SMARTGRAPH: AN ARTIFICIALLY INTELLIGENT GRAPH DATABASE

Hal Cooper[1], Garud Iyengar[1], and Ching-Yung Lin[2]

[1]Department of Industrial Engineering and Operations Research
[2]Department of Electrical Engineering, Columbia University, New York, USA

ABSTRACT

Graph databases and distributed graph computing systems have traditionally abstracted the design and execution of algorithms by encouraging users to take the perspective of lone graph objects, like vertices and edges. In this paper, we introduce the SmartGraph, a graph database that instead relies upon thinking like a smarter device often found in real-life computer networks, the router. Unlike existing methodologies that work at the subgraph level, the SmartGraph is implemented as a network of artificially intelligent Communicating Sequential Processes. The primary goal of this design is to give each ``router'' a large degree of autonomy. We demonstrate how this design facilitates the formulation and solution of an optimization problem which we refer to as the "router representation problem", wherein each router selects a beneficial graph data structure according to its individual requirements (including its local data structure, and the operations requested of it). We demonstrate a solution to the router representation problem wherein the combinatorial global optimization problem with exponential complexity is reduced to a series of linear problems locally solvable by each AI router.

KEYWORDS

Intelligent Information, Database Systems, Graph Computing

1. INTRODUCTION

Data is often thought of in the context of input and output, to be used or analyzed by some external program or process. The structure of graph data, however, can indicate useful information about how to best execute graph-based algorithms on that structure. This is demonstrated by a key refrain for existing graph computing paradigms; to "think like a vertex" [1]. In this work, we demonstrate the benefits of further integrating graph data with the analytics run on that data by creating an artificially intelligent graph database. When graphs have knowledge of their own properties, the ability to send messages to other graphs, run calculations concurrently, and perform self-modification, a graph is no longer a static source of data. It instead begins to resemble a network of routers. In this work we replace the "think like a vertex" mantra with "think like a router", using a router inspired abstraction to create a "SmartGraph" database. The method distinguishes itself from existing graph databases through the artificially intelligent router abstraction; the routers that are defined by the subgraphs they encapsulate manage graph representation, concurrency and execution of operations themselves as opposed to being simple static data managed by an external process.

Dhinaharan Nagamalai et al. (Eds) : ACSIT, SIPM, ICITE, ITCA - 2019
pp. 63–77, 2019. © CS & IT-CSCP 2019 DOI: 10.5121/csit.2019.90307

Asynchronous concurrent execution is difficult in traditional distributed graph computing systems like Pregel [2] in part because the "think like a vertex" mantra (that was chosen to make "reasoning about programs easier") is typically implemented using Bulk Synchronous Parallel [3] methodologies. This involves top-level maintenance of lists of "active" vertices (and/or edges) during each "superstep" that all perform the same vertex calculation. This can waste many computing cycles [4], since many graph algorithms do not converge at the same rate across different nodes. Proposed solutions to this problem often focus on maintaining sets of which nodes (or edges) need to be updated during each iteration, and therefore despite acknowledgment of asymmetric convergence rates, ultimately still use a synchronous iterative system.

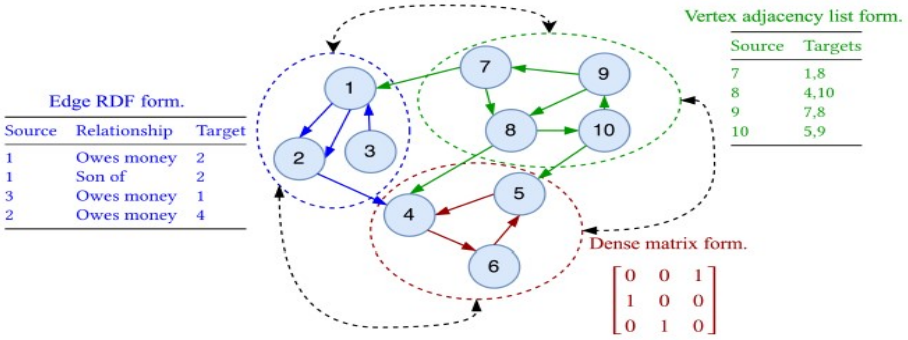

Figure 1. Toy graph (LHS) and a possible go routine router assignment

A key difference between the existing methods for graph computing and the system outlined in this paper is in having complex but manageable asynchronous concurrency of execution that eliminates the need for iterative supersteps and, more generally, the need for external macro-level management of the graph database and execution process. Though there exist previous works which have used subgraphs as a base unit for organizing computations, these works either limit their asynchronicity to "within subgraph" computations [5], or define subgraphs as connected components [6], [7]. In general, we wish to define subgraphs with less restrictions than existing methodologies allow in order to take advantage of more local structure, and to give these local structures individual autonomy.

There are many clear benefits to having highly concurrent asynchronous routers capable of "thinking for themselves". For example, routers can receive, execute, and transmit graph query operations without needing to know (and limit their speed to) the global supersteps, as well as automatically handle locally relevant operations concurrently with other routers; though we leave investigations of these properties to future work. In this paper, we focus on outlining the design of the Smart Graph itself and demonstrate how its asynchronous concurrency capabilities facilitate local artificial intelligence. In particular, we demonstrate the value of this approach through the "router representation problem", where "routers" use machine learning methods and solve local optimization problems to great effect (i.e. well approximating the globally optimal solution).

2. CONCURRENCY AND COMMUNICATING SEQUENTIAL PROCESSES

In a modern graph database, it is a functional requirement not only that the system be able to deal with large amounts of data, but that the system be able to deal with a large amount of different

requests with limited computational resources. The SmartGraph strives to facilitate massive concurrency involving graph operations on graph-based data by explicitly tying this graph structure into a concurrency management mechanism (through the router abstraction outlined in Section 3).

Concurrency is a property of a system that allows multiple processes (that may be related or entirely distinct) to have overlapping lifetimes. This does not necessarily mean that the multiple concurrent processes execute simultaneously at the hardware level (indeed, unlike parallelism, concurrency can be achieved on a single thread), it may simply refer to processes being able to be paused momentarily on a single thread, while other processes are given priority.

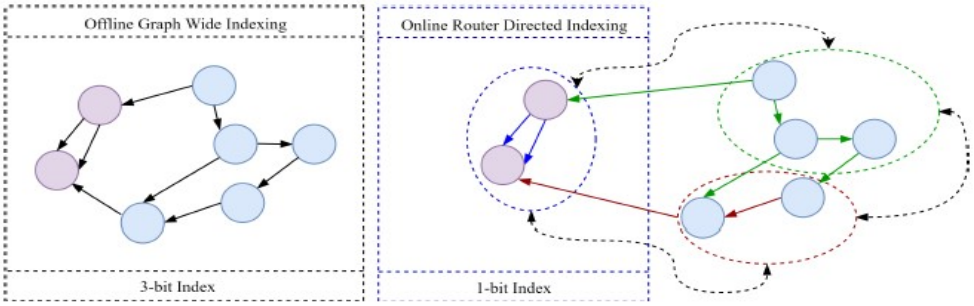

Figure 2. Toy comparison of graph-level and router-level vertex indexing

Communicating Sequential Processes are a method of implementing concurrency in programs based on message passing. This work takes some liberty with the theory of CSP as a whole, and instead focuses on CSP as it is implemented in the programming language Go (a.k.a. Golang) [8], developed by Google in 2009 with a focus on concurrency primitives as first-class citizens. Go is a language often described as a systems programming language, and is commonly used to design services, web-servers etc. It is with these capabilities in mind that we have created the Smart Graph in the Go language.

In Go, CSP is implemented through two main concurrency primitives: Go routines and channels. A go routine is essentially a very lightweight function that can be multiplexed onto different threads to be run concurrently with one another (and the Main, which is itself a goroutine). Goroutines communicate not by sharing memory, but “share memory by communicating” through the use of the channels. Of note is that although goroutines act like threads, they are not threads. This allows us to design a more lightweight concurrency mechanism, with many more go routines than available threads. This is ultimately of critical importance, as it allows us to simplify the management of graphs with a number of partitions at scales typically unheard of in the literature [9].

3. THE ROUTER

The SmartGraph concept is implemented in Go by explicitly tying goroutines to the graph-structured data that is to be explored or used in algorithms to be executed. Individual goroutines define (not merely control) the subgraphs of interest, with the relevant graph structure, vertex and edge properties defined in the variable size stack corresponding to the particular goroutine. The goroutines are our method of implementing the routers, and the highly concurrent functionality of

the goroutine is the mechanism through which we implement the communication and execution of analytics.

This approach contrasts strongly with the very strict definition of a subgraph (as connected components) used in prior works [6], [7] as it allows us to arbitrarily assign vertices and edges to routers. Though this may seem similar to partitioning done at the network or thread level in distributed graph computing [9], recall that goroutines are far more numerous than threads, and balancing occurs by multiplexing goroutines onto threads, in contrast to finding a partition that provides a balanced cut of the graph to all available threads. This allows us to partition the graph into many more pieces than might be indicated by the number of threads. Having separate graph structures in the routers facilitates taking advantage of local graph structure or optimizing with respect to locally requested operations. An example of this phenomenon, which we refer to as the "router representation problem" is explored in Section 4.

Through viewing this as the graph managing its own concurrent execution, one can do away with the top-level maintenance of sets of vertices, or algorithm iteration supersteps required by BSP methods. Concurrent execution is instead automatically handled by the router routines, as they will (through their status as a goroutine) immediately signal to the scheduler when they are ready (having received required data or messages from parent goroutines) to execute desired operations. This allows the system to be highly asynchronous, yet manageable thanks to the router abstraction. A basic example encapsulating a SmartGraph in goroutine routers is given in Figure 1. Observe on the LHS, a toy directed graph with no particularly noteworthy node or edge features. The RHS shows an example set of three goroutine routers that encapsulate the SmartGraph. Each router, defined by a dotted ellipse of a primary color, contains a subset of nodes and edges from the overall graph. The dotted black connections on the RHS represent channels between the source and destination routers.

The network router is useful not only as an abstraction model for efficiently implementing and executing graph algorithms, it also inspires useful functionality for the SmartGraph. Network routers have the ability to maintain useful information in memory, such as routing tables. They also possess the ability to perform custom routing logic independently of other routers. There are both obvious and subtle ways in which a graph database and analytics platform aping this functionality provides benefits. For example, a straight-forward method of employing the benefits of local representation is in saving memory address space as highlighted by Figure 2. The “local” aspect of the router AI can also be taken further than simple indices. With routers able to act independently, we introduce the notion of “local subgraph representations”, where the routers encapsulating a subgraph do so via different storage mechanisms and graph formats. For instance, one router AI may organize the subgraph it encapsulates using vertex adjacency lists, with another router AI learning to use an RDF framework. An example of this sort of router dependent representation is given in Figure 1. This approach has both computational benefits (since some operations have faster implementations in certain graph representations) and storage benefits (e.g. when using a sparse representation like CSR, or when compressing a subgraph consisting of nodes with high similarity).

4. THE ROUTER REPRESENTATION PROBLEM

The representation problem, as defined in this paper, is closely related to the more generic problem of selecting data structures either at compile-time or run-time, in order to minimize

memory usage, execution time, or some combination of the two. In contrast to this work, the literature focuses almost exclusively on optimization with respect to standard container objects (such as lists, sets, arrays, hash Maps etc) rather than graph specific data structures. A small number of papers due address the problem in a more graph specific context; for example, [10] directly considers the impact of basic graph operations and representations on execution time. However, it only uses the vertex adjacency list form and investigates how the single graph representation can be constructed by different container types, and so does not directly consider multiple graph specific data structures (i.e. alternatives to the adjacency list). Furthermore, [10] considers representations that "change" over time, but at any given time applies a single data structure to the entire graph. This means that the method cannot benefit from local graph structure. In contrast, [11] explicitly investigates different graph structure representations (both the adjacency list and the adjacency matrix), but again only considers their choice in application to the entire graph.

The data structure selection problem has been approached from three primary directions. The first is that of optimization, where benchmarking is used to create a function that approximates the execution time of a series of operations [10]. The second is using machine learning to generate rule sets that can be used to determine the choices of representations (e.g. "if BFS is called on a graph with density greater than 25 percent, use an adjacency matrix, else use an adjacency list") [12]. The final commonly used method is to simply provide a framework for implementing swap rules, and allow the user to specify precisely what those rules are manually [13].

Unique to this work, we explore methods for choosing local graph representations (that is, structural representations that apply locally rather than to the whole graph) by solving a set of optimization problems over learned models that consider both the local graph data, as well as the graph operations requested in relation to that data. Furthermore, we do so by deconstructing the problem such that a problem that initially appears exponential in complexity becomes linear in the number of routers by the number of router representations. In Section 4.1 we introduce the basics of the representation problem. In Section 4.2, we introduce our method for solving the representation problem under idealized circumstances, and in Section 4.3, outline how we learn the functions required to solve the problem in practice. This ultimately involves using average router behaviour as an approximation to input, which we argue in this section and demonstrate in Section 5, is an approximation that ultimately works very well.

4.1. REPRESENTATION METHODOLOGY

Let $G = (V,E)$ be a known property graph, i.e. a graph with properties attached to vertices and edges, where we allow multi-edges (e.g. representing different types of relationships between the same two objects). Let $P = (P_1,P_2,P_3,...,P_k)$ denote a partition of the edge set E. Let $V_i = \{u: (u,v)$ or $(v,u) \in P_i\}$. Note that we allow for a vertex v to be an element of more than one V_i. In a SmartGraph database, a separate router $\mathcal{R}_i$ encapsulates the subgraph (V_i,P_i).

We emphasize that the partitioning introduced here should not necessarily be considered in the same vein as traditional graph partitioning [9], as we are operating on a single machine unconstrained by threads, and as such, we do not overly care about partition balance.

Let S denote the set of all graph representations allowed by the SmartGraph. Let $R_i \in S$ denote the representation employed by router $\mathcal{R}_i$ for (V_i, P_i), $i = 1, ..., k$. In this work, the allowed set S of representations includes the Matrix Adjacency List (MAL; though we typically use the term adjacency list in full), wherein vertices are stored in a hash-table like structure, with lists of outbound vertices), and the Resource Description Format (RDF), which stores separate tables of edges and vertices. We deliberately use the term tables here, as the use of RDF can be thought of as functionally similar to implementing a graph database in a traditional relational database system. These two representations are by far the most common method of representing graphs in practice. For example, the RDF format is used in Spark, and it's GraphX [14] and GraphFrames [15] packages, whereas adjacency lists are used by Neo4j [16], a leading commercial graph database system.

There are many more graph representations, including adjacency matrices (often impractical in practice due to large storage requirements), sparse matrix representations (where-in a user is more focused on efficiently performing mathematical operations than graph operations; though the two are often closely connected), and representations with particular properties (such as allowing directed graphs only, disallowing multi-edges, cyles) etc. There are also custom representations designed to work efficiently with GPU operations [17], [18]. Thus it should be noted that the combination of representations used in this paper (and solved by the SmartGraph router abstraction herein) is not intended to argue that it will always result in the most efficient representative structure. Instead, we are arguing for a methodology for combining arbitrary different representation possibilities, and using the RDF and adjacency list as our representations for purposes of demonstration.

4.2. PREDICTING EXECUTION TIMES AND CHOOSING REPRESENTATIONS

We represent a database job $\mathbf{x} \equiv \{x_1, x_2, ..., x_M\}$ as a sequence of M known operations, that we classify as either "complex", or "basic". Complex operations are those that internally execute other operations (either complex or basic) to complete their execution, those that require knowledge of multiple routers (such as inspecting the number of vertices in a given router that are duplicated in other routers), and those that requiring complex input. What remains we describe as basic operations. We allow for directed acyclic dependence between operations of a job, i.e. all parents of an operation x_a, $\pi(x_a)$ must be executed before x_a. Let $\mathcal{B}$ denote the set of l basic operations. We assume that the average run time for a basic operation $b \in \mathcal{B}$ on a graph with n vertices and m edges, and representation $r \in S$ is given by a function $h(b, m, n, r)$ (which we assume is given, Section 4.3 will outline how we learn this and other functions).

Furthermore, we define a function $\mathbf{h}(\mathbf{b}, m, n, r)$, where $\mathbf{b} \in \mathbb{N}^l$ is a vector of basic operation counts, and $\mathbf{h}$ is the time to execute $\mathbf{b}$ (i.e. the linear combinations of functions h with appropriate parameters; where here we are not considering timing related to concurrent execution). Note that an operation initiated on router $\mathcal{R}_i$ may have to initiate calls to other routers in order to complete the operation. For example, since vertices are possibly duplicated, a "traverse neighbors" operation may require calls to other routers to check if those vertices exist there too, and get neighbors of such duplicates also. Let $c_{ij}(x_a, |x_a|) \in \mathbb{N}^l$ denote the number of

calls of basic operations $\mathcal{B}$ initiated on router $\mathcal{R}_j$ when the operation is initiated on router $\mathcal{R}_i$. We separately include the size of the input (e.g. number of vertices), $|x_a|$ involved in the operation for reasons that will become clear shortly. The approximate execution time $T(x_a,|x_a|,i,R)$ for an operation x_a initiated on router $\mathcal{R}_i$ is then given by

$$T(x_a,|x_a|,i,R) = \mathbf{h}(c_{ii}(x_a,|x_a|),|V_i|,|P_i|,R_i) + \sum_{j\neq i}\mathbf{h}(c_{ij}(x_a,|x_a|),V_j,|P_j|,R_j). \quad (1)$$

However, in practice jobs are sequences of operations, where the number of relevant graph objects for a job can depend upon its parents. For example, consider a job $\mathbf{p}:=\{x_1,x_2\}$ with $\pi(x_2)=x_1$ on a directed weighted graph G with partition P, where x_a is an operation "traverse edges from current vertex set if $w\geq\overline{w}$". For x_1, we assume that we have some initial vertex $z\in V_i$ as the current vertex set. However, we do not know in advance the size of the vertex set to which x_2 will apply. This means that in addition to knowing how many basic operations (and their type) $\mathbf{b}$ are required to execute a given operation, we also need a count of how many graph objects (e.g. vertices or edges) are filtered through by parent operations. We define $f_j(x_a)\in\mathbb{N}$ as a function that takes an operation and returns a count of the graph objects within partition part j that survived the filtering of the parent operations $\pi(x_a)$. Then the execution time of job $\mathbf{p}$ is given by

$$T(\mathbf{p},i,R) = T(x_1,|x_1|,i,R) + \sum_j T(x_2,f_j(x_2),j,R), \quad (2)$$

where the first term corresponds to execution x_1 on the initial router, and the sum represents running the subsequent operation on the "filtered" nodes. We have overloaded the notation T to correspond to runtimes for either jobs or operations, with the meaning clear from context. Note that $c_{ij}(x_a,1)\cdot|x_a|\neq c_{ij}(x_a,|x_a|)$ because there may be duplicates (e.g. multiple edges pointing to the same vertex) that could otherwise cause a vast overestimation (e.g. consider a fully connected graph) in the number of objects for the next operation, and therefore a large error in time estimation. It is easy to see how this approach extends to a much more general job $\mathbf{x}$ as

$$T(\mathbf{x},i,R) = \sum_{x_a\in\mathbf{x}}\sum_j T(x_a,f_j(x_a),j,R), \quad (3)$$

where we assume $f_i(x_1)$ is some known set, and $f_j(x_1)=\phi\ \forall i\neq j$. With this execution time function, we therefore wish to solve the following optimization problem:

$$\min_{\{R=(R_1,\ldots,R_k):R_i\in S\}} T(\mathbf{x},i,R) = \min_{\{R=(R_1,\ldots,R_k):R_i\in S\}}\sum_{x_a\in\mathbf{x}}\sum_j T(x_a,f_j(x_a),j,R). \quad (4)$$

The Problem (4) is a very complex combinatorial problem (as there are exponentially many, $k^{|S|}$, possible choices of the router representation vector R), where the representation of any given router i can influence the performance of operations sent to different routers j. Thus the choice of router representations must be made globally, with no opportunity for parallelization. However, we observe in Problem (4) that we have an important opportunity, by virtue of having established total counts of basic operations and their router occurrence locations.

Consider that the number of routers, and the length of the job $\mathbf{x}$ is finite. We can therefore interchange the order of summation on the RHS. Also, each complex operation ultimately results in a set of basic operations distributed across the routers, and that *we know this distribution of basic operations before solving the optimization problem* because it is not dependent upon representation because of how complex and basic operations are defined. This means, that supplied with appropriate c,f functions, we can rewrite the job $\mathbf{x}$ as a job $\mathbf{y}$, such that $\mathbf{y}$ consists only of basic operations (i.e. executing $\mathbf{y}$ executes $\mathbf{x}$). Furthermore, we can group the basic operations by router, such that for $\mathbf{y} = \{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_k\}$, $\mathbf{y}_i$ is the set of all basic operations for router $\mathcal{R}_i$. That is, we have the following very significant result:

$$\underset{R=(R_1,...,R_k):R_i \in S\}}{\operatorname{argmin}} \sum_{x_a \in \mathbf{x}} \sum_{j} T(x_a, f_j(x_a), j, R) = \bigcup_{i=1,...,k} \underset{R_i \in S}{\operatorname{argmin}} \boldsymbol{h}(\boldsymbol{y}_i, |V_i|, |P_i|, R_i), \tag{5}$$

where we are abusing notation in assuming $\mathbf{y}_i$ also has an appropriate vector form of counts of basic operations on $\mathcal{R}_i$. There are two key points to Problem (5). The first is that we have successfully decomposed a very complex interdependent problem with exponentially many possible solutions, to a linear number (in the number of routers) of problems with a linear number of solutions (in the number of representations), a vast reduction in problem complexity. Furthermore, these optimization problems are local to each partition part. This means that each router can locally solve the simple problem associated with itself (potentially concurrently), and these locally optimal solutions together combine to form the *globally* optimal solution of Problem (4).

4.3 LEARNING FILTERING, COUNTING, AND TIMING

In practice, we do not know the c_{ij}, f_j, and h functions unless we run the operations (which is contrary to our intent of choosing representations before execution). Instead, we seek to learn approximation functions $\overline{c}_{ij}$, $\overline{f}_j$, and $\overline{h}$. To learn these functions, we send sample operations to *each router* (note the contrast to our learning of basic operation runtimes, which is done as a global precomputation) and learn based upon the average responses of each router. In job $\mathbf{p}$, different routers may have different vertex degree (affecting basic operation counts and $\overline{c}_{ij}$ because edges must be checked, even if they do not satisfy the condition to be passed to the next operation), and different edge weight distributions (affecting $\overline{f}_j$ and the number of objects passed to the next operation).

Unlike in distributed graph computing systems, where a job might represent a call to compute PageRank etc. over the entire graph, we explicitly identify the database functionality of the SmartGraph as the reason to define *read-only* jobs as queries. We wish to have a range of operations that is sufficiently flexible such that they can be composed into more complex graph operations. This means that their combination should be sufficiently expressive. In the extensive literature on graph query languages [19]–[21], the expressiveness of query formulations has been a central area of research, with expressiveness often traded off against complexity and efficiency. In practice, industrial graph query languages like Cypher and Gremlin have not been theoretically analyzed to any significant degree due to the complexity facilitated by the wide range of possible query operations permitted [22].

Query methodologies typically fall into two categories; path queries [19], and pattern matching [23]. For purposes of demonstration within this paper, we choose to use a path query approach rather than a pattern matching approach. We choose this approach because it is clearly suitable for the methodology outlined in Section 4.2. Consider that a path query typically has a form $I \overset{cond.}{\rightarrow} J$ where we are seeking paths from graph object set I to (potentially unknown) graph object set J that must satisfy conditions $cond.$ along the path. There is an extensive literature on graph query algebras, and we refer the reader to [20] for a recent survey on the topic. For our purposes, we define path queries by chaining single hops with conditions on edges and/or vertices. Observe that this maps very neatly into the functions c_{ij}, f_j, and h. We have some path query that is decomposed into a series of stages, and conditions upon those stages. Each stage requires observing a certain number of graph objects (related to c_{ij}), only some of which survive to the next stage (related to f_j), and these are internally composed of basic graph operations (related to h, see Table 1) with timing dependent upon representation.

In contrast, we define *write-only* jobs as operations that modify the structure or properties of the graph. Indeed, the existence of properties within the graph database strongly differentiates this problem, even from the existing attempts at data structure optimization, as they focus entirely on graph structure [10], [11], [24]. The existence of properties is critical to our methodology, as it is not possible to ignore the schema of the graph when profiling simple operations as in existing works (e.g. profiling a GetVertex() operation is not sufficient, since the schema of the vertex added will have a large impact on performance). We leave support for write-operations to future work, in this version of the representation problem we focus on read operations as graph-querying, a fundamental read operation, is so elemental to the everyday usage of graph databases.

In-line with our view of routers as artificially intelligent structures, we learn approximations to $\overline{c}_{ij}$ and $\overline{f}_j$ independently at each router. That is, each router generates and executes sample queries according to the routers schema, and stores the resulting models for solving a problem akin to Problem (5), where we make the substitutions $c_{ij} \leftarrow \overline{c}_{ij}, f_j \leftarrow \overline{f}_j,$ and $h \leftarrow \overline{h}$. Thus we are using average router performance as a model for these functions, something that is facilitated at finer grains as we increase the number of routers (recalling that it is the router abstraction and the implementation of routers without go-routines that allows us to fathom router numbers in the tens of thousands, which is greatly distinct from the relatively few shards used in typical thread-focused graph computing).

Table 1. Examples of basic and complex operations

Basic Operation	Complex Operation
AddInternalEdge	AddExternalEdge
AddVertex	GetSharedVertices
RemoveEdge	ExploreNeighborsOutsideRouter
RemoveVertex	FilterEdges
GetVertex	FilterVertices
GetInternalEdge	RoutersConnected
VertexExists	
InternalEdgeExists	
UpdateInternalEdgeProperties	
UpdateVertexProperties	
ExploreNeighborsInRouter	

5. EXPERIMENTAL RESULTS

We use random forests [25] to learn each of the $\overline{c}_{ij}, \overline{f}_j, \overline{h}$ approximations owing to their ease of use combined with their ability to capture non-linear interactions. To demonstrate the efficacy of the method, we construct an artificial graph system consisting of four vertex types; Person, Product, Location, and Mail (with 300,400,500, and 600 vertices of each type respectively). We split the graph by vertex type, such that a given router primarily contains a single vertex type. As we allow edges both within the routers and between them, each router also contains a number of duplicated vertex types copied from other routers due to external edges. We constructed the graph using the Erdos-Renyi model [26] such that average degree is given as in Table 2. We note here that we do not expect our method to require a graph substantially similar to the graph presented here-in, or even to partition by vertex type as we have done. Instead, this example was generated for ease of understanding and reproducibility. We have similar designed the average between router degree to be much smaller than the within-router degree. This is because methods of graph partitioning typically involving minimize communication between partitions [9], and so we believe this a reasonable choice (note that this work does not deal with the question of graph partitioning, since it would need to be solved jointly with the assignment problem; such investigation is left to future work).

In terms of edge properties, we add two properties to each edge; a bivariate Gaussian with correlation ρ chosen $\sim \mathcal{U}[0, 1]$ for each edge type. We wish to demonstrate multiple properties in this early work in order to demonstrate that queries that interact with multiple properties are not an inconvenience for the described method.

5.1 BASIC OPERATIONS

For the purposes of learning basic operations, we simply run many instances of the basic operation on each edge and vertex type described in the schema, for varying sizes of the graph (that is; the only input available is the total number of vertices and edges in a graph). In this instance, we are not training specifically on the graph described in Table 2, merely graphs of

Table 2. Average out-degree for each vertex combination

	Person	Product	Location	Mail
Person	32	0.08	0	0.06
Product	0.1	26	0.12	0.04
Location	0.01	0.02	255	0.01
Mail	5	3	8	100

different sizes with the same vertex and edge types. For this reason, the learning of $\overline{h}$ functions, which we do independently for each representation type (RDF and MAL in our example), can be performed independently of any particular graph, only depending upon the schema. This indicates that if we expect multiple graphs with the same schema, we need only learn our approximations $\overline{h}$ once. In contrast, the methods described below require training on each individual graph.

5.2 COMPLEX TO BASIC COUNT MAPPING

In order to learn $\overline{c}_{ij}$, we generate a number (up to 10,000) of single-step test queries. These queries are of various types (e.g. filter a list of edges, filter in-bound from a list of vertices, filter out-bound from a list of vertices etc). In order to ensure that our methodology can be applied more broadly, we generate these test queries by randomly sampling vertices and edges from each router (the number of which is chosen uniformly from zero to the number of edges/vertices in the router). We then generate test queries randomly designating a property on the sampled graph object (in this paper, we only consider properties on edges), such that we are searching for edges that are greater than or less than the property at the sampled edge. Clearly this technique will generate queries that span the range of allowable property values, without any need to know in advance the distribution of any given property on the graph. We tag each query with a unique identifier, and track that identifier as results in the execution of basic operations (that is, for any given complex operation, we know exactly where basic operations occur, which operations they are, and how many of them occur).

We then teach a random forest to learn this mapping, such that test input includes the number of vertices or edges we wish to start the query from. In addition, we include the number of vertices shared between the source router (i.e. the router where the query is first initiated) and the other routers. We found that the inclusion of the latter was key to obtaining random forest models with very high accuracy. We observe that the random forest models quickly learn to become very accurate. We note that this is not a case of overfitting; all experiments used half of the samples for training, and half for evaluation. Indeed, the nature of our sampling methodology means that the sample space of possible queries is extraordinarily large. With $R^2 \to 1$ for many mapping functions, we observe the power of local learning, where the models quickly pick up on router characteristics.

5.3 LEARNING FILTERING

The method for learning filtering (that is, the number of objects that will survive a query) is very similar to that described above. However, here we explicitly use the parameter values of the queries as inputs to our random forest models. In addition to the relationships between the complex and basic operation tags, the system also keeps track of how many graph objects survive each filtering operation. As in the learning of $\overline{c}_{ij}$, we quickly obtain $R^2 \to 1$. Of course, we expect that the difficulty of learning the filtering mapping will increase as the number of variables that we perform filtering on increases; future work will investigate this potential issue. Nonetheless, as demonstrated by Figure 3, in this example the random forests are capable of learning the filtering mapping approximation to a very high degree of accuracy, given sufficient samples.

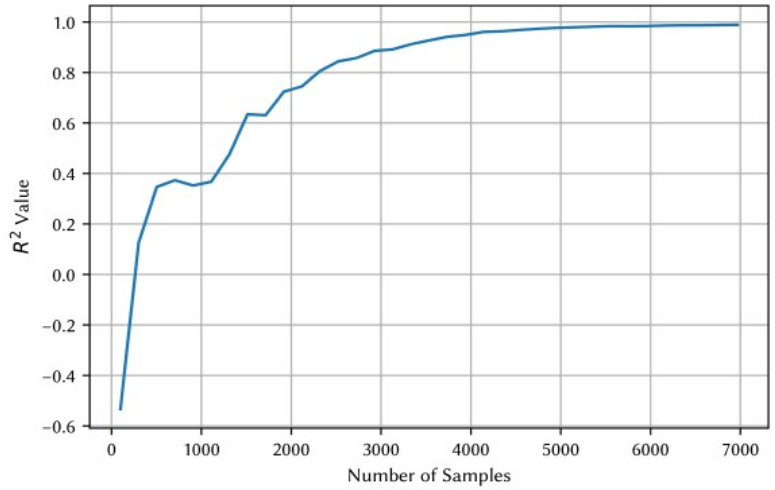

Figure 3. R^2 for learning vertex output and router locations given list of vertices and filtering conditions

5.4 THE REPRESENTATION PROBLEM

Finally we use all of the above methods to solve the representation problem. This problem involved sending 100 random two-step queries (that is, each query had an output that had to be fed into the next stage) to each of the four routers (as described by Table 2). We ran the queries in each of the sixteen possible sets of representation choices (in order to demonstrate the fully optimal solution), and then fed the query specifications to our learned models of $\overline{c}_{ij}, \overline{f}_j, \overline{h}$. We observed very encouraging results (as seen in Figure 3); using our machine-learning models gave us representations that on average were only ~14% slower than optimal, with the average router representation being more than 13 times slower than the optimal. As optimal representations run the gamut from all-MAL and all-RDF to everything in between, the value of an approach that gets a near optimal solution is particularly important given that the consequences of incorrect assignment can be so drastic. Note that finding the true optimal solution requires running the set of jobs with every possible combination of router assignments. As mentioned previously, this requires an exponential number of router representation tests. As the size of the jobs increases, it

becomes infeasible to look for an optimal representation through brute force (and doing so is redundant anyway, as completing the job is the purpose of the execution on the database). In contrast, the SmartGraph method will scale with the machine learning models used to learn $\overline{c}_{ij}, \overline{f}_j, \overline{h}$, and in many instances will not need to be retrained at all (e.g. if we keep the same graph and move from two-step to three-step traversal queries).

6. DISCUSSION AND FUTURE WORK

In this work we demonstrated a new methodology for constructing graph databases, and used the representation problem to demonstrate how giving artificial intelligence to "router-like" subgraphs can be used to solve highly complex problems. We showed that the representation problem is an exponentially large combinatorial optimization problem, but that it can be solved to near-optimality by having each router learn machine-learning model representations of

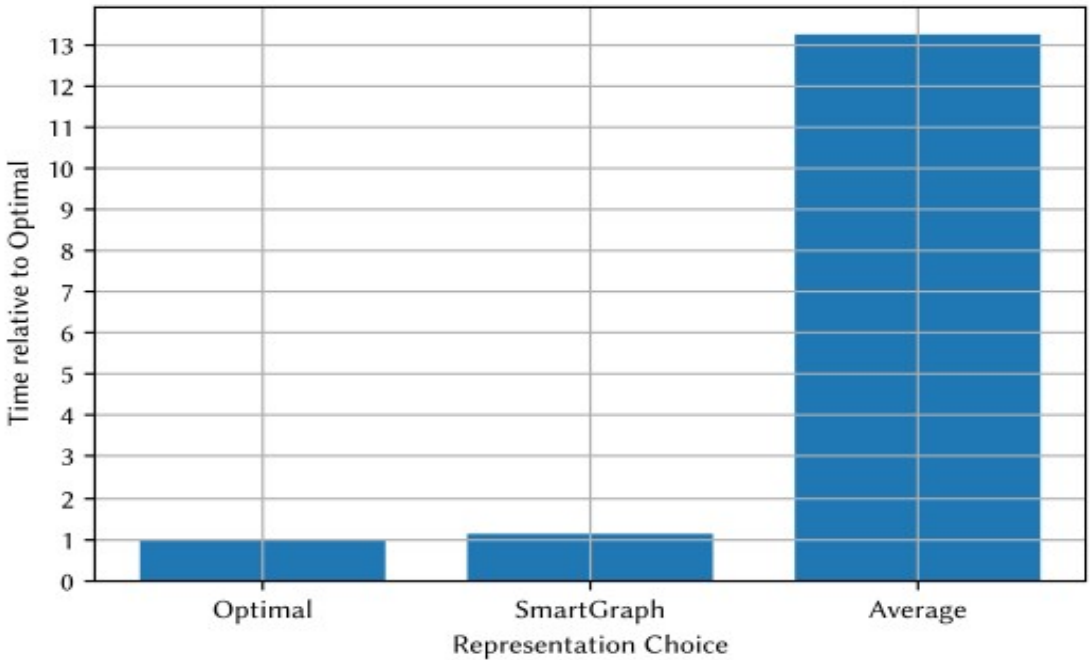

Figure 4. Representation Problem on our Test Cases

themselves and their neighboring routers. There is clearly more to explore in the representation problem alone. For example, solving the problem on real-world graphs (and investigating if scale-free properties are problematic for the approach), solving the problem with the inclusion of write methods, investigating how changing the number and size of routers influences the learning capability of each router, and exploring joint-learning with the graph partitioning problem. In addition, the SmartGraph itself is rife with potential for further research. For example, in investigating how the routers can facilitate completely asynchronous querying capabilities, the real-world performance metrics of SmartGraphs with an enormous number of routers, and more. By adding artificial intelligence at the "edge" of graph database technology, we increase the computing capabilities of the graph database, and continue to muddle the space between graph computing and graph databases.

REFERENCES

[1] R. R. McCune, T. Weninger, and G. Madey, “Thinking Like a Vertex,” ACM Comput. Surv., 2015.

[2] G. Malewicz et al., “Pregel,” Proc. 28th ACM Symp. Princ. Distrib. Comput. - Pod. ’09, p. 6, 2009.

[3] L. G. Valiant, “A Bridging Model for Parallel Computation,” Commun. ACM, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[4] V. Kalavri, S. Ewen, K. Tzoumas, V. Vlassov, V. Markl, and S. Haridi, “Asymmetry in Large-Scale Graph Analysis, Explained,” in Workshop on Graph Data Management Experiences and Systems, 2014.

[5] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, “From ‘think like a vertex’ to ‘think like a graph,’” Proc. VLDB Endow., 2013.

[6] Y. Simmhan et al., “GoFFish: A sub-graph centric framework for large-scale graph analytics,” in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2014, vol. 8632 LNCS, pp. 451–462.

[7] D. Yan, J. Cheng, Y. Lu, and W. Ng, “Blogel: A Block-centric Framework for Distributed Computation on Real-world Graphs,” in Proc. VLDB Endow., 2014.

[8] A. Donovan and B. W. Kernighan, The Go programming language. Addison-Wesley Professional, 2015.

[9] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, “Recent Advances in Graph Partitioning,” pp. 1–37.

[10] B. Schiller, C. Deusser, J. Castrillon, and T. Strufe, “Compile- and run-time approaches for the selection of efficient data structures for dynamic graph analysis,” Appl. Netw. Sci., pp. 1–22, 2016.

[11] A. Kusum, I. Neamtiu, and R. Gupta, “Safe and flexible adaptation via alternate data structure representations,” in Proceedings of the 25th International Conference on Compiler Construction - CC 2016, 2016.

[12] C. Jung, S. Rus, B. Railing, N. Clark, and S. Pande, “Brainy: effective selection of data structures,” in PLDI ’11:Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011.

[13] O. Kennedy and L. Ziarek, “Just-In-Time Data Structures,” in Onward!, 2015.

[14] R. S. Xin, J. E. Gonzalez, M. J. Franklin, I. Stoica, and E. AMPLab, “GraphX: A Resilient Distributed Graph System on Spark,” First Int. Work. Graph Data Manag. Exp. Syst., p. 2, 2013.

[15] A. Dave, A. Jindal, L. Li, R. Xin, J. Gonzalez, and M. Zaharia, “GraphFrames: An Integrated API for Mixing Graph and Relational Queries,” in Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems SE - GRADES ’16, 2016.

[16] J. Webber, “A Programmatic Introduction to Neo4J,” Proc. 3rd Annu. Conf. Syst. Program. Appl. Softw. Humanit., pp. 217–218, 2012.

[17] O. Green and D. A. Bader, "cuSTINGER: Supporting dynamic graph algorithms for GPUs," 2016 IEEE High Perform. Extrem. Comput. Conf. HPEC 2016, 2016.

[18] S. Che, B. M. Beckmann, and S. K. Reinhardt, "BelRed: Constructing GPGPU graph applications with software building blocks," 2014 IEEE High Perform. Extrem. Comput. Conf. HPEC 2014, 2014.

[19] P. Barceló, L. Libkin, A. W. Lin, and P. T. Wood, "Expressive Languages for Path Queries over Graph-Structured Data," ACM Trans. Database Syst., vol. 37, no. 4, p. 31:1--31:46, 2012.

[20] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, "Foundations of modern query languages for graph databases," ACM Comput. Surv., vol. 50, no. 5, p. 68, 2017.

[21] L. Libkin, W. Martens, and D. Vrgoč, "Querying Graphs with Data," J. ACM, 2016.

[22] F. Holzschuher and R. Peinl, "Performance of graph query languages: comparison of Cypher, Gremlin and native access in Neo4j," 16th Int. Conf. Extending Database Technol. EDBT' 13, no. March 2013, pp. 195–204, 2013.

[23] B. Gallagher, "Matching Structure and Semantics : A Survey on Graph-Based Pattern Matching," AAAI FS, 2006.

[24] B. Schiller, J. Castrillon, and T. Strufe, "Efficient data structures for dynamic graph analysis," in 11th International Conference on Signal-Image Technology & Internet-Based Systems, 2015.

[25] L. Breiman, "Random forests," Mach. Learn., vol. 45, no. 1, pp. 5–32, 2001.

[26] P. Erdos and A. Rényi, "On the evolution of random graphs," Publ. Math. Inst. Hung. Acad. Sci, vol. 5, no. 1, pp. 17–60, 1960.

AUTHORS

Hal Cooper (left) is a doctoral candidate in the Department of Industrial Engineering and Operations Research at Columbia University, where he is advised by Professor Garud Iyengar (right),who also serves as the Department Chair. Ching-Yung Lin (bottom) is an Adjunct Professor of the Department of Electrical Engineering at Columbia University. Ching-Yung is also the CEO of Graphen, Inc., and a former Chief Scientist for Graph Computing at IBM.