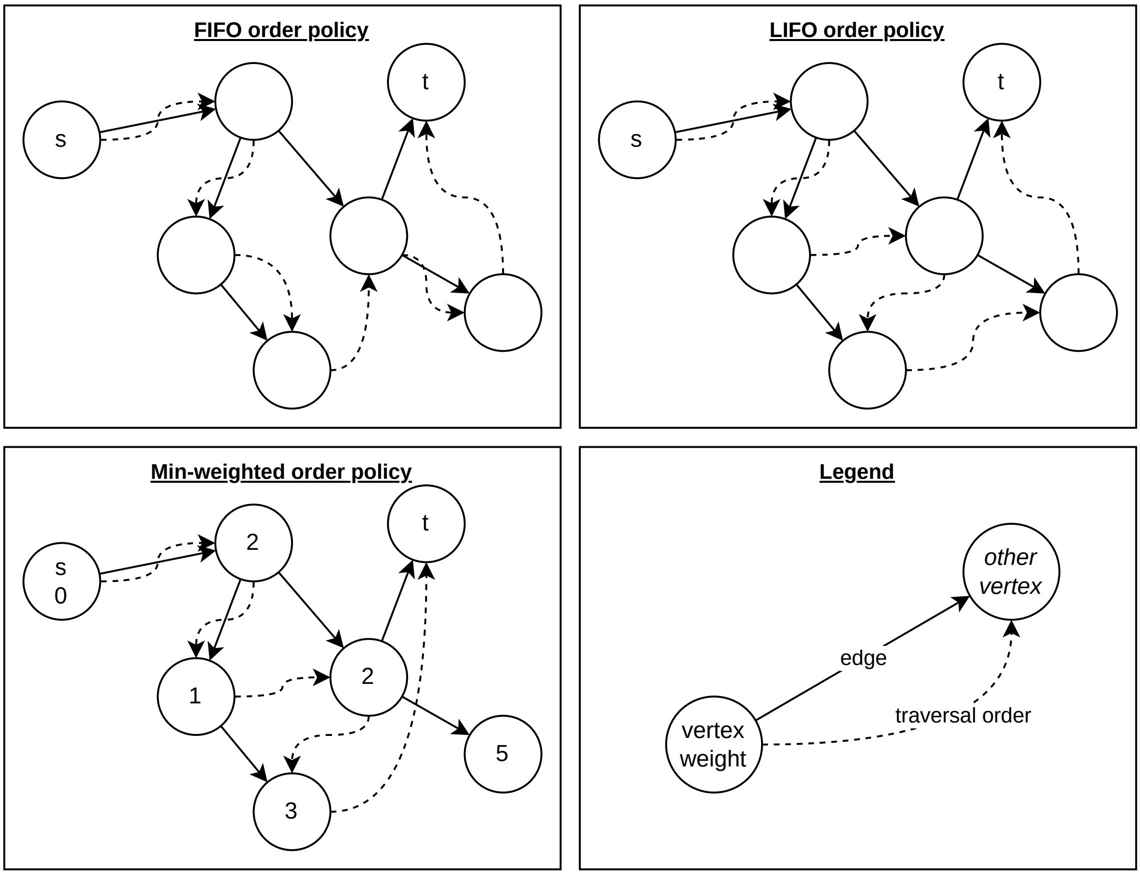


Figure 6.3: Various frontier ordering policies in action: FIFO, LIFO, Min-Weighted

As briefly mentioned earlier, the order in which elements from the frontier are visited is specified by the ordering policy. We specifically provide five different ordering policies that cover basic use cases: unordered (i.e., no ordering policy in place). These different policies are described in Table 6.2 and exemplified in Figure 6.3. An ordering policy defines the relation between elements presented at initialization or by expansion during visiting (input) and the thereafter following next visited elements (output). The trivial ordering (unordered/none) has an undefined relation between input and output. The basic orderings LIFO and FIFO define the relationship based on the input order. Min- and max-weighted ordering require the elements to be paired with an additional value, a priority. We observe that various graph traversal techniques that aim to improve on the basic ordering fit into this model without making assumptions about their implementation details (e.g., Δ-Stepping [52], Near-Far piling [20]). The initialization, ordering policy, termination and expansion logic together form the very basics of frontier exploration, visualized in Figure 6.4. In the remainder of this section, we expand on our invention by building more features on this basic framework.

So far, we have not clarified what exactly happens after an element has been visited. As drawn in Figure 6.1, the elements that were once in the frontier end up in the 'already explored' fraction of the graph. This is the default behavior of our frontier abstraction: once an element has been processed by the visit logic, it is marked as 'visited' (i.e., moved to the explored fraction). Elements by which the frontier is attempted to be expanded but that have already been visited are silently dropped. Effectively, a filter is placed on the 'expansion' → 'ordering policy' relation in Figure 6.4, populated by elements outputted by the ordering policy. Similarly, we equip the ordering policy with a filter for pending elements: elements that are already added to the frontier should not be added again. However, the ordering policy

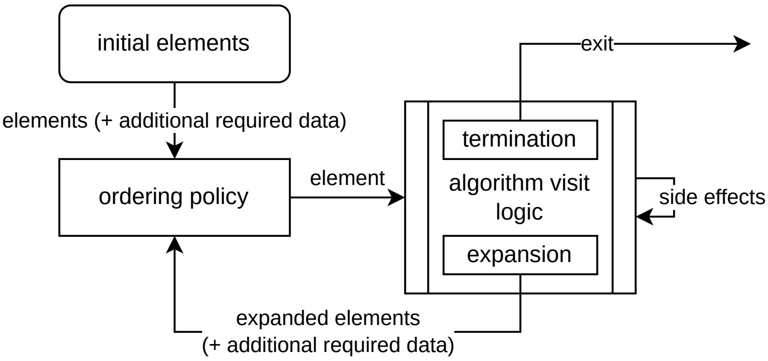


Figure 6.4: Block diagram of frontier exploration basics

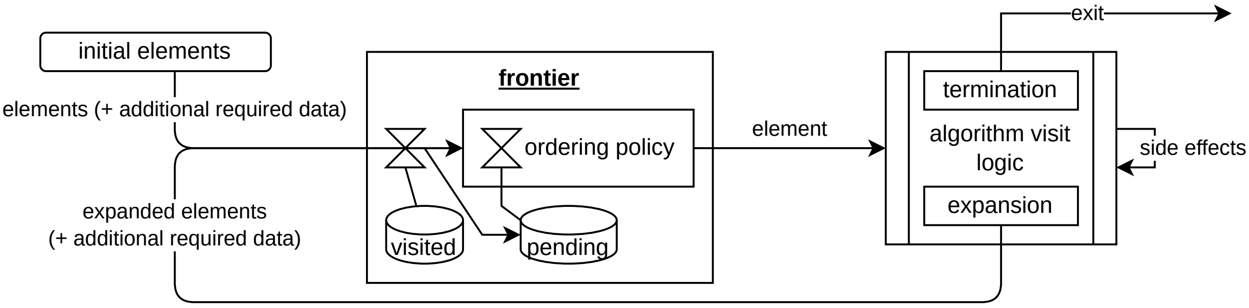


Figure 6.5: Block diagram of frontier exploration with default filters

has control over this filter to allow some elements to pass through. For example, a weighted ordering allows the priority of an element to be improved: an element with a better priority is not filtered out. Both filters are illustrated in Figure 6.5. The visited (first) filter can be ignored when the algorithm forces expansion into a certain element (revisiting), but the pending (second) filter cannot as it is tied to the definition of the ordering policy.

6.3.2 Element Enrichment by Automatic Tracking

Depth Tracking Under the concepts introduced so far, when an element is visited, there is no context provided to the algorithm in terms of where the element is positioned in the graph relative to the previously visited or pending elements. Under the concepts introduced so far, no additional information is provided to elements flowing out of the frontier. Some algorithms need to know how the element is positioned relative to previously seen elements. Common applications are depth or breadth limiting.

When requested by the algorithm, the frontier is equipped with a depth tracker which associates a depth value with each element that expands the frontier. Elements from initialization receive depth zero by definition. Elements from expansion during visit receive a depth value one greater than the current element. The depth information is passed along with the output of the frontier to the visit logic and may be used arbitrarily by the algorithm. The depth tracker only works when expansion is never forced: only without cycles can a depth value be uniquely assigned to each element in a meaningful way.

Reversing Visits The order in which we visit elements is specified by the ordering policy. Once an element is visited, it is never revisited unless expanded by force. Essentially, any visited element only has access to the information collected from elements already visited. However, some algorithms rely on information collected from elements that would be visited after itself. To accommodate for this, we provide a secondary variant of the visit block: the reverse visit.

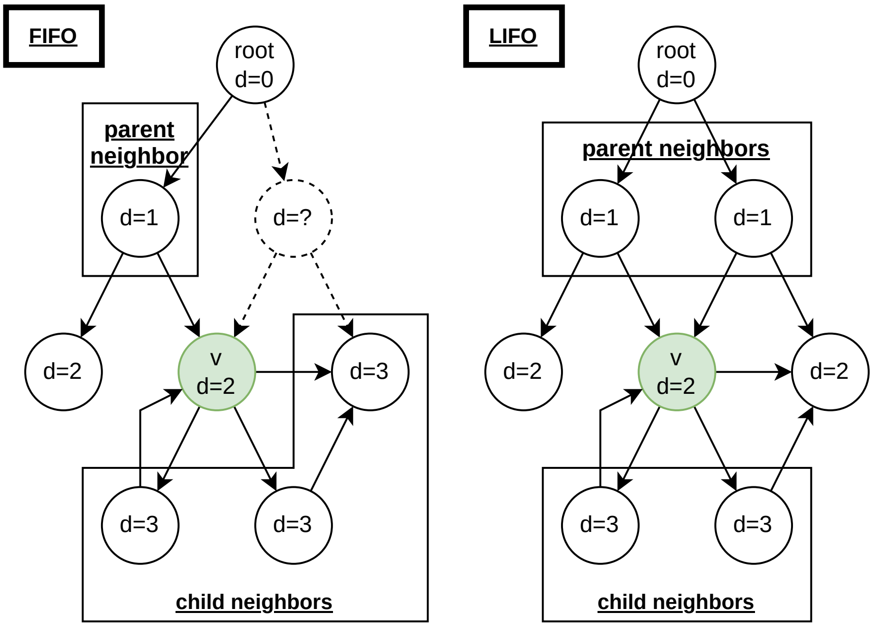


Figure 6.6: Classification of neighboring elements: parents and children

If the algorithm desires to, this is executed with the elements in the reverse order or more specifically when all elements that were caused to be visited by some element were visited. For example, if ‘a’ causes expansion to ‘b’ and ‘c’, ‘a’ is reverse-visited when ‘b’ and ‘c’ have been visited. The visiting of elements that we described earlier we refer to as the forward visit. Note that in the reverse visit, it is not possible to expand the frontier anymore. Unless specified, we continue to refer to the forward visit as ‘the visit’. It is only possible to make this distinction when expansion is never forced.

Parents–Children Tracking We extend the visit duality with an extra introspection feature: parents–children tracking. Specifically, the algorithm can request which elements led or could have led to the current element (parents) and into which elements the current element actually expanded (children). These questions can only be answered respectively during the forward and reverse visit. The degree to which this information is complete depends on the ordering policy and expansion. As the elements selected by these questions relate to the associated depths, only elements of which the depth has been established yet will be included. In Figure 6.6, the classifications are marked for a vertex in some graph where only a fraction of the graph has been explored (under a FIFO policy) and where the entire graph has been explored (under a LIFO policy). Note that if the left graph were explored further, the vertex with “d=?” would receive depth 1 too and it would appear in the parent neighbors of ‘v’.

6.3.3 Exploration Policies: Paths, Trees and Forests

To simplify pathfinding algorithms, we will now introduce several new concepts that allow for a tight integration with frontier exploration algorithms. Pathfinding algorithms, such as Dijkstra’s Algorithm, A* or any other best-first search algorithms are designed to answer the question ‘how to navigate from source (s) to destination (t), in the best way according to some metric (distance)?’. By using a weighted ordering policy, we can optimize the metric, but we cannot yet find out how to actually get there. When the destination has been found, the algorithm is tasked with finding out the path from source to destination. That is eventually the most important result that the algorithm should provide.

While it may seem possible to extract this information from the reverse visit, this is obtrusive, potentially inefficient and not always correct. Therefore, we introduce the concept of automatic path tracking. At each visit, the algorithm can request the path that would lead from one of the root elements to the current element. The critical difference with path tracking compared to parents–children tracking is that there is always one unique parent, independent of the ordering policy. With parents-children tracking, we query the neighboring elements that were or could have been parents, whereas in path tracking we only select the ultimately responsible parent. Note that the constructed path may not necessarily correspond with the order in which elements were visited. Another interpretation of this is that the frontier keeps track of a tree structure and is able to return a path from the root of the tree (tree tracking), assuming there is one initial element. Multiple initial elements generalize to keeping track of a forest (forest tracking).

We combine these ideas into the idea of an exploration policy which imposes new behavior on a frontier in order to guarantee some invariant. This is then used to provide information derived from the exploration. The different exploration policies with their features and behavior are listed in Table 6.7. Note that the None-policy is effectively the frontier configuration as drawn in Figure 6.4 and that the Setlike-policy is what is drawn in Figure 6.5. This implies that the remaining policies (Pathlike, Treelike and Forestlike) introduce new specialized behavior not seen before. Notice that the Pathlike-policy is trivially able to provide a path due to its constraints, for which it is not useful in practice. On the other hand, the Forestlike-policy is unable to provide paths due to tree fusion resulting on a potential loss of path from either root to any element (illustrated in Figure 6.8).

Exploration Policy	Features	Construction Mechanism	Initialization Filter/Behavior	Pre-Visit Filter/Behavior	Expansion Filter/Behavior
None	*None*	N/A	Any number of vertices or edges.	*None*	Element has not been visited. Can be forced.
Setlike	Preventing revisits by default.				
Pathlike	Provide a single path for the entire exploration, from the first to the last visited element.	By gradual extension along the explored elements.	Single vertex. Single edge.	*None*	Element has not been visited. At most one element.
Treelike	Provide a path on each visit from the global root. Provide a tree of the entire exploration.	By gradual extension along the element that caused expansion.	Single vertex. Any edges incident to a single vertex.	Edge: one endpoint not in tree yet.	Vertex: not in tree yet. Edge: one endpoint not in tree yet.
Forestlike	Provide a forest of the entire exploration.		Any number of vertices. Any edges that do not form a cycle.	Edge: one endpoint not in any tree OR both endpoint in distinct trees. → In the latter case, the trees are fused (a.k.a. merged or unioned).	Vertex: not in any tree yet. Edge: one endpoint not in the current tree.

Table 6.7: Exploration policies, their features and the constraints they impose

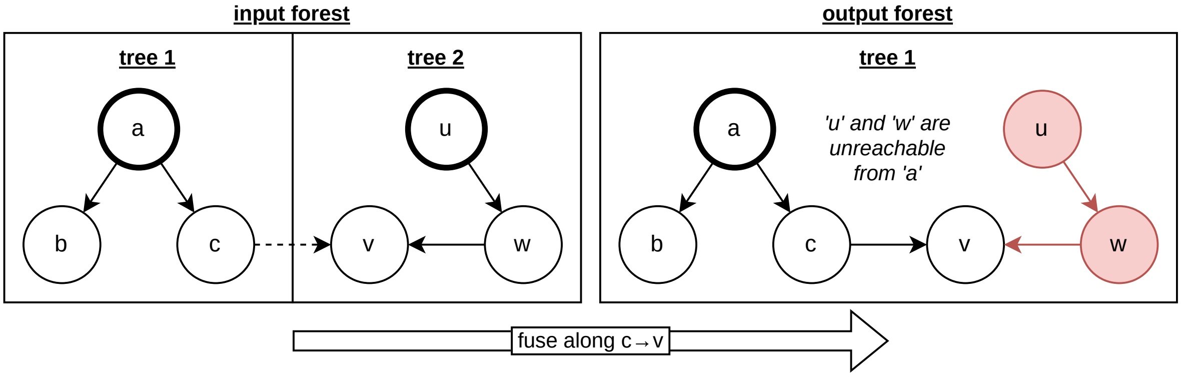


Figure 6.8: Infeasible scenario of tree fusion in a forest under directed edges, leading to loss of paths to some vertices

We make no further claims yet about how these exploration policies provide their information and how they enforce their filters and invariant. This is an implementation detail which we will cover later. The conceptual behavior, corresponding to 'Construction Mechanism' from Table 6.7 is illustrated in Figure 6.9. There, it is demonstrated that the frontier is initially expanded by vertex 'a' and 'b', causing an initial tree to be constructed over the edges incident to 'v' that caused this expansion. However, vertex 'a' is visited next and it discovers a new edge which provides a better priority for vertex 'b', by which the frontier is expanded. Although 'b' was already pending, due to the fact that a better priority was provided, the tracked tree is updated to reflect this fact. Next, 'b' is visited and if the path would be requested from 'v' to 'b', the path along the '3' and '1' edges would be provided as intended.

6.3.4 Parallelism

So far, we have only considered visiting elements from the frontier sequentially. At each visit iteration, one element is received from the frontier and the frontier is expanded by zero or more elements, repeating until the frontier is empty or the algorithm explicitly terminates. Our design allows for some frontier configurations to leverage parallelism by visiting multiple elements simultaneously. Processing elements in parallel improves overall algorithm throughput, insofar as the executing machine provides parallelism. We define parallel vis-

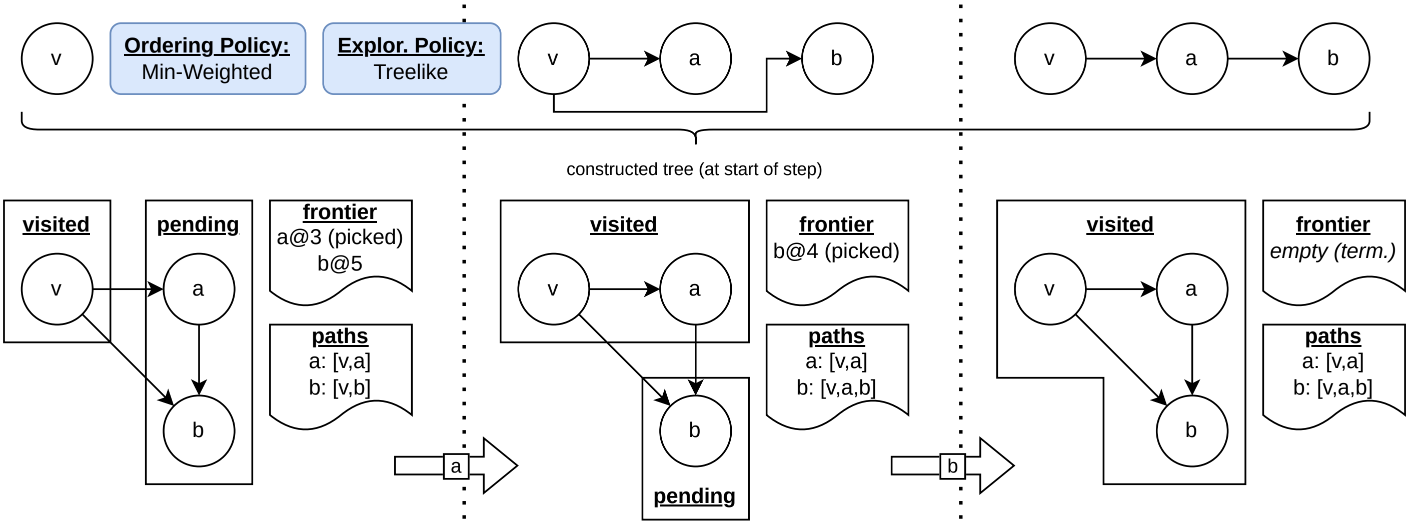


Figure 6.9: Gradual tree and path construction under a min-weighted treelike exploration

Projection Aspect	Value
visited tracking	set of elements that were visited
depth tracking	mapping from element to final associated depth, undefined/infinite for non-visited elements
path/tree/forest tracking	path/tree/forest data structure representing the exploration

Table 6.10: Projectable information of a frontier

its only for frontiers with an unordered or LIFO ordering policy. We cannot parallelize the FIFO order in general due to intrinsic limitations of DFS [61]. Weighted order is similarly not parallelizable in general, as the order of elements in the frontier can be arbitrarily and globally influenced by the elements and associated values by which the visit block expands the frontier.

6.3.5 Information Projection and Explicit Memory Management

Most of the information that is stored in the frontier so far is for internal purposes only, to enable certain behavior. While this leaves complete freedom of implementation up to the compiler, this information is not accessible to the algorithm except through the dedicated constructs that use this information indirectly. For example, most frontier configurations keep track of which elements have been visited for revisit deduplication purposes, yet the algorithm cannot directly access this information to inquire if a given arbitrary element has been visited. Similarly, another pattern we recognize in algorithms is the need to possess of this information after execution of the frontier exploration.

To some degree, it is possible to keep track of this manually by updating a set on each visit. However, this essentially leads to information duplication inside and outside the frontier. For this reason, we introduce the concept of information projection: a mechanism to extract internal information from the frontier without exposing any of the internal working of the frontier itself.

Table 6.10 lists what information is projectable.

Projection is closely tied to memory management. An implementation could reuse memory allocated to the projection destination for its internal purpose. This relates to the goal of reducing information duplication.

Consider the scenario where an algorithm that searches for a destination vertex in a graph from a starting vertex. Besides knowing whether or not the destination is reachable from the source, the algorithm also wants to know post-exploration which vertices were visited. Suppose the algorithm has allocated a boolean vertex property for this information. By requesting the frontier to project this information into this boolean vertex property, the runtime may decide to directly use this property for its own tasks involving access to a set of visited elements (to perform revisit deduplication).

6.4 Syntax Design

Figure 6.11 formalizes the grammar of all frontier exploration-related constructs.

(1, 3–6) We introduce a single complex type for frontiers, which captures all policy configuration options and element types.

$$
\begin{array}{rrcl}
(1) & t & = & \ldots \mid \mathit{fe}\,f \\
(2) & & \mid & \textbf{path}\ (\ \mathit{id}\ ) \mid \textbf{tree}\ (\ \mathit{id}\ ) \mid \textbf{forest}\ (\ \mathit{id}\ ) \\
(3) & \mathit{fe} & = & \textbf{none} \mid \textbf{treelike} \mid \textbf{forestlike} \mid \varepsilon \\
(4) & f & = & \textbf{unordered frontier}\ <\ \mathit{et}\ > \\
(5) & & \mid & \textbf{lifo frontier}\ <\ \mathit{et}\ >\ \mid \textbf{fifo frontier}\ <\ \mathit{et}\ > \\
(6) & & \mid & \textbf{min frontier}\ <\ \mathit{et}\ ,\ t\ >\ \mid \textbf{max frontier}\ <\ \mathit{et}\ ,\ t\ > \\
 & \mathit{is} & = & \ldots \\
(7) & & \mid & \textbf{init}\ \mathit{lr}\ \mathit{ft}\ ; \\
(8) & & \mid & \textbf{expand}\ \mathit{lr}\ \mathit{ft}\ \mathit{fp}\ ; \\
(9) & \mathit{ft} & = & (\ \mathit{lr}\ ) \mid (\ \mathit{lr}\ ,\ \textbf{priority} : e\ ) \\
(10) & \mathit{fp} & = & \textbf{force} \mid \textbf{then}\ b \mid \varepsilon \\
 & s & = & \ldots \\
(11) & & \mid & \textbf{bind visited of}\ \mathit{lr}\ \textbf{to}\ \mathit{lr} \mid \textbf{bind depth of}\ \mathit{lr}\ \textbf{to}\ \mathit{lr} \mid \textbf{bind tree of}\ \mathit{lr}\ \textbf{to}\ \mathit{lr} \mid \textbf{bind forest of}\ \mathit{lr}\ \textbf{to}\ \mathit{lr} \\
(12) & & \mid & \textbf{visit}\ \mathit{fv}^*\ \mathit{fx}\ b \\
(13) & & \mid & \textbf{visit}\ \mathit{fv}^*\ \mathit{fx}\ \textbf{parallel}\ \{\ \mathit{le}^*\ \mathit{is}^*\ \} \\
(14) & \mathit{fv} & = & \mathit{lr}\ (\ \mathit{fm}\ )\ [\ \mathit{fc}^*\ ] \\
(15) & \mathit{fm} & = & \mathit{id} \mid \mathit{id}\ \overset{\mathit{id}}{\longrightarrow}\ \mathit{id} \\
(16) & \mathit{fc} & = & \textbf{depth} : t\ \mathit{id} \mid \textbf{priority} : t\ \mathit{id} \mid \textbf{path} : t\ \mathit{id} \\
(17) & \mathit{fx} & = & \textbf{terminate if}\ (\ \mathit{le}\ )\ b\ \textbf{process} \mid \varepsilon
\end{array}
$$

Figure 6.11: Syntax of frontier exploration extension to GMIR.

(2) In order to support additional features, we also introduce new graph data types for paths, trees and forests. Although these are not specific to frontier exploration, they are currently not found in GMIR.

(7, 9) Before a frontier can be used to explore a graph, it must be initialized first. The `init` statement allows initial elements to be placed into the frontier. There are two variants, one for initializing with just an element and one for initializing an element with an associated priority value, used when the min or max weighted ordering policy is configured.

(8–10) The `expand` instruction uses a similar notation, but it is more flexible. Depending on the configured policies, an attempt to expand into a certain element can be prevented, for example if it has already been visited before. To override this, this instruction can be suffixed with `force` to ignore such checks. Otherwise, if not forced, a `then` block can be attached which is executed if the expansion was not prevented.

(11) An algorithm needs to be able to opt-in to the projection feature, which ties into memory management. These statements make the connection explicit by letting the algorithm pick what to bind. Doing so must be done carefully, as it can place strong constraints on the viability of compiler optimizations. For example, some ordering policies can provide the depth of any element by only keeping track of depth values in a single variable without the need for a full property. Explicitly binding to a property makes the depth values appear in this property, but without the presence of this property the code could have run more efficiently.

(12, 13) Once the frontier has been initialized and projections have been configured, the frontier can be repeatedly visited. The `visit` statement abstracts this and executes is body for each element flowing out of the frontier. This can be done sequentially (11) or in parallel (12), affecting the content of the body block. Parallel visits are only possible for unordered and LIFO frontiers.

Statements $\boxed{\Gamma, t \vdash s \text{ OK}}$ $\boxed{\Gamma \vdash is \text{ OK}}$

$$\frac{\Gamma \vdash lr_{\mathsf{fr}} : fe\,f \qquad \Gamma \vdash lr_{\mathsf{bind}} : \textbf{property} < et\mathsf{Of}\,f, \textbf{boolean} >}{\Gamma, \_ \vdash \textbf{bind visited of } lr_{\mathsf{fr}} \textbf{ to } lr_{\mathsf{bind}}\textbf{;} \text{ OK}}$$

$$\frac{\begin{array}{c}\Gamma \vdash lr : fe\,f \qquad et\mathsf{Of}\,f = \textbf{vertex}(\mathsf{G}) \\ \Delta = \Gamma \cup \{(id_{\mathsf{v}} : \textbf{vertex}(\mathsf{G}))\} \qquad \Delta \to E \vdash \overline{fc} \text{ OK} \qquad E \vdash fx \text{ OK} \qquad E, \mathsf{t} \vdash b \text{ OK}\end{array}}{\Gamma, \mathsf{t} \vdash \textbf{visit } lr\ (id_{\mathsf{v}})\,[\overline{fc}]\, fx\ b}$$

$$\frac{\begin{array}{c}\Gamma \vdash lr : fe\,f \\ et\mathsf{Of}\,f = \textbf{edge}(\mathsf{G}) \qquad \Delta = \Gamma \cup \{(id_{\mathsf{u}} : \textbf{vertex}(\mathsf{G})), (id_{\mathsf{e}} : \textbf{edge}(\mathsf{G})), (id_{\mathsf{v}} : \textbf{vertex}(\mathsf{G}))\} \\ \Delta \to E \vdash \overline{fc} \text{ OK} \qquad E \vdash fx \text{ OK} \qquad E, \mathsf{t} \vdash b \text{ OK}\end{array}}{\Gamma, \mathsf{t} \vdash \textbf{visit } lr\ (id_{\mathsf{u}} \overset{id_{\mathsf{e}}}{\to} id_{\mathsf{v}})\,[\overline{fc}]\, fx\ b}$$

$$\frac{\begin{array}{c}\Gamma \vdash lr_{\mathsf{f}} : \_\,\textbf{frontier} < et, t > \\ \Gamma \vdash lr_{\mathsf{el}} : et \qquad \Gamma \vdash e : t\end{array}}{\Gamma \vdash \textbf{expand } lr_{\mathsf{f}}\ (lr_{\mathsf{el}}, \textbf{priority} : e)\textbf{;}} \qquad \frac{\begin{array}{c}\Gamma \vdash lr_{\mathsf{f}} : \_\,\textbf{frontier} < et > \\ \Gamma \vdash lr_{\mathsf{el}} : et\end{array}}{\Gamma \vdash \textbf{expand } lr_{\mathsf{f}}\ (lr_{\mathsf{el}})\textbf{;}}$$

Context Variables and Termination $\boxed{\Gamma \to \Delta \vdash fc \text{ OK}}$ $\boxed{\Gamma \vdash fx \text{ OK}}$

$$\frac{t \text{ INTEGRAL}}{\Gamma \to \Gamma \cup \{(t : id)\} \vdash \textbf{depth} \ \ \textbf{:} \ \ t\ id \text{ OK}} \qquad \frac{}{\Gamma \vdash \epsilon \text{ OK}} \qquad \frac{\Gamma \vdash le : \textbf{boolean} \qquad \Gamma, \textbf{void} \vdash b \text{ OK}}{\Gamma \vdash \textbf{terminate if } (le)\ b}$$

Figure 6.12: Static semantics of the frontier exploration construct

(14–16) The frontier that is to be visited is expressed in the visit pattern, which consists of a reference to the frontier itself (*lr*), a match pattern (15, either a vertex or an edge) and zero or more context variables (16). The available context variables depend on the policies and projections configured.

(17) By default, the repeated visit stops once the frontier is completely drained. If for any reason termination needs to occur earlier, for example when the destination vertex has been identified, a boolean condition can be provided. When this condition holds, the associated block will be executed instead of the regular `visit` body.

6.5 Static Semantics

Figure 6.12 defines the static semantics rules for the frontier exploration operators. Before the frontier is visited, binds can be issued. We show the rule for binding the **depth** to a **boolean** property. This rule makes use of a convenience notation *et*Of, which extracts the element type *et* given a frontier type *f*.

For visiting the frontier, we show two rules for both vertex and edge frontiers, for the scenario where a single frontier is visited in sequential order. The referenced frontier *lr* must have its *et* match for the respective rule. From that, we extract the graph name G which we then use to introduce new variables according to the vertex or edge matching pattern. Any context variables $\overline{fc}$ may contribute to variables visible in the termination condition *fx* and the body block *b*. That is why we refine the context twice from $\Gamma \to \Delta \to E$.

For the frontier expansion, we provide rules for both weighted and unweighted frontier configurations. Rules for initialization (**init**) are essentially identical, given that we omit the force-or-then postfix term *fp*. The takeaway here is that weighted frontiers must always be expanded or initialized with a corresponding priority value and that non-weighted frontiers cannot be.

Finally we show a rule for one of the context variables, **depth**. This is the most basic one which provides the depth as either an **int** or **long** as requested. The other context variables work similarly, but have additional constraints on the type of frontier in which they are used. For example, binding **priority** is only possible in the context of weighted frontiers.

6.6 Integration with Normal Forms

The features introduced in this chapter do not interfere with GMIR's normal forms.

While the `visit` block iterates over elements, it is not strictly speaking a graph iterator. Graph iterators cannot be broken out of, which the termination condition basically can do. If we would mold the block into an iterator (e.g., `foreach (...: front(frontierRef)) ...`), there is no place for the termination condition. Therefore, the Top-Level Iterators NF does not apply to our additions to the language.

Chapter 7

Deep Dive: Community Detection

Community detection is an important problem in graph analytics. Typically, this is done by gradually grouping vertices that belong together according to some metric into communities. This process effectively builds larger structures from smaller substructures, i.e., it is agglomerative. These algorithms mostly operate in the following steps: they gradually associate similar vertices to the same cluster, starting from each vertex being associated to a unique community. This is repeated until a local stopping criterion is met. The formed communities can be interpreted as a new graph on which the process can be performed iteratively, until a global stopping criterion is met. This forms a hierarchical structure of communities, where communities consist of smaller and smaller communities until the original graph is reached.

Doing this in reverse is also possible. That is a divisive approach, the opposive of agglomerative. In this chapter, we only focus on agglomerative community detection. Please refer to the future work (Section 10.2) for more information about this.

7.1 Algorithm Overview

Key algorithms that currently dominate the commercial and academic community detection landscape are the Louvain method [10], Leiden [74] and Infomap [65]. Other popular algorithms are the label propagation algorithm (LPA) [8] and its more commonly used descendant speaker-listener label propagation algorithm (SLPA) [79].

Louvain Fundamentals The Louvain method of community detection [10] has been fundamental for other algorithms. Still to this day, it is a widely recognized and supported algorithm in commercial and open-source products. What sets Louvain apart from the earlier algorithms that it formed a highly flexible basis for many other innovations to follow. Summarizing from Chapter 2, the key to understanding the operation of the original Louvain algorithm is the concept of a hierarchy of communities and the modularity metric.

Consider an arbitrary graph in which we want to detect communities. Take this graph and associate each vertex to a distinct community, for example by using a property which is initialized to unique values. Each vertex is then checked to determine if it is beneficial to be associated to one of the communities to which the neighboring vertices are associated, according to the modularity metric. This process is repeated until the metric does not improve. This results in a graph where some vertices are associated to the same community and where some communities that initially represented one vertex now represent many or none at all.

Louvain does not stop there. In fact, it is key to the algorithm to use the previously gathered vertex-to-community assignment information and aggregate it into a new ordinary graph on which the aforementioned modularity improvement is run again. This effectively forms a hierarchy of graphs, where each vertex represents one or more vertices from the layer below.

Evolution to Leiden Many years later, the Leiden algorithm was proposed as the culmination of many incremental improvements from the Louvain method. The basic principles from Louvain remain, but some fundamental adaptations have been made.

The main problem Louvain provably suffered from, was its potential to construct disconnected communities. Moreover, there are also some graphs which the Leiden authors proved to be impossible to be detected correctly by Louvain, independent of the starting configuration. Leiden essentially performs Louvain first in a slightly weaker fashion, followed by a second refinement phase which guarantees connected communities. They also adapted a different metric than modularity to overcome the resolution limit issue [23], but remark that Leiden can be used with any arbitrary measure of quality.

Infomap Similarly, Infomap also takes significant inspiration from Louvain but approaches it in holistically from the viewpoint of information compression of the behavior of a random graph surfer, represented by the 'Map Equation' [64]. In practice, a basic implementation of the Infomap can be achieve by taking Louvain and using the Map Equation as the metric. Contrary to Louvain and Leiden, intermediate levels in the Infomap hierarchy have a clearly defined meaning.

Divisive Analysis Another early observation in 1990 was made by Kaufmann and Rosseeuw [43] that academic work essentially ignored divisive community detection, the opposite of agglomerative methods. Their proposed algorithm remains one of the few documented divisive algorithms to this day.

7.2 Essential Summary

We represent the communities that are detected by an algorithm in a graph-native way, as a community graph, containing community vertices and community edges. Such a community graph is related to the graph in which communities were detected (the base graph): vertices in a community graph represent one or more vertices from the base graph; edges between community vertices essentially summarize information from edges between base vertices that are associated to different communities.

This relationship between some ordinary graph and its community graph can be generalized to a hierarchy of communities. By considering a community graph as an ordinary property graph with vertices and edges and feeding it into the algorithm again, we get a community graph of another community graph. The result of this iterative community detection procedure is stored in a specialized hierarchical community detection structure.

The community association for each vertex is established by agglomeration. This occurs in three phases: initialization, arrangement and aggregation. First, the output, a community graph, needs to be initialized; typically each vertex starts off associated to a unique community only containing itself. Second, in the arrangement phase the algorithm is given control to optimize the vertex-to-community association. Finally, the aggregation phase takes the arranged association and aggregates information into the community graph, particularly edges and edge properties. In the arrangement phase, we provide two operations that modify the vertex-to-community association. (1) Move: one particular vertex is removed from one community and added to another. (2) Merge: two communities are merged into one, effectively moving all vertices associated to one community into the other.

Since community graphs extend the concept of property graphs, properties can also be defined on communities. In the initialization and arrangement phase, we allow modification of community vertex properties, but we delay the modification of community edge properties to the aggregation phase as the community edges are not yet final during the arrangement phase. Property updates during arrangement can only be caused by a vertex-community association being updated. Any updates unrelated to an arrangement could instead have been made during the initialization.

We provide a new abstraction for expressing these property updates in terms of effects causes by a move or merge. Moving or merging first of all causes the related vertex(es) to be removed from one community and captured in another, by definition. However, it also subtly affects edges incident to the moving vertex(es). For example, an edge could for example have both endpoints being associated to the same community, but due to a move one endpoint is associated to a different community, and so on.

We provide classifications for each way an edge may be affected and allow the algorithm to express side-effects based on this. Effectively, this allows for an arrangement-centric way of incrementally calculating statistics and other information that an algorithm may need for its optimization metric. This effect classification mechanism can be applied in both an effective (executing) and non-effective (simulating) way, committing to a change and observing effects for a 'dry-run', respectively.

We also define the concept of collapsing a community graph. This feature can be used to effectively reset the agglomerated community structure while retaining the formed communities. This operation models

In the concrete implementation, we introduce data types for each of the aforementioned graphs: (non-hierarchical) community graphs and hierarchical community graphs. Additionally, we define a special type for the community graph when it is under construction in the arrangement phase: the tentative community graph. We also offer additional helper constructs, such as the ability to declare multiple tentative community graphs. The agglomeration with three phases is captured in a block with three nested blocks, each representing one of the phases. Additionally, to set up the community detection data structures for the very first iteration, we also introduce a necessary global initialization phase that is only executed once at the beginning of the algorithm. Then, we model the collapse operation as a standalone statement. Finally, we introduce a 'projection' statement which instructs to continuously reflect the community structure into an ordinary vertex property. This is necessary for compatibility with other logic that does not use community detection constructs and as a potential lowering target.

7.3 Conceptual Design

From the algorithms we explored, it appears there are at least three distinct approaches to the problem of community detection. We only focus on agglomerative community detection where a strict and transitive hierarchy is formed.

All other abstractions that we introduce in the remainder of this section are related to the key concept of the *community graph*. This is a graph-native way of representing the relationship between ordinary vertices and the communities to which they belong and vice-versa. Vertices in a community graph are therefore synonymous with communities themselves. Edges represent relationships between communities, essentially in aggregated form of the edges between vertices from the represented graph belonging to different communities. This view on communities is what we continue working with on a conceptual level, but in practice this can be realized in different ways.

The most basic community detection algorithms suffice with a single agglomeration. However, more powerful algorithms run repeatedly to iteratively refine the community graph. The community graph constructed in one iteration is fed to the algorithm again as if it were an ordinary graph, which the algorithm arranges into a new community graph. This results in a hierarchy of community graphs, which we represent in a dedicated hierarchical community graph data structure, an ordered collection of community graphs. Algorithms that

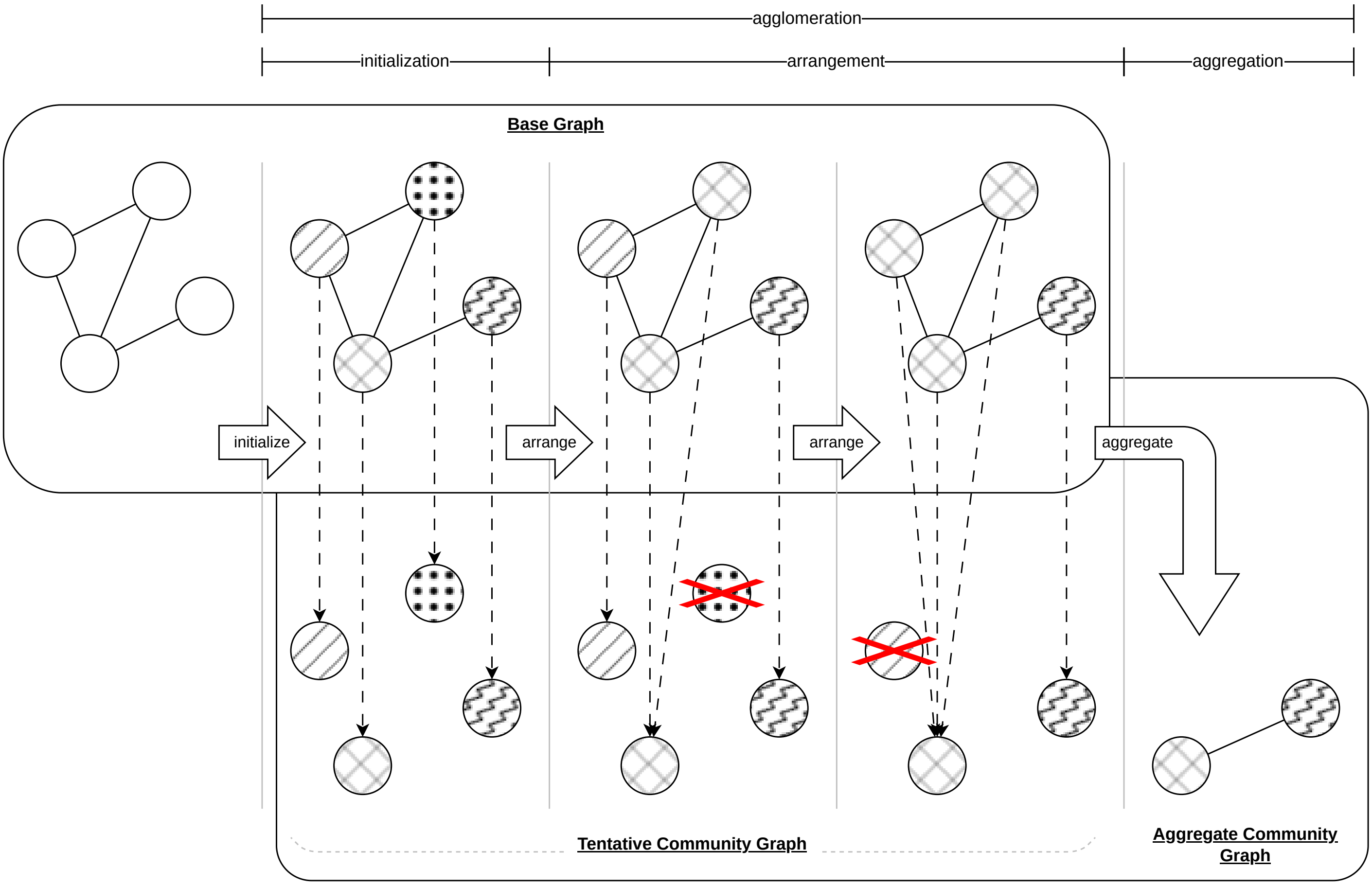


Figure 7.1: Base graph, tentative community graph and aggregate community graph under the three phases of agglomeration: initialization, arrangement, aggregation

construct such a hierarchy only need to work on a base graph. From this, we observe that such hierarchical algorithms are not truly hierarchical in nature as they present themselves. Both single-pass algorithms and hierarchically optimizing algorithms can therefore essentially be expressed in a uniform way. The only distinguishing factor is whether or not the optimization logic is reapplied. This observation is critical to enabling execution of these algorithms on runtimes that do not have any support for multi-level graphs.

We observe that the process of agglomeration, although gradual, is always separable in three major phases: initialization, arrangement and aggregation. Figure 7.1 describes the behavior. We take the base graph and assign each vertex a unique community, indicated by the fill of the vertices (initialization). The optimization logic of the algorithm is then free to perform zero or more arrangements (two are performed in Figure 7.1), changing the vertex-community association, indicated by the changing fill of the vertices (arrangement). As is demonstrated by the crosses over the community vertices, arrangement of vertices may cause loss of reference to a community. At the end, the base graph is aggregated into the final community graph, where each vertex represents one community, indicated by the correspondence in vertex fill (aggregation). Figure 7.1 shows how this aggregated community graph naturally follows. Any agglomerative algorithm can be modeled onto this under our community graph model, as long as the arrangement phase is not specified any further.

For hierarchical algorithms, the repeated execution of the optimization logic naturally extends the relation between base graph and aggregate graph into a hierarchy of graphs, as represented by the aforementioned community graph. At first, we have a base graph $Base_0$ from which we derive the aggregate community graph $Aggr_0$. The next time we apply the optimization, the base graph will be $Base_1 = Aggr_0$. Essentially, the aggregate graph of

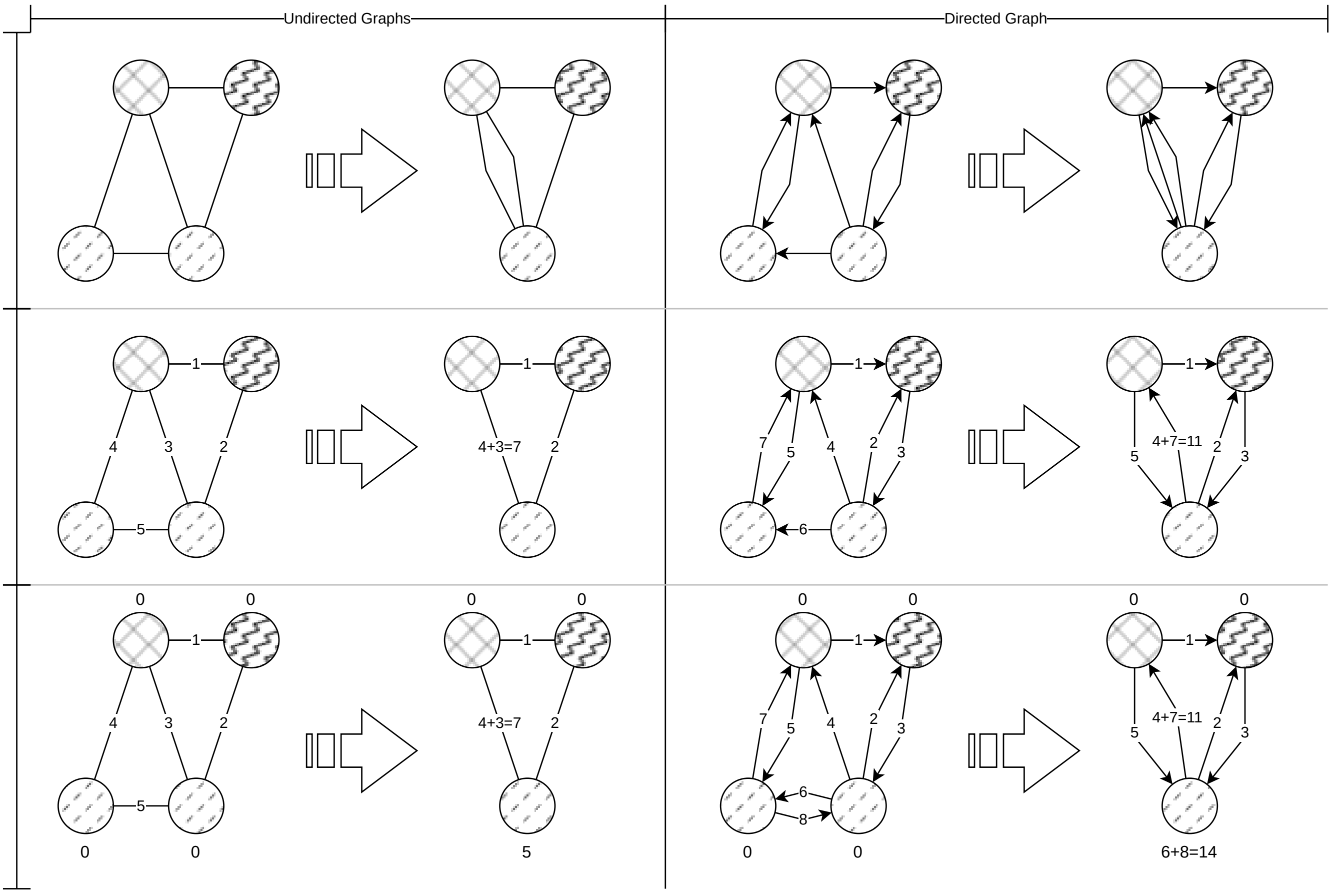

Undirected Graphs
Directed Graph
Edge Copy
Aggregate kept edges into community edges
Aggregate kept edges into community edges + Aggregate deleted edges into community properties
1
4
3
2
5
4+3=7
7
6
8
4+7=11
0
6+8=14

Figure 7.2: Variations of possible vertex and edge aggregation policies

one agglomeration becomes the base graph of the next aggregation, and so forth. Any level taken out from the hierarchy is a proper graph where each vertex has an indication to which community it belongs.

So far, we considered ordinary graphs without properties. We now continue to elaborate our concepts to proper property graphs. Since we decided to represent communities as vertices, these community-vertices can also be associated with properties, which we refer to as community-bound properties. As properties can also be defined on edges, similarly we also speak of community edge properties in the aggregate graph. Note that community edge properties are intentionally not made available in the tentative community graph as their values are not clearly defined until after aggregation.

Any state kept by agglomerative community detection algorithms is fully representable using such properties. The only way to preserve information between the base and aggregate graph is by having the algorithm provide some computation that reduces information from relevant elements of the base graph into information that can be expressed on a single vertex. This applies to both non-hierarchical and hierarchical construction; by similar reasoning as above, this concept applies similarly to both.

An algorithm needs to provide reduction rules separately for vertices and edges. As can be seen from Figure 7.1, the final community vertex represents both vertices and edges from the base graph, with at least one vertex. When it represents more than one vertex, it also represents the base edges between those base vertices. From this, it follows that the edges in the community graph represent the remaining edges, i.e. the edges from the base graph between vertices assigned to differing communities. Hence, we determine that there should be two distinct aggregations, each of which may be arbitrarily specific in terms of information complexity and number of properties. In Figure 7.2, various different approaches are shown.

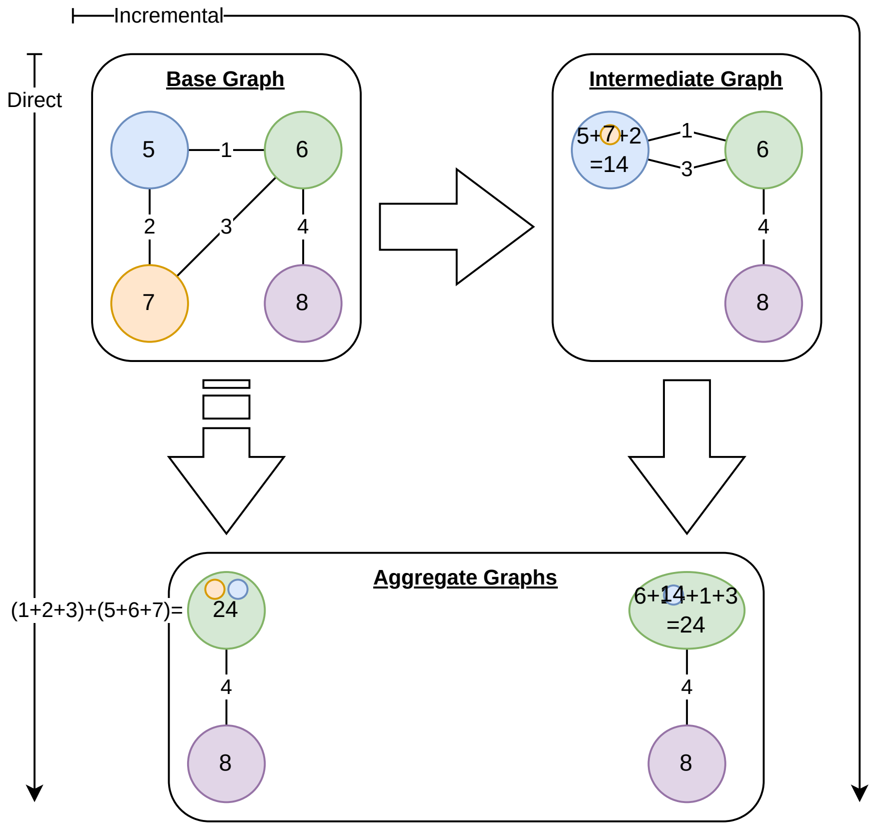


Figure 7.3: Direct versus incremental community property aggregation

The agglomerative nature allows the information aggregation of community vertices can be done incrementally, synchronous with how the arrangement steps gradually refine the vertex-community assignment. In fact, by providing incrementally computed community-bound property values, we enable algorithms that want to make well-informed decisions about their next arrangement to do so based on readily available information on the tentatively formed communities. In Figure 7.3, we show an example where a community vertex property is calculated as a direct reduction versus an incremental reduction. The result is the same, but the incremental form is able to provide information intermediately.

Consider again the example from Figure 7.3, where we sum arbitrary values defined on both vertices and edges. Not limited to summing aggregations, we observe that the terms involved in the incremental aggregation directly relate to the involved vertices and and edges. For example, in $5 + 7 + 2 = 14$, 5 originates from the property value of the initial community of the blue vertex, 7 from the property value of the orange community that is merged into the blue community, and 2 from the edge that would be deleted. Similarly, $6 + 14 + 1 + 3 = 24$ can be related to property values on related edges and vertices.

Due to our earlier assumption that the tentative community graph does not contain edges, we cannot directly apply this to the agglomeration. Moreover, assigning a vertex to a community during the arrangement phase is non-final; a follow-up arrangement may cause the vertex to leave that community and be assigned to another (already existing) community. The tentative community graph does contain vertices, which we should leverage to store properties on intermediately. In fact, once the aggregate graph is constructed, even if it were through a direct aggregation from the base graph, these properties will be defined anyway.

> **Key Idea**: only edges need to be carefully classified. From the context of the arrangement, it is always directly clear which vertices are involved: Merge involves all vertices

> assigned to either the source or the destination community, Move involves only the base vertex that is being moved. Additionally, both under Merge and Move, the source and destination community vertices are involved. All edges incident to the involved base vertices are involved too, but this is exactly what we just classified further. This classification is exemplified in Figure 7.4.

As demonstrated in Figure 7.4, we classify each edge that is involved in the arrangement in one of five classes. Notice that by the nature of the Merge arrangement, no edges will be captured between the source and destination community, as the source community disappears. Alternatively, we propose a second equally powerful classification which classifies such edges into four classes. The two perspectives are as follows:

Five-class model (as seen in Figure 7.4)

- Edges released between the source and destination community. (Purple)
- Edges captured between the source and destination community. Not applicable under Merge. (Yellow)
- Edges released from source community. (Red)
- Edges captured in the destination community. (Green)
- Edges of which one endpoint's community association changes from the source to destination community, but of which the other endpoint remains the same (relink). (Blue)

Four-class model

- Internal-to-internal edges: edges fully contained in the source community, which end up being fully contained in the destination community too.
- Internal-to-external edges: edges fully contained in the source community, which end up becoming edges between the source and destination community.
- External-to-internal edges: edges between the source and destination community.
- External-to-external edges: edges of which one endpoint's community association changes from the source to destination community, but of which the other endpoint remains the same (equivalent to relink edges).

> **Observation**: aggregate graphs will never contain self-edges. Edges in the aggregate graph represent edges in the base graph that have endpoints belonging to different community. By definition, this excludes self-edges. Self-edges, or any edge which is captured by both its endpoints being in the same community, are meant to be aggregated into community-bound properties, as partially shown in Figure 7.2. In hierarchical community detection, this means that the next base graph will not contain self-edges because it is directly derived from the previous aggregate graph.

Any algorithm expressed in our abstraction needs to provide logic to aggregate captured edges, unless it intentionally wants to discard this information. Therefore, any self-edge can be removed at any time by passing it through that aggregation. Any self-edge will eventually be passed through this aggregation anyway; because both endpoints are the same vertex, a self-edge will never be between two different communities. Therefore, we require any base

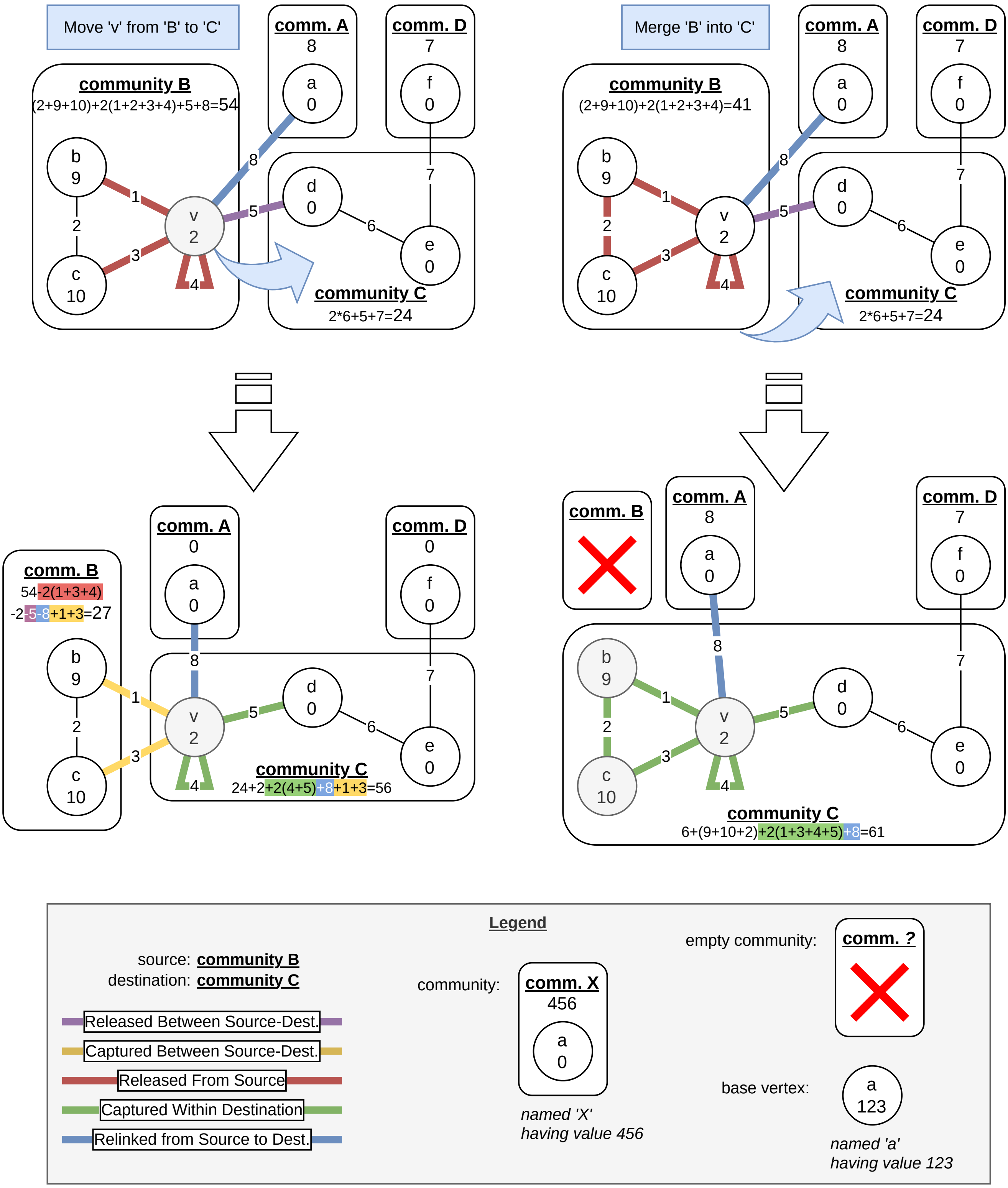


Figure 7.4: Classification of edges affected by moving and merging

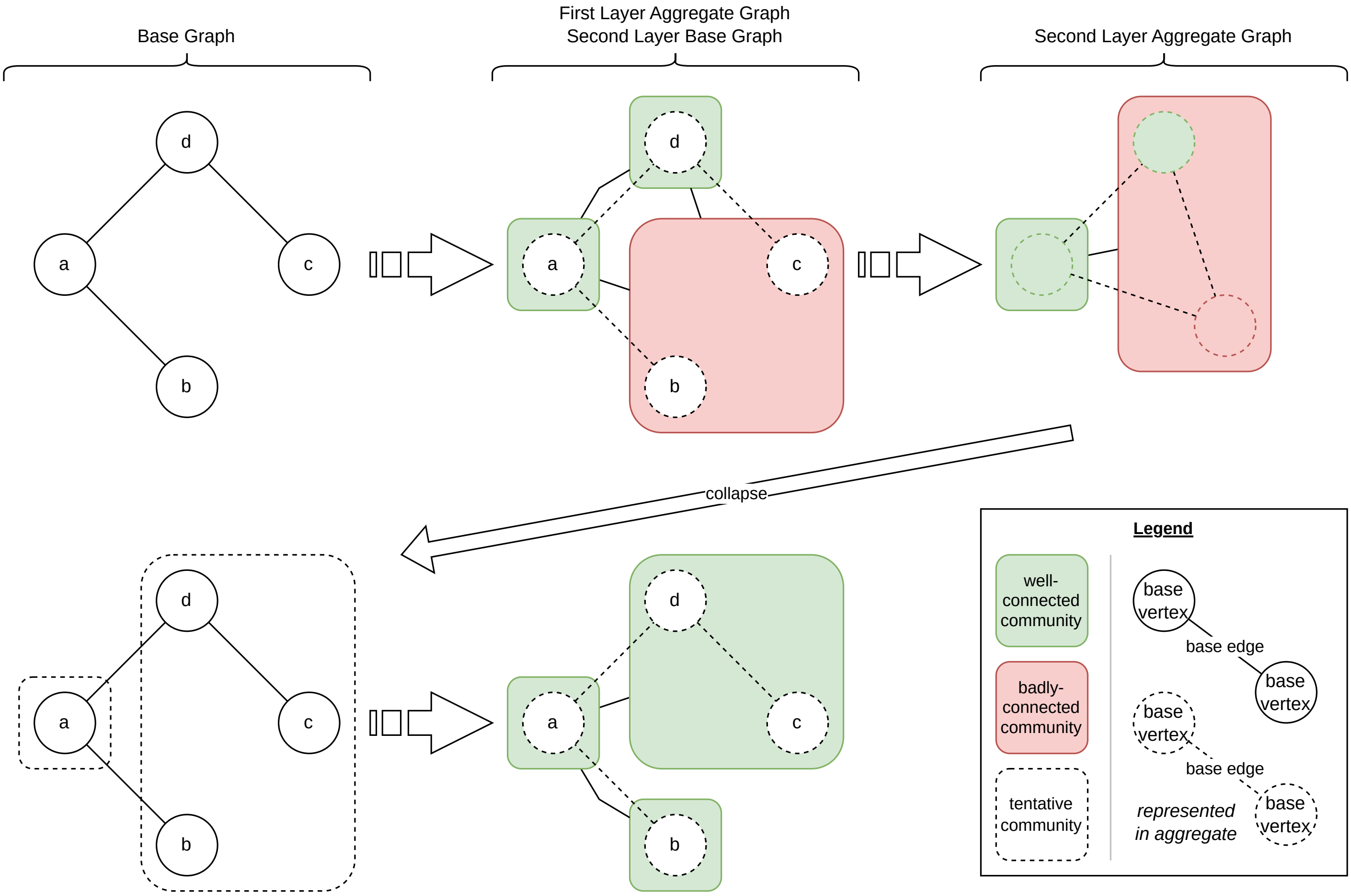


Figure 7.5: Community graph collapse

graph to be free of self-edges. This allows for runtime optimizations which for this reason do not have to care about self-edges beyond the initial construction of the base graph, which is non-specific to community detection.

Finally, we introduce the last concept: graph collapse. So far, we have only considered how we gradually group vertices tentatively into communities, which we then represent in an aggregate graph, which in turn may be fed to the algorithm again to construct a hierarchy. One disadvantage of this approach is that, once a vertex is aggregated into a community vertex, it is not possible to move it out of there. While this is inherent to the agglomerative construction model, it prevents some obvious optimizations from being expressible. Consider the first row of graphs drawn in Figure 7.5. In the aggregate graph, two communities exist (a singleton for vertex 'a' and a one composed of vertices 'b','c','d'). However, it is obvious from the graph structure that vertices 'b' and 'c','d' are unrelated unless 'a' is in the same community. While this is an intentionally exaggerated example, this is the problem that classic Louvain suffers from, as it only makes locally-informed decisions. To some degree the severity is ameliorated by hierarchical construction, but that is certainly no guarantee. To accommodate for algorithms that we provide the collapse operation.

Collapsing an aggregate graph essentially takes the community vertices from the aggregate graph and configures a tentative community graph from it, given an instance of a base graph which was previously used to reach the aggregate graph that is being collapsed. Community-bound properties are transferred to the newly formed tentative communities. This works both across a single agglomeration and a hierarchy of agglomerations. Effectively, this gives the algorithm a second chance and a different perspective to its community assignment, while retaining previous state. This is shown in the second row of graphs in Figure 7.5:

$$
\begin{array}{rrcl}
 & t & = & \ldots \\
(1) & & | & \textbf{community graph}\ (\ lr\ ) \\
(2) & & | & \textbf{hierarchical community graph}\ (\ lr\ ) \\
(3) & & | & \textbf{tentative community graph}\ (\ lr\ ) \\
(4) & & | & \textbf{community property}\ <\ et\ ,\ t\ > \\
 & s & = & \ldots \\
(5) & & | & \textbf{build initial}\ (\ lr\ \textbf{as}\ id\ \Rightarrow\ id^{*}\ pc^{*}\ )\ b \\
(6) & & | & \textbf{agglomerate}\ (\ lr\ \textbf{as}\ id\ \Rightarrow\ id^{*}\ )\ ci^{*}\ \textbf{arrange}\ b\ ca \\
(7) & & | & \textbf{collapse}\ lr\ ; \\
(8) & & | & \textbf{move graph}\ lr\ <\ lr^{*}\ >\ \textbf{into}\ lr\ <\ lr^{*}\ >\ ; \\
(9) & & | & lr\ =\ \textbf{community of}\ lr\ \textbf{in}\ lr\ ; \\
(10) & pc & = & lr\ \rightarrow\ lr \\
(11) & ci & = & \textbf{init}\ id\ \textbf{by singletons}\ b \\
(12) & & | & \textbf{init}\ id\ \textbf{by reflect}\ lr\ b \\
(13) & ca & = & \varepsilon\ |\ \textbf{aggregate}\ id\ \xrightarrow{id}\ id\ \textbf{into}\ id\ \xrightarrow{id}\ id \\
(14) & ld & = & \ldots\ |\ t\ id\ =\ \textbf{community of}\ lr\ \textbf{in}\ lr\ ; \\
 & is & = & \ldots \\
(15) & & | & \textbf{simulate}\ (\ lr\ \Rightarrow\ lr\ ,\ m\ )\ \{\ is^{*}\ \} \\
(16) & & | & \textbf{execute}\ (\ lr\ \Rightarrow\ lr\ ,\ m\ )\ \{\ is^{*}\ \} \\
(17) & m & = & \textbf{move}\ lr\ \textbf{from}\ lr\ \textbf{to}\ lr \\
(18) & & | & \textbf{merge}\ lr\ \textbf{into}\ lr \\
 & gi & = & \ldots \\
(19) & & | & \textbf{foreach}\ ce\ \textbf{case}\ \{\ cec^{*}\ \} \\
(20) & cec & = & cem\ :\ \{\ ld^{*}\ is^{*}\ \} \\
(21) & cem & = & \textbf{released between}\ |\ \textbf{captured between}\ |\ \textbf{captured between}\ id\ \textbf{and}\ id \\
 & & | & \textbf{released from}\ |\ \textbf{captured in}\ |\ \textbf{captured in}\ id \\
 & & | & \textbf{relinked} \\
 & lr & = & \ldots \\
(22) & & | & [\ \textbf{derived}\ (\ a\ \textbf{param}\ t\ )\ t\ ]\ id\ |\ [\ \textbf{derived}\ (\ m\ \textbf{local}\ t\ )\ t\ ]\ id \\
(23) & & | & [\ a\ \textbf{param}\ t\ ]\ lr\ ::\ id\ |\ [\ m\ \textbf{local}\ t\ ]\ lr\ ::\ id \\
 & ee & = & \ldots \\
(24) & & | & id\ :\ \textbf{communityVertices}\ (\ lr\ ) \\
(25) & & | & id\ :\ \textbf{verticesOfCommunity}\ (\ lr\ \Rightarrow\ lr\ ,\ lr\ ) \\
(26) & & | & id\ \xrightarrow{id}\ id\ :\ \textbf{communityEdges}\ (\ lr\ ) \\
(27) & & | & id\ \xrightarrow{id}\ id\ :\ \textbf{edgesOfCommunity}\ (\ lr\ \Rightarrow\ lr\ ,\ lr\ ) \\
(28) & & | & id\ \xrightarrow{id}\ id\ :\ \textbf{edgesBetweenCommunities}\ (\ lr\ \Rightarrow\ lr\ ,\ lr\ ,\ lr\ ) \\
(29) & & | & id\ \xrightarrow{id}\ id\ :\ \textbf{communityNeighbors}\ (\ lr\ ,\ lr\ ) \\
(30) & & | & id\ \xrightarrow{id}\ id\ :\ \textbf{neighborsInCommunity}\ (\ lr\ \Rightarrow\ lr\ ,\ lr\ ,\ lr\ ) \\
(31) & & | & id\ \xrightarrow{id}\ id\ :\ \textbf{neighborsOutCommunities}\ (\ lr\ \Rightarrow\ lr\ ,\ lr\ ,\ lr\ )
\end{array}
$$

Figure 7.6: Syntax of community detection extension to GMIR.

with the tentative community assignment being primed to 'a','b','c','d', the algorithm moves 'b' into a new community. The resulting communities are thus no longer badly-connected.

7.4 Syntax Design

Figure 7.6 formalizes the grammar of all community detection related constructs.

(1, 2) Two distinct types intended for usage by algorithm writers to distinguish between non-hierarchical and hierarchical community graphs. Both reference the underlying graph.

(3) A type for internal purposes only, which is never expected to be written explicitly by a programmer. This is the type used for the tentative community graph (TCG) within a single agglomeration step.

(4, 22, 23) Properties on community graphs are distinct from properties on ordinary graphs. This is necessary to easily distinguish between the significantly different behavior. In fact, community properties can and are often split into several ordinary properties upon lowering.

In order for community properties and their derived ordinary properties to be referenced correctly, we extend the reference annotation syntax. The **derived** keyword provides the connection to the community property from which it originated.

Community properties can also be passed to other functions. We provide the ascription operator (**::**) to disambiguate, as the derived properties are declared on the base graph and all TCGs.

(5, 10) Mandatory global initialization. This statements makes the connection between the underlying graph and the community graph explicit. The left-hand side of the fat arrow refers to the underlying graph and the right-hand side to the initial base graph. The base graph needs to have its declared community properties set up in the body, either by manual initialization or by copy (using 10).

(6, 11–13) The main agglomeration building block which performs a single arrange-and-aggregate step on a given community graph. The fat arrow denotes the names of the base graph and TCGs.

The concrete syntax notation works slightly differently from the conceptual description, but it captures the same meaning. At the start of each agglomeration, the TCGs need to be prepared. Most frequently, communities in the next TCG are formed for each vertex of the base graph (singletons, 11).

Alternatively, given a community-bound vertex property, base vertices that have the same value for that property can be initialized to be associated to the same community (12). This is useful for hierarchical algorithms that precompute some structure in a previous iteration that directly and efficiently needs to be reflected in the next.

After the main arrangement body block has run, the elements in the primary TCG are finalized by aggregation (13) into an aggregate graph which will serve as a base graph in the next iteration. If any community-bound edge properties exist, it is mandatory to match on edges from the base graph and the selected TCG. In the body of this match, aggregation logic must be provided to reduce one or more base edges into an aggregate edge.

(7) The complex collapse operation is captured in a single short statement. This statement does not perform the collapse operation immediately, it only signals that the next agglomeration of the give community graph should happen on the collapsed graph. It is only possible to execute this statement with

(8) For internal use only. This statement describes a variable rename-and-discard operation between graphs.

(9, 14) The community resolution expressions are used to retrieve the community vertex in the TCG to which the vertex is currently associated.

(15–18) These statements are the key to modifying the TCG. Move (17) and Merge (18) operations can either be simulated (15) for information gathering purposes or actually executed (16).

(19–21) The main purpose of placing statements in the body associated to a Move or Merge operation is to observe effects on edges. We introduce this specialization of the for-each loop specifically for the purpose of matching on those involved edges.

(24–31) Lastly, we provide a handful of element iterators to inspect the vertex-to-community association at any time. The iterators with a fat arrow are intended to be used during agglomeration, while the others are intended to be used in the normal scope instead.

7.5 Static Semantics

Figure 7.7 defines the static semantics rules for the community detection constructs.

The initial build requires the left-hand side of the fat arrow ($\Rightarrow$) to match the name of the underlying graph on which the community graph is defined. This also means that we only need to introduce id_{aggr} in the body b. The property copies are checked to only apply to edges and be symmetrical on the underlying and community graphs.

The main **agglomerate** block introduces the base graph as an ordinary non-hierarchical community graph and the aggregate graph as a TCG. The optional community initializers ($\overline{ci}$) must belong to one of the available aggregate graphs. In this figure, we only provide the case for a single aggregate graph, but the rule generalizes naturally.

The **execute** statement, which is typed identical to the **simulate** statement, is shown with the **move** operation. The involved local references must all match their types symmetrical to the fat arrow pattern. This is further propagated to the iterator statements of the associated body ($\overline{is}$).

Those other iterator statements are treated differently due to their additional context ($lr \Rightarrow lr$). In fact, only the **foreach** with **case**-matching is accepted. Case matching is only defined for edges in the neighborhood of the moving vertex. Excluded from this figure is a constraint to enforce the moving vertex to be equal to the reference used in the **neighbors** iterator.

7.6 Normal Form Integration

Similar to the other subdomains we dived into, community detection also interferes with GMIR's Top-Level Iterators NF. In this case, we encounter problems with the `foreach` case-matching which is expected to be placed in the body of an arrangement statement (i.e., `simulate` or `execute`).

An arrangement statement itself is not an iterator, but its body consists only of iterator statements. These can be lifted to the top-level, which loses the lexical scope information that case matching is possible in the first place. Outside of the body of a rearrangement statement, the effect case matching is not allowed. This means that the top-level declaration needs to be annotated that it only can be used from the scope of an arrangement statement.

In practice, this is not a problem. We provide transformations that transform arrangement statements and their effect-matching inner loops into existing statements and aggregation expressions. After this, the Top-Level Iterators NF can be applied without problems, as no constructs specific to community detection remain.

Statements $\boxed{\Gamma, t \vdash s \textsc{ ok}}\ \boxed{\Gamma \vdash pc \textsc{ ok}}\ \boxed{\Gamma, \overline{id} \vdash ci \textsc{ ok}}$

$$\frac{\begin{array}{c}\Gamma \vdash lr : \_\ \textbf{community graph}\ (id_{\text{base}}) \\ \Gamma \vdash \overline{pc} \textsc{ ok} \qquad \Gamma \cup \{((id_{\text{aggr}} : \textbf{tentative community graph}(id_{\text{base}}))\}, \text{t} \vdash b \textsc{ ok}\end{array}}{\Gamma, \text{t} \vdash \textbf{build initial}\ (lr\ \textbf{as}\ id_{\text{base}} \Rightarrow id_{\text{aggr}}\ \overline{pc})\ b}$$

$$\frac{\begin{array}{c}\Gamma \vdash lr_{\text{underlying}} : \textbf{property} < \textbf{edge}(\text{G}), t > \\ \Gamma \vdash lr_{\text{base}} : \textbf{community property} < \textbf{edge}(id_{\text{CG}}), t > \qquad \Gamma \vdash id_{\text{CG}} : \_\ \textbf{community graph}\ (\text{G})\end{array}}{\Gamma \vdash lr_{\text{underlying}} \to lr_{\text{com}} \textsc{ ok}}$$

$$\frac{\begin{array}{c}\Gamma \vdash lr : \_\ \textbf{community graph}\ (\text{G}) \\ \Delta = \Gamma \cup \{((id_{\text{base}} : \textbf{community graph}(\text{G})), ((id_{\text{aggr}} : \textbf{tentative community graph}(lr))\} \\ \Delta, [id_{\text{aggr}}] \vdash \overline{ci} \textsc{ ok} \qquad \Delta, \text{t} \vdash b \textsc{ ok}\end{array}}{\Gamma, \text{t} \vdash \textbf{agglomerate}\ (lr\ \textbf{as}\ id_{\text{base}} \Rightarrow id_{\text{aggr}})\ \overline{ci}\ b}$$

$$\frac{\begin{array}{c}id \in \overline{id} \qquad \Gamma \vdash id : \textbf{tentative community graph}\ (id_{\text{CG}}) \\ \Gamma \vdash lr : \textbf{community property} < \textbf{vertex}(id_{\text{CG}}),\ \textbf{long} > \qquad \Gamma, \textbf{void} \vdash b \textsc{ ok}\end{array}}{\Gamma, \overline{id} \vdash \textbf{init}\ id\ \textbf{by reflect}\ lr\ b}$$

Expressions $\boxed{\Gamma \vdash e : t}$

$$\frac{\begin{array}{c}\Gamma \vdash lr_{\text{v}} : \textbf{vertex}(id_{\text{cg}}) \\ \Gamma \vdash id_{\text{cg}} : \textbf{community graph}\ (\_) \qquad \Gamma \vdash lr_{\text{cg}} : \textbf{tentative community graph}\ (id_{\text{cg}})\end{array}}{\Gamma \vdash \textbf{community of}\ lr_{\text{v}}\ \textbf{in}\ lr_{\text{cg}} : \text{t}}$$

Iterator Statements $\boxed{\Gamma \vdash is \textsc{ ok}}\ \boxed{\Gamma, (lr \Rightarrow lr) \vdash is \textsc{ ok}}\ \boxed{\Gamma, (lr \Rightarrow lr) \vdash cec \textsc{ ok}}$

$$\frac{\begin{array}{c}\Gamma \vdash lr_{\text{v}} : \textbf{vertex}\ (lr_{\text{base}}) \\ \Gamma \vdash lr_{\text{src}} : \textbf{vertex}\ (lr_{\text{aggr}}) \qquad \Gamma \vdash lr_{\text{dst}} : \textbf{vertex}\ (lr_{\text{aggr}}) \qquad \Gamma, (lr_{\text{base}} \Rightarrow lr_{\text{aggr}}) \vdash \overline{is} \textsc{ ok}\end{array}}{\Gamma \vdash \textbf{execute}\ (lr_{\text{base}} \Rightarrow lr_{\text{aggr}}\ \textbf{, move}\ lr_{\text{v}}\ \textbf{from}\ lr_{\text{src}}\ \textbf{to}\ lr_{\text{dst}})\ \{\ \overline{is}\ \}}$$

$$\frac{\Gamma \cup \{(id_{\text{u}} : \textbf{vertex}(lr_{\text{base}})), \ldots\}, (lr_{\text{base}} \Rightarrow lr_{\text{aggr}}) \vdash \overline{cec} \textsc{ ok}}{\Gamma, (lr_{\text{base}} \Rightarrow lr_{\text{aggr}}) \vdash \textbf{foreach}\ (id_{\text{u}} \overset{id_{\text{e}}}{\to} id_{\text{v}}\ \textbf{:}\ \textbf{neighbors}\ (lr_{\text{base}}, \ldots))\ \textbf{case}\ \{\ \overline{cec}\ \}}$$

$$\frac{\Gamma \cup \{(id_{\text{c}} : \textbf{vertex}(lr_{\text{aggr}})), (id_{\text{d}} : \textbf{vertex}(lr_{\text{aggr}}))\} \to \Delta \vdash \overline{ld} \textsc{ ok} \qquad \Delta \vdash \overline{is} \textsc{ ok}}{\Gamma, (lr_{\text{base}} \Rightarrow lr_{\text{aggr}}) \vdash \textbf{captured between}\ id_{\text{c}}\ \textbf{and}\ id_{\text{d}}\ \textbf{:}\ \{\overline{ld}\ \overline{is}\}}$$

Figure 7.7: Key static semantics of the community detection construct

Chapter 8

Evaluation

In this chapter, we evaluate our proposed abstractions in three different ways. First, we discuss the general language design on a high level according to 'design dimensions' from Voelter [77]. Secondly, we show which common algorithms are and are not expressible, with explanations as to why. Finally, we demonstrate how our domain-specific abstractions operate in practice by explaining how they map onto existing constructs.

For general context, we implemented all these features as independent modular extensions in the existing GMIR compiler. This is a commercial product built on the Spoofax Language Workbench [12, 42, 68].

8.1 Voelter's Seven Design Dimensions

One of our primary sources for designing domain-specific abstractions is the well-known *DSL Engineering* book by Voelter [77]. In a chapter co-authored by Eelco Visser, seven design dimensions are given which DSLs should use as guidance. The authors note that design is inherently subjective to some degree, which provides reason for having a more objective evaluation framework.

The design dimensions are seven:

1. **Expressivity**: conciseness of programs in the related domain.
2. **Coverage**: what fraction of domain problems can be expressed.
3. **Semantics and Execution**: separable in static semantics and execution semantics, describing the statically observable behavior and runtime behavior of programs, achieved through transformations.
4. **Separation of Concerns**: clear division between different areas of the domain as a whole.
5. **Completeness**: ability of the abstraction to capture all necessary context without relying on external sources such as configuration files or on hand-written auxiliary code.
6. **Language Modularity**: reusability and integration with the existing DSL and potentially other languages.
7. **Concrete Syntax**: writability, readability, learnability and effectiveness.

As the goal of this thesis is to close the feature gap that arises from the lack of domain-specific abstractions in GMIR, we do not evaluate expressivity and concrete syntax. Moreover, GMIR is an IR which does not inherently benefit by being concise. In fact, the normalization features of GMIR intentionally explicate a lot of information in a very verbose and redundant way.

8.1.1 Coverage

The coverage of the DSL at hand and improving that is essentially the primary goal of this thesis. Most broadly speaking, GMIR is intended to cover the domain of algorithmic graph processing. In the state that it was at the start of this thesis project, it clearly lacked coverage in some areas. This is exactly what we addressed by introducing new abstraction in the three key areas—fixed-point iteration, frontier exploration and community detection.

To be precise, only our abstractions in the areas of frontier exploration and community increase coverage. The fixed-point iteration features do not as any `fix` loop can also be –albeit inefficiently– rewritten into a `do-while` construct and some helper variables. This contrasts with the other two areas that provide new data types and operations over those.

Frontier exploration allows primitive traversals such as BFS and DFS to be expressed. It also enables priority-based traversal for Dijkstra and similar algorithms. Additionally, MST algorithms such as those from Prim and Kruskal can be expressed. None of this was previously possible. To some extent, this was possible in Green-Marl, but not with the flexibility that we introduced.

Community detection was previously entirely not covered at all. While there are many different approaches to this problem that each have their unique requirements, we decided to introduce support for the most prominent ones. With support for the Louvain method and the derivatives Infomap and Leiden, we believe to cover the majority of current use cases.

These three aspects increase coverage, but certainly do not cover the entire domain. During our domain exploration, we already became aware of certain concepts that would be useful but infeasible to cover in this thesis. In Future Work (Section 10.2) we discuss in more detail which features specifically we believe are missing and how this could be addressed.

8.1.2 Semantics and Execution

For each of the three key areas of interest, we provided static semantics. Because GMIR lacks a formal static semantics specification, our static semantics are also not formally verified. We believe the static semantics rules are complete enough for practical purposes. We have verified them by using the Spoofax Testing Framework [41]. Furthermore, any transformation on the abstract syntax tree (AST) that uses explicated semantic information also implicitly performs consistency checks.

8.1.3 Separation of Concerns

As Voelter [77] states: "A domain may be composed from different concerns. Each concern covers a different aspect of the overall domain." We believe the three main areas of interest are sufficiently independent to be covering completely separate concerns. While it can be convenient to use abstractions from one in the other, it is not strictly necessary. For example, determining whether a local community assignment is stable and ready for aggregation can be done concisely with a fixed-point iteration, but need not be.

8.1.4 Completeness

GMIR programs are mostly self-contained. Our modifications have not made any changes to the degree to which this was the case. In practice, this means GMIR still relies on an external configuration of the Parallel Graph AnalytiX (PGX) server to perform housekeeping tasks, such as loading and persisting graphs.

8.1.5 Language Modularity

One of the main selling points of GMIR is its high degree of modularity. The clear separation between core and graph constructs allows for building out new features in an isolated way. Projects such as MLIR [46] have recently proven this fact once again. That is why we designed and implemented our extensions to GMIR in a similarly modular way. Similar to how we separated three main concerns, we also implemented them independently.

8.2 Expressing Algorithms

In the second part of our evaluation, we try to express various algorithms that would benefit from our abstractions. Although the algorithms covered here are not exhaustive, we believe they paint a representative picture of the algorithms that are typically used nowadays in the respective domains.

For each of the three subdomains we investigated we list various algorithms. We annotate the problems we faced while expressing them. Some algorithms are also simply not possible to express, but still very relevant to the subdomain; those we briefly list as well.

8.2.1 Fixed-Point Iteration

In the content chapters we have extensively covered both PageRank and the iterative WCC algorithms. Both work well and they were the main source of inspiration for the abstractions. PageRank is also sometimes used in a 'personalized' setting, where the calculated ranks are relative to one or more select vertices. This can be achieved by initializing the rank values leaning more towards those chosen vertices, which is independent of the fixed-point iteration itself.

We also briefly mentioned the HITS algorithm [45]. It appears that it does not benefit from the fixed-point operator, as it always iterates for a preconfigured amount of times. The 'authority' and 'hub' values are also intentionally updated within the loop itself and need to be directly visible.

8.2.2 Frontier Exploration

Basics Frontier exploration algorithms are more diverse. The essential graph traversals DFS and BFS are possible to express, including additional features such as reverse iteration. Due to the way we model the frontier data structure, the elements are always visited in pre-order, i.e., the element itself is visited before any of its descendants. The reverse visit effectively offers post-order traversal, but in-order traversal is impossible.

Dijkstra's algorithm [22] and related best-first search algorithms are equally expressible. In essence, Dijkstra's algorithm is a best-first search algorithm that minimizes the total distance so far. Alternative metrics can be used just as well by tuning the frontier configuration parameters. The typical application of such best-first search algorithms is path finding. This is fully supported and concisely achievable by enabling the path tracking feature.

The Bellman–Ford algorithm initially seemed relevant to this discussion too because it calculates shortest paths. Even though it is a path finding algorithm, it does not find these paths using a frontier, because it essentially considers the entire graph at once. Therefore, it is irrelevant to our discussion.

Heuristics A* [33] is often mentioned in discussions of Dijkstra's algorithm. It provides an extension with a heuristic to potentially decrease the overall search space. We successfully implemented A* using a consistent heuristic, but were unable to do so for any admissible heuristic. The problem we faced is that inconsistent heuristics do not guarantee optimality when a vertex is visited. This interferes with the termination condition, which can only contain a simple boolean expression. It is therefore impossible to indicate to the `visit` block that termination needs to be averted or delayed.

Optimizing Ordering Policies For the basic algorithms we discussed we provide ordering policies that cover their ordering requirements. Recent research has focused significantly on optimizing the ordering itself, separate from the rest of the algorithm logic. Although we believe this supports our claim that separating this into its own concern is beneficial, it is currently impossible to express custom/optimizing ordering policies. For example, Δ-stepping [52] is a common technique applied to achieve significant reductions in memory pressure.

Multiple Frontiers We laid some groundwork for frontier exploration with multiple frontiers, but did not include this in the final design. There are clearly some algorithms that benefit from exploring a graph concurrently from multiple starting points [1, 24, 70]. For example, bidirectional Dijkstra [73, 75] is a natural extension which explores the graph from both the source and destination. The main concerns with these multi-frontier algorithm are termination, synchronization and balancing.

When the frontiers 'meet', it is not necessarily a guarantee that any path that can then be constructed is actually optimal. Although the correctness is the concern of the algorithm itself, it impacts the progress that can be made in other frontiers. Ideally, we do not halt progress in other frontiers, but on the other hand it is wasteful to keep progressing if termination is imminent.

As long as termination is not applicable, the frontiers may or may not want to progress independently. Situations where frontiers become lopsided are the most important to avoid, as that defeats the purpose of having multiple in the first place. In our proposed design this cannot be captured, but we believe it to be necessary for successful support of this feature.

Minimum Spanning Trees Both Prim's and Kruskal's algorithms to compute MSTs can be expressed concisely. The only difficulty is with Prim's algorithm where we need to select a random starting vertex to use as the initial edge source. This is currently not possible in GMIR, but we can make the algorithm work by assuming a random vertex is passed in.

Prim's algorithm uses a treelike min-weighted frontier and is initialized by a set of edges incident to a random starting vertex. The priority values on the edges in the frontier are taken from a read-only weight property. Expansion happens in the edges incident to both endpoints of each visited edges. The treelike exploration policy automatically ensures only edges that extend the tree are visited. Therefore, the MST follows directly from the algorithm by projection and can be bound to an output parameter.

Kruskal's algorithm can be expressed even more concisely and does not require any expansion. The frontier is a forestlike min-weighted edge frontier, comparable to Prim's situation. The frontier is initialized by all edges and priorities corresponding to their weight. Then, the visit block is just let to run without any body (`visit kruskalFrontier (()-()) [] {}`). By nature of the forestlike exploration and the min-weight ordering policies, it follows that an MST is constructed.

We also considered Borůvka's or Sollin's MST algorithm and Chu–Liu/Edmonds' algorithm for minimum spanning arborescences (MSAs). While both appear to be not far related from the other MST algorithms, we can unfortunately not express either.

8.2.3 Community Detection

Community detection is inherently the most complex subdomain we covered. We only evaluate the our design against a few large algorithms and use-cases.

Already from the initial domain exploration, we have continuously mentioned the three most important agglomerative algorithms Louvain, Infomap and Leiden. In practice, we successfully implemented Louvain completely and Leiden partially. Due to time constraints, we did not implement Infomap, but we believe it is completely feasible to do so. We verified our Louvain implementation by cross-checking results with a commercial implementation. We also worked together with other teams at Oracle to get our implementation working in other products, such as natively on PGX.D.

The major blocking issue we faced with Leiden is the implementation of the probability mass function. In Louvain, vertices are moved between communities based on which move is best. However, in Leiden, this happens with a probability relative to the exponential of their quality. The current state of GMIR is not expressive enough to capture this.

Second to that is the lack of random iteration. The original Louvain algorithm only works with sequential deterministic iteration, but Leiden explicitly requires random iteration for accuracy reasons. We considered enriching the for-each loop and aggregator forms with a `random` keyword, but this only opened up new challenges that we were not ready to face.

As we focus only on agglomerative algorithms, other well-known community detection algorithms such as SLPA are not strictly relevant to this evaluation. During the domain exploration and later experimental phase, we still attempted to implement both LPA and SLPA to see if there was a common ground to build upon. The main reason why these algorithms do not work in GMIR as of now is due to the lack of random iteration, just as we see in Leiden.

8.3 Operational Semantics by Lowering

As we described earlier, part of our realized implementation are transformations that allow running the new features with only minimal changes on existing platforms. Inevitably, some features are missing from the existing platforms, which is partially what this thesis tries to uncover. Disregarding what is missing, these lowering transformation essentially define the operational semantics of the new constructs in terms of existing operational semantics.

Similar to the static semantics, there are no formal operational semantics defined for GMIR. Therefore, we only highlight some key transformations to support a general understanding of the constructs. Moreover, our actual implementation in the Stratego language comprises of more than 1,000 SLOC, which is too much to cover here.

8.3.1 Fixed-Point Iteration

Basics The basic usage of the fixed-point loop can be modeled as a do-while loop with some extra features. A do-while loop checks the loop guard at the end of the body, similar to how the fixed-point body is always executed at least once.

We model the safeguard iteration limit as a local variable which is initialized by the lower bound. At the end of every iteration, the value is incremented by `1`. The loop guard is then implemented as a condition which verifies that the upper bound has not been reached yet. If there is no upper bound, the condition is always true. The loop guard is combined in a logical conjunction with the result of the user-provided **`until`** clause.

Variable Versioning The key feature of the fixed-point loop is the automatic variable versioning and related accessors for those subject variables. Depending on the type of subjects, versioning implementations are lowered differently. If a subject is marked **`inplace`**, this lowering is not applicable.

We implement scalar variables by introducing a temporary variable which holds the new value. After the body and termination condition are completed, the deferred value is assigned to the current value.

A nontrivial implementation is required for properties and more complex data types. Properties cannot simply be reassigned, but need to be copied instead. Different platforms have significantly different ways of representing properties at runtime. This is abstracted through a builtin function, which we use instead of a simple reassignment.

The same logic applies to other complex data types. For example, this feature integrates with community detection.

Explicit Boolean Change Tracker Another key feature is the automatic termination upon subject stabilization in the absence of a termination condition. Some platforms, such as an SQL database, can report whether a variable was *touched*, but usually not whether it was actually modified to a meaningfully different value. However, that is exactly the information we need to determine when all subject variables have stabilized.

To achieve this, we introduce some additional helper local variables for each subject: a flag and a reference value. The flag indicates whether we detected a change so far and the reference value is what we compare to. In practice, this reference value can be the same variable that holds the current value.

Both for scalars and properties, the functionality is more or less the same. We insert statements around each place where the variable is written. These compare the value that is about to be written with the current value. If there is a meaningful difference, the flag is set.

At the end of the body, we collect all flags and incorporate this into the do-while loop guard.

8.3.2 Frontier Exploration

Explicit API The first step in lowering frontier exploration constructs is to make all domain-specific statements and expressions explicit. We do this by introducing a broad range of new builtin functions, which essentially represent the same functionality. This is less accessible for a user, but internally more convenient to work with, especially because the initial statics analysis already proved that the structure is acceptable.

Still, we need to introduce some new syntactical features to support this lower form which mainly uses builtin functions. The `visit` block repeats by default. We lower this into a foreach loop using new element iterators such as `frontVertices` and `frontEdges`.

General-Purpose Data Structures Platforms that support general-purpose data structures can be leveraged for implementations of frontiers. This second lowering is effectively a follow-up which specializes the intermediate API. The corresponding lowerings look similar to how frontier explorations are traditionally implemented in general-purpose programming languages.

Because we have full control over the initial lowering, we can expect certain fixed patterns in the API form. This significantly simplifies the implementation of this secondary lowering, which only has to be able to support program shapes emitted by the first.

8.3.3 Community Detection

Affected Edges The first step in lowering commnuity detection constructs is the lowering of case-matching on affected edges. This lowering removes the **`foreach`** with `case` body and replaces it with conventional foreach loops. These foreach loops use the iterator patterns that are reserved for internal use, such as `neighborsInCommunity` and `neighborsOutCommunities`.

Depending on the configuration of the community graph element type and the specific arrangement operation used (Move or Merge), some case branches can be combined. The different scenarios are described in Table 8.1.

Effect	Move	Merge
released between	neighborsInCommunity(Base => Aggr, movingVertex, targetCommunity)	edgesBetweenCommunities(Base => Aggr, sourceCommunity, targetCommunity)
captured between	neighborsInCommunity(Base => Aggr, movingVertex, sourceCommunity)	N/A
released from		edgesOfCommunity(Base => Aggr, sourceCommunity)
captured in	neighborsInCommunity(Base => Aggr, movingVertex, targetCommunity)	edgesOfCommunity(Base => Aggr, sourceCommunity) + edgesBetweenCommunities(Base => Aggr, sourceCommunity, targetCommunity)
relinked	neighborsOutCommunities(Base => Aggr, movingVertex, sourceCommunity, targetCommunity)	edgesOutCommunity(Base => Aggr, sourceCommunity, targetCommunity)

Table 8.1: Overview of which lower iterators should be used to cover which effects under both Move and Merge, disregarding self-edges

G	v.com (before)	Base ⇒ Aggr		v.com (after)
v#0	5L	v#0	v#0	5L
v#1	4L	v#1	v#2	8L
v#2	4L	v#1	v#2	8L
v#3	8L	v#2	v#2	8L

Table 8.2: Gradual community projection across hierarchy

Community Graph to Property After the first lowering, all remaining domain-specific syntax is related to managing the community graphs. There are many small constructs, which make this lowering particularly hard to implement. In summary, what we do is we get rid of all community types and operators. The result is a program where communities are represented by long vertex properties.

The 'heavy lifting' of constructing the initial graph and consecutive aggregation is still handed off to builtin functions, as follows.

- `procedure void createMappedNullCommunityGraph(in graph Base, out graph Tent, out property<vertex(Base), vertex(Tent)> belongsTo)`
  - Initializes an ordinary graph Tent with as many vertices as in `Base`, each of which is assigned to a unique vertex from Base in the `belongsTo` property.
- `procedure void createPreMappedNullCommunityGraph(in graph Base, in property<vertex(Base), long > primer, out graph Tent, out property<vertex(Base), vertex(Tent)> belongsTo)`
  - Special case of `createMappedNullCommunityGraph` where the `belongsTo` mapping is initialized differently. Each vertex from `Base` that has the same value for `primer` is instead associated with the same vertex from `Tent`.
  - This is intended to be used with the `collapse` instruction. Typically, `com` from `finalizeCommunityGraph` is fed into `primer`.
  - This is also used for `init ... by reflect ....`
- `procedure void finalizeCommunityGraph(in graph G, in graph Base, in-out graph TentAggr , in property<vertex(Base), vertex(TentAggr)> belongsTo, in-out property<vertex (G), long > com)`
  - Takes a base graph `Base`, another graph `TentAggr` and a `belongsTo` property, which were initialized by one of the previous functions.
  - Constructs aggregate edges in the graph `TentAggr` based on edges from `Base`, in accordance with the `belongsTo` mapping.
  - Updates the projection property `com` in accordance with `belongsTo`, respecting the previous values in `com` to support hierarchical projection too. Table 8.2 demonstrates this.

Chapter 9

Related work

In this chapter we cover several different areas in which academic and commercial work has progressed, related to our topic. We briefly outline the differences and similarities and point out where our ideas could be applied.

9.1 Graph Analysis Landscape

While this thesis focuses on imperative algorithmic graph processing, the full landscape of graph analysis is much broader. We can classify at least four different approaches, each of which have their unique strengths and weaknesses. In the following subsections, we compare each with their respective pros and cons.

9.1.1 SQL-like languages

Graphs are typically stored in some form of database geared towards representing graphs. For all intents and purposes of graph processing, it is not necessarily relevant how exactly this data is represented but only that we can analyze it. A major advantage of using databases is their ability to perform work close to the source of data before transmitting a subset of data to the requester.

Traditionally, the intent of the data to be requested is declaratively modeled using SQL. By extension, graph databases have evolved SQL to support graph-native operations. Currently most used are openCypher [25] and PGQL [62]. PGQL is used exclusively in Oracle products, while openCypher is derived from Neo4j's proprietary Cypher and supported in other vendors' graph databases [50, 72]. Other vendors develop their own competing SQL-like languages [27, 30]. Recently, ISO/IEC have published GQL, a standard [38] which takes ideas from PGQl, openCypher and G-CORE [2]. Separately a few months earlier, ISO/IEC also extended SQL with property graph syntax through SQL/PGQ [39].

The clear advantage of querying graphs this way is that there is only a small learning curve. Many programmers are already familiar with some SQL dialect and should be able to to quickly learn how to query graphs.

The major disadvantage of an SQL-like approach is the declarative nature of SQL and the non-graph specific baggage that comes with it. Queries alone are insufficient to implement arbitrary algorithms. This is partially resolved with imperative extensions such as the standardized SQL/PSM [40] or other more strongly deviating dialects such as GSQL [30] or AQL [6]. However, this only makes sense in a context where the usage of SQL is mandatory or where this is the only feasible interface to the desired database. DSLs are particularly suited for this situation if some of these requirements are loosened up.

To overcome some of these limitations, some vendors compromise by implementing popular graph algorithms as built-in functions [5]. Those are implemented directly in the database runtime itself with an opaque implementation in a different programming language. The algorithm implementations may be modified or extended by a database administrator, but it is not a first-class feature.

9.1.2 Machine-local processing libraries

Another approach to graph processing is locally without the use of a database, or only indirectly for long-term persistence. This is achieved through graph processing libraries for general-purpose languages. Such libraries typically abstract the memory lay-out of the graph and provide various read and write access points to efficiently iterate or filter elements. It is then mostly up to the programmer itself to build arbitrary algorithms around such a library. Well-known libraries include Boost Graph Library [67] and NetworkX [32, 53]. Over the last years, significant efforts have been made to optimize the performance of algorithms using these libraries [15, 44, 69].

Advantages of these local implementations are simplicity, convenience and predictability. While their implementations may still be complex, their usage is usually intentionally not. Particularly NetworkX, which has been around for almost 20 years and is implemented in Python tends to scripting use cases, which is less often seen in other approaches.

The main disadvantage is scalability. Such libraries are only intended to be used on a single machine. While they attempt to exploit local parallelism, CPU and memory limits are guaranteed to be hit when dealing with massive graphs.

9.1.3 Graph-parallel systems

This is where graph-parallel systems come into play, being able to handle graphs with millions or billions of vertices and edges. In 1990, Valiant published the bulk-synchronous parallel (BSP) model [76], long before distributed graph processing became more relevant in the last 10-20 years. This formed the basis for Google's highly-influential Pregel model [49, 81], which extends BSP with graph-specific abstractions. This was used in their proprietary implementations, but has also been picked up by major vendors such as Neo4j and Apache Spark [80]. Over the last 10 years, more and more frameworks targeting large-scale parallel graph processing have been developed [17, 19, 26, 29, 36]. While less popular, similar research has been done into running graph analysis on GPUs with a focus of integrating machine learning (ML) [51, 57, 78].

The primary benefit of these graph-parallel systems or frameworks is their scalability. They are designed to handle very large graphs that span many machines to run logic in parallel. To be able to efficiently leverage such systems, it is necessary to know more about its implementation details compared to any other approach. On the contrary, without being informed about how the system works, it is easier to misuse the system unintentionally. While that also poses a risk for the previously covered library approach, the detriment of improper use is amplified due to the risk of nullifying the benefits gained by the near-horizontal scaling that this approach operates on.

9.2 Other Domain-Specific Languages

As mentioned in the introduction, several DSLs other than GMIR have been developed that target graph analytics: Gremlin [4], Falcon [18] and GraphIt [82]. We compare all and discuss their relation to GMIR and our three aspects of new abstraction.

Gremlin [4] (2009) Gremlin can be considered both a query language and an algorithm language. It is a fluent API designed to be used from Groovy, so it is neither SQL-like or a DSL. Repetition, somewhat similar to our fixed-point abstraction is already present. Full frontier exploration as we introduced is currently impossible without supplementary Groovy code. The same problem can be seen when attempting to perform nontrivial community detection.

The Gremlin API appears to be flexible and adaptable enough to potentially support the constructs to support our abstractions. It has many use-case-specific feature already and is still evolving. More configurable bookkeeping or the introduction of a frontier data type is sufficient to support frontier exploration. Similarly, community detection can be directly implemented by extending Gremlin with APIs similar to our proposal.

Falcon [18] (2015) Falcon's primary focus was providing a tunable high-level abstraction for parallel graph algorithms to be compiled to heterogeneous hardware configurations, something the authors critique Green-Marl for not supporting. For its time, it was also novel in supporting algorithms that work on a changing graph.

The Falcon DSL is implemented as a graph extension to the C language, providing various new data types and syntax constructs for both structure and tuning. Only primitive data structures (point, edge, graph, set, collection) are provided. Therefore, implementing either frontier exploration or community detection suffers from exactly the same problems as in PGX Algorithm and Green-Marl: it is not abstract enough. However, this does also mean that the missing features can be implemented relatively straightforwardly in a similar way to how we extended GMIR.

GraphIt [82] (2018) and GraphIR [13] (2021) The GraphIt DSL and its successor IR encapsulation GraphIR are a standalone languages, similar to Falcon and GMIR in many ways. Unique to GraphIt is its split-DSL approach, an explicit separation between an algorithm language and a scheduling language. Due to the focus on developing such scheduling language for tuning purposes, due algorithm language itself is rather simple and therefore suffers from the same lack of higher-level algorithmic data types as Falcon and Green-Marl. Similarly, the missing features can be implemented as in Falcon, but consideration needs to be taken with respect to the scheduling language which needs to be kept in sync.

Chapter 10

Conclusions & Future Work

Expressing graph algorithms concisely, efficiently and correctly is and remains a broad and open-ended problem. The main problem we found with current approaches is the existence of a large conceptual gap between algorithm theory and realized implementations. Domain-specific language (DSL) are a proven tool for helping close this gap, which is why the Green-Marl intermediate representation (IR) (GMIR) language was introduced a few years ago. Because GMIR was intentionally been developed to counter the rigidness of Green-Marl, its expressiveness is limited. Our goal was to close the gap by identifying areas of improvement to broaden the algorithm support or expressiveness in GMIR.

In this chapter, we briefly summarize the work we did and draw our final conclusions on that based on the preceding evaluation. Following this, we discuss our work to provide input for future work.

10.1 Conclusions

- We performed an initial broad domain analysis. Based on the outcome of that, we selected three subdomains which we focused on: fixed-point iteration, frontier exploration and community detection.
- We performed more extensive domain analysis for each of these subdomains to distill the essence of these algorithms. This formed the basis for the syntax and static semantics which we introduced to abstract these concepts.
- We implemented all syntax and static semantics in the commercial Oracle GMIR compiler using the Spoofax Language Framework, following the footsteps of Voelter's seven design dimensions [77].
- We successfully implemented many algorithms using our extended syntax, directly showing that GMIR has become more expressive.
- By means of lowerings, we further demonstrated that these abstractions narrow the conceptual gap and can be meaningfully compiled to different existing commercial graph processing frameworks, such s PGX.D [36, 66] and the Oracle RDBMS.
- We filed U.S. patent applications for our abstractions for frontier exploration and community detection.

10.2 Future Work

This still leaves work to be done in various areas and opens doors for other future work in adjacent areas. We identify at least four clear areas based on the experience gathered during the execution of this project and based on the conclusions drawn above. We cover each in no particular order in limited detail, as that is far our of this thesis' scope.

10.2.1 User-Defined Exploration Policies

We proposed several different exploration policies to configure frontiers with. Some optimizations such as Δ-*stepping* [52] can currently not be implemented. It seems that optimizations that would be opaque to the processing block itself can be fully implemented as a user-defined exploration policy.

10.2.2 Divisive Community Detection

The opposite of this is divisive community detection, which seems similar but appears to be fundamentally different; moving and merging are no longer applicable. We intentionally only covered agglomerative community detection, as we realized early on that these differences are significant. Even though there appear to be less algorithms that perform divisive community detection, it does come with some unique benefits that are worth exploring.

One potential advantage over agglomerative community detection parallelization. Such an algorithm starts off with one large community, representing all vertices from the underlying graph. Once the first layer is finalized and several new communities are created, each community can be considered independently.

10.2.3 Multiple Membership & Fuzzy Logic

In our community detection design, a vertex is always associated to exactly one community, which in turn, represents one or more vertices. This is far from a perfect model of how real-world communities are formed. Consider for example a person (vertex) being part of a tight social circle with family and friends, but simultaneously being less strongly part of a community of colleagues at work. This information can only be captured if a vertex can be associated to more than one community at a time. Alternative algorithms such as clique percolation method (CPM) partially address this [56], but DSL support is currently lacking.

As already hinted at, not all community memberships are equal. Moreover, not all membership is certain. We only considered algorithms where the resulting community structure does not capture any of this multiplicity or uncertainty. Ideally, this information is propagated throughout the algorithm itself and to the caller. While separate, these two aspects both seem to involve probabilities. Supporting this correctly likely requires usage of fuzzy logic, for which some algorithms have been designed [9, 28, 48].

10.2.4 Random Walks and Sampling

In community detection, many algorithms run deterministically or at least behave correctly under deterministic execution. Some benefit from nondeterminism to reach convergence faster. However, some algorithms explicitly rely on randomness to ensure correct results [79]. First-class support for this currently exists to some degree, as a modifier on iterators.

During development, we also observed a common pattern in several scientific and proprietary algorithms that perform random walks to gather information from a neighborhood [59, 63]. This was performed by a relatively lengthy manual process. There seem to be some opportunities here, as a random walk can be seen as a controlled form of local frontier exploration. Such a frontier would be sequential/linear and have an exploration policy that selects the next hop either arbitrarily or based on some relative probability value. Additional considerations need to be made for features such as random restarts and mechanics to aggregate collected information.

10.2.5 Higher-Level Abstractions and Syntactic Sugar

The new abstractions described in this thesis were evaluated on GMIR, an IR-level language. Despite this, out abstractions are general in nature. We only used GMIR as it was the most suitable vehicle to perform our work on. Moreover, having performed our work in the context of an IR provided some degree of worry-free design as users typically do not interact with IRs. For example, we did not introduce any syntactic sugar, but there were some occasions arose where some syntactic sugar would have been convenient.

An IR is typically mapped to from a higher-level language in which users functionally express algorithms. For any of our new abstractions to be accessible from a higher-level language, corresponding abstractions need to be devised. It is likely that most if not all can be ported directly into a higher-level language. Still, there are opportunities for even higher-level features or syntactic sugar features, which is what we discuss in this subsection.

Automatic Priority Values Weighted frontiers need to have each element associated with some priority value. Sometimes, these values are static to algorithm and correspond directly to values of some property bound to the frontier elements. Both during initialization and expansion, these values are looked up by performing a property access to the same property. This could be captured by syntactic sugar where the name of the property in question is provided at the declaration site of the frontier. This makes the code more concise, less bug-prone and friendlier to refactoring.

Size Tracking For example, in agglomerative community detection, some algorithms keep track of the 'size' of communities. This term is very ambiguous, but one potential meaning given to it is the amount of vertices that are recursively a member of a given community. In other words, any community starts off with size 1, as they are all singletons by definition on the first layer. A move affects the source community by -1 and the target community by $+1$, whereas a merge nullifies the source community and adds all to the target community. The value upon aggregation is propagated to the next layer, effectively causing a move of one second-layer vertex to change the 'size' of the communities between which it moves to be modified by the amount of first-layer vertices are represented by that vertex. More precisely, this can be referred to as the recursive size of communities.

Performing this recursive size tracking manually is trivial but requires a handful of lines of code. While easy, it is typically an auxiliary value that algorithms need to keep track of and therefore secondary priority. Still, it is critical that this information is updated correctly at every location where vertices are rearranged.

Centralized Hooks In the proposed design, effects can be specified on a rearrangement instruction only. Taking the example of size tracking from above, this means that the size update instructions need to be placed on each rearrangement instruction. By allowing the effects to be specified global to the agglomeration, this redundancy can be resolved. A desugaring pass would simply copy the agglomeration-global effects to each related rearrangement.

Community Property Inheritance In community detection, communities can have properties associated. Our proposed design separates such properties from those on 'ordinary' vertices or edges. Even though in a hierarchical community construction communities from base layers are represented by ordinary vertices, they are defined on a graph distinct from the input graph and therefore do not share any properties.

A typical use-case is a distance property on edges, used to calculate how similar vertices are. Under community detection, the values of that property are used one-to-one in the first layer. On consecutive layers, aggregated values are used, which are calculated by summing the property values of related edges. For both cases together, the proposed design forces a separate community-bound property to be declared and initialized with copied values. This effectively results in a problem related to the *Automatic Priority Values*, but more generalized.

Default Aggregation Values of community-bound properties need to be aggregated to reflect the structural agglomeration. In practice, values are often summed. The design at hand always requires all aggregations to be provided explicitly. It may be feasible to assume that properties are aggregated by sum by default. This would reduce the code size even further, especially when combined with *Automatic Priority Values* or *Community Property Inheritance*.

Extended Projection: Spanning Trees of Communities Both in frontier exploration and community detection, we have the concept of projection to concisely expose certain derived information in more generic data structures. Currently, under community detection it is only possible to project the cross-hierarchy vertex-community assignment into a long property. The way that communities are formed under agglomeration appears to be related to spanning trees. It is unclear what this information would be relevant for, but the projections of community structures can be extended.

Bibliography

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. “Fast exact shortest-path distance queries on large networks by pruned landmark labeling”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’13. New York, NY, USA: Association for Computing Machinery, June 22, 2013, pp. 349–360. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2465315. URL: https://dl.acm.org/doi/10.1145/2463676.2465315 (visited on June 21, 2024).

[2] Renzo Angles et al. “G-CORE: A Core for Future Graph Query Languages”. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ’18. New York, NY, USA: Association for Computing Machinery, May 27, 2018, pp. 1421–1432. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3190654. URL: https://dl.acm.org/doi/10.1145/3183713.3190654 (visited on June 4, 2024).

[3] Hendrik van Antwerpen et al. “Scopes as types”. In: *Proceedings of the ACM on Programming Languages* 2 (OOPSLA Oct. 24, 2018), 114:1–114:30. DOI: 10.1145/3276484. URL: https://dl.acm.org/doi/10.1145/3276484 (visited on May 30, 2024).

[4] *Apache TinkerPop: Gremlin*. URL: https://tinkerpop.apache.org/gremlin.html (visited on May 14, 2024).

[5] *APOC user guide for Neo4j 5 - APOC Documentation*. Neo4j Graph Data Platform. URL: https://neo4j.com/docs/apoc/5/ (visited on Nov. 4, 2024).

[6] *AQL Documentation*. URL: https://docs.arangodb.com/3.12/aql/ (visited on May 31, 2024).

[7] B. Bebee et al. “Amazon Neptune: Graph Data Management in the Cloud”. In: International Workshop on the Semantic Web. 2018. URL: https://ceur-ws.org/Vol-2180/paper-79.pdf (visited on May 31, 2024).

[8] Kamal Berahmand and Asgarali Bouyer. “LP-LPA: A link influence-based label propagation algorithm for discovering community structures in networks”. In: *International Journal of Modern Physics B* 32.06 (Mar. 2018). Publisher: World Scientific Publishing Co., p. 1850062. ISSN: 0217-9792. DOI: 10.1142/S0217979218500625. URL: https://www.worldscientific.com/doi/abs/10.1142/S0217979218500625 (visited on July 17, 2024).

[9] James C. Bezdek, Robert Ehrlich, and William Full. “FCM: The fuzzy *c*-means clustering algorithm”. In: *Computers & Geosciences* 10.2 (1984), pp. 191–203. ISSN: 0098-3004. DOI: 10.1016/0098-3004(84)90020-7. URL: https://www.sciencedirect.com/science/article/pii/0098300484900207 (visited on Aug. 7, 2024).

[10] Vincent D. Blondel et al. “Fast unfolding of communities in large networks”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Oct. 2008), P10008. ISSN: 1742-5468. DOI: 10.1088/1742-5468/2008/10/P10008. URL: https://dx.doi.org/10.1088/1742-5468/2008/10/P10008 (visited on May 9, 2024).

[11] Houda Boukham et al. "A Multi-target, Multi-paradigm DSL Compiler for Algorithmic Graph Processing". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2022. New York, NY, USA: Association for Computing Machinery, Dec. 1, 2022, pp. 2–15. ISBN: 978-1-4503-9919-7. DOI: 10.1145/3567512.3567513. URL: https://dl.acm.org/doi/10.1145/3567512.3567513 (visited on May 7, 2024).

[12] Houda Boukham et al. "Spoofax at Oracle: Domain-Specific Language Engineering for Large-Scale Graph Analytics". In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. Open Access Series in Informatics (OASIcs). ISSN: 2190-6807. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 5:1–5:8. ISBN: 978-3-95977-267-9. DOI: 10.4230/OASIcs.EVCS.2023.5. URL: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.EVCS.2023.5 (visited on Dec. 14, 2024).

[13] Ajay Brahmakshatriya et al. "Taming the Zoo: The Unified GraphIt Compiler Framework for Novel Architectures". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). ISSN: 2575-713X. June 2021, pp. 429–442. DOI: 10.1109/ISCA52012.2021.00041. URL: https://ieeexplore.ieee.org/abstract/document/9499863 (visited on June 17, 2024).

[14] Sergey Brin and Lawrence Page. "The anatomy of a large-scale hypertextual Web search engine". In: *Computer Networks and ISDN Systems* 30.1 (Apr. 1998), pp. 107–117. ISSN: 01697552. DOI: 10.1016/S0169-7552(98)00110-X. URL: https://linkinghub.elsevier.com/retrieve/pii/S016975529800110X (visited on May 9, 2024).

[15] Benjamin Brock et al. *The GraphBLAS C API Specification: Version 2.0.0*. Nov. 15, 2021. URL: https://graphblas.org/docs/GraphBLAS_API_C_v2.0.0.pdf.

[16] Jochem Broekhoff. "Extracting LLVM Intermediate Representation from Agda". PhD thesis. Delft: Delft University of Technology, June 27, 2022. 20 pp. URL: https://repository.tudelft.nl/record/uuid:6ed6ad26-18c3-4427-b99a-c6241f7102c7 (visited on Feb. 24, 2025).

[17] Rong Chen et al. "PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs". In: *ACM Transactions on Parallel Computing* 5.3 (Jan. 22, 2019), 13:1–13:39. ISSN: 2329-4949. DOI: 10.1145/3298989. URL: https://dl.acm.org/doi/10.1145/3298989 (visited on June 4, 2024).

[18] Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. "Falcon: A Graph Manipulation Language for Heterogeneous Systems". In: *ACM Transactions on Architecture and Code Optimization* 12.4 (Dec. 22, 2015), 54:1–54:27. ISSN: 1544-3566. DOI: 10.1145/2842618. URL: https://dl.acm.org/doi/10.1145/2842618 (visited on May 17, 2024).

[19] Pengjie Cui et al. "CGgraph: An Ultra-Fast Graph Processing System on Modern Commodity CPU-GPU Co-processor". In: *Proceedings of the VLDB Endowment* 17.6 (May 3, 2024), pp. 1405–1417. ISSN: 2150-8097. DOI: 10.14778/3648160.3648179. URL: https://dl.acm.org/doi/10.14778/3648160.3648179 (visited on June 18, 2024).

[20] Andrew Davidson et al. "Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014 IEEE 28th International Parallel and Distributed Processing Symposium. ISSN: 1530-2075. May 2014, pp. 349–359. DOI: 10.1109/IPDPS.2014.45. URL: https://ieeexplore.ieee.org/abstract/document/6877269 (visited on June 21, 2024).

[21] Alin Deutsch et al. *TigerGraph: A Native MPP Graph Database*. Jan. 24, 2019. DOI: 10.48550/arXiv.1901.08248. arXiv: 1901.08248[cs]. URL: http://arxiv.org/abs/1901.08248 (visited on June 3, 2024).

[22] Edsger W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (Dec. 1, 1959), pp. 269–271. ISSN: 0945-3245. DOI: 10.1007/BF01386390. URL: https://doi.org/10.1007/BF01386390 (visited on June 24, 2024).

[23] Santo Fortunato and Marc Barthélemy. "Resolution limit in community detection". In: *Proceedings of the National Academy of Sciences* 104.1 (Jan. 2, 2007). Publisher: Proceedings of the National Academy of Sciences, pp. 36–41. DOI: 10.1073/pnas.0605965104. URL: https://www.pnas.org/doi/full/10.1073/pnas.0605965104 (visited on Sept. 23, 2024).

[24] Leonardo Fraccaroli et al. "FAST-CON: a Multi-source Approach for Efficient S- T Connectivity on Sparse Graphs". In: *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. 2023 IEEE High Performance Extreme Computing Conference (HPEC). ISSN: 2643-1971. Sept. 2023, pp. 1–6. DOI: 10.1109/HPEC58863.2023.10363544. URL: https://ieeexplore.ieee.org/document/10363544 (visited on June 21, 2024).

[25] Nadime Francis et al. "Cypher: An Evolving Query Language for Property Graphs". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, May 27, 2018, pp. 1433–1445. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3190657. URL: https://dl.acm.org/doi/10.1145/3183713.3190657 (visited on May 31, 2024).

[26] Xinbiao Gan et al. "GraphCube: Interconnection Hierarchy-aware Graph Processing". In: *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. PPoPP '24. New York, NY, USA: Association for Computing Machinery, Feb. 20, 2024, pp. 160–174. ISBN: 9798400704352. DOI: 10.1145/3627535.3638498. URL: https://dl.acm.org/doi/10.1145/3627535.3638498 (visited on June 10, 2024).

[27] *Graphs in AQL | ArangoDB Documentation*. URL: https://docs.arangodb.com/stable/aql/graphs/ (visited on May 15, 2024).

[28] Steve Gregory. "Fuzzy overlapping communities in networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2011.2 (Feb. 2011), P02017. ISSN: 1742-5468. DOI: 10.1088/1742-5468/2011/02/P02017. URL: https://dx.doi.org/10.1088/1742-5468/2011/02/P02017 (visited on July 16, 2024).

[29] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. "Making pull-based graph processing performant". In: *ACM SIGPLAN Notices* 53.1 (Feb. 10, 2018), pp. 246–260. ISSN: 0362-1340. DOI: 10.1145/3200691.3178506. URL: https://dl.acm.org/doi/10.1145/3200691.3178506 (visited on June 10, 2024).

[30] *GSQL Language Reference - GSQL Language Reference*. TigerGraph Documentation. URL: https://docs.tigergraph.com/gsql-ref/current/intro/ (visited on May 14, 2024).

[31] José Rolando Guay Paz. "Introduction to Azure Cosmos DB". In: *Microsoft Azure Cosmos DB Revealed: A Multi-Model Database Designed for the Cloud*. Ed. by José Rolando Guay Paz. Berkeley, CA: Apress, 2018, pp. 1–23. ISBN: 978-1-4842-3351-1. DOI: 10.1007/978-1-4842-3351-1_1. URL: https://doi.org/10.1007/978-1-4842-3351-1_1 (visited on May 31, 2024).

[32] Aric Hagberg. *NetworkX first public release (NX-0.2)*. E-mail. Apr. 12, 2005. URL: https://mail.python.org/pipermail/python-announce-list/2005-April/003924.html (visited on June 17, 2024).

[33] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (July 1968). Conference Name: IEEE Transactions on Systems Science and Cybernetics, pp. 100–107. issn: 2168-2887. doi: 10.1109/TSSC.1968.300136. url: https://ieeexplore.ieee.org/document/4082128 (visited on Feb. 25, 2025).

[34] Taher H. Haveliwala. "Topic-sensitive PageRank". In: *Proceedings of the 11th international conference on World Wide Web*. WWW '02. New York, NY, USA: Association for Computing Machinery, May 7, 2002, pp. 517–526. isbn: 978-1-58113-449-0. doi: 10.1145/511446.511513. url: https://dl.acm.org/doi/10.1145/511446.511513 (visited on Aug. 16, 2024).

[35] Sungpack Hong et al. "Green-Marl: a DSL for easy and efficient graph analysis". In: *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. New York, NY, USA: Association for Computing Machinery, Mar. 3, 2012, pp. 349–362. isbn: 978-1-4503-0759-8. doi: 10.1145/2150976.2151013. url: https://dl.acm.org/doi/10.1145/2150976.2151013 (visited on May 9, 2024).

[36] Sungpack Hong et al. "PGX.D: a fast distributed graph processing engine". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. New York, NY, USA: Association for Computing Machinery, Nov. 15, 2015, pp. 1–12. isbn: 978-1-4503-3723-6. doi: 10.1145/2807591.2807620. url: https://dl.acm.org/doi/10.1145/2807591.2807620 (visited on June 10, 2024).

[37] Sungpack Hong et al. *The Green-Marl Language Specification*. Mar. 1, 2024. url: https://pgx.us.oracle.com/releases/stable/otn/odocs/latest/specification.pdf (visited on June 10, 2024).

[38] ISO/IEC 39075:2024. *Information technology — Database languages — GQL*. Version 1. Geneva, CH, Apr. 2024. url: https://www.iso.org/standard/76120.html (visited on June 28, 2024).

[39] ISO/IEC 9075-16:2023. *Information technology — Database languages SQL - Part 16: Property Graph Queries (SQL/PGQ)*. Version 1. Geneva, CH, June 2023. url: https://www.iso.org/standard/79473.html (visited on Oct. 28, 2024).

[40] ISO/IEC 9075-4:2023. *Information technology — Database languages SQL - Part 4: Persistent stored modules (SQL/PSM)*. Version 7. Geneva, CH, June 2023. url: https://www.iso.org/standard/76585.html (visited on Jan. 10, 2025).

[41] Lennart C.L. Kats, Rob Vermaas, and Eelco Visser. "Integrated language definition testing: enabling test-driven language development". In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, Oct. 22, 2011, pp. 139–154. isbn: 978-1-4503-0940-0. doi: 10.1145/2048066.2048080. url: https://dl.acm.org/doi/10.1145/2048066.2048080 (visited on Feb. 21, 2025).

[42] Lennart C.L. Kats and Eelco Visser. "The Spoofax language workbench". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. OOPSLA '10. New York, NY, USA: Association for Computing Machinery, Oct. 17, 2010, pp. 237–238. isbn: 978-1-4503-0240-1. doi: 10.1145/1869542.1869592. url: https://dl.acm.org/doi/10.1145/1869542.1869592 (visited on May 30, 2024).

[43] Leonard Kaufmann and Peter J. Rosseeuw. "Divisive Analysis (Program DIANA)". In: *Finding Groups in Data*. Section: 6. John Wiley & Sons, Ltd, 1990, pp. 253–279. isbn: 978-0-470-31680-1. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470316801.ch6 (visited on Aug. 21, 2024).

[44] Jeremy Kepner et al. "Mathematical foundations of the GraphBLAS". In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 2016 IEEE High Performance Extreme Computing Conference (HPEC). Sept. 2016, pp. 1–9. DOI: 10.1109/HPEC.2016.7761646. URL: https://ieeexplore.ieee.org/abstract/document/7761646 (visited on June 10, 2024).

[45] Jon M. Kleinberg. "Authoritative sources in a hyperlinked environment". In: *J. ACM* 46.5 (Sept. 1, 1999), pp. 604–632. ISSN: 0004-5411. DOI: 10.1145/324133.324140. URL: https://dl.acm.org/doi/10.1145/324133.324140 (visited on Aug. 16, 2024).

[46] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Feb. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308. URL: https://ieeexplore.ieee.org/abstract/document/9370308 (visited on May 14, 2024).

[47] *Leiden algorithm*. In: *Wikipedia*. Page Version ID: 1221225623. Apr. 28, 2024. URL: https://en.wikipedia.org/w/index.php?title=Leiden_algorithm&oldid=1221225623 (visited on May 9, 2024).

[48] Wenjian Luo et al. "Community Detection by Fuzzy Relations". In: *IEEE Transactions on Emerging Topics in Computing* 8.2 (Apr. 2020), pp. 478–492. ISSN: 2168-6750. DOI: 10.1109/TETC.2017.2751101. URL: https://ieeexplore.ieee.org/abstract/document/8031356 (visited on June 18, 2024).

[49] Grzegorz Malewicz et al. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. SIGMOD '10. New York, NY, USA: Association for Computing Machinery, June 6, 2010, pp. 135–146. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807184. URL: https://dl.acm.org/doi/10.1145/1807167.1807184 (visited on May 13, 2024).

[50] *Managed Graph Database - Amazon Neptune - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/neptune/ (visited on May 14, 2024).

[51] Ke Meng et al. "A pattern based algorithmic autotuner for graph processing on GPUs". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. New York, NY, USA: Association for Computing Machinery, Feb. 16, 2019, pp. 201–213. ISBN: 978-1-4503-6225-2. DOI: 10.1145/3293883.3295716. URL: https://dl.acm.org/doi/10.1145/3293883.3295716 (visited on June 17, 2024).

[52] U. Meyer and P. Sanders. "Δ-stepping: a parallelizable shortest path algorithm". In: *Journal of Algorithms*. 1998 European Symposium on Algorithms 49.1 (Oct. 1, 2003), pp. 114–152. ISSN: 0196-6774. DOI: 10.1016/S0196-6774(03)00076-2. URL: https://www.sciencedirect.com/science/article/pii/S0196677403000762 (visited on June 10, 2024).

[53] *NetworkX — NetworkX documentation*. URL: https://networkx.org/ (visited on June 17, 2024).

[54] M. E. J. Newman and M. Girvan. "Finding and evaluating community structure in networks". In: *Physical Review E* 69.2 (Feb. 26, 2004). Publisher: American Physical Society, p. 026113. DOI: 10.1103/PhysRevE.69.026113. URL: https://link.aps.org/doi/10.1103/PhysRevE.69.026113 (visited on May 24, 2024).

[55] Lawrence Page et al. "The PageRank Citation Ranking : Bringing Order to the Web". In: The Web Conference. Nov. 11, 1999. (Visited on May 31, 2024).

[56] Gergely Palla et al. "Uncovering the overlapping community structure of complex networks in nature and society". In: *Nature* 435.7043 (June 2005). Publisher: Nature Publishing Group, pp. 814–818. ISSN: 1476-4687. DOI: 10.1038/nature03607. URL: https://www.nature.com/articles/nature03607 (visited on July 25, 2024).

[57] Sungwoo Park, Seyeon Oh, and Min-Soo Kim. "INFINEL: An efficient GPU-based processing method for unpredictable large output graph queries". In: *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. PPoPP '24. New York, NY, USA: Association for Computing Machinery, Feb. 20, 2024, pp. 147–159. ISBN: 9798400704352. DOI: 10.1145/3627535.3638490. URL: https://dl.acm.org/doi/10.1145/3627535.3638490 (visited on June 10, 2024).

[58] *PGX Algorithm Specification*. May 2, 2019. URL: https://docs.oracle.com/cd/E56133_01/24.2.2/PGX_Algorithm_Language_Specification.pdf (visited on May 31, 2024).

[59] Pascal Pons and Matthieu Latapy. *Computing communities in large networks using random walks (long version)*. Dec. 12, 2005. DOI: 10.48550/arXiv.physics/0512106. arXiv: physics/0512106. URL: http://arxiv.org/abs/physics/0512106 (visited on Sept. 26, 2024).

[60] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. "Near linear time algorithm to detect community structures in large-scale networks". In: *Physical Review E* 76.3 (Sept. 11, 2007). Publisher: American Physical Society, p. 036106. DOI: 10.1103/PhysRevE.76.036106. URL: https://link.aps.org/doi/10.1103/PhysRevE.76.036106 (visited on July 16, 2024).

[61] John H. Reif. "Depth-first search is inherently sequential". In: *Information Processing Letters* 20.5 (June 12, 1985), pp. 229–234. ISSN: 0020-0190. DOI: 10.1016/0020-0190(85)90024-9. URL: https://www.sciencedirect.com/science/article/pii/0020019085900249 (visited on Dec. 10, 2024).

[62] Oskar van Rest et al. "PGQL: a property graph query language". In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. GRADES '16. New York, NY, USA: Association for Computing Machinery, June 24, 2016, pp. 1–6. ISBN: 978-1-4503-4780-8. DOI: 10.1145/2960414.2960421. URL: https://dl.acm.org/doi/10.1145/2960414.2960421 (visited on May 31, 2024).

[63] Alejandro Pérez Riascos and José L. Mateos. "Random walks on weighted networks: a survey of local and non-local dynamics". In: *Journal of Complex Networks* 9.5 (Oct. 1, 2021), cnab032. ISSN: 2051-1329. DOI: 10.1093/comnet/cnab032. URL: https://doi.org/10.1093/comnet/cnab032 (visited on May 8, 2024).

[64] M. Rosvall, D. Axelsson, and C. T. Bergstrom. "The map equation". In: *The European Physical Journal Special Topics* 178.1 (Nov. 1, 2009), pp. 13–23. ISSN: 1951-6401. DOI: 10.1140/epjst/e2010-01179-1. URL: https://doi.org/10.1140/epjst/e2010-01179-1 (visited on Aug. 22, 2024).

[65] Martin Rosvall and Carl T. Bergstrom. "Maps of random walks on complex networks reveal community structure". In: *Proceedings of the National Academy of Sciences* 105.4 (Jan. 29, 2008). Publisher: Proceedings of the National Academy of Sciences, pp. 1118–1123. DOI: 10.1073/pnas.0706851105. URL: https://www.pnas.org/doi/full/10.1073/pnas.0706851105 (visited on May 9, 2024).

[66] Nicholas P. Roth et al. "PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine". In: *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. GRADES'17. New York, NY, USA: Association for Computing Machinery, May 19, 2017, pp. 1–6. ISBN: 978-1-4503-5038-9. DOI: 10.1145/3078447.3078454. URL: https://dl.acm.org/doi/10.1145/3078447.3078454 (visited on June 10, 2024).

[67] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library*. Boost C++ Libraries. Sept. 27, 2000. URL: `https://www.boost.org/doc/libs/1_85_0/libs/graph/doc/` (visited on June 17, 2024).

[68] Spoofax Team. *Spoofax - Reference*. URL: `https://www.spoofax.dev/references/` (visited on Jan. 17, 2025).

[69] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. "GraphGrind: addressing load imbalance of graph partitioning". In: *Proceedings of the International Conference on Supercomputing*. ICS '17. New York, NY, USA: Association for Computing Machinery, June 14, 2017, pp. 1–10. ISBN: 978-1-4503-5020-4. DOI: `10.1145/3079079.3079097`. URL: `https://dl.acm.org/doi/10.1145/3079079.3079097` (visited on June 10, 2024).

[70] Manuel Then et al. "The more the merrier: efficient multi-source graph traversal". In: *Proceedings of the VLDB Endowment* 8.4 (Dec. 1, 2014), pp. 449–460. ISSN: 2150-8097. DOI: `10.14778/2735496.2735507`. URL: `https://dl.acm.org/doi/10.14778/2735496.2735507` (visited on May 30, 2024).

[71] Yuanyuan Tian et al. "IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, May 31, 2020, pp. 345–359. ISBN: 978-1-4503-6735-6. DOI: `10.1145/3318464.3386138`. URL: `https://dl.acm.org/doi/10.1145/3318464.3386138` (visited on May 21, 2024).

[72] TigerGraph. *TigerGraph Announces Commitment to Support openCypher in GSQL*. TigerGraph Press Releases. Oct. 19, 2022. URL: `https://www.tigergraph.com/press-article/tigergraph-announces-commitment-to-support-opencypher-in-gsql/` (visited on Jan. 10, 2025).

[73] Matthew Towers. *Bidirectional Dijkstra*. Matthew Towers' homepage. May 30, 2020. URL: `https://www.homepages.ucl.ac.uk/~ucahmto/math/2020/05/30/bidirectional-dijkstra.html` (visited on May 15, 2024).

[74] V. A. Traag, L. Waltman, and N. J. van Eck. "From Louvain to Leiden: guaranteeing well-connected communities". In: *Scientific Reports* 9.1 (Mar. 26, 2019). Publisher: Nature Publishing Group, p. 5233. ISSN: 2045-2322. DOI: `10.1038/s41598-019-41695-z`. URL: `https://www.nature.com/articles/s41598-019-41695-z` (visited on May 9, 2024).

[75] Gintaras Vaira and Olga Kurasova. "Parallel Bidirectional Dijkstra's Shortest Path Algorithm". In: *Databases and Information Systems VI*. IOS Press, 2011, pp. 422–435. DOI: `10.3233/978-1-60750-688-1-422`. URL: `https://ebooks.iospress.nl/doi/10.3233/978-1-60750-688-1-422` (visited on June 17, 2024).

[76] Leslie G. Valiant. "A bridging model for parallel computation". In: *Communications of the ACM* 33.8 (Aug. 1, 1990), pp. 103–111. ISSN: 0001-0782. DOI: `10.1145/79173.79181`. URL: `https://dl.acm.org/doi/10.1145/79173.79181` (visited on May 24, 2024).

[77] Markus Voelter. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. In collab. with Sebastian Benz et al. 2013. 558 pp. URL: `https://voelter.de/data/books/markusvoelter-dslengineering-1.0.pdf` (visited on June 18, 2024).

[78] Yangzihao Wang et al. "Gunrock: GPU Graph Analytics". In: *ACM Transactions on Parallel Computing* 4.1 (Aug. 23, 2017), 3:1–3:49. ISSN: 2329-4949. DOI: `10.1145/3108140`. URL: `https://dl.acm.org/doi/10.1145/3108140` (visited on June 17, 2024).

[79] Jierui Xie, Boleslaw K. Szymanski, and Xiaoming Liu. "SLPA: Uncovering Overlapping Communities in Social Networks via a Speaker-Listener Interaction Dynamic Process". In: *2011 IEEE 11th International Conference on Data Mining Workshops*. 2011 IEEE 11th International Conference on Data Mining Workshops. ISSN: 2375-9259. Dec. 2011, pp. 344–349. DOI: `10.1109/ICDMW.2011.154`. URL: `https://ieeexplore.ieee.org/document/6137400` (visited on May 9, 2024).

[80] Reynold S. Xin et al. "GraphX: a resilient distributed graph system on Spark". In: *First International Workshop on Graph Data Management Experiences and Systems*. GRADES '13. New York, NY, USA: Association for Computing Machinery, June 23, 2013, pp. 1–6. ISBN: 978-1-4503-2188-4. DOI: `10.1145/2484425.2484427`. URL: `https://dl.acm.org/doi/10.1145/2484425.2484427` (visited on June 4, 2024).

[81] Da Yan et al. "Pregel algorithms for graph connectivity problems with performance guarantees". In: *Proceedings of the VLDB Endowment* 7.14 (Oct. 1, 2014), pp. 1821–1832. ISSN: 2150-8097. DOI: `10.14778/2733085.2733089`. URL: `https://dl.acm.org/doi/10.14778/2733085.2733089` (visited on May 24, 2024).

[82] Yunming Zhang et al. "GraphIt: a high-performance graph DSL". In: *Proceedings of the ACM on Programming Languages* 2 (OOPSLA Oct. 24, 2018), 121:1–121:30. DOI: `10.1145/3276491`. URL: `https://dl.acm.org/doi/10.1145/3276491` (visited on May 17, 2024).

Acronyms

API application programming interface

AST abstract syntax tree

BFS breadth-first search

BNF Backus–Naur form

BSP bulk-synchronous parallel

CPM clique percolation method

DFS depth-first search

DSL domain-specific language

GM Green-Marl

GPL general-purpose language

GMIR Green-Marl IR

HITS hyperlink-induced topic search

IR intermediate representation

LPA label propagation algorithm

ML machine learning

MSA minimum spanning arborescence

MST minimum spanning tree

NF normal form

PGX Parallel Graph AnalytiX

PPR personalized PageRank

SLOC source lines of code

SLPA speaker-listener label propagation algorithm

SSSP single-source shortest path

TCG tentative community graph

WCC weakly-connected components

Appendix A

Eenvoudige Samenvatting

Grafen zijn fundamentele wiskundige structuren die in de praktijk erg geschikt zijn om mee te modelleren. Een graaf bestaat uit knopen die die onderling met lijnen zijn verbonden. Een wegennetwerk kan bijvoorbeeld worden gemodelleerd door wegen als lijnen en verkeersknooppunten (kruispunten, rotondes, etc.) als knopen te zien. Als een weg eenrichtingsverkeer is, of als beide rijrichtingen los vastgelegd moeten worden, kunnen pijlen in plaats van lijnen gebruikt worden. Dit noemen we dan een gerichte graaf.

In de praktijk zijn lijnen en knopen niet voldoende om de werkelijkheid te beschrijven. Daarom kennen we vaak eigenschappen toe aan lijnen en/of knopen, in de vorm van een label met bijbehorende informatie. In Figuur 1.1 is een kleine gerichte graaf getekend met eigenschappen op de pijlen. Dit is een eenvoudig model waar de knopen personen representeren en de pijlen banktransacties tussen die personen. De pijlen hebben een eigenschap 'amount' (hoeveelheid) die beschrijft hoeveel geld er met een transactie (pijl) gemoeid is.

Grafen met eigenschappen zijn handig om informatie in op te slaan, maar dit doen we niet zonder reden. Het doel van informatie zo systematisch vastleggen is uiteindelijk om het programmatisch te analyseren. algoritmes die hiervoor ontworpen zijn zijn talrijk, zoals we verderop zullen zien. Om zulk soort algoritmes in de praktijk te brengen, zijn er grofweg twee hoofdzaken waar men mee te maken krijgt. Allereerst moet het algoritme uitgedrukt worden, waar meestal een programmeertaal voor gebruikt wordt. Vervolgens moet het ook daadwerkelijk uitgevoerd worden met de opgeslagen grafen. Hoewel de werking van de uitvoerende systemen invloed heeft op wat überhaupt mogelijk is, is dit niet relevant voor deze samenvatting. In deze scriptie houden we ons voornamelijk bezig met het uitdrukken van zulk soort algoritmes, los van beperkingen van dit soort systemen.

De focus ligt op GMIR, een zgn. domein-specifieke programmeertaal voor graafalgoritmes [11]. Dat een programmeertaal domein-specifiek is wil zeggen dat het ontworpen is om binnen een bepaald domein, in dit geval graafalgoritmes, gebruikt te worden. Waar algemene programmeertalen zich op diverse gebruiksdoeleinden richten, zijn domein-specifieke talen voor afgebakende doelen binnen hun domein bedoeld. Zodoende is het niet mogelijk om iets anders met GMIR te doen dan algoritmes voor grafen te schrijven.

Het probleem met GMIR is echter dat het te strikt afgebakend is. GMIR is oorspronkelijk door Oracle van de grond af opgebouwd als vervanger van de voorganger Green-Marl. Als tegenreactie op het feit dat Green-Marl te algemeen bleek te zijn, is GMIR bewust uiterst doelgericht ontwikkeld, wat tot een relatief beperkte taal leidde. De mate waarin GMIR gebrekkig is hebben wij alleereerst onderzocht.

In drie verschillende gebieden hebben we gebreken gevonden. De tweede stap van ons onderzoek was om hier een zinvolle invulling voor te geven. Per gebied hebben we GMIR uitgebreid met nieuwe grammatica en semantiek om meer algoritmes in dat gebied uit te kunnen drukken. Tegelijkertijd hebben we rekening gehouden om deze nieuwe toevoegingen in de geest van GMIR te houden. Dat betekent bijvoorbeeld dat de grammatica dezelfde stijl aanhoudt en dat de mate van complexiteit op een gelijk niveau blijft.

Samenvattingen per gebied zijn hieronder in Appendix A.1 te vinden.

De ideeën van het tweede en het derde gebied zijn in de loop van onze aanstelling bij Oracle ingediend als patentaanvraag in de V.S.

Uiteindelijk hebben we alles ook daadwerkelijk geïmplementeerd in een Oracle-interne compiler met behulp van het Spoofax-framework [12, 42]. Om het tot op zekere hoogte mogelijk te maken om algoritmes die nu wel in GMIR uitgedrukt worden ook uit te voeren, hebben we ook bepaalde transformaties geïmplementeerd die dit realiseren. Let wel dat deze transformaties die de nieuwe syntax naar bestaande syntax normaliseren veelal werken door gebruik te maken van functies waarvan aangenomen wordt dat deze een platformspecifieke implementatie hebben (zgn. ingebouwd gedrag). Daarmee wordt in principe afbraak gedaan aan de domeinspecificiteit, maar dat is acceptabel omdat dit exclusief voor intern gebruik is.

A.1 Samenvatting per gebied

Gebied 1: herhalingen Het was al mogelijk om herhalingen uit de drukken, maar niet op een idiomatische[1] manier. Een typische toepassing van herhalingen in de context van grafen is het bepalen van de PageRank-score[2] [55]. In Figuur 2.3 wordt dit visueel weergegeven door grootte van de knopen te relateren aan de score. Aanvankelijk is de score voor alle knopen gelijk, maar door een deel van de score als het ware door de pijlen te laten 'stromen' krijgen knopen met meer inwaartse pijlen een hogere score. Geleidelijk stabiliseert de score, afhankelijk van de grootte van de graaf.

In elke herhaling worden zo dus nieuwe scores bepaald op basis van de voorgaande score. Ervan uitgaande dat de score als knoopeigenschap is vastgelegd, betekent dat in een naïeve oplossing dat er twee eigenschappen nodig zijn die bij elke herhaling omgewisseld moeten worden. Om dit te voorkomen, hebben wij een speciale herhalingslus geïntroduceerd die bewust is van eigenschappen die betrokken zijn bij de herhaling. Globaal bestaat er slechts een eigenschap, de score, maar binnen de herhaling kunnen beiden de huidige en opvolgende eigenschappen benaderd worden. Daarmee wordt effectief de verantwoordelijkheid van het beheren van de twee eigenschappen en het omwisselen daarvan uit handen gegeven.

Gebied 2: grensvlakverkenning Een manier om een (grote) graaf te benaderen in behapbare stukken is door elk element – knoop, lijn of pijl – los te bekijken. Gangbaarder en nuttiger is om dit systematischer te doen door rekening te houden met de omgeving waarin zich de elementen bevinden. Neem bijvoorbeeld een willekeurige knoop om mee te beginnen om vervolgens alle verbonden knopen te nemen en zo door. Afhankelijk van de keuze van dit startpunt in combinatie met de structuur van de graaf volgt hieruit ook een volgorde in alle knopen. Dit staat in sterk contrast met een willekeurige volgorde; de gestructureerde volgorde bevat namelijk ook informatie an sich.

Een praktischere toepassing kunnen we zien in Dijkstra's algoritme [22], het meest bekende algoritme om het kortste pad tussen twee knopen te bepalen. In Figuur 2.7 zijn een tussentijdse situatie en het uiteindelijk bepaalde kortste pad getekend. In deze voorbeeldsituatie is de verkenning bij knoop 's' gestart en is 't' het doel. In Figuur 2.7a is te zien dat een deel van de graaf verkend is, maar een ander deel niet (gestippeld). Op elke knoop is met een cijfer aangeduid wat de kortste afstand is tot daar vanaf de start ('s'). Gaandeweg meer mogelijke paden ontdekt worden, kan de afstand soms ingekort worden. De knopen waarvoor dit nog mogelijk is zijn grijs gearceerd, die zijn dus nog niet 'definitief'; niet-gearceerde

[1] D.w.z. op een manier die gebruikelijk is voor het domein; in termen van het vakgebied

[2] Typisch gebruikt door Google om webpagina's te rangschikken

knopen zijn dat dus wel. Dijkstra's strategie is om de meestbelovende knoop te kiezen om mee verder te gaan, dat is dus die met de korste afstand.

Hoewel het voorbeeld hier slechts met een kleine graaf werkt, is het duidelijk te zien dat er een driedeling is: afgehandelde, te bezoeken en nog niet ontdekte knopen. De knopen die al ontdekt zijn in de buurt van eerder bezochte knopen, maar nog niet behandeld zijn, vormen als het ware een grensvlak tussen een bekende en een onbekende fractie van de algehele graaf.

Als we de ontwikkeling van dit grensvlak gedurende de verkenning volgen, observeren we dat dit gedrag uitgedrukt kan worden in twee parameters: een rangschikkingsbeleid en een verkenningsbeleid. Het rangschikkingsbeleid van een algoritme schrijft voor hoe ontdekte elementen gerangschikt worden. In het geval van Dijkstra's algoritme gebeurt dit op basis van de waarde van een eigenschap – afstand hier. Andere algoritmes hebben geen speciale wens, of doen dit simpelweg op basis van de volgorde van ontdekking. Voor meer details over het verkenningsbeleid, zie Hoofdstuk 6; voor eenvoudige grensvlakerkenning is dit meestal niet relevant.

Deze ideeën hebben wij onder een paar bouwblokken samengevoegd. Zodoende is het mogelijk om in GMIR het grensvlak te configureren met het rangschikkingsbeleid. Om de verkenning af te trappen, moet eenmalig ten minste een element opgegeven worden. Voor elk element dat daarna volgens het rangschikkingsbeleid uit het grensvlak geselecteerd wordt, kan verdere logica gespecificeerd worden. Dit is normaliter waar een algoritme de relevante informatie verzamelt en ook besluit welke aangelegen knopen verder verkend zouden moeten worden.

Gebied 3: gemeenschapsstructuurbepaling Netwerkstructuren die vaak als grafen vastgelegd worden bevatten vaak verborgen structuren. Neem bijvoorbeeld een graaf van een sociaal netwerk waarmee interacties tussen mensen zijn gemodelleerd. Als een groep mensen vaak met elkaar omgaat, is dat een aanwijzing dat ze mogelijk een bepaalde gemeenschap vormen. Hoewel dit voor een mens relatief eenvoudig te bepalen is, is dit algoritmisch niet zo. Dit kan op allerlei manieren worden aangepakt en uitgelegd, wat er ook toe leidt dat algoritmes die dit soort structuren in een graaf kunnen detecteren relatief complex zijn en altijd een zekere foutmarge hebben.

De meest bekende techniek om algemene gemeenschapsstructuren in grafen te ontdekken is de Louvain-methode [10], vernoemd naar de locatie van de auteurs. Dit algoritme neemt de graaf zoals deze is en wijst initieel aan elke knoop een unieke identificatie toe waarmee eigenlijk aangeduid wordt dat elke knoop een losse gemeenschap is. Vervolgens wordt elke knoop afzonderlijk bekeken. Door elke knoop met aangelegen knopen te vergelijken wordt besloten of het aannemelijk is dat de knoop niet bij een van die gemeenschappen behoort. Zo ja, dan wordt unieke identificatie gewijzigd. Dit proces wordt herhaald totdat geen enkele knoop meer tussen gemeenschappen schuift.

Zodoende worden gemeenschappen gevormd die elk een of meerdere knopen bevatten. Dit kan een nuttig eindresultaat zijn, maar in de praktijk zien we dat gemeenschappen vaak een zekere hierarchie hebben. Alle sociale interacties wereldwijd zouden bijvoorbeeld alleereerst opgedeeld kunnen worden per land, om vervolgens per taal of dialect gesplitst te worden. Deze verfijning in de geemeenschapsstructuur kan nog veel verder gaan tot op het niveau van vriendengroepen bijvoorbeeld.

Het algoritme zover beschreven levert doorgaans een relatief fijnmazige gemeenschapsstructuur – het niveau van vriendengroepen. Louvain schrijft ook de techniek voor om hier een hierarchie in aan te brengen. Dit gebeurt door de gevormde gemeenschappen in een nieuwe graaf als knopen te modelleren, waarop het eerder beschreven algoritme opnieuw uitgevoerd wordt. Belangrijk hiervoor is dat de informatie van de originele graaf correct samengevat wordt om de hogere hierarchie zinvol te informeren.

In de praktijk levert Louvain redelijke resultaten op, maar er zijn scenario's waarin het fundementeel incorrecte resultaten oplevert. In de loop der tijd zijn er andere algoritmes zoals Infomap [63] en Leiden [74] gepubliceerd die deze problemen aanpakken. Beiden bouwen op de fundamentele structuur van Louvain, maar breiden het uit met uiteenlopende verbeteringen.

Om al deze nuances die we in Hoofdstuk 6 uitgebreider behandelen vast te leggen in GMIR hebben we uiteindelijk een resultaat geleverd vergelijkbaar met gebied 2. We wijden een gespecialiseerde veriant van de gewone graaf aan de graaf die de gemeenschappsstructuur omvat. Dit doen we bewust om onderscheid te maken en om de andere onderdelen eenvoudig toepasbaar te maken. Vervolgens maken we de aanname dat dat elke knoop initieel een losse gemeenschap vormt. Waar nodig kan het algoritme hierin sturen en een andere bron opgeven waarmee voorgevormde gemeenschappen overgenomen worden. Dan hebben we het hoofdbouwblok waarmee een niveau van de hierarchie geconstruëerd kan worden door gebruik te maken van de twee gemeenschapsbewerkingen 'samenvoegen' (van twee gemeenschappen) en 'verplaatsen' (van een knoop van de ene naar een andere gemeenschap). Ter ondersteuning van het bepalen van de kwaliteit van zo'n bewerking is het ook mogelijk om dit de simuleren en de effecten te observeren alvorens het daadwerkelijk toe te passen. Dit hoofdbouwblok abstraheert alle verdere boekhouding die plaatsvindt om de groeperingen en verschuivingen daadwerkelijk te realiseren. Wanneer een volledige hierarchie wenselijk is, kan dit blok in een herhalingslus geplaatst worden (zie gebied 1).