Subgraph Pattern Matching: Models, Algorithms, and Techniques

by

Arash Jalal Zadeh Fard

(Under the Direction of John A. Miller and Lakshmish Ramaswamy)

Abstract

Subgraph pattern matching is a fundamental operation for many applications, and it is exhaustively studied in its classical forms. Nevertheless, there are newly emerging applications, like analyzing hyperlinks of the web graph and analyzing associations in a social network, that need to process massive graphs in a timely manner. Regarding the extremely large size of these graphs and knowledge they represent, not only new computing platforms are needed, but also old models and algorithms should be revised. In recent years, a few pattern matching models have been introduced that can promise a new avenue for pattern matching research on extremely massive graphs. In this research, we study a family of subgraph pattern matching models called graph simulation, and propose two new models, called *strict and tight simulation*, to increase their efficiency while preserving the quality of their results. Moreover, we propose a new set of conditions, namely *cardinality restriction*, that can improve the expressiveness of most models in this family.

Several graph processing frameworks like Pregel have recently sought to harness shared nothing clusters for processing massive graphs through a vertex-centric, Bulk Synchronous Parallel (BSP) programming model. However, developing scalable and efficient BSP-based algorithms for pattern matching is very challenging on these frameworks because this prob-

lem does not naturally align with a vertex-centric programming paradigm. We design and implement novel distributed algorithms based on the vertex-centric programming paradigm for efficient subgraph pattern matching. Our algorithms are fine tuned to consider the challenges of pattern matching on massive data graphs. Furthermore, we present an extensive set of experiments involving massive graphs (millions of vertices and billions of edges) to study the effects of various parameters on the scalability and performance of the proposed algorithms.

Regarding the fact that pattern matching can be considered as an important type of queries for a graph database- either centralized or distributed- we study the problem of pattern containment and caching techniques specified for subgraph pattern matching. The proposed caching technique works based on the *tight simulation* model. Nevertheless, it is also possible to use it for subgraph isomorphic queries. We identify the main challenges of such a system, and our experiments show the effectiveness of the proposed solutions.

INDEX WORDS: Query graphs, Distributed algorithms, Graph simulation, Subgraph pattern matching, Catching techniques, Big data

Subgraph Pattern Matching: Models, Algorithms, and Techniques

by

Arash Jalal Zadeh Fard

B.S., Iranian University of Science and Technology, 1994

M.S., Amirkabir University of Technology, 2000

A Dissertation Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2014

Subgraph Pattern Matching:
Models, Algorithms, and Techniques

by

Arash Jalal Zadeh Fard

Approved:

Major Professors:	John A. Miller Lakshmish Ramaswamy
Committee:	Hamid R. Arabnia Krzysztof J. Kochut

Electronic Version Approved:

Julie Coffield
Interim Dean of the Graduate School
The University of Georgia
August 2014

Subgraph Pattern Matching: Models, Algorithms, and Techniques

Arash Jalal Zadeh Fard

July 21, 2014

Acknowledgments

I would like to thank my advisor, Dr. Miller, and my co-advisor, Dr. Ramaswamy, for their crucial research guidance, without which this dissertation would not have been possible. Also, I would like to thank the other professors in my advising committee, Dr. Arabnia and Dr. Kochut, for their support and valuable advice during the years I was trying to make progress in my research. I am also grateful to all the faculty and staff members of the computer science department in UGA. I will be always thankful for the opportunity that I was given in this department to learn and to grow.

Furthermore, I appreciate incredible support of my wife, Dr. Sahar Sadjadian, during my PhD program. I am also grateful to my parents, Fereshteh and Ali-Asghar, for all their help and support not only during this adventure, but also through my whole life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are fundamental to representing and analyzing relationships in many diverse domains. Although graph storage, querying and mining have been extensively studied over the past several decades, massive scales of modern application domains such as social networks, genomics and the World Wide Web have reignited interest in highly scalable graph processing platforms and algorithms [17, 55, 67].

Graph pattern matching forms a uniquely important class of graph queries. In simple words, the purpose of graph matching is finding correspondence between vertices and edges of two graphs. A broad set of related problems are studied under the general topic of graph matching. Well-known homomorphism and isomorphism matching problems are traditional models when similarity of two graphs is evaluated. Decision problem associated to graph homomorphism is NP-complete [10]. Graph isomorphism is neither known to be NP-complete nor to be tractable. Nevertheless, regarding its importance in computational complexity theory, a new complexity class is defined as GI-complete which contains any problem that can be reduced to graph isomorphism in polynomial time [16]. A related problem is finding the maximum common subgraph-isomorphism (MCS) which is also NP-hard [34].

In the category of subgraph pattern matching, which is more related to our work, the goal is finding similar instances of a smaller graph, usually called pattern or query graph, in a bigger graph, usually called data graph. Traditionally, subgraph isomorphism has been the most famous subgraph matching model, and its associated decision problem is well-known to be NP-complete. Despite its intractability, subgraph isomorphism has many real-world applications; hence, during the last decades many researchers have actively tried to design more efficient algorithms for finding all exact subgraph pattern matches. The most famous exact subgraph isomorphic algorithms are Ullmann [70] and VF2 [21]. Several algorithms have also been introduced in the recent years, like GraphQL [37], QuickSI [63], SPath [78], GADDI [76], TurboIso [36], and DualIso [62].

In general, subgraph pattern matching algorithms, can be classified as *exact* or *inexact* algorithms. *Inexact pattern matching algorithms* have been considered interesting mainly because finding inexact similar matches is more useful when the data graph is noisy. For inexact matching, some metrics are defined to measure similarity of the results to the pattern. Another, classification for subgraph pattern matching algorithms is being *optimal* or *approximate*. An optimal algorithm guarantees to find a correct solution; e.g., for exact matching all the correct matches, and for inexact matching the closest match or a correctly ranked list of matches. In comparison, an approximate algorithm does not guarantee to find a correct solution; e.g., for exact matching, some, but not all the matches; and for inexact matching, a close match, but not the closest. Good surveys for subgraph pattern matching are [20] and [33]. Moreover, in [48] Lee et al. present a recent survey about subgraph isomorphism algorithms.

Several new subgraph pattern matching models are introduced during recent years in order to address the limits of the traditional models, like p-homomorphism [26], graph simulation [40], and strong simulation [52]. Graph simulation provides a practical alternative to subgraph isomorphism by relaxing its stringent matching conditions. This allows matches to

be found in polynomial time. Furthermore, some researchers [17, 25, 52, 30] argue that graph simulation is more appropriate than subgraph isomorphism for modern applications such as social network analysis because it yields matches that are conceptually more intuitive.

In recent years, several variants of *graph simulation* are introduced like *dual simulation* and *strong simulation*. These models form a spectrum with respect to the stringency of the matching conditions – graph simulation is the least restrictive whereas strong simulation is the most restrictive. In other words, strong simulation is conceptually closest to subgraph isomorphism whereas graph simulation is the farthest. It is also noteworthy that, in general, the more restrictive the model, the less scalable the corresponding algorithm.

While polynomial time algorithms exist for graph, dual and strong simulations, they still do not scale to the massive graphs that characterize modern applications such as social networks. Therefore, a natural question is whether these algorithms can be parallelized and implemented efficiently on shared nothing clusters. In recent years, several graph processing frameworks have been proposed that enable programmers to easily and efficiently implement distributed graph algorithms on shared nothing clusters. These include Pregel [55], Giraph [4] and GPS [60] which are characterized by the BSP programming paradigm. In these platforms, the programmer expresses algorithms in terms of the actions performed by a generic vertex of the graph.

In recent years, researchers have designed vertex-centric algorithms for various graph problems. Surprisingly, to the best of our knowledge, there are no studies on vertex-centric approaches for graph simulation models. While there is some recent work on distributed basic graph simulation [54], the proposed algorithm is not based on the vertex-centric programming paradigm. Moreover, there is no distributed algorithm for dual or strong simulation.

We have introduced two new pattern matching models: *strict simulation* [32] and *tight simulation* [31]. They improve upon *strong simulation* [52], which was the state of the art pattern matching model in the graph simulation family in both in terms of efficiency and

quality of the result. We have also introduced a new condition, called *cardinality restriction*, that can improve the expressiveness and the quality of the results for all these models.

Furthermore, we have designed and implemented vertex-centric distributed algorithms for different pattern matching models in the graph simulation family. We have comprehensively studied their behaviors through an extensive set of experiments.

Although graph simulation models are tractable, they are still highly computation-intensive for massive graphs. Most existing subgraph pattern matching systems treat each incoming query completely independently from previous queries. We contend that reusing query results can yield significant performance benefits. Hence, we have studied the pattern containment, and have designed cache techniques specified for subgraph pattern matching. We have proved the correctness of our technique. Also, we examined the effectiveness of our techniques on real-life datasets.

The main contributions of this research can be summarized as follows:

- We identify the bottlenecks that severely limit the scalability and performance of strong simulation pattern matching model. Towards ameliorating these limitations, we propose two new graph simulation models called *strict simulation* and *tight simulation*. They are greatly faster, and surprisingly, we improve the quality of the result as well as efficiency.

- We introduce a new concept, called *cardinality restriction*, that can be applied to all the models in the graph simulation family to improve their expressiveness.

- We introduce novel vertex-centric distributed algorithms for five simulation models. Our algorithms include several techniques to mitigate performance bottlenecks.

- We present a detailed experimental study on the benefits and costs of the proposed distributed algorithms for various graph simulation models.

- We present a novel mechanism for reusing the result of a graph pattern matching query to answer some other new queries. This mechanism can be potentially used in a cache system to improve the response time of a query engine that runs based on the tight simulation model. This technique can be ultimately used for subgraph isomorphism as well because the results of tight simulation always contain the results of subgraph isomorphism.

- In order to design our caching technique and prove its correctness, we have investigated some of the properties of graph simulation models. Regarding the fact that these models enjoy polynomial algorithms, we believe the newly discovered properties can spur further research about using these models in the field of graph pattern matching.

- We present a detailed empirical study on the performance gain of reusing the results of subgraph pattern matching queries, and explore the affects of some parameters that can play a pivotal role in a basic cache system for this purpose.

Chapter 2

Background

In this section we present some background knowledge required for the understanding of our algorithms. First, we discuss the vertex-centric framework for distributed graph processing, and then we explain different models of pattern matching.

2.1 Vertex-centric Graph Processing

Although some researchers have used the MapReduce platform [22, 43], and especially its free implementation Hadoop [6], to analyze massive graphs, this platform is not well-suited for iterative algorithms including most graph algorithms. In [55] Google presented a new platform, called Pregel, specifically designed for running graph algorithms. This platform introduces a vertex-centric paradigm based on the BSP computing model.

Bulk Synchronous Parallel (BSP) was first proposed by Valiant [71] as a model of computation for parallel processing. Computation in this model is a series of *supersteps*. As it is shown in figure 2.1, each superstep contains three ordered stages: (1) concurrent computation, where different processes run concurrently, (2) communication where all processes exchange their messages, and (3) barrier synchronization in which every process waits for

others to reach the same state before going to the next superstep. In the vertex-centric programming model proposed in Pregel, each vertex of the data graph is a computing unit which can be conceptually mapped to a process in the BSP model. At the time of loading, the data graph is partitioned among several *workers*. Each worker is a real process which handles the computation for each vertex. Each vertex initially knows only about its own label and its outgoing edges. Then, vertices can exchange messages through successive supersteps to learn about each other or to accomplish a computing task. Within each superstep, vertices are executing the same predefined function. When a vertex believes that it has accomplished its tasks, it votes to halt and goes to inactive mode. A vertex remains inactive until it is triggered externally by a message from another vertex. When all vertices become inactive the algorithm terminates.

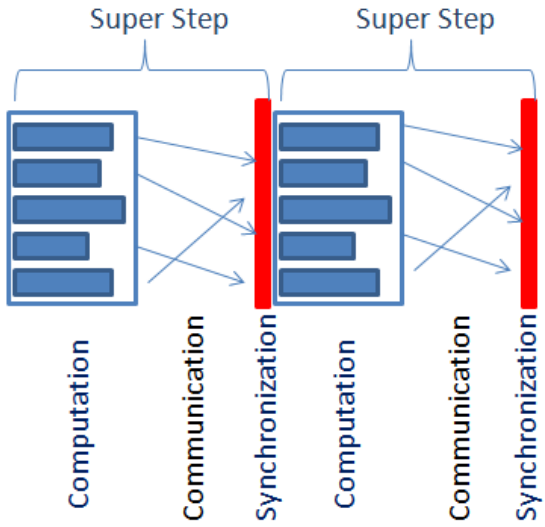


Figure 2.1: Bulk Synchronous Parallel (BSP) computation model

The vertex-centric approach is a pure message passing model in which users focus on a local action for each vertex of the data graph. Its usage of the BSP model makes it inherently free of deadlocks. Moreover, it can provide very high scalability, and by design it is well-suited for distributed implementation. A few other open source projects that follow

the same idea as Pregel have been introduced recently, namely GPS [60], Apache Giraph [4], Apache Hama [2], and Signal/Collect [66].

We have used GPS, which can be considered a free distribution of Pregel, to implement our distributed algorithms. GPS is written in Java and has an extended API to provide a type of global communication among vertices. It also supports the dynamic repartitioning of a data graph to balance the workload during computation.

2.2 Different Pattern Matching Models

The goal of a pattern matching algorithm is to find all the matches of a given graph, called a query graph, in an existing larger graph, called a data graph. To define it more formally, assume that there is a data graph $G(V, E, l)$, where V is the set of vertices, E is the set of edges, and l is a function that maps the vertices to their labels. Given a query graph $Q(V_q, E_q, l_q)$ the task is to find all subgraphs of G that match the query Q. $G'(V', E', l')$ is a subgraph of G if and only if (1) $V' \subseteq V$; (2) $E' \subseteq E$; and (3) $\forall u \in V' : l'(u) = l(u)$.

Here, we assume all vertices are labeled, all edges are directed, and there are no multiple edges. Without loss of generality, we also assume a query graph is a connected graph because the result of pattern matching for a disconnected query graph is equal to the union of the results for its connected components. In this paper, we use pattern and query graph interchangeably.

2.2.1 Subgraph Isomorphism

Subgraph isomorphism is the most famous model for pattern matching. It preserves all topological features of the query graph in the result subgraph. However, finding all subgraphs that are isomorphic to a query graph is an NP-hard problem in the general case. By definition, subgraph isomorphism describes a bijective mapping between a query graph $Q(V_q, E_q)$

and a subgraph of a data graph $G(V, E)$, denoted by $Q \trianglelefteq_{iso} G$. That is, assuming $G'(V', E')$ is a subgraph of G, graph Q will be subgraph isomorphic of G if there is a bijective function f from the vertices of Q to the vertices of G' such that (u, v) is an edge in Q if and only if $(f(u), f(v))$ is an edge in G' [70]. It should be noticed that function f ensures that u and $f(u)$ have the same labels.

2.2.2 Graph Simulation

Another model, graph simulation, permits faster algorithms by relaxing some restrictions on matches.

Definition 2.1 *Pattern $Q(V_q, E_q)$ matches data graph $G(V, E)$ via* graph simulation*, denoted by $Q \trianglelefteq_{sim} G$, if there is a binary relation $R \subseteq V_q \times V$ such that (1) if $(u, u') \in R$, then u and u' have the same label; (2) for every $u \in V_q$ there is a $u' \in V$ such that $(u, u') \in R$; (3) $\forall (u, u') \in R[(u, v) \in E_q \Rightarrow \exists v' \in V : (v, v') \in R \wedge (u', v') \in E]$.*

Intuitively, graph simulation only preserves the child relationships of each vertex. The result of pattern matching is a maximum match set of vertices. The *maximum match set*, $R_m \subseteq V_q \times V$, is the biggest relation set between Q and G with respect to $Q \trianglelefteq_{sim} G$. The *result match graph*, $G_r(V_r, E_r)$, as suggested in [52], is a subgraph of G that can represent R_m. By definition, G_r is a subgraph of G which satisfies these conditions: (1) $(u, u') \in R_m \Leftrightarrow u' \in V_r$; (2) $\forall (u, u'), (v, v') \in R_m \ [(u', v') \in E_r \Leftrightarrow (u, v) \in E_q]$.

A quadratic time algorithm for graph simulation was first proposed in [40] with applications to the refinement and verification of reactive systems. This model and its extensions have been studied especially in recent years [25, 54] because of their new applications in analysis of social networks [17].

2.2.3 Dual Simulation

Dual simulation improves on graph simulation by taking into account not only the children of a query node, but also the parents. That is, a vertex in a data graph becomes a dual match with a vertex in a query graph if and only if (1) it has the same label, (2) a subset of its children match all the children of its correspondent vertex in the query graph, (3) a subset of its parents also match all the parents of its correspondent vertex.

Definition 2.2 *Pattern $Q(V_q, E_q)$ matches data graph $G(V, E)$ via* dual simulation*, denoted by $Q \trianglelefteq^{D}_{sim} G$, if (1) $Q \trianglelefteq_{sim} G$ with a binary match relation $R_D \subseteq V_q \times V$, and (2) for every $(u, u') \in R_D$, if there is a $w \in V_q$ such that $(w, u) \in E_q$ then there exists a $w' \in V$ such that $(w, w') \in R_D$ and $(w', u') \in E$.*

In dual simulation as in graph simulation, we maintain the concept of a maximum match set and a result match graph. Here, the result match graph is also a single graph which might be connected or disconnected. In [52] a cubic algorithm for dual simulation is proposed.

2.2.4 Strong Simulation

Strong simulation adds a locality property to dual simulation. Shuai Ma et al. [52] introduce the concept of a ball to define locality. A ball b in $G(V, E)$, denoted by $\hat{G}[v, r]$, is a subgraph of G that contains all vertices not further than a specified radius r from a center $v \in V$; moreover, the ball contains all edges in G that connect these vertices (i.e., it is an induced connected subgraph).

In order to measure the distance between vertices of a graph, we consider the edges to be undirected. Therefore, given two vertices u and v in a connected graph, the distance from u to v is defined as the minimum number of edges in an undirected path from u to v. The diameter of a connected graph is then defined as the greatest distance between any pair of nodes in the graph.

Definition 2.3 *Pattern $Q(V_q, E_q)$ matches data graph $G(V, E)$ via* strong simulation*, denoted by $Q \trianglelefteq^S_{sim} G$, if there exists a vertex $v \in V$ such that (1) $Q \trianglelefteq^D_{sim} \hat{G}[v, d_Q]$ with maximum dual match set R^b_D in ball b where d_Q is the diameter of Q, and (2) v is member of at least one of the pairs in R^b_D. The connected part of the result match graph of each ball with respect to its R^b_D which contains v is called a* maximum perfect subgraph *of G with respect to Q.*

In contrast to the previous types of simulation, strong simulation may have multiple maximum perfect subgraphs (MaxPGs) as its result. As indicated by [52], although strong simulation preserves many of the topological characteristics of a pattern graph, its result can be computed in cubic time. Moreover, the number of MaxPGs is bounded by the number of vertices in a data graph, while subgraph isomorphism may have an exponential number of match subgraphs.

2.2.5 Comparing the result of the models

Figure 2.2 provides an example to show the difference in the results of the mentioned pattern matching models. This example is about finding a team of specialists on a social network like LinkedIn, inspired by an example in [54]. A vertex in this example represents a member, and a label represents a member's profession. A directed edge from a member a to a member b indicates that member a has endorsed member b. Because the query nodes here have distinct labels, we only specify data graph node ID's for a match rather than specifying the full match relation. (The relation set between pattern and data graph can be inferred from the labels.)

Given the pattern and the data graph, the maximum match set for graph simulation consists of all vertices except {2,13}. It is clear that vertices 1, 3, and 14 are not appropriate matches either, because the system analysts represented by vertices 3 and 14 are not endorsed by any database designers. Dual simulation removes these inappropriate ver-

tices; its maximum match set contains all vertices except {1,2,3,13,14}. However, there are still some vertices that do not provide very meaningful matches. For example, the cycle {15,16,17,18,19,20} creates a very big subgraph match, which is not desirable for such a small pattern. Applying strong simulation then shrinks the result match graph to a reasonable size; the result here is the set of vertices {4,5,6,7,8,9,10,11,12}. In contrast, there are two isomorphic subgraphs corresponding to these two set of vertices: {4,6,7,8} and {5,6,7,8}.

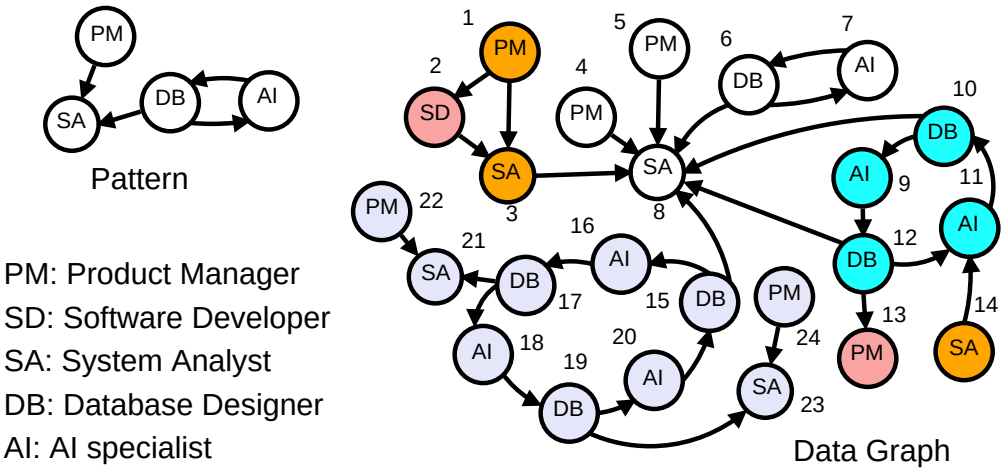


Figure 2.2: An example for different models

Chapter 3

Strict Simulation

3.1 Introducing Strict Simulation

Strict simulation is a novel modification of Strong simulation that not only substantially improves its performance, but also maintains a better quality of result because of its revised definition of locality. The locality restriction defined in strong simulation is the main reason for its long computation time because the size of balls can be potentially very big. Here, the size of a ball means the number of vertices that it contains.

Proportional to the diameter of Q and average degree of vertices in G, each ball in strong simulation can be fairly bulky. Furthermore, due to communication overhead during ball creation, the problem is exacerbated in distributed systems where a data graph is partitioned among different nodes. In order to mitigate this overhead, we introduced a pattern matching model, named strict simulation in [31].

3.2 Comparing Strict with Strong Simulation

Strict simulation is more scalable and preserves the important properties of strong simulation. It is shown in [52, 23] that: (1) if $Q \trianglelefteq^{G}_{iso}$, then $Q \trianglelefteq^{S}_{sim} G$; (2) if $Q \trianglelefteq^{S}_{sim} G$, then $Q \trianglelefteq^{D}_{sim} G$; and (3) if $Q \trianglelefteq^{D}_{sim} G$, then $Q \trianglelefteq^{G}_{sim}$. We show that strict simulation and another new model we introduce in the next chapter are not only more efficient than strong simulation, but also more stringent; i.e., their results are getting closer to subgraph isomorphism while they become computationally more efficient.

Figure 3.1a shows the flowchart of the centralized algorithm for strong simulation. In this algorithm, the match relation of dual simulation, R_D, is computed first. Then, a ball $\hat{G}_s[v, d_Q]$ is created for each vertex v of the data graph contained in R_D. Members of the ball are selected regardless of their membership in pairs of R_D. At the next step, the dual match relation is projected on each ball to compute the result of strong simulation. Finally, the maximum perfect subgraph (MaxPG) is extracted by constructing the match graph on each ball and finding the connected component containing the center.

In comparison, Figure 3.1b illustrates the centralized algorithm for strict simulation. The key difference between them is that in strict simulation the duality condition is enforced before the locality condition; i.e., balls are created from the dual result match graph, G_D, rather than from the original graph. As the balls in strict simulation are often significantly smaller, this seemingly minor difference between the two algorithms has a profoundly positive impact on running time.

There are the same numbers of balls in strict and strong simulation; however, the extracted MaxPG from each ball in strict simulation is always a subgraph of the result in strong simulation on a ball with the same center. Duplicate MaxPGs may be produced from different balls, and in the case of strict simulation a result MaxPG might be subgraph of another one. In a post-processing phase, the duplicate results are filtered, and only the smaller

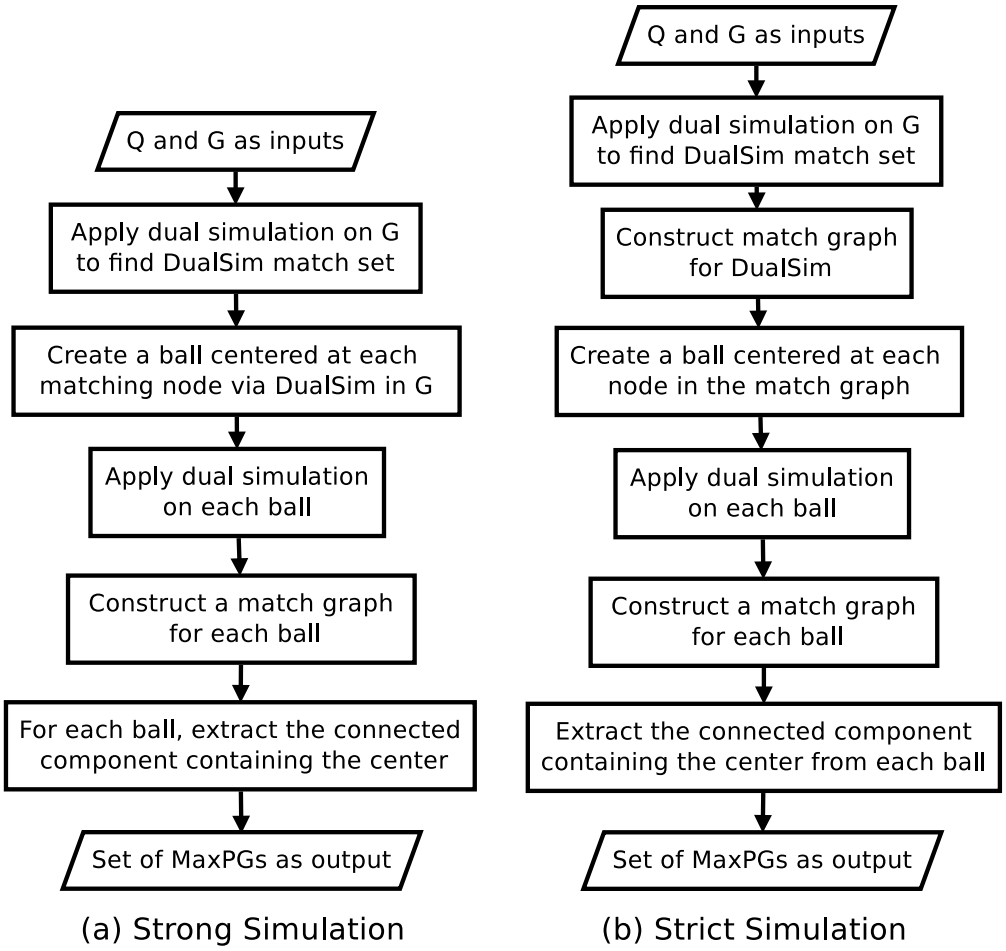


Figure 3.1: Comparing the algorithms of strong and strict simulation

result is kept when it is a subgraph of another result. Any possible isomorphic subgraph match of the query will be preserved in the result of strict simulation.

Figure 3.2a shows an example that highlights the difference between the results of these two types of simulation. In this example, all vertices except 9 will appear in the dual result match graph. For strong simulation, a ball centered at vertex 2 would contain all the vertices in the data graph. In contrast, because the dual match graph would not contain vertex 9 and some of the edges, the corresponding ball for strict simulation would only contain the set of vertices {1,2,3,4}. Therefore, the MaxPG resulting from strong simulation will contain

all vertices except 9, while the one resulting from strict simulation will contain only vertices 1, 2, and 3. One can verify that this is the only MaxPG which will result from any ball of strict simulation.

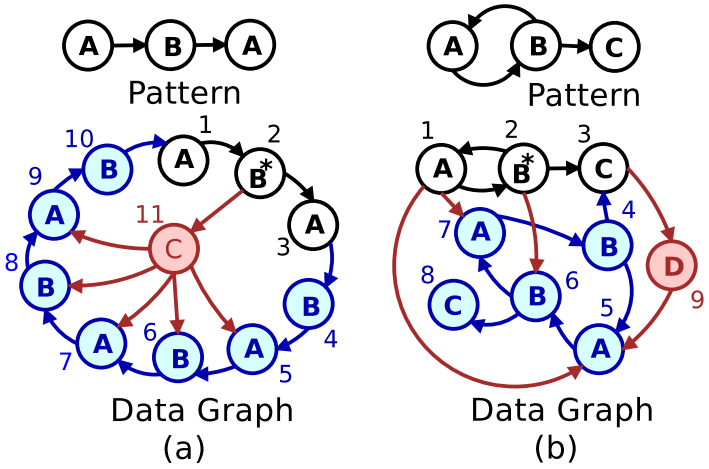


Figure 3.2: Comparing strict and strong simulation

Figure 3.2b illustrates another example. Here, one can verify that resulting MaxPG of strong simulation for a ball centered at vertex 2 will contain vertices 1 to 8. In comparison, the resulting MaxPG for strict simulation will contain the set of vertices {1,2,3}.

It is noteworthy that there is no bound on the number of vertices in a MaxPG resulted by strict simulation. Neither is there any bound on the diameter of MaxPGs with respect to the diameter of the query. Figure 3.3 displays an example where the query, Q, has 6 vertices and diameter $d_Q = 4$. The data graph, G, has been intentionally selected big to show how the size of subgraph result can be bigger than the query. All the vertices of G in this example are member of results match graph with respect to $Q \trianglelefteq^{D}_{sim} G$. There are three labels in Q and each on two vertices.

A color code is used to depict the relationship between the vertices in Q and G. The vertices with green bodies in Q are in dual match relation with the green vertices in G. It is the same for the blue vertices. When the vertex 12 in G is selected as the center of a

ball with radius 4, one can verify that all the other vertices except 4 and 5 will be in the ball. After applying dual simulation on the ball, all the vertices that are displayed with red borders will remain in the subgraph result. It means there are 28 vertices in the resulted MaxPG. Moreover, one can verify that the diameter of the MaxPG is 10, which is more than twice bigger than d_Q.

We have also defined a *post-processing phase* after filtering the balls in strict simulation. In this phase all MaxPGs are compared with each other, and any result that is supergraph to any other will be omitted from the set of MaxPGs. Filtering the subgraph results in this process makes it possible to keep only more stringent results.

Figure 3.4 present an example that shows how the post-processing phase will filter the results and makes the the total number of vertices in results of strict simulation smaller than the total number of vertices in strong simulation. The query, Q in this example has three vertices and its diameter is $d_Q = 2$.

All the vertices in the data graph, G, are in dual relation with the vertices in the Q. It can be observed that because of dense connectivity, all the subgraph results of strong simulation contain all the vertices; therefore, if the post-processing is applied a MaxPG with all the vertices will be left as the final result. In contrast the subgraph results generated by strict simulation have different number of vertices because the edges displayed in red are not part of result dual match graph. For example, the MaxPG that is generated from a ball centered at vertex 3 contains all the vertices, but the MaxPG produced from the ball centered at vertex 2 contains only vertices 1, 2, and 3. Here, only the smaller MaxPG with three vertices will remain after post-processing phase.

3.3 Properties of Strict Simulation

We formally define strict simulation as follows.

Definition 3.1 *Pattern $Q(V_q, E_q, l_q)$ matches data graph $G(V, E, l)$ via strict simulation, denoted by $Q \trianglelefteq^{\Sigma}_{sim} G$, if there exists a vertex $v \in V$ such that (1) $v \in V_D$ where $G_D(V_D, E_D, l_D)$ is the result match graph with respect to $Q \trianglelefteq^{D}_{sim} G$; (2) $Q \trianglelefteq^{D}_{sim} \hat{G}_D[v, d_Q]$ where $\hat{G}_D[v, d_Q]$ is a ball extracted from G_D, and d_Q is the diameter of Q; (3) v is a member of the result MaxPG.*

A MaxPG for strict simulation is defined the same as a MaxPG for strong simulation. Using the definition and the properties of dual simulation presented in [52], [23], properties of strict simulation can be proved as follows.

Theorem 3.1 *For any query graph Q and data graph G such that $Q \trianglelefteq^{\Sigma}_{sim} G$, there exists a unique set of maximum perfect subgraphs for Q and G.*

Proof: It is proved that the result of dual simulation is unique. Therefore, the balls extracted from its result match graph and consequently their result after applying dual filter would be unique. □

Proposition 3.1 *(1) If $Q \trianglelefteq^{G}_{iso}$, then $Q \trianglelefteq^{\Sigma}_{sim} G$. (2) if $Q \trianglelefteq^{\Sigma}_{sim} G$, then $Q \trianglelefteq^{S}_{sim} G$.*

Proof: (1) Any subgraph isomorphic match in G is also dual match to Q; therefore, it will appear in the result of dual match graph. Clearly, a ball with radius d_Q (diameter of Q), and centered on a vertex of this subgraph will contain its whole vertices. The subgraph later will be also in the result of dual filter on the ball. (2) G_D is a subgraph of G, and the distance between any pair of vertices in G_D is smaller than their distance in G. Therefore, any ball in strict simulation will be a subgraph of the corresponding ball in strong simulation with the same center. Consequently, when there is a subgraph result in the ball of strict simulation after applying dual filter, it will be also preserved in the result of strong simulation. □

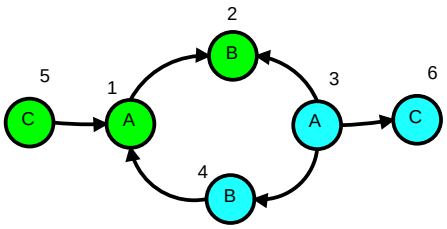
Proposition 3.2 *The number of maximum perfect subgraphs produced by* $Q \trianglelefteq^{\Sigma}_{sim} G$ *is bounded by the number of vertices in* G*.*

Proof: G_D is a subgraph of G, and the number of balls in strict simulation equals to the number of vertices in G_D. Moreover, not more than one MaxPG can be produced from each ball. Therefore, their total number is bounded by the number of vertices in G. □

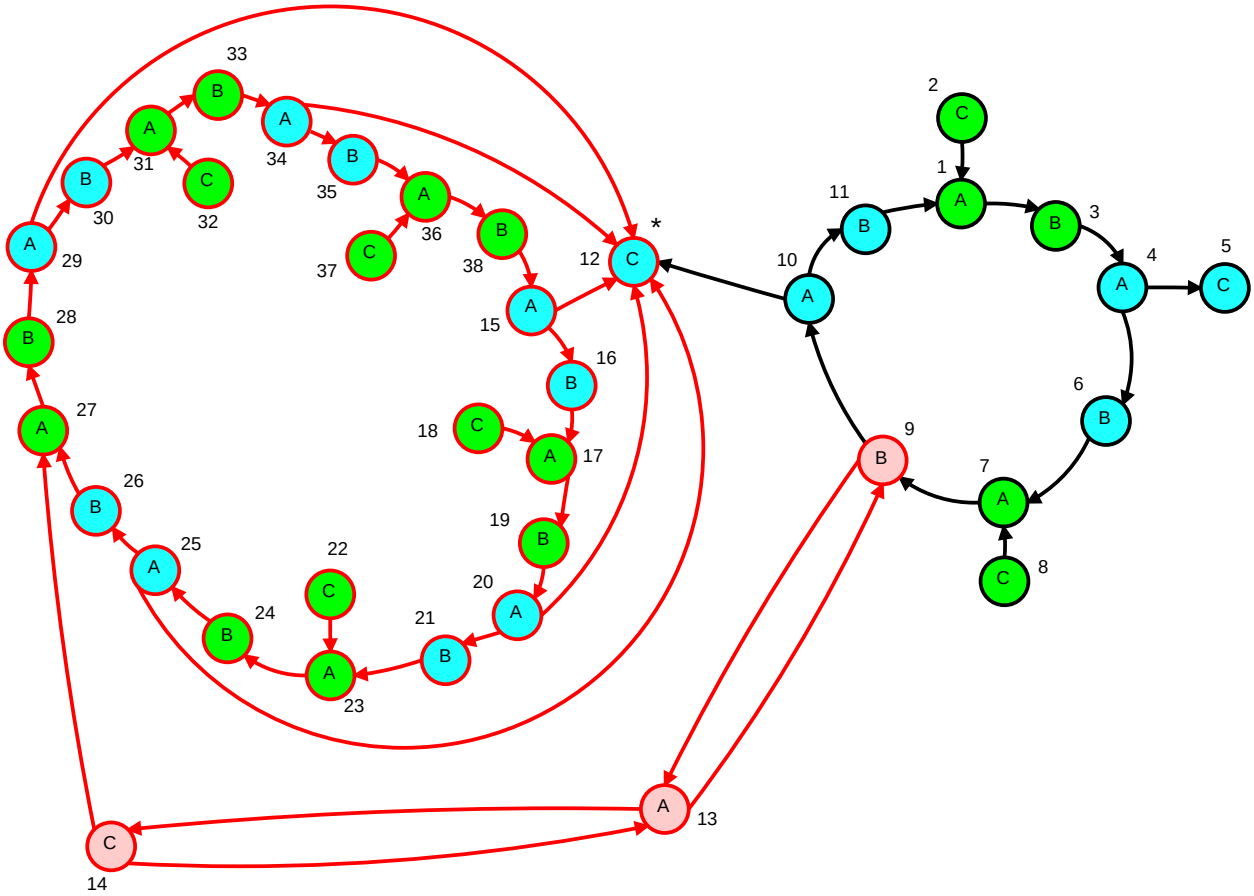
Theorem 3.1 ensures that strict simulation yields a unique solution for any query/data graph pair. Proposition 3.1 indicates that strict simulation is a more stringent notion than strong simulation, though still looser than subgraph isomorphism. Proposition 3.2 gives an upper bound on the number of possible matches for a query graph in a data graph. It should be noted that the number of matches in subgraph isomorphism can be exponential to the number of vertices in G. The following theorem also shows that the asymptotic time complexity of strict simulation is the same as that of strong and dual simulation.

Theorem 3.2 *For any query graph* Q *and data graph* G*, the time complexity for finding all maximum perfect subgraphs with respect to strict simulation is cubic.*

Proof: It is proved that time complexity for finding G_D is cubic. Moreover, it is proved that time complexity of strong simulation is cubic. Regarding the fact that each ball in strict simulation is a subgraph of its corresponding ball in strong simulation, time complexity of strict simulation is also cubic. □

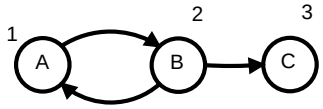


a) A query graph, Q

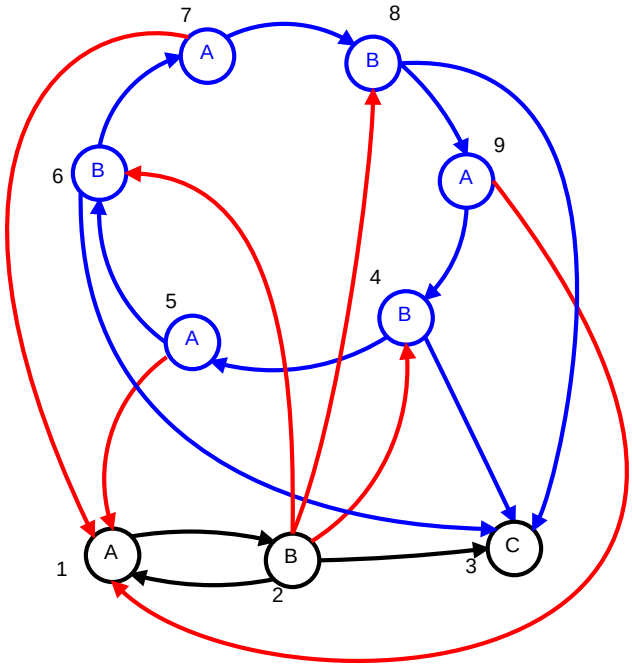


b) A data graph, G

Figure 3.3: Bound neither on the number of vertices nor the diameter of the result



a) A query graph, Q



b) A data graph, G

Figure 3.4: The effect of post-processing on MaxPGs

Chapter 4

Tight Simulation

4.1 Introducing Tight Simulation

Strict simulation reduces the computation time of Strong simulation by decreasing the size of the balls, but the number of balls remains the same. The number of balls can be big when the number of vertices left after of dual simulation is big which exacerbates the performance. Moreover, it is still desirable, both in terms of computation time and the quality of the result, to shrink the size of the balls. In this chapter, we introduce a new novel pattern matching model, named tight simulation, which not only decreases the size of the balls further in comparison to strict simulation, but also reduces the number of balls. This will make the search for the pattern faster and the resulting subgraphs more stringent.

In this chapter we talk about eccentricity of vertices, and canter and radius of graphs. The eccentricity of a vertex in a graph, Q, is its maximum distance from any other vertex in the graph. The vertices of the graph with the minimum eccentricity are the centers of the graph, and the value of their eccentricity is the radius, r_Q, of the graph. The maximum value of eccentricity equals to the diameter of the graph, d_Q. It is proved that $r_Q \leq d_Q \leq 2r_Q$ [29].

4.2 Comparison of Tight and Strict Simulation

A summary of the centralized algorithm for tight simulation is compared to strict simulation in Figure 4.1. The main difference between these two is on the phases of preprocessing the query graph, and ball creation. In the phase of preprocessing, a single vertex, $u \in Q$, is chosen as a candidate match to the center of a potential ball on the data graph. The appropriate radius of the ball is also calculated in this phase. Then, in the phase of ball creation, only those vertices of the data graph which are in the dual match set of u will be picked as the center of balls. The pre-calculated radius of such a ball is always between the radius and the diameter of the query graph. It should be noticed that only the vertices in the result of dual match graph will be used for ball creation, similar to strict simulation.

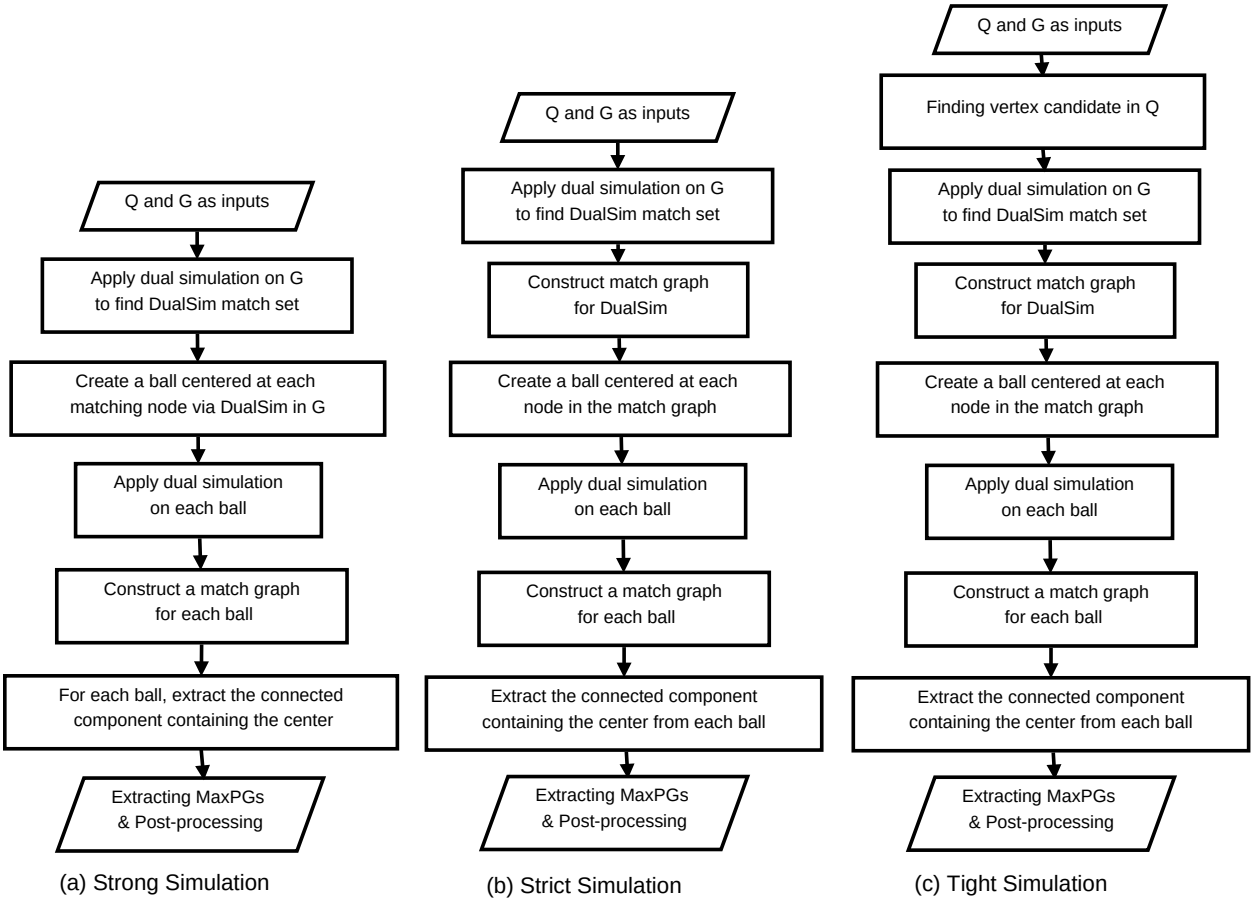


Figure 4.1: Comparing the algorithms of strong, strict, and tight simulation

We introduce multiple selectivity criteria for finding the candidate vertex and radius out of the query graph. A vertex $u \in Q$ with the minimum eccentricity (a center of Q) which has the highest ratio of degree to label frequency (in Q) will be picked as the candidate vertex. Selecting one of the centers of Q as the candidate vertex makes it also possible to select the candidate radius of balls equal to the radius of Q. It is the tightest ball which also preserves all the subgraph isomorphic matches of the query graph in the result. Among the potential vertex candidates those with the highest degree and lowest label frequency present higher selectivity condition. In the case that there are several vertices with same selectivity score, one of them will be selected randomly.

In a centralized algorithm, it is possible to postpone selecting the candidate vertex and radius until the end of dual simulation phase and right before ball creation. One may consider that a vertex $u \in Q$ with the smallest dual match set in G would be the best vertex candidate. In this case, the candidate radius would be equal to the eccentricity of u in Q. Although this choice might be a good option in a centralized algorithm, it is not viable for a distributed algorithms based on vertex-centric framework. In such a distributed environment every vertex of G will learn if it is a member of a dual match set to a few vertices in Q, but no global view of this set would be available. Proposed vertex-centric distributed algorithms will be explained in the next section.

The results of tight simulation are subgraphs of the corresponding results of strict simulation while they always contain all the subgraph isomorphic matches. Therefore, the results of tight simulation are closer to subgraph isomorphism in comparison to strict simulation. It should be noted that the post-processing phase explained in the last section will be applied to the results of strong, strict, and tight simulation in the same way.

Figure 4.2 shows an example which displays the difference between the results of tight versus strict and strong simulation. In this example, all the vertices shown in the data graph will remain in the dual match graph. Clearly, the vertex labeled B in the pattern is its center;

hence, will be picked as the candidate vertex. Therefore, vertices {2,4,6,10,12,14} will be picked as the center of balls with radius $r_Q = 1$ in tight simulation. Only the ball centered at 2 can result a MaxPG which contains these vertices {1,2,3}. In contrast in strict and strong simulation, a ball with radius $d_Q = 2$ will be created for any vertex of the data graph. One can verify that for the ball created on vertex 1, strict simulation results a MaxPG containing {1,2,3,4,5,6,7,8}, and strong simulation results a MaxPG containing all the vertices.

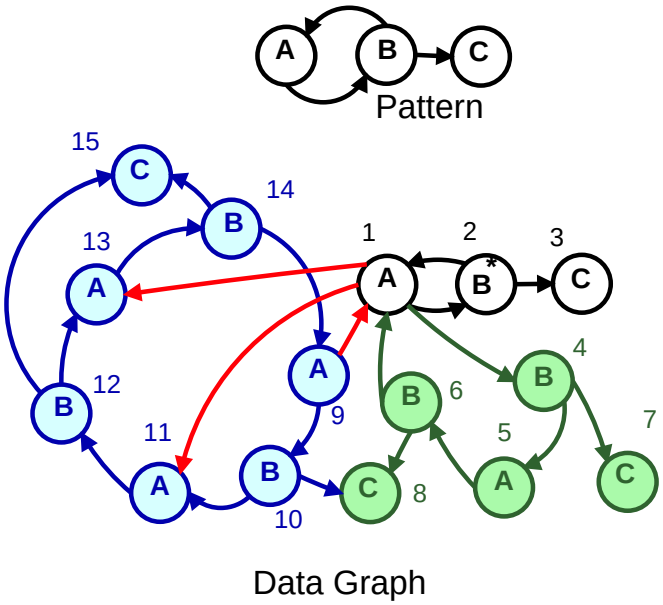


Figure 4.2: Comparing tight with strict and strong simulation

For tight simulation, we also use a post-processing phase to filter the resulted MaxPGs. Similar to what was explained for strict simulation, we filter any MaxPG that is supergraph to any other MaxPG. Furthermore, It should be noticed that similar to strict simulation, there is neither a bound on the number of vertices in a resulted MaxPG nor its diameter.

In figure 4.3 the real-world application of these models for advertisement targeting is illustrated through another example. Here we use the idea of Amazon product co-purchasing graph. This graph is collected by crawling Amazon website [49]. If a product i is frequently co-purchased with product j, the graph contains a directed edge from i to j. Figure 4.3a shows the query pattern. Here, labels are different book departments in Amazon. Figure 4.3b displays the data graph (or part of it). Figure 4.3c shows the result of tight simulation and figure 4.3d shows the results of strict or strong simulation.

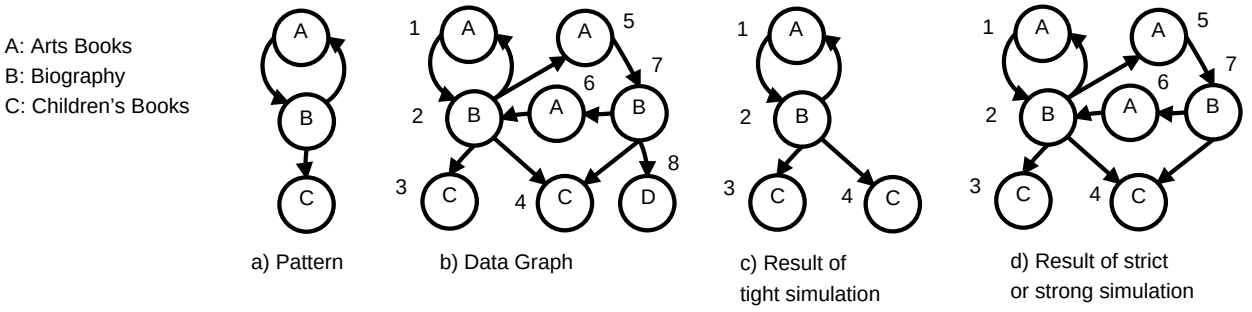


Figure 4.3: An example for graph pattern matching

4.3 Properties of Tight Simulation

We formally define tight simulation as follows.

Definition 4.1 *Pattern $Q(V_q, E_q, l_q)$ matches data graph $G(V, E, l)$ via tight simulation, denoted by $Q \preceq^T_{sim} G$, if there are vertices $u \in Q$ and $u' \in G$ such that (1) u is a center of Q with highest defined selectivity; (2) $(u, u') \in R_D$ where R_D is dual relation set between Q and G; (3)$Q \preceq^D_{sim} \hat{G}_D[u', r_Q]$ where $\hat{G}_D[u', r_q]$ is a ball extracted from $G_D(V_D, E_D, l_D)$ which is the result match graph with respect to $Q \preceq^D_{sim} G$, and r_Q is the radius of Q; (4) u' is a member of the result MaxPG.*

The criterion for selectivity of u in Q is the ratio of its degree to its label frequency. The definition of MaxPG is also similar to its definition for strong and strict simulation. Similar to strict simulation, we can assert and prove the properties of tight simulation as follows.

Theorem 4.1 *For any query graph Q and data graph G such that $Q \trianglelefteq^T_{sim} G$, there exists a unique set of maximum perfect subgraphs for Q and G.*

Proof: It is proved that the result of dual simulation is unique. The candidate vertex selected from the query is also unique when the query and the selectivity criteria are fixed. Therefore, the balls created in tight simulation and their result after dual filter will be also unique. □

Proposition 4.1 *(1) If $Q \trianglelefteq^G_{iso}$, then $Q \trianglelefteq^T_{sim} G$. (2) if $Q \trianglelefteq^T_{sim} G$, then $Q \trianglelefteq^\Sigma_{sim} G$.*

Proof: (1) Any subgraph isomorphic match in G is also dual match to Q; therefore, it will appear in the result of dual match graph. The dual match to candidate vertex of Q is one of the vertices of the isomorphic match and will be selected as the center of a ball with radius r_Q. As the candidate vertex was the center of Q, the isomorphic match will be entirely enclosed in the ball and therefore will appear in the result of tight simulation. (2) When there is a subgraph result in the ball of strict simulation, it will also appear as a part of the result of strict simulation because there is a corresponding ball in strict simulation created on G_D with a bigger radius. In other words, a ball in tight simulation is always a subgraph of its corresponding ball in strict simulation. □

Proposition 4.2 *The number of maximum perfect subgraphs produced by $Q \trianglelefteq^T_{sim} G$ is bounded by the number of vertices in G.*

Proof: The results of tight simulation are subset of the results of strict simulation. The number of maximum perfect subgraphs produced by $Q \trianglelefteq_{sim} G$ is bounded by the number of vertices in G; hence, it is the same for $Q \trianglelefteq^T_{sim} G$. □

Theorem 4.2 *For any query graph Q and data graph G, the time complexity for finding all maximum perfect subgraphs with respect to tight simulation is cubic.*

Proof: The time complexity for finding the center and the radius of $Q(V_q, E_q, l_q)$ is $(|V_q|^3|)$. The procedure of tight simulation is similar to strict simulation, but it deals with a smaller number of balls which are most likely smaller than the corresponding balls in strict simulation; therefore, its time complexity must be smaller as well. Taking into account that the time complexity of dual simulation phase is cubic, we can conclude that it is the same for tight simulation. □

Chapter 5

Distributed Pattern Matching

In contrast to the usual graph programming models, an algorithm in the vertex-centric programming model should be designed from the perspective of each vertex of a graph. In this chapter, we present distributed algorithms for different types of graph simulation based vertex-centric programming model.

5.1 Distributed Graph Simulation

Figure 5.1 shows a summary of our algorithm for a vertex in distributed graph simulation. Initially, we distribute the query graph among all workers. The cost of this distribution is negligible because the size of query is small and the total number of workers is limited to the number of processing elements in the system. Different tasks in different supersteps are distinguished using an if-else ladder. Moreover, the BSP framework ensures that all vertices are always at the same superstep.

A Boolean flag, named match, is defined for each vertex in G in order to track if it matches a vertex in Q. It is initially assumed that the vertex is not a match. Then at the first superstep, the match flag becomes true if its label matches the label of a vertex in Q.

Superstep 1:

- Set *match* flag true if there is any vertex in query with same label
  - Make a local match set, *matchSet*, of potential match vertices
  - Ask children about their status
- Otherwise vote to halt

Superstep 2:

- If the flag is true reply back with *matchSet*
- Otherwise vote to halt

Superstep 3:

- If *match* flag is true evaluate the members of *matchSet*
  - In the case of any removal from *matchSet*, inform parents and set *match* flag accordingly
  - Otherwise vote to halt
- Otherwise vote to halt

Superstep 4 and beyond:

- If there is any incoming removal message reevaluate *matchSet*
  - In the case of any removal from *matchSet*, inform parents and set *match* flag accordingly
  - Otherwise vote to halt
- Otherwise vote to halt

Figure 5.1: Summary of Distributed Graph Simulation algorithm

In this case, a local match set, named *matchSet*, is created to keep track of its potential matches in Q. Each vertex, then, learns about the *matchSet* of its children during the first three supersteps and keeps them in a local list for later evaluation of graph simulation conditions.

Any match is removed from the local *matchSet* if it does not satisfy the simulation conditions. The vertex should also inform its parents about any changes in its *matchSet*. Consequently, any vertex that receives changes in its childrens *matchSet* reflects those changes in its list of match children and reevaluates its own *matchSet*. The algorithm can terminate

after the third superstep if no vertex removes any match from its *matchSet*. This procedure will continue in superstep four and beyond until there is no change. To guarantee the termination of the algorithm, any active vertex with no incoming message votes to halt after the third superstep. At the end, the local *matchSet* of each vertex contains the correct and complete set of matches between that vertex and the vertices of the query graph.

Figure 5.2 displays an example for distributed graph simulation. Here, all the vertices of the data graph labeled a, b, and c make their match flag true at the first superstep, and then vertices 1, 2, and 5 send messages to their children. At the second superstep only vertices 5, 6, and 7 will reply back to their parents. At the third superstep, vertices 1, 5, 6, 7, and 8 can successfully validate their matchSets, but vertex 2 makes its flag false, because it receives no message from any child. Therefore, vertex 2 sends a removal message to vertex 1. This message will be received by vertex 1 at superstep four. It will successfully reevaluate its match set, and the algorithm will finish at superstep five when every vertex has voted to halt (there is no further communication).

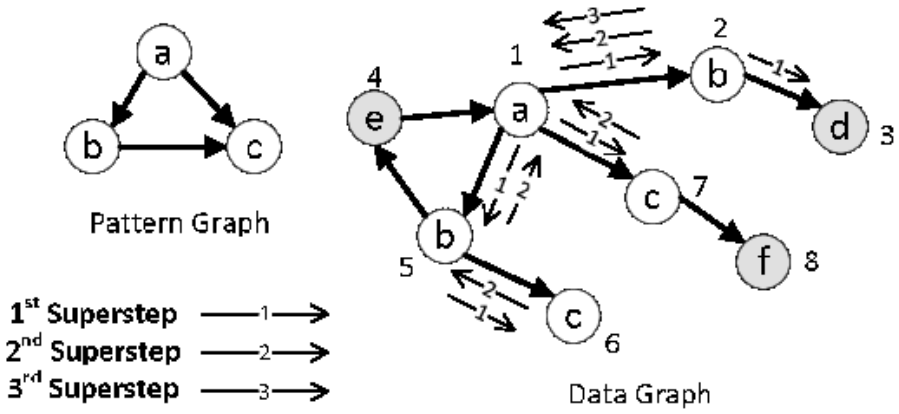


Figure 5.2: An example for distributed graph simulation

The pseudo code of our algorithm is also displayed in figure 5.3. Here, *chMatch* is the local list that each vertex keeps the learned *matchSet*s of its children. Each condition of the if-else ladder in the pseudo code will be executed simultaneously by all the active vertices of the data graph at the same superstep.

To clarify the algorithm further, we explain its running behavior for the pattern and data graphs presented in figure 5.4. The IDs of vertices stored in *matchSet* or $chMatch$ are displayed in figure 5.5 for each superstep. Initially, both sets are empty for all vertices and their *match* flags are $false$ as well. In the first superstep, all vertices except 5 find a potential match based on their labels, so they change their *match* flags to *true*. For the sake of space, the vertices whose *matchSet*s become empty are not displayed. In the second superstep, each vertex learns the IDs of its parents and replies back with its *matchSet*. No changes in either *matchSet* or $chMatch$ are expected. At the third superstep, each vertex learns about the *matchSet*s of its children; consequently, vertex 4 discovers that it is not an appropriate match to $q2$ and makes its membership $false$. Vertex 4 informs vertex 3 of its removal. Thus, in superstep 4, vertex 3 will also remove itself from membership and inform vertex 2 of this change. Because vertex 2 can rely on vertex 1 to satisfy its required child relationships, no changes in *matchSet* occur and the algorithm terminates in the fifth superstep.

5.1.1 proof of correctness

The correctness of the proposed distributed algorithm can be derived from the following lemmas.

Lemma 5.1 *The proposed distributed algorithm for graph simulation will eventually terminate.*

Proof: The algorithm will terminate when all the vertices vote to halt and become inactive. After the third superstep, only the vertices are active which have received removal messages. A removal messages is sent from a vertex to its parents when it removes a member of its matchSet. The total number of members of all matchSets is finite; therefore, the algorithm will terminate eventually in a finite time when all the matchSets become empty in the worst case. □

Lemma 5.2 *At the end of the proposed algorithm, the matchSet of each vertex contains the correct and complete set of matches for that vertex.*

Proof: At the first superstep, each vertex creates its matchSet from any vertex in Q with the same label. Hence, any potential match initially becomes a member of this set. The set is filtered during the next supersteps, and it is expected that it will contain only correct matches at the end. In other words, the completeness condition of the set is satisfied at the first superstep, and we should only prove the correctness of its members when the algorithm terminates.

BSP computational model ensures that all vertices are synchronized at the beginning of each superstep. Having this property in mind, the set of supersteps 4 and beyond in the proposed algorithm (Figure 5.1) is very similar to a while loop. Therefore, this lemma can be proved using loop invariant theorem [41]. Here, the invariant is the validation of each matchSet with respect to the local list of match children. At the end of the third superstep,

each vertex has a matchSet which its members are validated based on the information gathered from the matchSet of its children. The guard condition for iterating through supersteps 4 and beyond is receiving at least one removal message. The invariant condition is true at the beginning of each superstep, and will be also true at the end of the superstep because the vertices that have received any removal message will update their list of match children accordingly and reevaluate their matchSets. According to lemma 5.1, the guard condition will become false after a finite number of iterations. Existence of no removal message means that all vertices have satisfied the children condition. Therefore, the invariant is true after termination; i.e., each member of a matchSet is a correct match. □

Figure 5.4 demonstrates the number of supersteps in the worst case. One can verify that if the length of path 1,2,3,4 is increased to L_p in such a way that the label pattern of vertices 3 and 4 are repeated, the algorithm will need $L_p + 1$ supersteps to terminate. In general, the minimum number of supersteps is 3, and its upper bound is $O(|E|)$.

5.2 Distributed Dual Simulation

The distributed algorithm for dual simulation is a smart modification of the distributed algorithm proposed for graph simulation in the previous subsection. Indeed, we extend the algorithm to check parent relationship as well. Therefore, each vertex also needs to keep track of the matchSets of its parents.

At the first superstep each vertex sends not only its ID, but also its label to its children. At the second superstep a vertex can infer the matchSets of its parents from the received labels and store them. Having this initial list at the second superstep allows each vertex to verify the parent relationships for each of the candidate matches in its matchSet. Very similar to the idea explained in the previous subsection for child relationships, removals from matchSet caused by evaluating the parent relationship must be reported to the children.

The rest of the algorithm remains similar to the algorithm for graph simulation, with a few small modifications to consider the evaluation of a vertex with respect to its parent relationships.

The proof of correctness for this algorithm is very similar to the proof of correctness for the graph simulation algorithm. Similarly, the upper bound on the number of required supersteps for the distributed algorithm of dual simulation is $O(|E|)$.

5.3 Distributed Strong, Strict, and Tight Simulation

The algorithms for distributed strong, strict, and tight simulation are built on top of the algorithm for distributed dual simulation. In the case of strong and strict simulation, each vertex that has successfully passed the filter of dual simulation will create a ball around itself. Recalling their definitions, each ball in strong simulation is an induced connected subgraph of the data graph; whereas, each ball in strict simulation is an induced connected subgraph of the dual match graph. In the case of tight simulation, only those vertices that find themselves a dual match of the candidate vertex of Q will create a ball around itself.

The distributed algorithms that we have designed and implemented for strong and strict simulation follow the flowcharts of Figure 3.1, but in a distributed fashion. The first step for applying dual simulation is the same as the distributed algorithm for dual simulation. At the end of the dual simulation phase, each vertex of the data graph has a matchSet that contains the IDs of vertices of Q that match to that vertex. Any vertex with a non-empty matchSet which is qualified to make a ball centered at itself, finds the member of its ball in a breadth-first search (BFS) fashion.

Ball creation phase takes $2(R-1)$ supersteps where R is the selected radius for the ball. For strong and strict simulation $R = d_Q$, while it is $R = r_Q$ for tight simulation. Moreover, all the vertices in the neighborhood are considered as members of a ball in strong simulation.

In contrast for strict and tight simulation, only vertices with the match flag set to true will answer the request for neighborhood information. In the latter case, while the center vertex of the ball receives neighborhood information, it adds a vertex to the ball only if there is a corresponding edge in the pattern graph. Eventually, the center vertex of each ball will perform the rest of the computation on that ball in a sequential fashion. Because vertices are distributed among workers, this phase can be considered an embarrassingly parallel workload.

5.4 Experimental Study

This section is dedicated to experimental study which aims to evaluate the new pattern matching models and the proposed distributed algorithms. Regarding the distributed algorithms, the study attempts to learn about their bottlenecks and design trade-offs with respect to the properties of their inputs. We implemented our distributed algorithms on the GPS platform [60], which is similar to Googles proprietary Pregel system.

The parameters for data graphs are the number of vertices, denoted by $|V|$, the density of the graph, denoted by parameter α, where $|E| = |V|^{\alpha}$, and the number of distinct labels, denoted by l. In all experiments, $l = 200$, unless it is mentioned explicitly. The parameters for queries are also the number of vertices, denoted by $|V_q|$. Another parameter in the experiments is the number of workers denoted by k.

5.4.1 Experimental setting

We used both real world and synthesized datasets in our experiments. In terms of real world datasets, we used uk-2002 with 18,520,486 vertices and 298,113,762 edges; ljournal-2008 with 5,363,260 vertices and 79,023,142 edges; and amazon-2008 which has 735,323 vertices and 5,158,388 edges [7, 15].

We used graph-tool [5] to synthesize small and medium size randomly generated data graphs, but because of its memory limits we also implemented our own graph generator to synthesize large semi-randomly generated graphs. The input parameters of our graph generator are the number of vertices, the average number of outgoing edges deg_O, and the number of distinct labels. It picks a random integer between 0 and $2deg_O$ as the number of outgoing edges for every vertex. Then, for each outgoing edge, the endpoint of the edge is randomly selected. The label of each vertex is also a randomly picked integer number between 1 and l.

To generate a pattern graph, we randomly extract a connected subgraph from a given dataset. Our query generator has two input parameters: the number of vertices and the desired average number of outgoing edges. Unless mentioned otherwise, the average number of outgoing edges is set to such a value that $\alpha = 1.2$. In order to randomly extract a query with n vertices and average degree d from a given data graph, we first randomly pick one vertex of the data graph. Then, we generate a random number between 1 and d. We add this number of neighbors to the query and continue this process in a BFS fashion until the required number of vertices are added to the query.

The experiments were conducted using GPS on a cluster of 8 machines. Each one has 128GB DDR3 RAM, two 2GHz Intel Xeon E5-2620 CPUs, each with 6 cores. The intra-connection network is 1Gb Ethernet. One of the machines plays the role of master.

5.4.2 Experimental results

The results of the experiments are categorized in four groups. It should be mentioned that distributed strong simulation is so slow on large datasets that we could run it only on fairly small datasets to be compared with strict and tight simulation. It is also noteworthy that presenting running time or speedup of different models in the same chart is only for studying their scalability. In other words, when the quality of pattern matching increases from graph

simulation to dual and then strong simulation, the running time also increases. Strict and tight simulation are exceptions; i.e., we observe decrease in running time when the quality increases from strong to strict and then tight simulation.

We also performed a set of experiments to compare the quality of the results of strong, strict, and tight simulation. For comparison, we measured a few parameters in their set of subgraph results including: the number of subgraph results, their total number of distinct vertices, their total number of distinct edges, and the average and standard deviation of their diameters. We found that the number of subgraph results is increasing and their diameters are decreasing while we change the model from strong to strict and from strict to tight simulation. However, our experiments have yet shown no significant difference in either the total number of distinct vertices or the total number of distinct edges.

Experiment 1- Running time and Speedup

We examine the running time and speedup of the proposed algorithms to study their performance and scalability (Figure 5.6). It can be observed that the running time of tight simulation is always less than the running time of strict simulation. In the experiment displayed in Figure 5.6a, we could not run the test on a single machine because of memory limits. Therefore, we extrapolate its running time.

As expected, the BSP model scales very well on bigger datasets. Moreover, all types of simulation exhibit a filtering behavior, meaning that they start by processing a large set and then refine it; this causes light workload at the final supersteps.

Experiment 2- Impact of pattern

Figure 5.7 shows that running time increases as the size of the query becomes bigger, which is not surprising. The behavior remains similar across datasets with different numbers of vertices. The running times of strict and tight simulations are similar for small patterns

because the difference in the overhead of ball-creation phase is negligible. However, it can be observed that the difference between their running-time increases with increase in the size of pattern.

Experiment 3- Impact of dataset

In the first experiment of this group (Figure 5.8a), we compare the running time of different pattern matching models with respect to the number of vertices in the data graph. As expected, the running time increases with growth in the size of data graph. The ratio between the running times of graph simulation and dual simulation shows the difference between their computational complexities. The increasing difference between the running times of strict and dual simulation can be explained by the fact that the number of balls increases proportionally to the number of vertices in the result of dual simulation. However, the rate of increase in difference of tight and strict simulation is very smaller because even in the case of tight simulation all the active vertices after dual-filtering phase will contribute to the ball creation although the number of balls is smaller as well.

Figure 5.8c shows the total number of supersteps for the same set of experiments. It is not surprising that tight simulation needs less number of supersteps to terminate because the radius of its ball is smaller. The stable number of supersteps indicates the scalability of the algorithms with respect to the number of vertices in data graph. That is, increase in the size of the data graph mostly increases the local computation of the workers not their communication.

Figure 5.8b shows the impact of the density of a data graph on running time. An increase in running time is to be expected. The unchanged cost of ball creation (difference between dual, strict and tight simulation) reveals a good feature of strict and tight simulation; because the density of the data graph does not have a big impact on the density of the dual match graph, the balls do not necessarily increase in size as the density of the data graph increases.

The total number of supersteps for these experiments is reported in Figure 5.8d. The small changes in the number of supersteps indicate the scalability of the algorithms with respect to the density of data graph.

Experiment 4- Comparison of strong, strict, and tight simulation

The difference in behavior between the algorithms for strong, strict, and tight simulation can be seen in Figure 5.9. Because of the high cost of ball creation in distributed strong simulation, it was only possible to test it on fairly small datasets.

Figure 5.9a and Figure 5.9b compare the running time of the three algorithms on two different datasets and a range of query sizes. The differences between strong simulation and the other two are huge on both datasets, though they are small for $|V_q| \leq 10$. The running time of tight simulation is always slightly better than strict simulation.

Figure 5.9c shows the total size of the communication in the system per superstep for strong, strict, and tight simulation. The chart shows three phases for the procedure. The first phase is the dual simulation phase that occurs before superstep 6. The ball creation occurs between supersteps 6 and 19 for strong and strict simulation, while it finishes at superstep 13 for tight simulation. Supersteps 20 and 21 in strong and strict algorithms correspond to processing balls and terminating the algorithms. This phase occurs in supersteps 14 and 15 of tight simulation. It is clear that, there is no communication in these supersteps.

There is a difference in communication at the first superstep because in strong simulation every vertex needs to learn about its neighborhood regardless of its matching status. Strict and tight algorithms perform exactly the same in the first phase. The exponential increase of communication size in strong simulation during creating balls is because of the involvement of all vertices in that process.

The jagged shape of communication is because of our BFS-style algorithm for discovering balls, which contains requests at one superstep and responses at the next. Expectedly, the communication size of tight simulation is less than strict during the second phase.

Figure 5.9d displays the difference between the numbers of active vertices in the three different types of simulation. Although the number of balls is smaller in tight in comparison to strict simulation, it has the same number of active vertices in the phase of ball creation because all the vertices of dual-match result graph stay active in this phase.

After the dual simulation phase, though the number of active vertices in strict and tight simulation does not experience a significant change, the number of active vertices in strong simulation increases exponentially because inactive vertices become active again during ball creation.

```
procedure VERTEX(u, inMessages)
    match ← false
    if superstep = 1 then
        retrieve labelIndex containing the labels of Q
        matchSet ← labelIndex(label(u))
        if matchSet ≠ ∅ then
            match ← true
        if matchSet has any non-leaf node then
            send id of u to its children
        else vote to halt
    else if superstep = 2 then
        if match = true then
            store received ID of parents and return label(u)
        else vote to halt
    else if superstep = 3 then
        find chMatch from received labels
        Δmatch ← evaluateChildren()
        if Δmatch ≠ ∅ then
            send Δmatch to all parents
            if matchSet = Δmatch then
                match ← false
        else vote to halt
    else if superstep ≥ 4 and inMessages ≠ ∅ then
        if match = true then
            for all ΔchMatch ∈ inMessages do
                chMatch ← chMatch\ΔchMatch
            Δmatch ← evaluateChildren()
            if Δmatch ≠ ∅ then
                send Δmatch to all parents
                if matchSet = Δmatch then
                    match ← false
            else vote to halt
        else vote to halt
    else if superstep ≥ 4 and inMessages = ∅ then
        vote to halt
end procedure
```

Figure 5.3: Distributed graph simulation algorithm

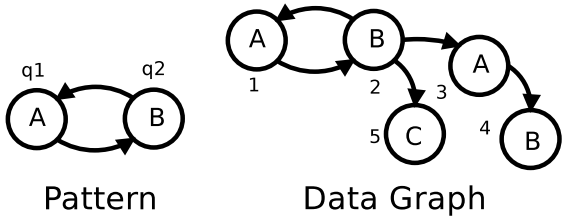
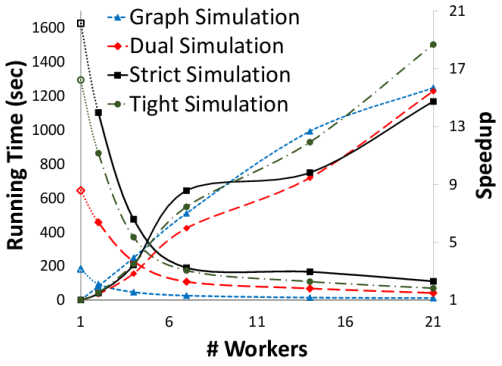


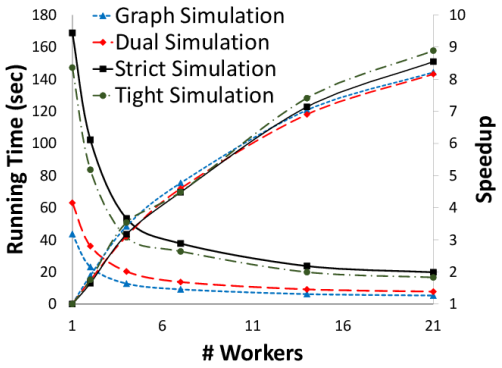
Figure 5.4: An example for the number of supersteps

initialization
1,2,3,4,5: $matchSet = \{\}$, $chMatch = \{\}$
superstep = 1
1,3: $matchSet = \{q1\}$, $chMatch = \{\}$
2,4: $matchSet = \{q2\}$, $chMatch = \{\}$
superstep = 2
1,3: $matchSet = \{q1\}$, $chMatch = \{\}$
2,4: $matchSet = \{q2\}$, $chMatch = \{\}$
superstep = 3
1: $matchSet = \{q1\}$, $chMatch = \{q2\}$
2: $matchSet = \{q2\}$, $chMatch = \{q1, q1\}$
3: $matchSet = \{q1\}$, $chMatch = \{q2\}$
superstep = 4
1: $matchSet = \{q1\}$ $chMatch = \{q2\}$
2: $matchSet = \{q2\}$ $chMatch = \{q1, q1\}$
superstep = 5
1: $matchSet = \{q1\}$, $chMatch = \{q2\}$
2: $matchSet = \{q2\}$, $chMatch = \{q1\}$
finish

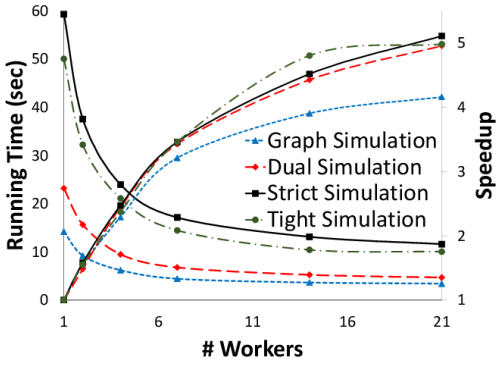
Figure 5.5: Example for running distributed graph simulation algorithm



(a) Synthesized ($|V| = 1e8, \alpha = 1.2$)

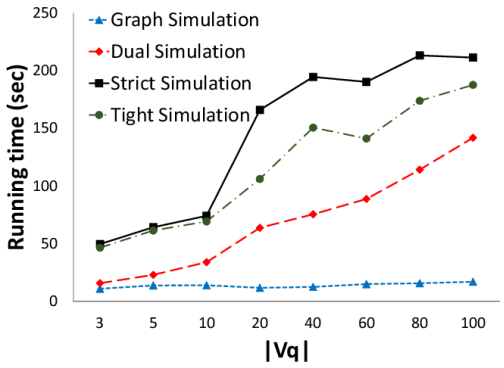


(b) uk-2002-hc ($|V| \approx 1.8e7, |E| \approx 2.9e8$)

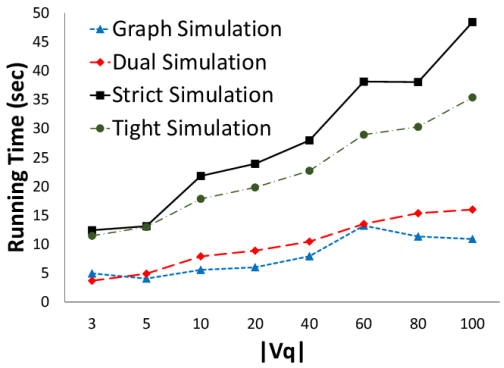


(c) ljournal-2008 ($|V| \approx 5.3e6, |E| \approx 7.9e7$)

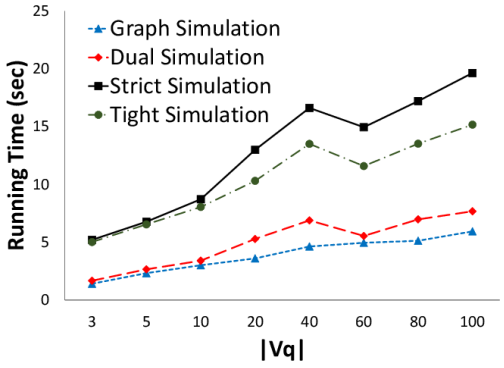
Figure 5.6: Running time and speedup of the distributed algorithms ($l = 200, |V_q| = 20$)



(a) Synthesized ($|V| = 1e8, \alpha = 1.2$)

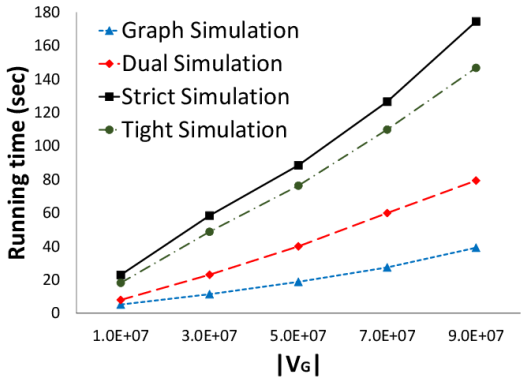


(b) uk-2002-hc ($|V| \approx 1.8e7, |E| \approx 2.9e8$)

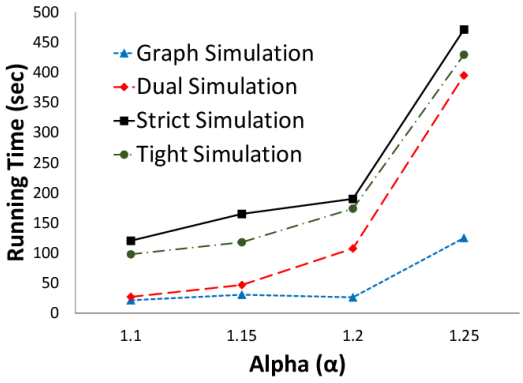


(c) ljournal-2008 ($|V| \approx 5.3e6, |E| \approx 7.9e7$)

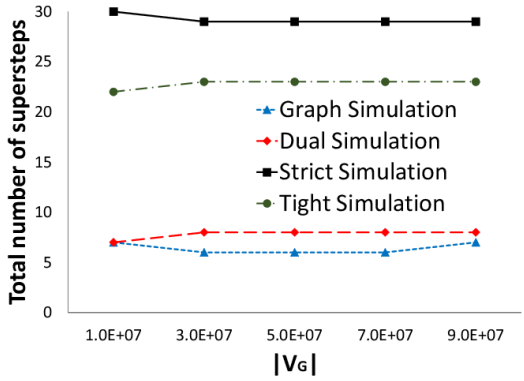
Figure 5.7: Impact of pattern for both real-world and synthesized datasets ($l = 200, k = 7$) on running time



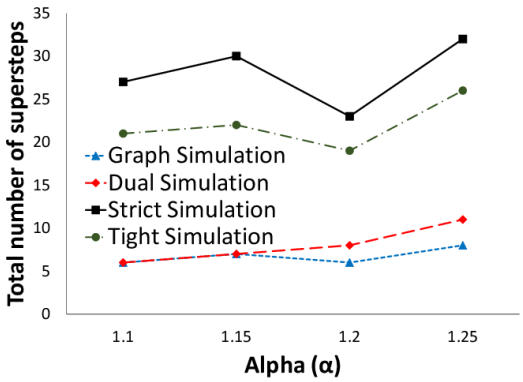
(a) Running time, synthesized ($\alpha = 1.2$)



(b) Running time, synthesized ($|V| = 1e8$)

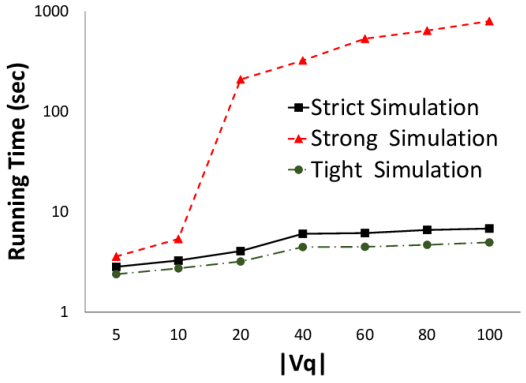


(c) Number of supersteps, synthesized ($\alpha = 1.2$)

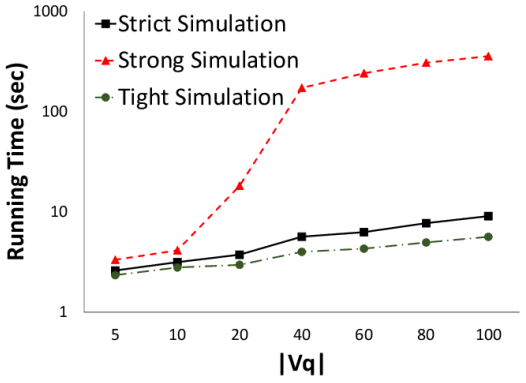


(d) Number of supersteps, synthesized ($|V| = 1e8$)

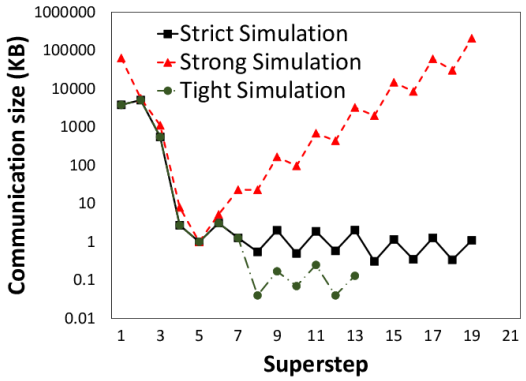
Figure 5.8: Impact of size and density of G ($|V_q| = 20, l = 200, k = 7$) on running time and the number of supersteps



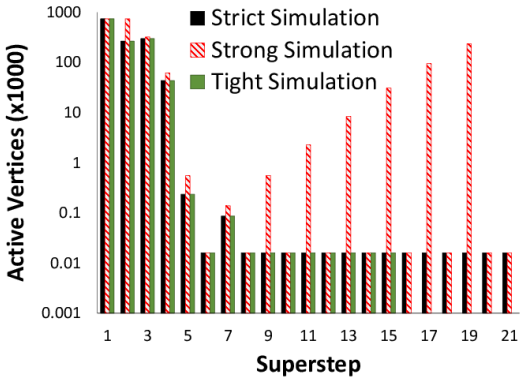
(a) Running time, synthesized ($|V| = 1e6, \alpha = 1.2$)



(b) Running time, amazon-2008 ($|V| \approx 7.3e5, |E| \approx 5.1e6$)



(c) Communication size, amazon-2008



(d) Number of active vertices, amazon-2008

Figure 5.9: Comparison of strong, strict, and tight simulation for real-world and synthesized datasets ($l = 200, k = 14$)

Chapter 6

Cardinality Restricted Simulation

Dual simulation, the way it was first introduced in [52], does not consider cardinality of vertices in the child or parent relationships. We found that in many real-world graphs, like co-purchasing and citation networks, distribution of labels are very skewed; e.g., there are communities that many vertices have the same label. Dual simulation produces not very meaningful results in these regions when some of the vertices in the query have the same label because they all may match to the same vertex or very simple connection of vertices in the data graph.

Here, we introduce a modified version, called *cardinality restricted* or *CAR-dual simulation*, in which the number of match children or parents with the same label in the data graph should not be less than their correspondents in the query. This extra condition does not increase the time complexity of dual simulation, but it improves its expressiveness. Our experiments show a significant drop in the number of vertices in the results of CAR-dual simulation in comparison to dual simulation.

All the models defined so far based on dual simulation; e.g., strong [52], strict [32], and tight simulation [31], can be revised to employ CAR-dual simulation. This modification, not only improves the quality of their results, but also can impact their performance because

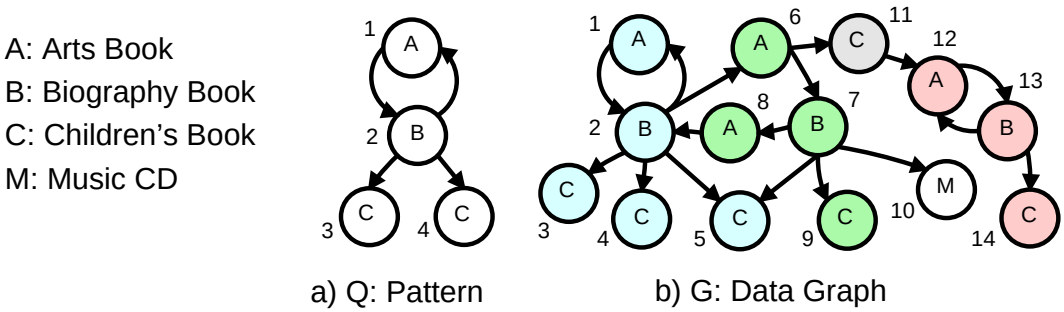


Figure 6.1: An example for comparing cardinality restricted models with their older counterparts

they will be constructed on less number of vertices. A smaller result is clearly more amenable for cache storage. Following, we present the formal definition of these new models.

The difference of models is illustrated through an example in figure 6.1. This example is inspired from Amazon product co-purchasing network [49], where if a product i is frequently co-purchased with product j, the graph contains a directed edge from i to j. Here, each letter inside the vertex is the category of the product and represents its label. Moreover, each number beside a vertex represents its ID number. The subgraph matching results of this example are displayed in table 6.1.

In figure 6.1, subgraph isomorphic match of pattern Q to data graph G has three subgraph results. As it is displayed in table 6.1, the result of graph simulation contains all the vertices of G except 10 because they all have the same label and meet child relationship with respect to their counterpart vertices in Q. Only vertex 11 will be omitted in the result of dual simulation because it does not satisfy parent relationship.

To find the result of tight simulation, we first need to find candidate vertices for the center of balls. The *candidate vertices* in G are those that are dual match to the center of Q. In our example 2 is the center of the query; therefore, vertices 2, 7, and 13 are candidate vertices. Moreover, the distance for finding the neighbor vertices equals to the radius of Q,

Table 6.1: Results of different pattern matching models

Model	Symbol	Subgraph Results
subgraph isomorphism	$Q \trianglelefteq_{iso} G$	$f(1,2,3,4) \rightarrow (1,2,3,4)$ $,(1,2,3,5),(1,2,4,5)$
graph simulation	$Q \trianglelefteq_{sim} G$	$R(1,2,3,4) \rightarrow (\{1,6,8,12\},\{2,7,13\}$ $,\{3,4,5,9,11,14\},\{3,4,5,9,11,14\})$
dual simulation	$Q \trianglelefteq^{D}_{sim} G$	$R(1,2,3,4) \rightarrow (\{1,6,8,12\},\{2,7,13\}$ $,\{3,4,5,9,14\},\{3,4,5,9,14\})$
tight simulation (based on $\trianglelefteq^{D}_{sim}$)	$Q \trianglelefteq^{T}_{sim} G$	$R(1,2,3,4) \rightarrow (1,2,\{3,4,5\}$ $,\{3,4,5\}),(12,13,14,14)$
CAR-dual simulation	$Q \trianglelefteq^{CD}_{sim} G$	$R(1,2,3,4) \rightarrow (\{1,6,8\},\{2,7\}$ $,\{3,4,5,9\},\{3,4,5,9\})$
CAR-tight simulation (based on $\trianglelefteq^{CD}_{sim}$)	$Q \trianglelefteq^{CT}_{sim} G$	$R(1,2,3,4) \rightarrow (1,2,\{3,4,5\}$ $,\{3,4,5\})$

which is one in our example. As it is displayed in table 6.1, the result of tight simulation in our example will be two subgraphs.

It should be noticed that in this example, vertex 13 in G is a dual match to vertex 2 in Q because the vertex 14 is a dual match to both vertices 3 and 4 at the same time. In example of figure 6.1, vertices 12, 13, and 14 are excluded in the result of CAR-dual simulation. Consequently, *CAR-tight simulation*, which is defined based on the new dual simulation, will have a single subgraph result as it is displayed in table 6.1. Following, we present the formal definition of these new models.

6.1 CAR-dual Simulation

We first present formal mathematical definition of dual simulation, and based on that we define CAR-dual simulation. By definition [52], pattern Q matches data graph G via *dual simulation*, denoted by $Q \trianglelefteq^{D}_{sim} G$, if there is a binary relation $R_D \subseteq V_Q \times V_G$ such that it meats the following conditions. (1) Having the same label: $(u, u') \in R_D \Rightarrow l_Q(u) = l_G(u')$; (2) All vertices of Q are covered: $\forall u \in V_Q, \exists u' \in V_G : (u, u') \in R_D$; (3) Child relationship: $\forall (u, u') \in R_D[(u, v) \in E_Q \Rightarrow \exists v' \in V_G : (v, v') \in R_D \wedge (u', v') \in E_G]$ (4) Parent relationship: $\forall (u, u') \in R_D[(w, u) \in E_Q \Rightarrow \exists w' \in V_G : (w, w') \in R_D \wedge (w', u') \in E_G]$.

The result of this model is defined as a *maximum dual match set*, $R_D \subseteq V_Q \times V_G$, which is the biggest relation set between Q and G with respect to $Q \trianglelefteq^{D}_{sim} G$. The *result dual match graph*, $G_D(V_D, E_D, l_D)$, for this model is a subgraph of G that can represent R_D. By definition, G_D is a subgraph of G which satisfies these conditions: (1) $(u, u') \in R_D \Leftrightarrow u' \in V_D$; (2) $\forall (u, u')(v, v') \in R_D[(u, v) \in E_Q \Leftrightarrow (u', v') \in E_D]$.

To define CAR-dual simulation, we add cardinality condition to the children and parents with the same labels. Other definitions remain the same.

Definition 6.1 *Pattern Q matches data graph G via* CAR-dual simulation*, denoted by $Q \trianglelefteq^{CD}_{sim} G$, if there is a relation R_D with respect to $Q \trianglelefteq^{D}_{sim} G$, and for every $(u, u') \in R_D$: (1) the number of children of u' in G_D with a particular label is not less than the number of children of u in Q with the same label; (2) the number of parents of u' in G_D with a particular label is not less than the number of parents of u in Q with the same label.*

6.2 CAR-tight Simulation

The only difference between CAR-tight simulation and tight simulation, defined in [31], is its construction based on CAR-dual simulation instead of dual simulation. To find the

neighborhood of a vertex to enforce locality condition, it uses a concept called *ball* [52]. A ball b in G, denoted by $\hat{G}[c, r]$, is a subgraph of G that contains all vertices in distance r from the vertex c (including c). The vertex c is called the center of the ball, and the integer value r is called the radius of the ball. Moreover, the ball contains all edges in G that connect these vertices (i.e., it is an induced connected subgraph). Given two vertices u and v in a connected graph, the *distance* between them is defined as the minimum number of edges in an undirected path which connects them.

Definition 6.2 *Pattern Q matches data graph G via* CAR-tight simulation*, denoted by $Q \trianglelefteq^{CT}_{sim} G$, if there are vertices $u \in Q$ and $u' \in G$ such that (1) u is a center of Q with highest defined selectivity; (2) $(u, u') \in R_D$ where R_D is maximum dual match set with respect to $Q \trianglelefteq^{CD}_{sim} G$; (3) $Q \trianglelefteq^{CD}_{sim} \hat{G}_D[u', r_Q]$ with maximum dual match set R^b_D, where $\hat{G}_D[u', r_Q]$ is a ball extracted from $G_D(V_D, E_D, l_D)$. G_D is the result dual match graph with respect to $Q \trianglelefteq^{CD}_{sim} G$, and r_Q is the radius of Q; (4) u' is member of at least one of the pairs in R^b_D.*

The connected part of the result match graph of each ball with respect to its R^b_D which contains u' is called a maximum perfect subgraph (MaxPG) of G with respect to Q. The subgraph results of the model is actually distinct set of these MaxPGs. The criterion for selectivity of u in Q is the ratio of its degree to its label frequency.

After this point, when we mention about dual or tight simulation, we mean their new modification unless it is explicitly expressed. Moreover, we may use dual-sim and tight-sim for abbreviation. It is straightforward to show that *CAR-tight simulation*, preserves all the nice features of the old tight simulation including the fact that it contains all the results of subgraph isomorphism.

6.3 Empirical studies

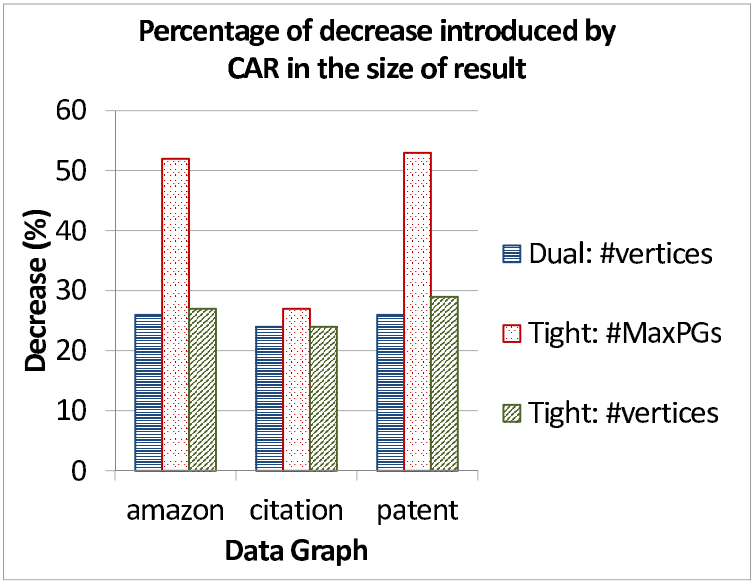
We used three real-world labeled graphs for our experiment: (a) Patent graph [35], a dataset on US patents where each vertex represents a patent and an edge from i to j means that patent i cites patent j. There are about 3.7M vertices and 16M edges in this graph. The label of the vertices are 37 different subcategories of the patents. (b) Citation graph [68], a citation network of papers where an edge from vertex i to vertex j means paper i has cited paper j. It has about 2.2M vertices and 4.3M edges. We selected publishing year of the papers as the label of vertices, which means 80 different labels. (c) Amazon product co-purchase network [49], with about 548K vertices and 1.7M edges. Each vertex is a product, and an edge from i to j means that product i is frequently co-purchased with product j. We used the first category of each product as its label, which led to 104 distinct labels for all vertices.

For our experiments, we randomly extract a set of base queries from each data graph. These queries are from five different sizes: $(10, 3)$, $(15, 3)$, $(20, 4)$, $(25, 5)$, and $(30, 5)$. We extract 100 randomly extracted queries for each size; hence, the base query set for each data graph contains 500 unique queries. As these queries are directly extracted from data graph, they all have some results. The experiments were performed on a machine that has 128GB DDR3 RAM, and two 2GHz Intel Xeon E5-2620 CPUs, each with 6 cores.

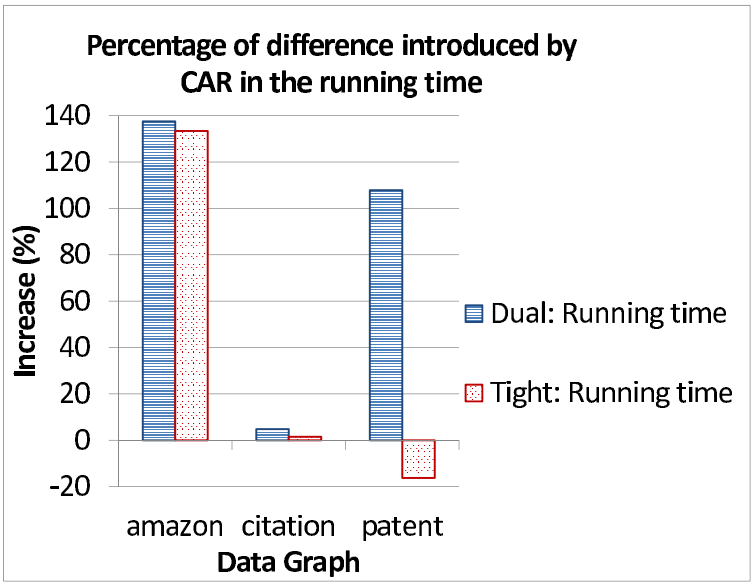
It is expected that the result of *Cardinality Restricted* (*CAR* for abbreviation) version of dual and tight simulation to be more stringent than their old counterparts. Our experiments summarized in figure 6.2 show this property in practice. We used the base query set and performed dual, CAR-dual, tight, and CAR-tight simulation for each query graph, and measured the percentage of decrease in the number of vertices in the results. For CAR-tight simulation, we also measured the decrease in the number of subgraph results.

As it is displayed in figure 6.2a, we observed more than 25% decrease in the average number of vertices in the results of both models for all three data graphs under the test. Moreover, the average number of subgraph results in CAR-tight simulation decreased significantly for amazon and patent data graphs. The number of subgraph results has dropped faster than the number of their containing vertices because of the post-processing phase of tight simulation. At this phase, all subgraph results that are the superset of any other one are filtered. While some subgraphs share many vertices in common, they may not be filtered because of their small differences. Removing less meaningful vertices in CAR-tight simulation makes many of these subgraphs been filtered.

The cost of achiving more meaningful and more stringent results by CAR modification may be longer running time. Figure 6.2b shows the percentage of increase in the running time of dual and tight simulation after CAR modification. Clearly, the running time of the algorithm for CAR-dual is longer than dual because it needs to check extra conditions. However, it is more likely that a decrease in the number of balls can compensate the extra cost of CAR-dual to some extent. Indeed, we could observe that the running time of CAR-tight decreased about 16% on average for Patent data graph.



(a) More stringent results)



(b) Difference in running time

Figure 6.2: Effect of CAR modification

Chapter 7

Effective Caching Techniques for Accelerating Pattern Matching Queries

7.1 Introduction to the Cache System

To speedup the process of finding subgraph isomorphism matches several indexing techniques are introduced like [42] and [19]; however, as it is argued in [67] the construction time and the storage capacity for complicated indices make them unsuitable for large graphs.

Notwithstanding the fact that the time complexity of the new graph pattern matching models, like tight simulation, are polynomial, the size of the graphs representing social networks or web graphs have become so massive that it would be challenging both in terms of response time and required computing resources to query graph patterns inside them. Therefore, a cache system specifically designed for graph pattern matching seems very appealing. Surprisingly, there is very limited research on the design of such a system.

Almost all the previous research works on subgraph pattern matching treat each query as completely independent. In other words, they do not reuse the results from one query to answer another different query. In a specific domain, the recent works in [56, 65] present preliminary ideas for reusing the results of SPARQL queries. The other related work, which very recently published, is in [27] for answering graph pattern queries using views.

The lack of studies in this area is mostly because of intrinsic limits of traditional subgraph pattern matching models. For example, let us assume that G_1 is a graph corresponding to the collection of the results of subgraph isomorphism of query graph Q_1 on a large data graph G. Then, let us have a new query Q_2 which contains a subgraph isomorphic match of Q_1. Although it can be concluded that the results of subgraph isomorphism of Q_2 on G will contain G_1, but it is not sufficient by itself to find these results. Indeed, in subgraph isomorphism model, a new query can be answered using the results of several old queries only if it can be decomposed to those old queries. This idea is sometimes used for indexing a graph [57, 73]. However, indexing a massive data graph is not feasible.

In contrast to subgraph isomorphism, we show that the new concepts introduced by the family of graph simulation models can be employed to design an efficient mechanism for reusing the result of old queries to answer new ones. In an analogy to the previous example, assuming that $G1$ contains the results of tight simulation match of the Q_1 on G, the result of tight simulation of the new query Q_2 on G can be answered using only $G1$ provided that Q_2 is a special type of tight simulation match to Q_1.

As the result of subgraph isomorphism is always embedded in the result of tight simulation, this technique can also be used to speedup response time of queries for traditional type of queries. Intuitively, it would be a significant gain in the performance if we can answer the new popular queries using a relatively small graph which is resulted by the previous queries instead of repeatedly using the main massive data graph.

Existence of popular queries is very common in real-life scenarios. For example, the social network of Facebook has currently more than 1 billion users and every user has links to many different objects [3]. It is reported in [12], however, that less than 10% of the data were accessed during a 6-day trace. The benefits of using a cache system are clear in this example.

7.2 Motivation

Given a query graph Q and a data graph G, the idea is storing its results of subgraph pattern matching in a cache-like system, and using that to answer new queries. This technique can reduce the average response time of queries provided that there are popular new queries which their answers are contained in the answer of Q. Hence, the main challenge in designing an effective content-aware cache system for subgraph pattern matching is the problem of pattern containment.

The main focus of this paper is about pattern containment problem. Let us assume that Q_{old} is an old graph query on G and its set of results is A_{old}. The subproblems are (1) how to store the pair of Q_{old}, A_{old} in the cache space; (2) receiving a new query Q_{new} different from Q_{old}, how we can evaluate the relation between the two to realize if the answer of the new query is contained in A_{old}. The goal is answering more number of new queries without referring to G.

Each time that a new query is received, it should be compared to all old queries in the cache to find out if there is any match. For a large graph, it is likely that after awhile the number of stored queries in the cache grows to a large number; therefore, the overhead of search process in the cache can become restricting. We have designed simple methods to filter the queries in the cache space when we search for a match to a new query. The implemented mechanism for searching the cache shows very efficient in our experiments.

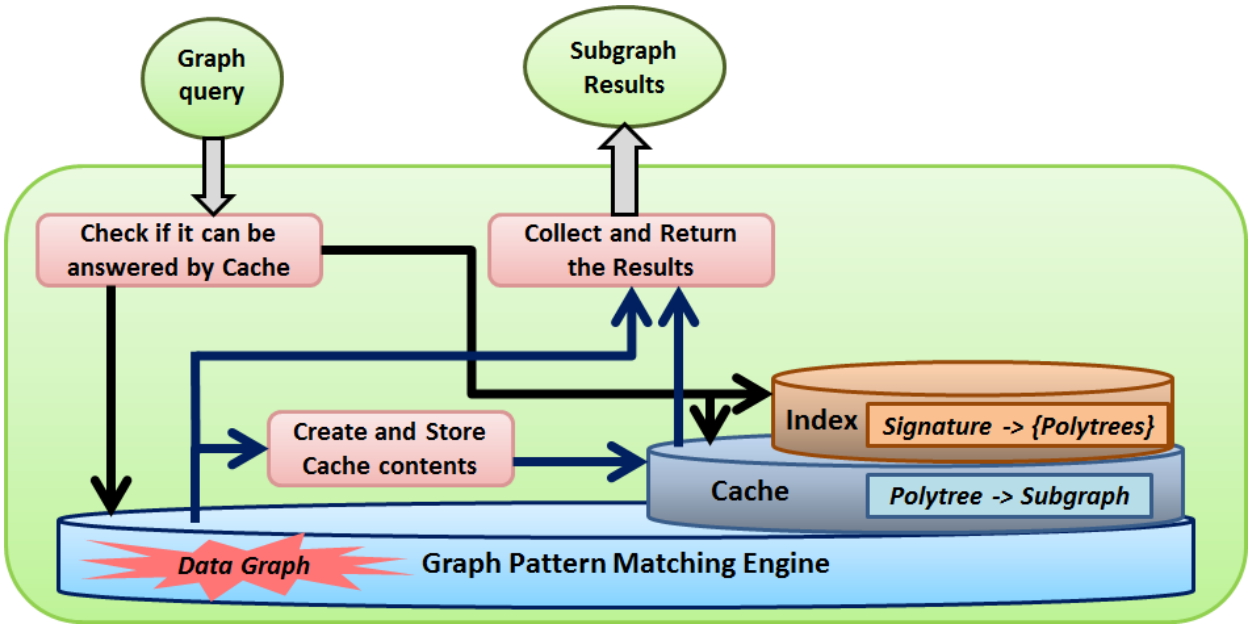


Figure 7.1: The overall architecture of the proposed cache system

When the cache space becomes full, an appropriate replacement mechanism should be available to remove some of the old contents from the cache in order to open space for new queries. Cache replacement strategies are extensively studies in other contexts [59]. The most popular cache replacement policies are based on replacing *least frequently used (LFU)*, or *least recently used (LRU)* items. As a future work, it might be worthwhile to design a specific replacement policy for pattern matching queries. At this work, we have only implemented a simple replacement mechanism which works based on LFU policy in order to test our proposed caching mechanism. We have not tested LRU policy, but we do not expect it to significantly affect our results.

7.3 Architecture

The caching system proposed in this paper works based on tight simulation model; nevertheless, it can also be used to retrieve subgraph isomorphic matches because the result of tight simulation always contains the result of subgraph isomorphism. The overall architecture of the proposed cache system is depicted in figure 7.1. "Graph Pattern Matching Engine" is the main query processing engine that maintains the data graph. It can be a centralized or a distributed system. Cache system resides on top of this engine.

At the warm-up phase of the system, or when a query cannot be answered using the contents of the cache, it will be submitted to the main query engine. Then its result will be stored in the cache space for the future usage. Nevertheless, instead of storing the original query, we retain its spanning polytree. Using polytree as a more inclusive representation will improve the cache hit rate and reduce cache overhead because of its special properties. By definition, polytree is a directed acyclic graph whose underlying undirected graph is a tree. The data structure of the cache space is a key-value map from a polytree to a subgraph. To find the appropriate map-value for this polytree, we first find its tight-simulation result in the data graph and then extract the induced subgraph corresponding to all the vertices in this result. The pair of polytree and its corresponding induced subgraph is stored in the cache space.

In order to facilitate the search in the cache space, we have designed a simple indexing system based on the *signature* of the polytrees. The data structure of the index is a key-value map from a signature to a set of polytrees. When a new query matches to a polytree based on pattern containment conditions, all of its expected results can be retrieved from the polytree's correspondent subgraph; therefore, there would be no need to refer to the original data graph. We will explain about signature and pattern containment conditions in section 7.4.

7.4 The caching mechanism

As it was mentioned earlier, we store pairs of polytree-subgraph in the cache space. We show that any new query which is *cover-tight match* to a polytree can be answered using its corresponding subgraph. We call graph Q_2 *cover-tight match* to graph Q_1, if there is a tight-simulation relation from Q_1 to Q_2 and this relation covers all the vertices in Q_2. Using polytree of an old query graph instead of itself has several advantages. First, it can be used to answer a wider range of new queries. Moreover, a polytree has a number of properties that makes it possible to compute its dual-simulation set instead of tight-simulation for our purpose. Dual-simulation is always faster than tight-simulation. We explain these properties in the next section in detail.

Figure 7.2 shows the main idea. The initial received query is displayed in part (a). Its extracted polytree is illustrated in part (b). After storing the pair of this polytree and its pattern matching result based on tight simulation in the cache, queries in (c) and (d) can be answered without using data graph because they are *cover-tight match* to the polytree.

The designed system has two main modules, each containing several steps as follows. The flowchart of these steps are also displayed in figure 7.3.

7.4.1 Storing the result of an old query

When a query cannot be answered using the contents of the cache, we store an appropriate key-value item to the cache space. The steps for creating such a key-value element follows. Each steps is explained using the example displayed in figure 7.2.

- We first find the spanning polytree of the query graph. For example, when the query in figure 7.2a is received as a new query which cannot be answered using cache contents, its spanning polytree will be the query displayed in figure 7.2b. To find the polytree, we first find a candidate vertex in the query, the same way that we find it for tight

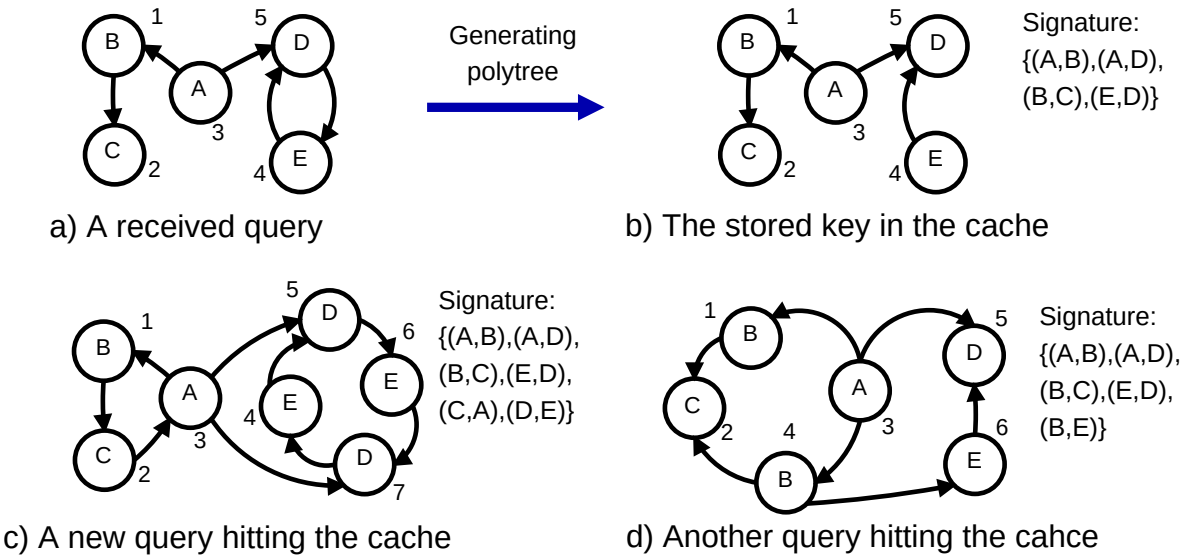


Figure 7.2: An example for the procedure of caching and reusing the results

simulation. That is, we find a center vertex in the query which has the highest ratio of degree to the frequency of its label. Then we traverse the underlying undirected graph of the query in a BFS fashion to find its spanning tree. The spanning ploytree is then created by adding direction to the edges. It should be noticed that the undirected graph may have multiple edges; for instance, the multiple edges between between vertices 4 and 5 in our example. In this case, when the undirected edge in the tree is converted to the directed edge in the polytree, it can have any direction.

- We perform a CAR-dual simulation for the polytree on the data graph, and find the set of vertices in the data graph that are present in the resulting dual-relation set. We will show in the section 7.5 that these set of vertices are the same as the set of vertices in the maximum perfect subgraphs if we had performed tight-simulation.

- We extract an induced subgraph of the data graph using the set of vertices found in the previous step.

- The pair of the polytree and the induced subgraph will be stored in the cache space as a key-value pair.

- In order to facilitate the later search in the cache, the *signature* of the new polytree is calculated. Here, we define the *signature* of each graph as its set of *label-edges*. An *label-edge* for an edge is the pair of the labels of its source vertex and its target vertex. The signature of the graphs in figure 7.2 are displayed for example. We store a map from any created signature to the set of polytrees with the same signature. Therefore, the new polytree will be added to such a set of polytrees with the same signature.

We explain the concept of our defined *graph signature* further using figure 7.2c. One can verify that there are nine edges in this graph. Considering the label of the vertices, it can be inferred that *label-edge* will be a pair of labels for each edge; for example, the label-edge for the edge $(1, 2)$ is (B, C). As another example, the label-edge for both edges $(4, 5)$ and $(6, 7)$ is (E, D). As we define the signature as the set of label-edges, the number of pairs in the signature can be smaller than the number of the edges in the graph. Consequently, the signature of the graph in 7.2c contains only six label-edges.

7.4.2 Search in the cache

When a new query is received, the cache space should be searched for any old polytree which is a cover-tight match to this query. Clearly, it can be very time consuming to check all the polytrees which are available in the cache space. Therefore, we use a simple filtering technique based on the stored signatures.

A polytree can be a candidate match to a new graph query if it has exactly the same set of labels and its signature is a subset of the signature of the new query. Let us assume that the polytree in figure 7.2b and its corresponding subgraph are stored in the cache space, and the new query is the graph in figure 7.2d. It should be noticed the new query has six vertices

and seven edges, while the polytree has five vertices and four edges. Therefore, only a smart caching technique can measure their similarity, and guarantee the existence of all the results of the new query in the cache. The main steps of this phase are:

- The signature of the new query is calculated. For example, the signature of the query graph in figure 7.2d contains five pairs of label-edges.

- The signature of the new query is compared against all the signatures stored in the cache to find the set of candidate match polytrees. There two comparison conditions: (1) the two signature must have exactly the same set of individual labels; (2) the signature of the polytree must be a subset of the signature of the new query.

  In our example, the new query has a set of five labels, A, B, C, D, E which is the same as the polytree in figure 7.2b. If we compare the set of their signatures we can see that they also satisfy the second condition; i.e., the signature of the polytree is a subset of the signature of the new query.

- Any polytree found through the previous step as candidate match will be compared with the new query to check for cover-tight match. Indeed, the new query might be cover-tight match with several polytrees; this means that any of them can be used to answer the new query. Therefore, when the first tight-cover match is found the other candidates will be ignored.

  There is only one candidate polytree in our example. In order to check if the graph in figure 7.2d is a cover-tight match to the polytree in figure 7.2b, their CAR-tight simulation should be calculated. However, it will be proved in section 7.5 that it would be enough to calculate only the maximum CAR-dual relation set. One can verify that the vertices of the new query graph are in such a CAR-dual relation set from the polytree to the query graph. Therefore, the new query is a cover-tight match to the polytree.

- When a cover-tight match is found among candidate polytrees, its correspondent value in the cache space, which is an induced subgraph of the data graph, will be used to perform tight simulation and the results will be returned as the final results.
- When there is no cover-tight match present in the cache for the new query, it should be answered using the original data graph.

Another example illustrated in figure 7.4. Assuming that the query displayed in part (a) has no match in the cache, its polytree is created as it is shown in part (b) of the figure. Then the corresponding subgraph of this polytree will be found from the data graph, and the pair of the polytree and its corresponding subgraph will be stored in the cache. Having this entry in the cache, not only the original query can be answered using the cache content, but also the queries displayed in (c) and (d) can be answered because they are all cover-tight match to the polytree.

7.5 Proof of correctness

In this section, we present the theoretical concepts that prove the correctness of our approach. Here, we state the materials only for CAR-dual and CAR-tight simulation which were introduced in chapter 6, but they are also valid for the previous version of these models. Theorem 7.1 guarantees the pattern containment in our proposed approach. In order to prove the theorem, we first need to define a few new concepts and prove a few lemmas.

Definition 7.1 *A graph Q_2 is a* cover-tight match *to another graph Q_1 when all of its vertices are present in the MaxPGs returned by $Q_1 \trianglelefteq_{sim}^{CT} Q_2$.*

Definition 7.2 *Assuming that R_D is maximum match set for $Q \trianglelefteq^{CD}_{sim} G$, we define $R_{minD} \subseteq R_D$ as a minimum dual match set when (1) it is a complete match set; i.e., $u \in V_Q \Rightarrow \exists u' \in G : (u, u') \in R_{minD}$ (2) removing any vertex of data graph which participate in the match set from R_{minD} makes it incomplete. We also call the subgraph of G corresponding to a R_{minD}, a minimum dual match subgraph.*

Lemma 7.1 *Any maximum perfect subgraph produced by CAR-tight simulation is a union of some minimum dual match subgraphs.*

Proof: Let us assume that $X(V_X, E_X)$ is a maximum perfect subgraph produced by $Q \trianglelefteq^{CT}_{sim} G$. Therefore, X is a connected graph, and any $u \in V_X$ is in a CAR-dual relationship with a vertex in Q according to the definition of maximum perfect subgraph. To prove this lemma, it would be clearly enough to show that u is member of at least one minimum dual match subgraph, like M, which is also a subgraph of X. We create M as a subgraph for X that initially contains single vertex u. Then, we add minimal number of parents and children of u in X to M (including the necessary edges) that it relies on them to satisfy CAR-dual match conditions.

For any added new vertex to M, we continue to add its minimal set of parents and children in the same way. Clearly, at some point there would be no need to add any extra vertex to M because the existent vertices are all satisfied the CAR-dual conditions. This will certainly happen because of the definition of X. At this point, M is a minimal dual match subgraph because at each step we added only the vertices which their existence were necessary for CAR-dual match conditions. It is also clear that we always kept it a subgraph of X. □

The example illustrated in figure 7.5 can clarify the proposed concept by lemma 7.1. Assuming that the graph depicted in part (a) is a query Q, its maximum perfect subgraph can be a graph like X depicted in part (b) of the figure. First, It is clear that all the vertices

in X are in CAR-dual relationship with the vertices in Q, and X is a result dual match graph. Second, although both vertices 3 and 5 are centers of Q, 3 will be selected as the candidate vertex because it has lower label frequency. Moreover, the radius of Q is 3. Third, the only match to candidate center in X is 5, and a ball centered at this vertex with radius 3 contains all the 9 vertices of X. If we look closer at X, we can see that it is actually a union of three minimum dual match subgraphs. The set of vertices in each of these subgraphs are: {1,2,3,4,5,8,9}, {1,7,5,8,9}, and {2,6,5,8,9}.

Lemma 7.2 *When a query graph Q is a polytree with diameter d_Q, none of its minimum dual match subgraphs on G has a diameter bigger than d_Q.*

Proof: Let us assume that $d_Q = n$ and $M(V_M, E_M)$ is a minimum dual match subgraph with respect to $Q \trianglelefteq^{CD}_{sim} G$ with diameter $d_M = m$. We use proof by contradiction for proving this lemma. Let us assume that $m > n$. It means that there is a shortest path, P, in *undirected* underlying of M that its length is m. Also, let us assume that the distinct vertices on this path are $< v_1, v_2, ..., v_n, ..., v_m >$. Clearly, v_1 is a match to a vertex in Q, like u_1. Because of the definition for M, v_2 must be match to a neighbor (either parent or child) of u_1, like u_2.

If we keep traversing on P, we will find more match vertices in Q. These vertices must be all distinct because first, Q is a polytree; second, if we assume that v_i and v_j, where $j > i$ are match to the same u_i, then regarding the fact that there is no other link between the vertices of P (because it is a shortest path between v_1 and v_m), it means that they belong to different minimum dual match subgraphs inside M; this is in contradiction with our assumption that M is a minimum dual match subgraph. Moreover, adding distinct vertices u_i cannot go beyond n steps because it means the longest path in Q is traversed. As Q is a polytree, u_n cannot have any other neighbor except u_{n-1}; therefor, v_{n+1} must be a match to u_{n-1} and this leads to contradiction as we stated earlier. Hence, m cannot be bigger than n. □

Proposition 7.1 *When Q is a polytree, the set of vertices in MaxPGs returned by $Q \trianglelefteq_{sim}^{CT} G$ is the same as the set of vertices in the result dual match graph returned by $Q \trianglelefteq_{sim}^{CD} G$.*

Proof: It is clear that the result dual match graph returned by $Q \trianglelefteq_{sim}^{CD} G$ is a union of several minimum dual match subgraphs. It is also known that any of these minimum dual match subgraphs has at least one candidate vertex for creating the CAR-tight balls. Moreover, none of these minimum dual match subgraphs has diameter bigger than the diameter of Q according to lemma 7.2. Therefore, every ball and consequently any MaxPG will contain all the minimum dual match subgraphs that share its center. Furthermore, lemma 7.1 implies that a MaxPG cannot have any vertex which is not member of a minimum dual match subgraph. Hence, we can conclude that the set of vertices in MaxPGs is the same as the set of vertices in the result dual match graph. □

Lemmas 7.1 and 7.2 are used to prove proposition 7.1, and then the proposition is used to prove theorem 7.1. Moreover, this propsition indicates that for extracting the induced subgraph, which should be stored for a polytree in the cache, it would be enough to run dual-simulation instead of tight-simulation. It should be noticed that dual-simulation is a preliminary step in calculating tight-simulation; therefore, the extra cost of ball creation would be avoided. For the same reason, it would be enough to run dual simulation when a new query should be compared to the existent polytree to realize if it is a cover-tight match for that polytree.

Theorem 7.1 *Consider G as a data graph and Q_1 as a query graph. Given that P is a spanning polytree of Q_1, and V_P is the set of all vertices in the MaxPGs resulting by $P \trianglelefteq_{sim}^{CT} G$, for Q_1 or any other new query Q_i that is a cover-tight match to P, the set of the vertices, V_R, in the MaxPGs resulting by $Q_i \trianglelefteq_{sim}^{CT} G$ is a subset of V_P ($V_R \subseteq V_P$).*

Proof: As Q_i is a cover-tight match to P, we can conclude that any vertex in Q_i is in a CAR-dual relation with a vertex in P with respect to $P \trianglelefteq^{CD}_{sim} Q_i$. It is also clear that any vertex in V_R is in a CAR-dual relation with a vertex in Q_i with respect to $Q_i \trianglelefteq^{CD}_{sim} G$. As the matching conditions in the latter relation can be considered as more restricted conditions in the former relation, we can conclude that any vertex in V_R is in a CAR-dual relation with a vertex in P. Moreover, V_P contains all the vertices in the maximum dual match set with respect to $P \trianglelefteq^{CD}_{sim} G$ according to proposition 7.1. Hence, $V_R \subseteq V_P$. □

7.6 Empirical studies

We have conducted several sets of experiments to evaluate the effectiveness of the proposed caching technique. We call a query that cannot be answered by cache a *miss query*, otherwise a *hit query*. We calculate hit-rate ratio by finding the percentage of hit queries in a workload of queries. The *response time* of a query is the elapsed time from submitting the query until receiving its results either from cache or main engine.

7.6.1 Experimental setting

We have implemented a basic cache system in Java. A data graph is provided in adjacency-list format from a text file. The queries are also fed to the system by their files, which have the same format. For submitting a workload of queries, the path to the folder of query files is passed to the program.

All the experiments were performed on a machine that has 128GB DDR3 RAM, and two 2GHz Intel Xeon E5-2620 CPUs, each with 6 cores. Moreover, we used JDK-1.7.0_55 to compile and run our program.

We observed that the probability distribution of the label of vertices can greatly affect the performance of the algorithms. Regarding the fact that there is no study on the distribution

of the labels in real-world graphs, we decided not to use any synthesized graph for our tests. Instead, we used three real-world labeled graphs for our experiment: (a) Patent graph [35], a dataset on US patents where each vertex represents a patent and an edge from i to j means that patent i cites patent j. There are about 3.7M vertices and 16M edges in this graph. The label of the vertices are 37 different subcategories of the patents. (b) Citation graph [68], a citation network of papers where an edge from vertex i to vertex j means paper i has cited paper j. It has about 2.2M vertices and 4.3M edges. We selected publishing year of the papers as the label of vertices which means 80 different labels. (c) Amazon product co-purchase network [49], with about 548K vertices and 1.7M edges. Each vertex is a product, and an edge from i to j means that product i is frequently co-purchased with product j. We used the first category of each product as its label, which led to 104 distinct labels for all vertices.

To generate a query graph with guaranteed result in data graph, we randomly extract a connected subgraph from a given dataset. Our query generator has a pair of input parameters $(n, \bar{d})$, where n is the number of vertices in the query, and $\bar{d}$ is the desired average degree of each vertex. The program first randomly picks a vertex of the data graph, and then randomly extracts between 0 and $\bar{d}$ neighbors of this vertex regardless of the edge-direction. It continues adding the local vertices in a BFS-fashion until the requested number of vertices are extracted.

7.6.2 Experimental results

The experiments are categorized in two groups. First, we study the potential speedup that can be achieved by reusing the results of queries. Then, we examine cache-hit ratio and performance improvement achieved by our caching technique for a synthesized workload.

For our experiments, we randomly extract a set of base queries from each data graph. These queries are from five different sizes: $(10, 3)$, $(15, 3)$, $(20, 4)$, $(25, 5)$, and $(30, 5)$. We

randomly extract 100 queries for each size; hence, the base query set for each data graph contains 500 unique queries. As these queries are directly extracted from data graph, they all have some results.

Speedup

We use the base query set of each data graph to test the speedup that can be achieved by our caching mechanism. We first submit each new individual query to the system and have their corresponding cache contents been created. Then, we submit all the queries again and measure speedup in their response time caused by the cache system. Table 7.1 displays the average speedup of queries for each data graph. The speedup of each query depends on many features in both data graph and query graph. Intuitively, one can expect a relation between the number of vertices in the output of a query and its caching speedup. The Pearson-Product-Moment Correlation of these two variables are displayed in table 7.1. The degree of freedom in this experiment is 488. Considering significance level 0.01 and two-tailed probabilities, the critical value is 0.115. It indicates a statistically significant correlation between the two variables. We can use this property to determine a criterion for the queries that are not worth storing their results in the cache.

Charts in figure 7.6 show the caching speedup of base queries versus the number of vertices in their results, which is normalized with the total number of vertices in the data graph. After drawing the Power-regression line of the data points, we can select 0.7% as a common optimal criterion for storing the result of queries. In other words, we do not consider the queries that the number of vertices in their results is bigger than this limit for storing in the cache. In our experiments, they are 9.6%, 1.2%, and 8.8% of queries respectively for Amazon, Citation, and Patent graphs.

Table 7.1: Speedup achieved by caching

Data graph	Amazon	Citation	Patent
Average speedup	99	487	887
Meadian speedup	60	92	41
Pearson Correlation between the number of vertices in the results and the caching speedup of individual queries	-0.37	-0.39	-0.29

Cache Hit

To evaluate the proposed caching technique, an appropriate workload of queries is needed. Because of the lack of such a workload, we had to create our own. In general, any caching technique is useful when there are popular queries that are asked during the time. In our context, pattern containment technique improves cache hit ratio with utilizing the cache to answer not only the same repeated queries but also similar queries.

In order to create an appropriate *test-set* of queries for each data graph, we use the 500 queries randomly extracted in the previous experiment as the set of base queries; i.e.; they are 5 different sizes, each 100 queries. Then, we create a set of similar queries by incrementally adding a vertex to each base query and connecting the new vertex to a random number of old vertices with random directions. The label of the new vertex is also randomly selected from the set of available labels. We continue this incremental process until 5 steps; therefore, we synthesize 5 new query graphs for each base query. At the end, we have 2500 newly synthesized queries.

This approach for generating new queries resembles the behavior of real users who tweak their old queries to explore more interesting patterns. It is noteworthy to mention that there is no guarantee that newly synthesized queries meat the similarity condition (cover-tight match) proposed in this paper in order to enjoy pattern containment. Moreover, many of

these synthesized queries may not have any match in data graph, similar to the real-world situation.

We form the *test-set* of queries from the union of base queries and synthesized queries; hence, there are 3000 unique queries in the test-set. The workload is a sequence of queries randomly picked from the the test-set. We do not eliminate a selected query from the set; therefore, there is a chance that it is selected again. To provide a reasonable chance for each unique query to be picked once, we set the number of queries in the workload 5 times bigger than the number of unique queries; that is, the workload comprises of 15000 queries.

Figure 7.7 illustrates the measured hit-rate ratio versus different cache sizes. The horizontal axis displays the cache size based on its ratio to the total number of unique queries in the test-set, and the vertical axis shows the ratio of the number of queries hit the cache to the total number of queries in the workload. *Least Frequently Used (LFU)* policy is implemented for *replacement* when the cache becomes full. As it is expected, the hit-rate ratio goes towards saturation after some point.

Improvement in the average response time of CAR-tight simulation queries versus the cache size is displayed in figure 7.8. Table 7.2 presents more detail information about this experiment. In this table, the *average response time without cache* represents the situation when the size of the cache is zero, and the *average response time with cache* corresponds to unlimited cache space. It can be observed that the behavior of Citation graph has been slightly different from the other two graphs; e.g., slower increase in the hit-rate ratio, and higher ratio of response times comparing with and without cache. This difference can be explained using features of the graph; i.e., Citation graph is sparser than the others and has less number of undirected cycles. These features cause shorter response times, less number of vertices in the results, and less similarity among the queries in the *test-set*. Less similarity among the queries will lead to lower hit-rate ratio, and having smaller subgraphs corresponding to the polytrees makes response times from cache even faster.

Table 7.2 also shows that the main overhead of the cache system, which is search time, is negligible in our tests. Average store time includes the spent times for extracting the polytree, finding its CAR-tight simulation results in data graph, and extracting corresponding induced subgraph. These steps can be executed in parallel to the main engine; hence, the store time is not considered in the response time.

Table 7.2: More detail information about cache behavior

Data graph	Amazon	Citation	Patent
Average response time without cache, CAR-tight (msec)	905	607	3482
Average response time with cache, CAR-tight (msec)	329	14	306
Average response time without cache, $\trianglelefteq_{iso}$ (sec)	591	49	200
Average response time with cache, $\trianglelefteq_{iso}$ (sec)	130	1.2	175
Average number of vertices in cache for each polytree	990	478	5281
Average search time in cache (msec)	2	6	2
Average store time in cache (msec)	567	793	4315

We have also tested cache performance for subgraph isomorphism. We used *DualIso* algorithm [62] to find the first 1000 subgraph isomorphic matches. Exactly the same caching mechanism used in this test; however, we ran *DualIso* on data graph when the query did not hit the cache, and on its correspondent subgraph when it hit the cache. The average response time versus the size of the cache is illustrated in figure 7.9. Similar to the tight simulation, the average response times with and without cache are also reported in table 7.2.

Figure 7.10 shows the warm-up behavior of the cache for different data graphs. The horizontal axis is the percentage of queries submitted from the workload, and vertical axis is the cumulative percentage of the queries which hit the cache. As one may expect, there is a transition phase for warm-up and then the curve becomes linear. Comparing with figure 7.7, it can also be observed that the pace of the transition is faster for a data graph with higher cache-hit ratio.

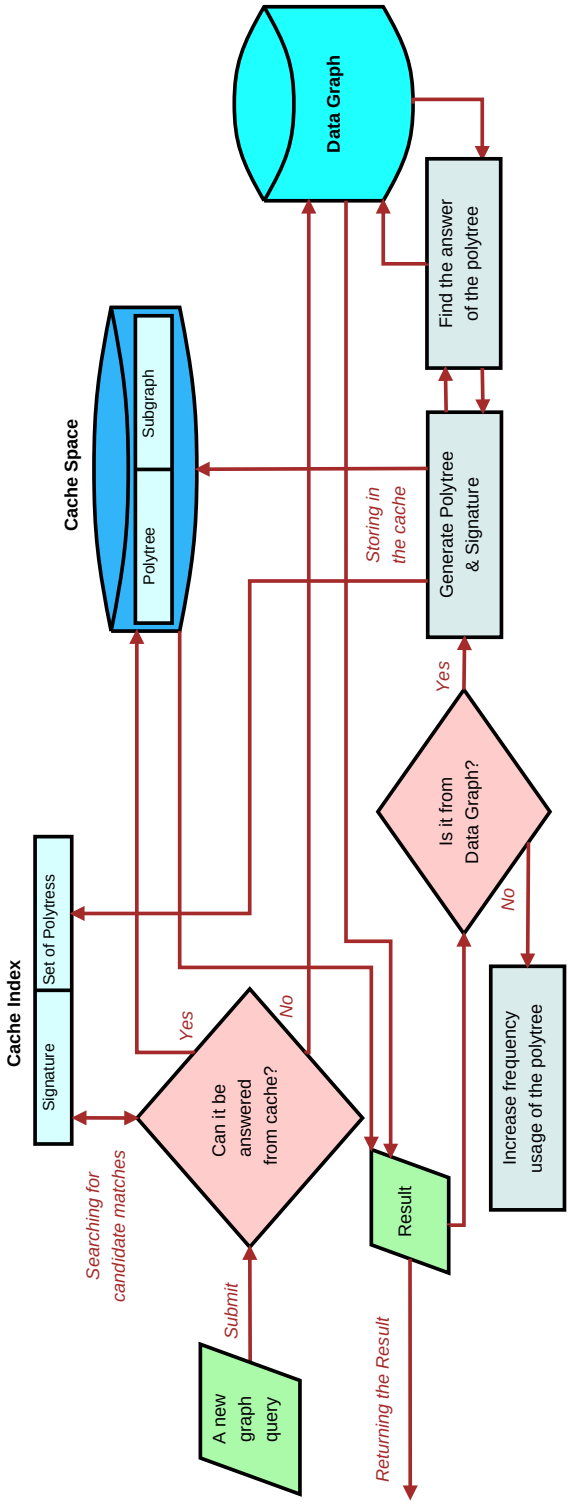


Figure 7.3: The overall procedure of the proposed mechanism

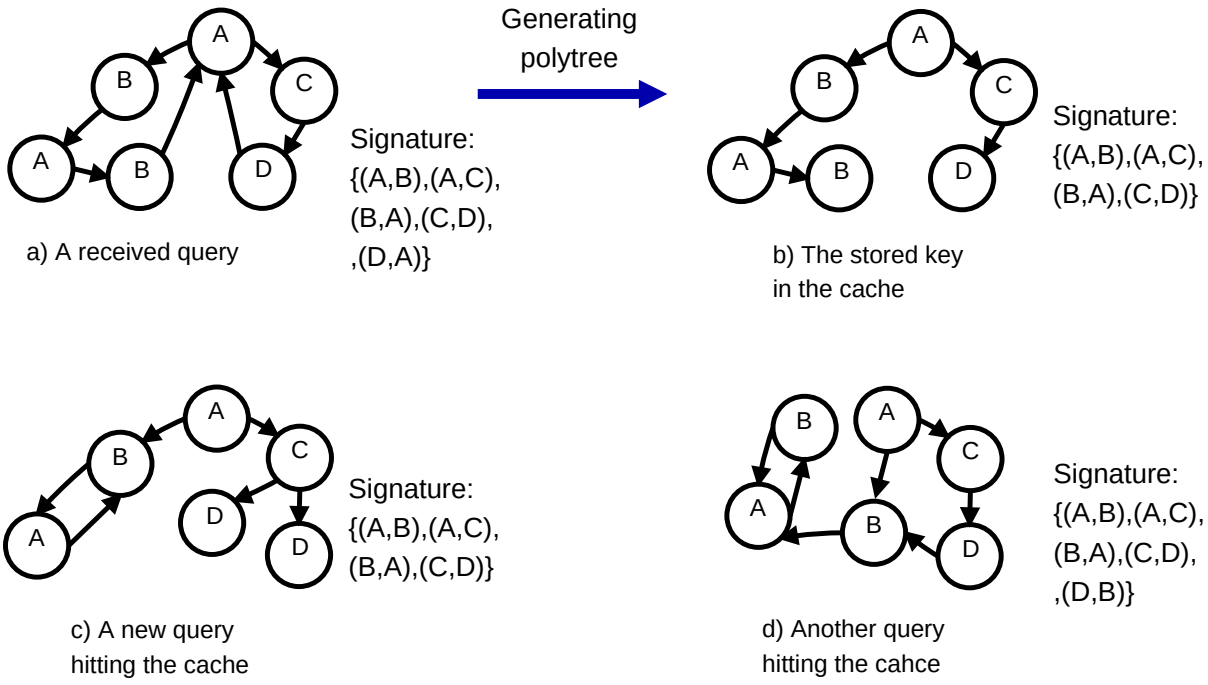
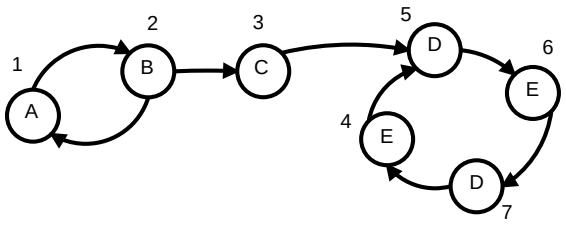
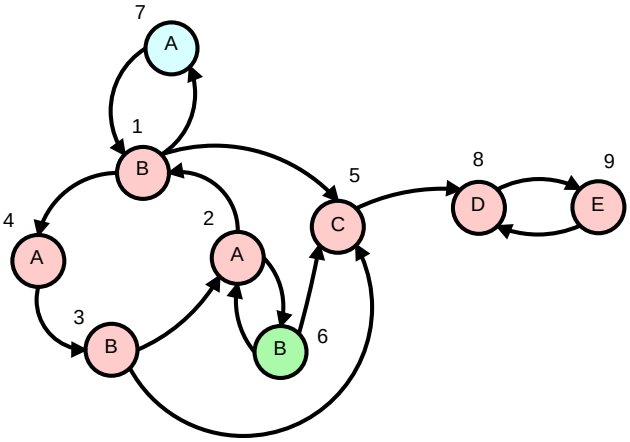


Figure 7.4: An example for caching the graph queries

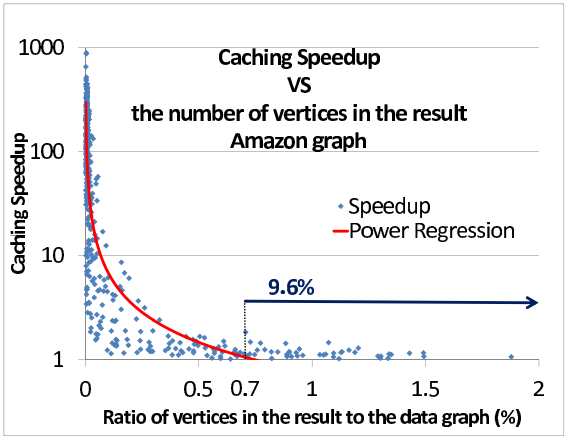


a) A query, Q

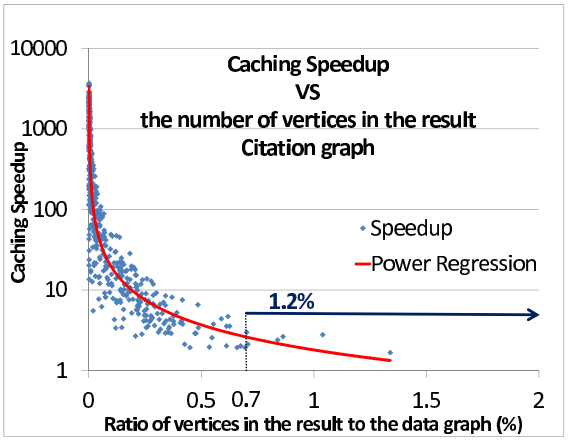


b) A maximum perfect subgraph, X

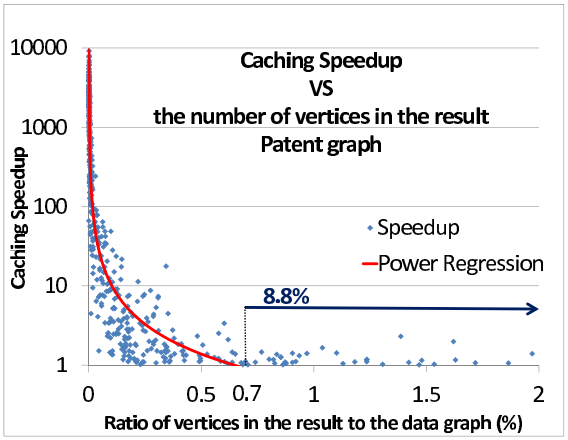
Figure 7.5: An example about a MaxPG and its minimum dual match subgrapgs



(a) Amazon graph



(b) Citation graph



(c) Patent graph

Figure 7.6: Caching speedup

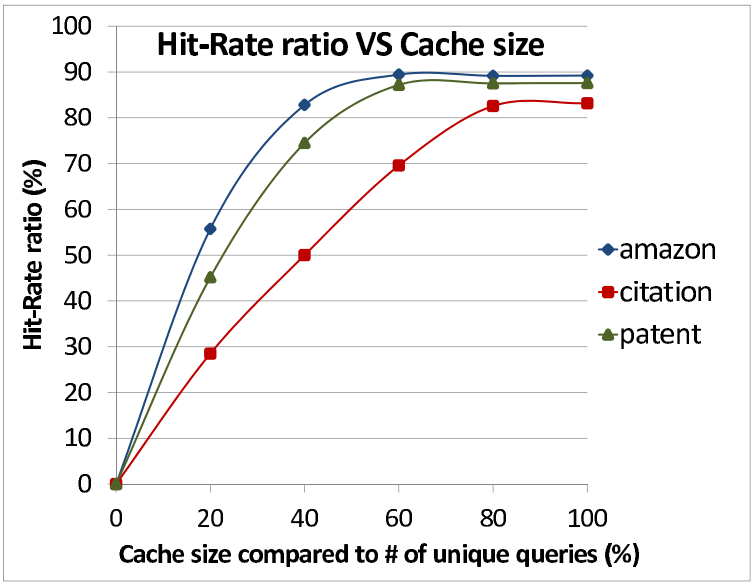


Figure 7.7: Hit-rate ratio

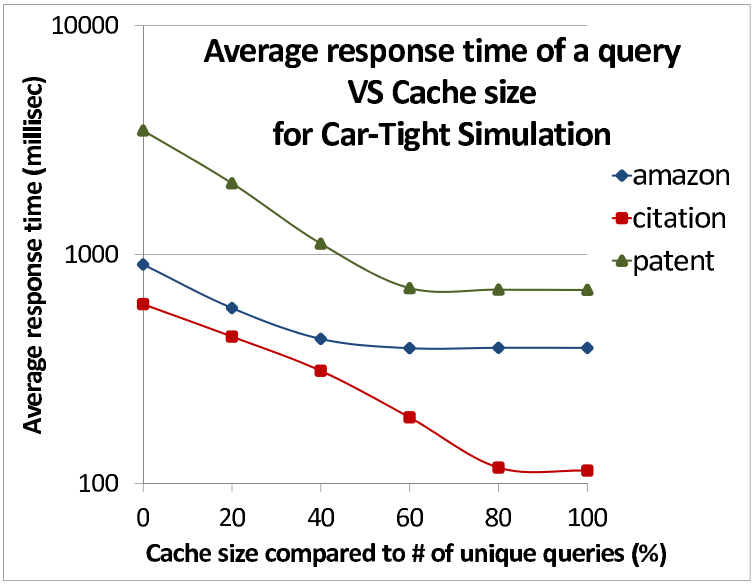


Figure 7.8: Average response time for CAR-tight simulation

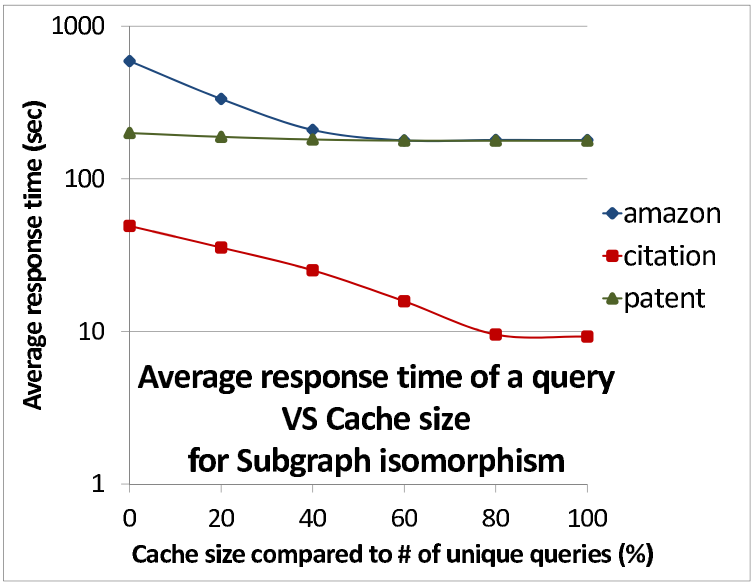


Figure 7.9: Average response time for subgraph isomorphism

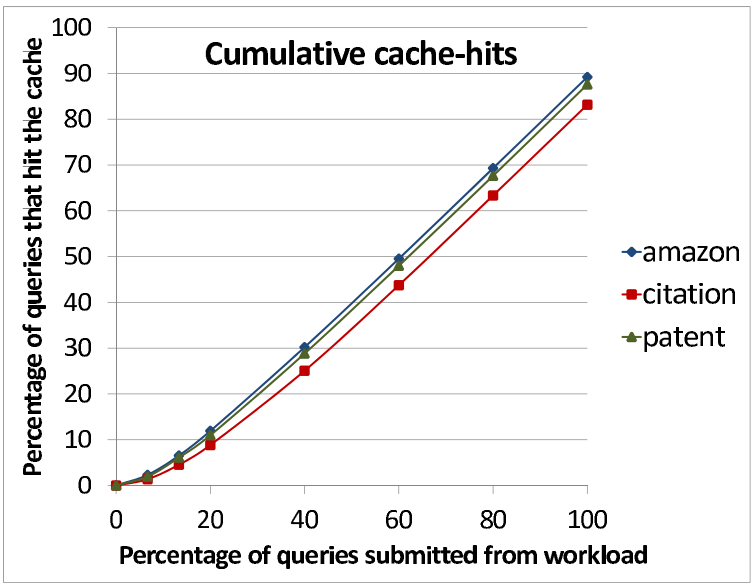


Figure 7.10: Cumulative cache hits

Chapter 8

Related Work

In this chapter, the most important related works are reviewed. We introduce several pattern matching models which is related to the context of this research, but not introduced earlier in this document. In this research we have only used vertex-centric BSP programming model for designing our distributed algorithms. Here, we review some other alternative distributed computing models which can be considered for future work. Furthermore, we will also present a brief review of graph partitioning which can be considered important for distributed graph algorithms. The status-of-the-art of related work for caching techniques is presented at the end.

8.1 A few other new pattern matching models

The problem of subgraph pattern matching and its different solutions is exhaustively studied in the literature. We surveyed related work in two categories: pattern matching models, and distributed algorithms for pattern matching on massive graphs.

There are varieties of models for graph pattern matching. Many matching models find similarity between graphs based on exact structural matching like subgraph isomorphism

and its approximate algorithms. On the other hand, there are some matching models that take into account the meaning of attributes of vertices and edges and their relationship in addition to graph structure. We are interested in the recent type of matching which is also called semantic graph matching [33].

A few graph pattern matching models introduced during recent years to response the emerging need of applications to find a proper semantic graph match like p-homomorphism [26] and bounded simulation [25]. Our work closely looks at the new extensions of graph simulation, namely dual simulation and strong simulation [52].

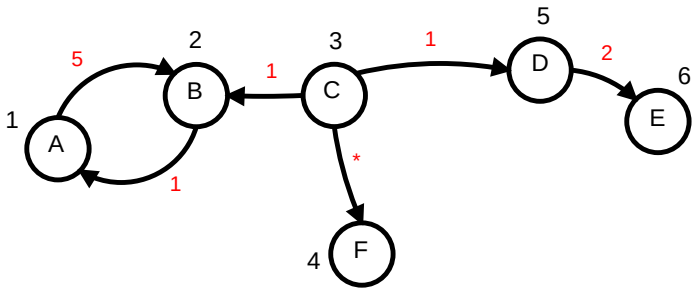
8.1.1 Bounded Simulation

Bounded simulation [25] is an extension of graph simulation where instead of edge-to-edge mapping there is edge-to-path mapping. That is, similar to graph simulation each vertex of a data graph should satisfy children condition to be in match relation with a vertex in a query graph; however, the children can be in a bounded distance instead of being the adjacent a child. It is noteworthy to mention that graph simulation is a special case of bounded simulation where all the edges have bound labels "1". A cubic algorithms is proposed in [25] for finding the results of bounded simulation of a query graph on a data graph. The example displayed in figure 8.1 can clarify the concept.

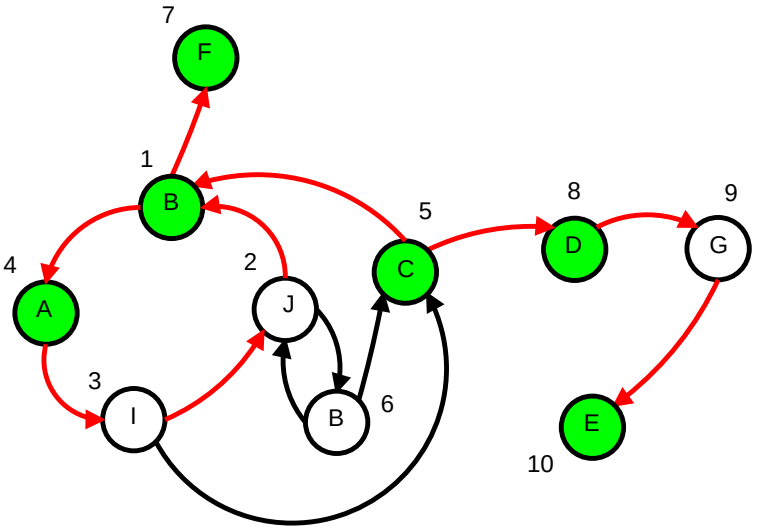
Part (a) of figure 8.1 is a sample query graph for bounded simulation. In this type of simulation, each edge of the query has a numeric label which represents the maximum length of a path that can be matched to the edge. When the label is specified as "*" instead of a number, it indicates no bound of the length of matching path.

In our example, existing "*" as the label on the edge from vertex 3 to vertex 4 means that the match vertex to 3 must have a path to a vertex that is match to 4. The same way, label 5 on the edge from vertex 1 to vertex 2 means that match vertex to 1 must have a path with length not longer than 5 to a vertex that is match to 2. The result of bounded

simulation of this query on the data graph is displayed on part (b) of the figure. This result contains vertices 1, 4, 5, 7, 8, 10.



a) A query graph, Q



b) A data graph, G

Figure 8.1: An example about bounded simulation

8.1.2 p-homomorphism

P-homomorphism (p-hom) [26] is an extension to graph homomorphism. Similar to relation between graph homomorphism and graph isomorphism, 1-1 p-homomorphism (1-1 p-hom)

can be defined as a special case of p-hom, which can be considered as an extension to graph isomorphism. An important notion for defining p-hom is called *similarity matrix* `mat()`. Consider two labeled graphs $G_1(V_1, E_1, L_1)$ and $G_2(V_2, E_2, L_2)$. For each pair of $(u, v) \in V_1 \times V_2$, $\texttt{mat}(u, v)$ is a real number in $[0, 1]$, which indicates the similarity of the two vertices. In other words, $\texttt{mat}(u, v) = 0$ means that there is no similarity between vertices u and v. On the other hand, $\texttt{mat}(u, v) = 1$ means that these two vertices are completely similar.

Graph G_1 is p-homomorphism (p-hom) to graph G_2 with respect to a similarity matrix `mat()` and a threshold ξ, if there is a function σ from each $u \in V_1$ to a $v \in V_2$ such that (1) $\texttt{mat}(u, \sigma(u)) \geq \xi$, and (2) for each edge (u, u') in E_1 there is a path $v \rightsquigarrow v'$ in E_2 such that $v = \sigma(u)$ and $v' = \sigma(u')$; i.e., each edge in G_1 is mapped to a path in G_2. 1-1 p-homomorphism (1-1 p-hom) is a special case that σ defines a 1-1 mapping (injective) from vertices in V_1 to V_2. It should be noticed that subgraph isomorphism is a special case of 1-1 p-hom.

Figure 8.2 shows an example for explaining these two models. In this example, we assume that $\texttt{mat}(u, v) = 1$ if u and v have the same label, and $\texttt{mat}(u, v) = 0$ otherwise. One can realize that there is a p-hom mapping from $G1$ to $G2$, but there is no 1-1 p-hom mapping from $G1$ to $G2$ because the vertices 2 and 4 in $G1$ are mapped to the single vertex 1 in $G2$. In comparison, there is not only a p-hom mapping from $G1$ to $G3$, but also a 1-1 p-hom mapping.

Fan et al. have shown in [26] that the decision problems for p-hom and 1-1 p-hom are NP-complete. They have even proved that these problems are approximation-hard. Nevertheless, they have proposed approximation algorithms for these two problems, and have examined their effectiveness for Web site matching using real-life and synthesized datasets.

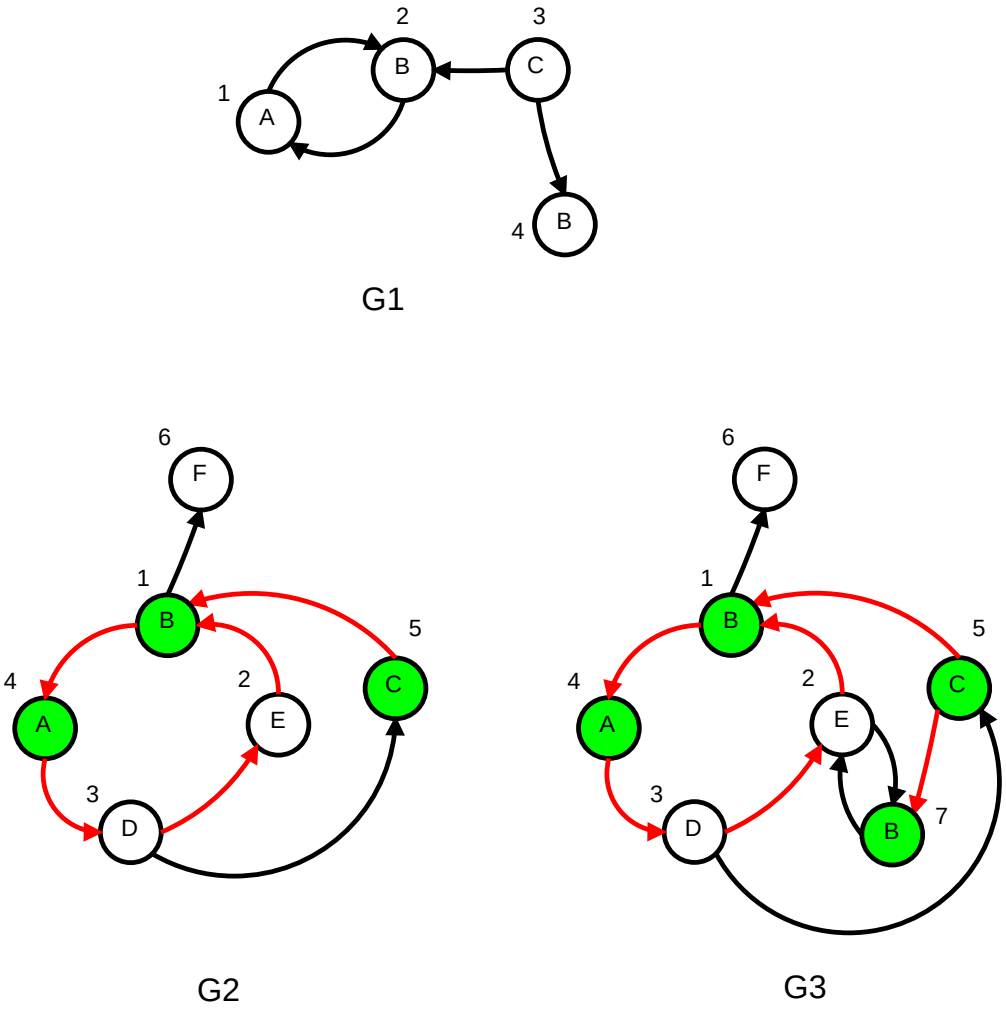


Figure 8.2: An example about p-hom and 1-1 p-hom

8.2 Alternative distributed computing models

In [52] that introduces dual and strong simulation, the idea of a distributed algorithm is talked in abstract, and it is not based on a vertex-centric approach. There are also very interesting theoretical investigation about different types of simulation presented in [23], but it never becomes close to a distributed algorithm for strong simulation.

In [54] a distributed algorithm for graph simulation is proposed. They have also analyzed distributed algorithms for graph simulation and identified three complexity measures for their analysis: (1) visit time, which measures maximum visiting time of a machine in a cluster system and indicates the complexity of interactions, (2) makespan, which is response time to the query from submitting query until when its answer is ready, (3) data shipment, which is the total size of messages exchanged between machines of the cluster system during computation. They have implemented and tested their algorithm on a cluster of 16 machines; however, neither the algorithm nor its implementation is adapted to a widely in use cluster computing platform like GPS, but it is implemented all in Python.

There are also some approaches proposed based on indexing to increase efficiency of answering graph pattern matching queries; for example, the framework proposed in [42], or the algorithm introduced in [19]. However, creating and storing indexes for massive data graphs does not seem to be feasible.

In [67] a distributed algorithm is introduced for finding isomorphic subgraph matches of a given query graph in a huge data graph. They deploy graphs on Trinity [13] which provides a distributed memory cloud environment. This work focuses on efficient web-scale graph processing, so it avoids exhaustive usage of indices which is very common in most of graph processing systems. Indeed, they only use a simple string index which maps node labels to node IDs. It is argued that the time needed for constructing indices and the capacity for their storage would be infeasible in web-scale graph processing. Index size and time of the proposed approach, in addition to the query processing time, is compared with estimation for some other existent systems. Conceptually similar to our approach, their main idea is based on graph exploration, but they cannot avoid expensive joins because of the intrinsic limits of subgraph isomorphism.

Fan et al. have studied the theoretical concerns about distributed graph simulation in [28]. They have a new notion called *parallel scalable* to measure scalability of a distributed

algorithm for graph simulation. Given a pattern query Q and a data graph G that is fragmented into $(F_1, F_2, ..., F_n)$ and distributed among several sites $(S_1, S_2, ..., S_n)$, they study the conditions that an algorithm A for distributed graph simulation can be parallel scalable, according to their definition, in *response time* and *data shipment*. The distributed algorithm A is parallel scalable in response time if its computation time is determined only by the largest fragment, F_m, and the size of Q. Moreover, it is parallel scalable in data shipment if its total required data shipment is a function of the size of Q and the number of fragments.

Then, they prove an *impossibility theorem* that indicates it is impossible to have such an algorithm that is either parallel scalable in response time or data shipment. Nevertheless, they show that the response time of a distributed algorithm for graph simulation can be bounded to three parameters: $|Q|$, $|F_m|$, and $|V_f|$, which is the number of vertices in G with edges across different fragments. Also, they show that data shipment can only depend on Q and the number crossing edges $|E_f|$. At the end, they design and examine a distributed algorithm for graph simulation which considers all these parameters.

Ma et al. have designed an implemented a distributed algorithm for strong simulation. They do not use any general platform for their implementation. Instead, they have implemented their own simple platform, which consists of two main modules: *Coordinator module* and *Site module*. Their algorithm completely rely on the locality conditions of strong simulation. They assume that the partitions of a data graph is distributed among the nodes of a cluster. The single *Coordinator*, running on the master node, is responsible to distribute the query graph among all the *Site* processes, which are running on the client nodes of the cluster.

Each ball can be processed independently after creation. All the balls can be built locally except those that need vertices in the other partitions. To address the problem, they create part of the balls centered on the *boundary vertices* of each partition and send them to all the Sites the center has direct neighbor (parent or child). After all the Sites receive all

the exchanged partial balls, they update their partitions by incorporating the shipped balls. After this point, all the balls can be created and processed locally on each Site. As an optimization technique to reduce the number of shipping balls, they find a *maximum partial dual match relation* on each partition. It is similar to the ordinary maximum dual match relation except for the boundary vertices where only their labels will be checked for matching not their parent and child relationship conditions.

Following, we review a few generic distributed programming environment for specifically provided for distributed graph algorithms beside the vertex-centric BSP that was explained earlier.

8.2.1 Asynchronous Vertex-centric

Although designing and implementing a distributed algorithm on an asynchronous platform is more difficult than a synchronous one, it may achieve higher efficiency. In general in an asynchronous platform, it is upon the algorithm designer to take care of problems like data races and deadlocks. Nevertheless, the authors of GraphLab platform [51] attempt to provide an asynchronous distributed environment for graph algorithms (iterative machine learning algorithms in general) that as much as possible insulate users from the complexities of such a system. It is implemented in C++, but it has also API in Python.

The abstraction model of GraphLab consists of several parts [50]: *data graph*, *update function*, *scheduling primitives*, *consistency model*, and *sync mechanism*. Similar to what is presented in Pregel-like systems each vertex is a computing unit; i.e., it is a vertex-centric programming model. According to the definition of its authors, the *data graph* in its abstraction represents the data blocks and computational dependencies.

Moreover, the *update functions* describe local computation. The *scheduling primitives* state the order of the computations and may rely on the data. The data *consistency model* specifies how much the local computations can overlap. The *sync mechanism* is also provided for the users to be able to aggregate global states.

Signal/Collect [66] is another parallel graph processing platform that provides an asynchronous vertex-centric programming environment. It is implemented in Scala. Similar to GraphLab, each vertex is a computing unit. Communication between vertices occurs through the edges using signal and collect mechanism. Edges signal and vertices collect. When an edge signals, it computes a message based on the state of its source vertex. This message is sent through the edge to the target vertex. When the target vertex collects, it uses the received message to update its state. All these signals and collects in parallel across the graph until all the signals have been collected and all the vertices have been converged to their final state.

8.2.2 Graph-centric

For running any distributed graph algorithm, it is necessary to partition the data graph among different processing elements. In vertex-centric programming models, either synchronous like Pregel [55] or asynchronous like GraphLab [51], an algorithm should be designed from perspective of a vertex. It means the partitioning information is hidden from the users. However, this information can be used for many algorithms to increase efficiency; i.e., shortening their running time by reducing either network communication that is needed for synchronization (e.g. in Pregel) or scheduling overhead that is imposed by ensuring data consistency (e.g. in GraphLab).

To address the mentioned problem, Tian et al. have proposed a graph-centric programming model in [69]. Under this model, the programmer can utilize partition structure of the graph to avoid unnecessary message passing or scheduling machinery. They have imple-

mented their model, called Giraph++, as an add-in for Apache Giraph [4] and have examined its applicability for several graph algorithms.

The proposed graph-centric model is indeed an extension to the vertex-centric model. In their model, they define vertices in very similar way; however, they divide them to two group: *internal vertices* and *boundary vertices*. Every vertex of the real data graph is considered internal vertex for exactly one partition that it belongs to. This partition is called the *owner* of the internal vertex. A local copy of an internal vertex on another partition, where it has an edge to a vertex of that partition, is called a boundary vertex on that partition. One can imagine that a vertex may have copies on several partitions. Moreover, they provide the compute method for a partition class in contrast to the vertex-centric platforms where the compute method belongs the vertex class. That is, subgraph partitions are programming units, and there would be no need for the vertices of the same partition to exchange any message. Indeed, all the communication will be among different partitions and through boundary vertices.

8.3 Graph partitioning

Before a distributed graph algorithm can be executed on a system with underlying distributed architecture, the graph should be partitioned among different nodes of the system. The way the graph is partitioned can greatly affect the efficiency of the algorithm.

The classical definition of graph partitioning, in simple words, is dividing the vertices of a graph to roughly equal groups while the number of edges among these groups is minimal. A more formal definition of this concept follows [18].

Given a graph $G(V, E)$, a p-way partition of the graph is a mapping $\Phi : V \rightarrow [1...p]$ of its vertices into p subsets $S_1, S_2, ..., S_p$ where $\bigcup_i S_i = V$ and $S_i \bigcap S_j = \emptyset$ where $i \neq j$. Every partition generates a set of cut edges, E_c, defined as the subset of E whose endpoints lie in distinct partitions; formally:

$$E_c = \{(v_i, v_j) | (v_i, v_j) \in E, \Phi(v_i) \neq \Phi(v_j)\}$$

The weight of each subset, $|S_i|$, is defined to be the number of vertices mapped to that subset by Φ.

Given a graph as input, the graph partitioning problem seeks to find a p-way partition in which each subset contains roughly the same number of vertices ($|S_i| \leq \lceil |V|/p \rceil$) and the number of cut edges, $|E_c|$, is minimized. Usually, each partitioned subset represents data and computation that should be assigned to a single processor, and the cut edges represent the inter-processor communication required by the distribution of workload among different processors. In other variations, it is possible to assign weight to vertices and edges; therefore, instead of the same number of vertices, the total weight of vertices in each partition should be equal. The same way, instead of minimizing the number of cut edges, the their total weight should be minimized.

The classic definition is based on using edge separator which is a group of edges whose removal breaks the graph into disjoint subsets. A related problem tries to break the graph into subsets using a vertex separator of minimum size.

Figure 8.3 displays an example for graph partitioning. In this example, there are the graph is split to four partitions. There are five cut-edges in this partitioning, which are: $(4, 12)$, $(6, 5)$, $(9, 22)$, $(10, 16)$, and $(17, 18)$.

Multi-level algorithms and spectral heuristics have been shown to be very effective for partitioning graph abstractions derived from physical topologies, such as finite-element meshes

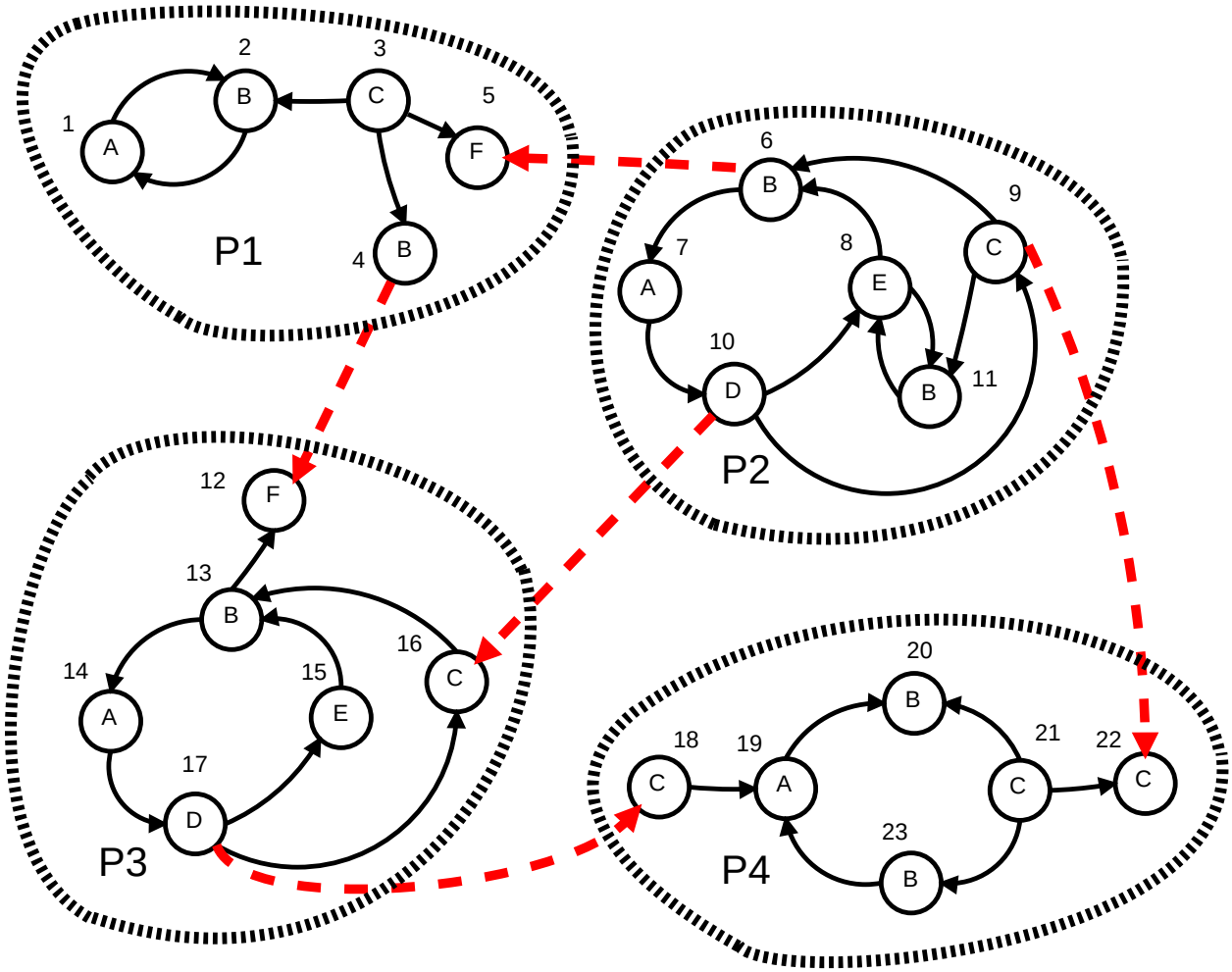


Figure 8.3: An example for graph partitioning

arising in scientific computing. Software packages implementing these algorithms (e.g., Chaco [39] and Metis [45, 46]) are freely available, computationally efficient, and produce high-quality partitions in most cases.

Multi-level partitioning algorithms, first obtain a sequence of successive approximations of the original graph [9]. Each of these approximations represents a problem whose size is smaller than the size of the initial graph. This procedure continues until the graph contains a manageable number of vertices. At this point, these algorithms compute a min-cut par-

titioning of that graph. Since the size of this graph is quite small, even simple algorithms lead to reasonably good solutions.

The final step of these algorithms is to take the partitioning computed at the smallest graph and use it to derive a partitioning of the original graph. This is usually done by propagating the solution through the successive better approximations of the graph and using simple approaches to further refine the solution. Since the successive finer graphs have more degrees of freedom, such refinements improve the quality of the resulting partitioning.

8.3.1 Challenges

Applying the classic definition of graph partitioning faces some challenges in real-world applications. The main challenges are listed as follows.

Time complexity

When the desired number of partitions for graph partitioning is selected as $p = 2$ the problem would be equivalent to well-known Minimum Bisection problem which is NP-complete. Due to the importance of the problem much effort has been made to develop efficient heuristics and approximation algorithms. If p is not constant the problem is hard to approximate within any finite factor [11].

Although extensive efforts are made during the last decades to find approximate algorithms and heuristics to efficiently solve graph partitioning problem, it is still an active area of research. Because of its importance graph partitioning was one of the implementation challenges of the 10th DIMACS hold in February 2012 [1].

Shortcomings of the classic edge-cut metric

Classic approach of minimizing cut edges has several major flaws [38]. First, although it is not widely acknowledged, cut edges are not always proportional to the total communication

volume. Second, in addition to the message size, the latency of communication is also an important factor in a parallel computer. Graph partitioning approaches try to minimize the total volume but not the total number of messages. However, depending on the machine architecture and problem size, message latency can be more important than message volume.

Third, the slowest processor generally limits the performance of a parallel application. Even when the computational work is well balanced, the communication effort might not be. Rather than minimizing the total communication volume or even the total number of messages, we may instead wish to minimize the maximum volume and/or number of messages handled by any single processor.

Last, on many machine architectures the time to send a message depends upon the distance between the sending and receiving processors. Although most modern machines have some mechanism that enables a single message to travel quickly between distant processors, the communication network is usually handling many messages simultaneously. A message between distant processors ties up many wires that cannot be used by other messages. To avoid message contention and improve the overall throughput of the message traffic, it is preferable to have communication restricted to nearby processors.

Small-world Network Analysis and Partitioning

Many real-world graphs (networks), especially graphs corresponding to social networks, have a fairly small diameter; this phenomenon is called small-world [72]. Partitioning of these types of graphs using traditional edge-cut approach will not result a good computational balance among the nodes of a distributed memory system. SNAP [14] is a parallel network analysis framework which is designed for efficiently partitioning of massive small-world graphs. In its core it uses graph clustering methods for partitioning.

A key problem in social network analysis is that of finding communities, dense components, or detecting other latent structure. This is usually formulated as a graph clustering

problem, and several algorithms are suggested for finding communities and also several criteria have been proposed for measuring the quality of their result. SNAP finds graph clusters based on modularity measure. Intuitively, modularity is a measure that is based on optimizing intra-cluster density over inter-cluster sparsity [8].

Power-law graphs

Many real world graphs like Web-graph, instant messenger graphs, biological networks, and various social networks are power-law graphs. A power-law graph is a graph whose degree distribution follows a power-law function [9]. More precisely, a function of the form $f = \alpha d^{\beta}$, where f is the number of vertices whose degree is d and $\beta < 0$. These graphs have a large number of vertices with very low degree and a few vertices with relatively high degrees. Classic graph partitioning approaches do not create very well balanced subgraphs for graphs that follow power-law property. Abou-Rjeili et al. [9] present new multilevel graph partitioning algorithms that are specically designed for partitioning graphs whose degree distribution follows a power-law curve. They use graph clustering methods to find well-balanced partitions.

8.4 Caching techniques for subgraph pattern matching

There are rare previous studies about caching techniques for subgraph pattern matching queries. In the field of semantic web, many RDF engines cache intermediate results of SPARQL queries in order to speedup the computation of other queries when they have common triple patterns [47]. These techniques rely on triple structure of RDF datasets and the fact that a SPARQL query is mainly composed of triple patterns. Then, they try to mitigate the cost of expensive joins between the results of common triple patterns.

Martin et al. [56] have proposed a caching system for SPARQL queries. Nevertheless, their cache system is *content-blind*; i.e., a new query can be answered using the cache only if it is identical to an old query stored in the cache. Indeed, their contribution is about cache maintenance; that is, updating the cache contents according to the changes in the underlying knowledge bases.

A *content-aware* caching technique for SPARQL queries is introduced in [65]. The authors propose a few conditions for containment checking of conjunctive SPARQL queries with simple filter conditions. The first difference from our work is that their work is about SPARQL queries. Properties of SPARQL queries, which fall in the category of path pattern queries, are different from the type of subgraph pattern queries that we study in this research. Because of the nature of SPARQL queries and the way they store their results, they also need to define *evaluability* notion because containment conditions cannot guarantee that a new query can be answered using the cache contents.

Another related work has been published very recently in [27]. The authors have investigated pattern containment problem for *graph simulation* [40] and *bounded simulation* [25] models. They statically generated a set of *views* from a given data graph. Each view represents an interesting part of the data graph. Then, they define their containment conditions and show how the queries can be evaluated to learn if their answers are contained in a view. Their experiments eventually shows that the response time of the queries will improve when they can be answered using initially defined views. The main difference to our work lies in the intrinsic differences of *view* and *cache*. A cache space for graph queries has a dynamic characteristic; therefore, new issues like cache size, cache-hit ratio, and query replacement arise that are not considered under concept of views. Moreover, our work is based on a different pattern matching model that suits a completely different set of applications.

Chapter 9

Conclusions and Future Work

In the recent years, graph analytics on massive graphs has become an important subfield of big data analytics especially for applications related to social networks. A good number of distributed programming platforms are implemented specifically for *big graph analytics*, such as Pregel [55], GPS [61], Apache-Giraph [4], GraphLab [51], Pegasus [44], GraphX [75], and Signal/Collect [66]. Because of the dramatical increase in the size of the graphs, the old graph algorithms are not efficient for solving the new problems. Therefore, it is necessary to revise them or design new models and algorithms that suit better to the new problems.

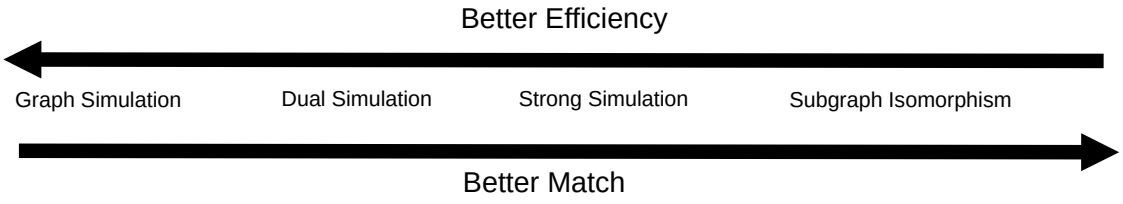
One of the important problems in graph analytics is subgraph pattern matching [33]. Subgraph pattern matching has been increasingly used in analysis of social networks [17, 25]. Although it is extensively studied, the search for efficient methods of answering subgraph pattern matching still remains open. Using *graph simulation models* is relatively a new approach for subgraph pattern matching. Different models are introduced in this family during recent years that are mainly aiming to provide alternative solutions to traditional subgraph pattern matching models [31, 32, 52, 25, 26, 74]. It is shown that these new models can be very useful for new applications like analyzing social networks because of their promising properties [17]. The first and the most important property of these models

is the tractability of their algorithms for finding all matches; in comparison, the algorithms for subgraph isomorphism or subgraph homomorphism are NP-complete. Moreover, the number of subgraph results returned by these models is linear to the number of the vertices in a data graph, while it can be exponential for the traditional models. Also, *graph simulation models* can return meaningful results for new applications that are not captured by traditional subgraph pattern matching models [54, 17, 24, 23, 58, 30].

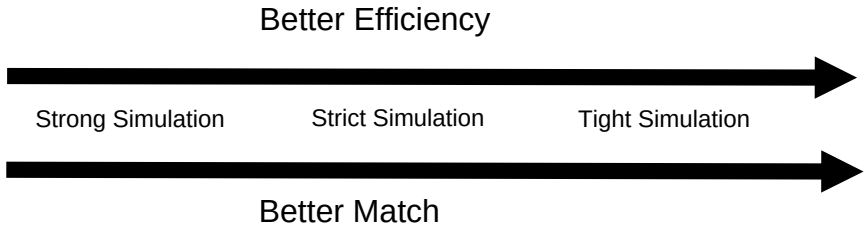
In this research we introduce two new graph pattern matching models in the family of graph simulation, called *strict simulation* and *tight simulation*. These models are indeed smart improvement of a model called *strong simulation* [53], and can be considered as alternatives to subgraph isomorphism on massive graphs. Figure 9.1 compares several related pattern matching models in this family with each other and with subgraph isomorphism in terms of both efficiency and the quality of results. It is traditionally thought that when the result of a pattern matching model becomes closer to subgraph isomorphism, the corresponding algorithm will become computationally less efficient. This behavior is depicted in figure 9.1a for graph simulation, dual simulation and strong simulation. However, we show that moving from strong simulation towards strict simulation, and then tight simulation not only improves efficiency, but also makes the results more stringent; i.e., their results become closer to subgraph isomorphism (figure 9.1b).

Furthermore, we have introduce the concept of *cardinality restriction* for dual simulation. Adding this restriction to dual simulation does not increase the time complexity of its algorithms, but improves its expressiveness. Moreover, new version of dual simulation, called *CAR-dual*, can be used to make updated versions of the other models that are defined based on dual simulation; i.e., strong simulation, strict simulation, and tight simulation. These updated models become more stringent in comparison to their older counterparts.

It seems that it is still possible to improve the available pattern matching models in the graph simulation family. For example, a straightforward but important extension to the



(a) Comparison of $Q \trianglelefteq_{sim} G$, $Q \trianglelefteq^{D}_{sim} G$, $Q \trianglelefteq^{S}_{sim} G$, and $Q \trianglelefteq_{iso} G$



(b) Comparison of $Q \trianglelefteq^{S}_{sim} G$, $Q \trianglelefteq^{\Sigma}_{sim} G$, and $Q \trianglelefteq^{T}_{sim} G$

Figure 9.1: Comparing different pattern matching models

current graph simulation models is adding label to the edges. This extension will make these model applicable to a wider range of applications. This will also require the algorithms and different related definitions to be revised.

In this research, we also designed and implemented distributed algorithms for graph simulation, dual simulation, strong simulation, strict simulation, and tight simulation. All these distributed algorithms were designed and implemented based on vertex-centric BSP programming model. Although, this programming has many advantages and many platforms are implemented based on that, it has also some disadvantages. The property of synchronizing all the supersteps at the end of each superstep may add a noticeable overhead at the last supersteps when only small number of vertices left active.

For future work, it might be worthwhile to try distributed algorithms designed for other distributed computing models. For example, an asynchronous vertex-centric programming environment like GraphLab [51] seems to be a good candidate for this purpose. Another promising alternative might be Graph-centric programming model suggested in [69].

We have also proposed a novel pattern containment approach based on tight simulation model. Furthermore, we have designed and implemented an efficient cache system based on this technique. This is one of the first works that investigates caching techniques for subgraph pattern matching queries.

Although the employed filtering mechanism for searching the cache has been very efficient in our experiments, it might be worthwhile for future work to investigate more complex indexing algorithms in order to support very heavy workloads in extremely massive datasets. This problem is very similar to the problem that is studied under the title of *supergraph query processing* in the literature [77, 64]. However, the problem in the context of our research has two main distinguishing differences: (1) All the available index systems are based on subgraph isomorphism and there is no work about other matching models particularly simulation; (2) the queries gradually populate the cache space; therefore, unlike the previous works the index should be formed and evolved dynamically.

The cache replacement policy that we have implemented in this project is very similar to the policies in other contexts. Nevertheless, it seems appealing to design new replacement policies specifically revised for subgraph pattern matching. For example, it should be possible to merge two graph queries stored in the cache when they have a large overlap.

Another future work can be adding support for time evolving data graphs. Real-life data graphs are time evolving [30]; i.e., there are minor changes in their structure through the time. The content of cache for pattern matching queries should be updated according to these changes using incremental algorithms. Incremental algorithms for subgraph pattern matching are in their infancy stages. In [24], Fan et al. have investigated incremental algorithms for graph simulation, bounded simulation, and subgraph isomorphism.

References

[1] 10th DIMACS. http://www.cc.gatech.edu/dimacs10/.

[2] Apache Hama. http://hama.apache.org/.

[3] Facebook. http://newsroom.fb.com.

[4] Giraph website. http://giraph.apache.org/.

[5] graph-tool website. http://projects.skewed.de/graph-tool/.

[6] Hadoop website. http://hadoop.apache.org/.

[7] Laboratory for web algorithmics. http://law.di.unimi.it/datasets.php.

[8] Engineering graph clustering: Models and experimental evaluation. *J. Exp. Algorithmics*, 12:1.1:1–1.1:26, June 2008.

[9] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, April 2006.

[10] Karl A. Abrahamson, Rodney G. Downey, and Michael R. Fellows. Fixed-parameter tractability and completeness iv: On completeness for w[p] and {PSPACE} analogues. *Annals of Pure and Applied Logic*, 73(3):235 – 276, 1995.

[11] Konstantin Andreev and Harald Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[12] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196, 2013.

[13] H. Wang B. Shao and Y. Li. The trinity graph engine. 2012.

[14] D.A. Bader and K. Madduri. Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, April 2008.

[15] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 595–602, New York, NY, USA, 2004. ACM.

[16] Kellog S Booth and Charles J Colbourn. *Problems polynomially equivalent to graph isomorphism*. Computer Science Department, Univ., 1979.

[17] J. Brynielsson, J. Hogberg, L. Kaati, C. Mårtenson, and P. Svenson. Detecting social positions using simulation. In *Advances in Social Networks Analysis and Mining (ASONAM), 2010 International Conference on*, pages 48–55. IEEE, 2010.

[18] Bradford L Chamberlain et al. Graph partitioning algorithms for distributing workloads of parallel computations. *University of Washington Technical Report UW-CSE-98-10*, 3, 1998.

[19] Jiefeng Cheng, J.X. Yu, Bolin Ding, P.S. Yu, and Haixun Wang. Fast graph pattern matching. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 913 –922, april 2008.

[20] D. CONTE, P. FOGGIA, C. SANSONE, and M. VENTO. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(03):265–298, 2004.

[21] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, Oct 2004.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[23] W. Fan. Graph pattern matching revised for social network analysis. In *Proceedings of the 15th International Conference on Database Theory*, pages 8–21. ACM, 2012.

[24] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 925–936, 2011.

[25] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: from intractable to polynomial time. *Proc. VLDB Endow.*, 3(1-2):264–275, September 2010.

[26] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. Graph homomorphism revisited for graph matching. *Proceedings of the VLDB Endowment*, 3(1-2):1161–1172, 2010.

[27] Wenfei Fan, Xin Wang, and Yinghui Wu. Answering graph pattern queries using views. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 184–195, March 2014.

[28] Wenfei Fan, Xin Wang, Yinghui Wu, and Dong Deng. Distributed graph simulation: Impossibility and possibility. *Proceedings of the VLDB Endowment*, 7(12), 2014.

[29] Martin Farber. On diameters and radii of bridged graphs. *Discrete Mathematics*, 73(3):249–260, 1989.

[30] Arash Fard, Amir Abdolrashidi, Lakshmish Ramaswamy, and John A. Miller. Towards efficient query processing on massive time-evolving graphs. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012 8th International Conference on*, pages 567 –574, oct. 2012.

[31] Arash Fard, M Usman Nisar, John A Miller, and Lakshmish Ramaswamy. Distributed and scalable graph pattern matching: Models and algorithms. *International Journal of Big Data (IJBD)*, 1(1), 2014.

[32] Arash Fard, M.U. Nisar, L. Ramaswamy, J.A. Miller, and M. Saltz. A distributed vertex-centric approach for pattern matching in massive graphs. In *Big Data, 2013 IEEE International Conference on*, pages 403–411, Oct 2013.

[33] Brian Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS*, 6:45–53, 2006.

[34] Michael R Garey and David S Johnson. Computer and intractability. *A Guide to the NP-Completeness. Ney York, NY: WH Freeman and Company*, 1979.

[35] Bronwyn H Hall, Adam B Jaffe, and Manuel Trajtenberg. The nber patent citation data file: Lessons, insights and methodological tools. Technical report, NBER Working Paper 8498, 2001.

[36] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 337–348, New York, NY, USA, 2013. ACM.

[37] Huahai He and Ambuj K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 405–418, New York, NY, USA, 2008. ACM.

[38] Bruce Hendrickson and Tamara G Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519 – 1534, 2000. Graph Partitioning and Parallel Computing.

[39] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 28. ACM, 1995.

[40] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 453–, 1995.

[41] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[42] Wei Jin and Jiong Yang. A flexible graph pattern matching framework via indexing. In *Scientific and Statistical Database Management*, pages 293–311. Springer, 2011.

[43] U. Kang, C.E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.

[44] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2), 2011.

[45] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96 – 129, 1998.

[46] George Karypis and Vipin Kumar. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN*, 1998.

[47] Tomas Lampo, Mara-Esther Vidal, Juan Danilow, and Edna Ruckhaus. To cache or not to cache: The effects of warming cache in complex sparql queries. In Robert Meersman, Tharam Dillon, Pilar Herrero, Akhil Kumar, Manfred Reichert, Li Qing, Beng-Chin Ooi, Ernesto Damiani, DouglasC. Schmidt, Jules White, Manfred Hauswirth, Pascal Hitzler, and Mukesh Mohania, editors, *On the Move to Meaningful Internet Systems: OTM 2011*, volume 7045 of *Lecture Notes in Computer Science*, pages 716–733. Springer Berlin Heidelberg, 2011.

[48] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proc. VLDB Endow.*, 6(2):133–144, December 2012.

[49] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. The dynamics of viral marketing. *ACM Trans. Web*, 1(1), May 2007.

[50] Yucheng Low. *GraphLab: A Distributed Abstraction for Large Scale Machine Learning*. PhD thesis, Carnegie Mellon University, 2013.

[51] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, pages –1–1, 2010.

[52] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. Capturing topology in graph pattern matching. *Proc. VLDB Endow.*, 5(4):310–321, December 2011.

[53] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Transactions on Database Systems (TODS)*, 39(1):4, 2014.

[54] Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. Distributed graph pattern matching. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 949–958, 2012.

[55] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 6–6, 2009.

[56] Michael Martin, Jrg Unbehauen, and Sren Auer. Improving the performance of semantic web applications with sparql query caching. In *The Semantic Web: Research and Applications*, volume 6089, pages 304–318. Springer Berlin Heidelberg, 2010.

[57] B.T. Messmer and H. Bunke. Efficient subgraph isomorphism detection: a decomposition approach. *Knowledge and Data Engineering, IEEE Transactions on*, 12(2):307–323, Mar 2000.

[58] M.U. Nisar, A. Fard, and J.A. Miller. Techniques for graph analytics on big data. In *Big Data (BigData Congress), 2013 IEEE International Congress on*, pages 255–262, June 2013.

[59] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, December 2003.

[60] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. Technical report, Stanford University, 2012.

[61] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM.

[62] Matthew Saltz, Ayushi Jain, Abhishek Kothari, Arash Fard, John A Miller, and Lakshmish Ramaswamy. Dualiso: An algorithm for subgraph pattern matching on very large labeled graphs. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, pages 498–505. IEEE, 2014.

[63] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, August 2008.

[64] Haichuan Shang, Ke Zhu, Xuemin Lin, Ying Zhang, and R. Ichise. Similarity search on supergraph containment. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 637–648, March 2010.

[65] Yanfeng Shu, Michael Compton, Heiko Mller, and Kerry Taylor. Towards content-aware sparql query caching for semantic web applications. In *Web Information Systems Engineering WISE 2013*, volume 8180, pages 320–329. Springer Berlin Heidelberg, 2013.

[66] Philip Stutz, Abraham Bernstein, and William W. Cohen. Signal/Collect: Graph Algorithms for the (Semantic) Web. In P.F. Patel-Schneider et al., editor, *International Semantic Web Conference (ISWC) 2010*, volume LNCS 6496, pages pp. 764–780. Springer, Heidelberg, 2010.

[67] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.*, 5(9):788–799, May 2012.

[68] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: Extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 990–998, 2008.

[69] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3), 2013.

[70] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[71] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[72] Duncan J Watts and Steven H Strogatz. Collective dynamics of small-worldnetworks. *nature*, 393(6684):440–442, 1998.