Improving the Dual Cardinality Simulation Algorithms

by

(Under the Direction of John Miller)

ABSTRACT

Graph pattern matching is typically defined in terms of subgraph isomorphism, which makes it an NP-complete/NP-hard problem. Isomorphism algorithms requires bijective functions which can be too restrictive to identify patterns in real-world applications. Moreover, real-world graphs may contain some noise and the problem of finding the exact match can be very expensive. In order to avoid the combinatorial worst-case time complexity of subgraph isomorphism, we extend prior work on dual cardinality simulation. According to our experiments, this type of graph simulation offers high precision with good performance in large graphs. Precision is acquired because dual cardinality simulation checks the constraint in which the number of matching children or parents with the same label in the data graph should not be less than their correspondents in the query graphs. For improving the performance, we have introduced the concept of count sets which are computed before dual cardinality simulation is executed. Experiments are done on large graphs using synthetic and real-world graphs.

INDEX: Subgraph Isomorphism, Cardinality, Graph Database, Graph Simulation, Pattern matching

Improving the Dual Cardinality Simulation Algorithms

by

Luis Anggelo Ibarra

B.S, Pontificia Universidad Catolica del Peru, Peru, 2011

A thesis submitted to the Graduate Faculty
of University of Georgia in partial fulfillment
of the
requirements for the degree

Master of Science

2018

Improving the Dual Cardinality Simulation Algorithms

by

Luis Anggelo Bernaola Ibarra

Approved:

Major Professor:	John A Miller
Committee:	Maria Hybinette
	Budak Arpinar

Electronic Version Approved

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
May 2018

Acknowledgements

I would like to say thank you to my major professor Dr. John Miller for his help throughout this research project. Also I would like say more than thank you to my mom, my uncle and sister.

Contents

Acknowledgements **iv**

List of Figures **vi**

1 Introduction **1**

2 Background **3**

2.1 Graph representation 3

2.2 Subgraph Pattern Matching Problem 4

2.3 Types of Pattern Matching 5

2.4 Graph Database Systems 9

2.5 QUERY LANGUAGES. 10

3 Dual Cardinality Algorithms **12**

3.1 Dual Cardinality Simulation 12

3.2 Dual Cardinality Isomorphism 16

4 Experimentation **18**

4.1 Introduction 18

4.2 Effects that impact the runtime 19

4.3 Precision and runtime 26

4.4 Cardinality in graphs 41

5 Conclusion and Future Work **47**

Bibliography **49**

List of Figures

2.1 A data graph and a query graph 4
2.2 SubGraph Isomorphism. 6

3.1 Cardinality and Isomorphism 17

4.1 data size effect for uniform graphs 20
4.2 data size effect for power law graphs 20
4.3 query size effect for uniform graphs 21
4.4 query size effect for power law graphs 21
4.5 label size effect for uniform graphs 22
4.6 label size effect for power law graphs 23
4.7 label size effect for amazon-2008 dataset 23
4.8 uk-2014 data set for label=200 24
4.9 data graph density effect 25
4.10 Precision for uniform graphs 27
4.11 runtime for uniform graphs 27
4.12 Precision for power law graphs 28
4.13 runtime for power law graphs 28
4.14 Precision for power law graphs with different query size 29
4.15 runtime for power law graphs with different query size 29
4.16 Precision for amazon-2008 dataset 30
4.17 runtime for amazon-2008 dataset 30
4.18 Precision for graphs with different density 31
4.19 runtime for graphs with different density 31
4.20 Precision vs runtime for uniform graphs 32
4.21 Precision vs runtime for power law graphs 32
4.22 Precision vs runtime for power law graphs with different query size 33
4.23 Precision vs runtime for amazon-2008 dataset with different label size 33
4.24 Precision vs runtime for graphs with different density 34
4.25 runtime for edge labeled power law graphs with different graph size 35
4.26 runtime for edge labeled power law graphs with different graph size 36
4.27 runtime for edge labeled power law graphs with different edge label size 37
4.28 runtime for edge labeled power law graphs with different edge label size 38
4.29 Mutable vs Immutable for uniform graphs 39

4.30 Mutable vs Immutable for power law graphs 40
4.31 version 1 vs vesion 2 for uniform graphs- graph size effect 42
4.32 version 1 vs vesion 2 for power law graphs- graph size effect . . . 42
4.33 version 1 vs vesion 2 for uniform graphs- query size effect 43
4.34 version 1 vs vesion 2 for power law graphs- query size effect . . . 44
4.35 version 1 vs vesion 2 for uniform graphs- label size effect 45
4.36 version 1 vs vesion 2 for power law graphs- label size effect . . . 46

Chapter 1

Introduction

The practical relevance of the graph pattern matching problem is reflected in a variety of applications ranging from chemical documentation, computer vision, knowledge discovery, biology, cheminformatics, dynamic network traffic, and recently, social networks. Graph pattern matching is typically defined in terms of subgraph isomorphism. Subgraph isomorphism tries to find all subgraphs of G that are isomorphic to a query graph Q. That is, a match of Q is a subgraph G' of G such that there exists a bijective function f from the vertices of Q to the vertices of G' , and (a) for each vertex v in G' , v and $f(v)$ have the same label, and (b) there exists an edge from v to v' in Q if and only if $(f(v), f(v'))$ is an edge in G' . This makes graph pattern matching np-complete, and obstructs its scalability in finding exact matches. Moreover, all known algorithms have an exponential worst-case behavior.

To reduce the complexity, Simulations like Tight[6], Strict[13] and Dual Simulation[1] have been adopted for pattern matching. These simulations with a polynomial-time complexity can identify matches with fewer constraints on the topology of the graphs than subgraph isomorphism.

In this research project, we extend the dual cardinality simulation, introduced by Arash Fard[8], that tries to solve the problem of subgraph isomorphism strictness. Although this algorithm is not as fast as Dual Simulation, it exhibits a good average performance and offers more precision than dual simulation. In chapter 2 we discuss the required background for this thesis, including how graphs are represented and the most known pattern matching algorithms. In chapter 3 we introduce our version for the dual cardinality algorithm. In chapter 4 we see the experimental results in synthetic and real world graphs. In chapter 5 we present our conclusions based on the experimentation results and we discuss the future work.

Chapter 2

Background

In this section, we discuss graph terminology and its representation in ScalaTion[19], different types of pattern matching algorithms, graph databases and the most popular query languages.

2.1 Graph representation

Throughout this thesis, we have considered only directed graph, with vertex labels; this can be defined as $G(V, E, L, l)$, where

V = set of vertices
$E \subseteq \{(u,\ v)\ |$u$\in V,\ v \in V$ and $u \neq v$ $\}$ set of directed edges
L = set of labels, $l : V \rightarrow L$ (vertex labeling function)

The base type of set L can be set at configuration time. In this research, we have used integer or strings as the label type. We denote outgoing edges for a vertex v as $adj(v)$ where for some vertex $v \in V$, $adj(v) = \{v' : (v,v') \in E\}$. We sometimes refer to the vertices in $adj(v)$ as children of v and also refer to v as the parent of all the vertices in $adj(v)$. We have assumed that all the vertices are labeled. We also assumed that the query graph is a connected graph, otherwise it can be decomposed into multiple query graphs.

In ScalaTion a class called Graph is used to represent graphs in memory, vertices of the graphs are represented in memory as the indices of an array and each vertex is mapped to its labels using a separate array called label. The children of a vertex v are represented by an adjacency set called ch where $ch(v)$ contains all the children of vertex v. If the parameter $inverse$ is true, the parents of a vertex v are also represented by an adjacency set. We will need the parent set in the DualCAR simulation; and this inverse adjacency set will give us rapid accesses to parent vertices for any vertex v. The last parameter called $name$ just specifies the name of the graph.

class Graph $(ch : Array[SET[Int]], label : Array[TLabel], inverse : Boolean, name : String)$

2.2 Subgraph Pattern Matching Problem

The problem of subgraph pattern matching is defined as follows. Let $G(V, E, L, l)$ be a graph, where V is the set of vertices, E is the set of edges, $l : V \to L$ (vertex labeling function). Let $Q(Vq, Eq, Lq, lq)$ be the query graph where Vq is the set of vertices, Eq is the set of edges and $lq : Vq \to Lq$. The goal of subgraph pattern matching is to find all the subgraphs from the data graph G that match the pattern graph Q. Therefore, $G'(V', E', L', l')$ is a subgraph of G if and only if

1. $V' \subseteq V$
2. $E' \subseteq E$
3. $\forall u \in V' : l'(u) = l(u)$

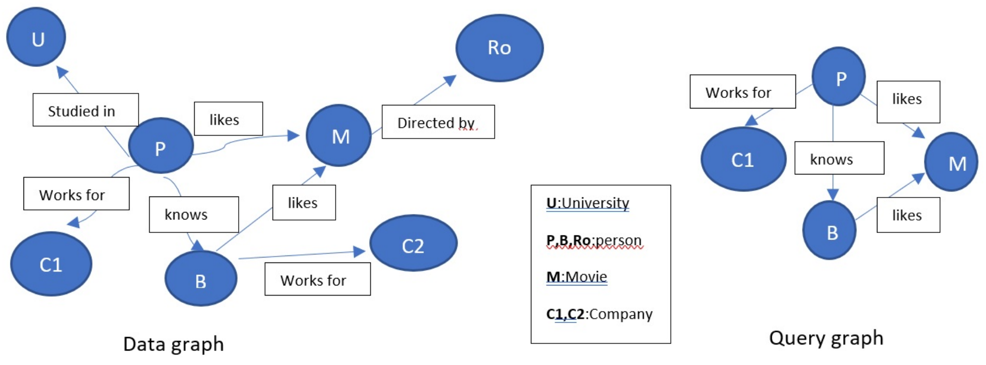

Figure 2.1: A data graph and a query graph

In figure 2.1 we can see an example of subgraph pattern matching where entities are represented as vertex labels and edge labels represent the relationships between them.

2.3 Types of Pattern Matching

In this section, we present the different pattern-matching problems. For each of them we provide a basic idea on its functionality in order to know how they perform matches for a given pattern.

2.3.1 Subgraph Isomorphism

In this problem we are looking for precise matches between two given graphs, a large data graph G, and a small query graph Q. This can be defined as a bijective function between a query graph $Q(Vq, Eq, Lq, lq)$ and a subgraph of a data graph $G(V, E, L, l)$. Thus $G'(V', E', L', l')$ is said to be a subgraph isomorphic match to Q if

1. $V' \subseteq V$
2. $E' \subseteq E$
3. there exists a bijective function $f : Vq \rightarrow V'$ such that

(a) An edge $e = (u, v) \in Eq \Leftrightarrow (f(u), f(v)) \in E'$
(b)$\forall v \in Vq, l(v) = l'(f(v))$

Ullmann was the first well known algorithm for subgraph isomorphism [5]. It laid a foundation for other pattern matching algorithms such as DualIso,VF2 and GraphQL. Because this problem is said to be NP-hard there has been much research going on to reduce the complexity to polynomial time and some of them will be discussed below. In figure 2.2 we can see an example of subgraph isomorphism.

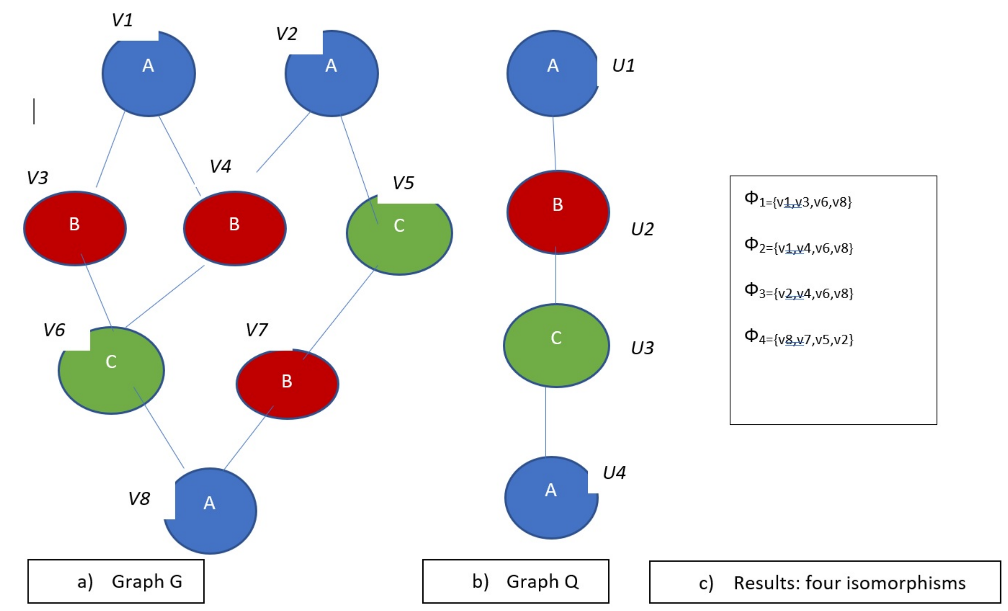

Figure 2.2: SubGraph Isomorphism.
(a) represents the data graph G, (b) represents the query graph Q and (c) represents the result match set. We have 4 different match results. For vertex u1, we have 3 possible values v1,v2 and v8. For vertex u2 we have 3 possible values: v3, v4 and v7. For vertex u3 we have 2 possible values: v6 and v7.For vertex u4 we have 2 values: v8 and v2

2.3.2 VF2

VF2[3] is an algorithm for graph isomorphism and subgraph isomorphism suited for dealing with large graphs. In order to select the next query vertex, unlike Ullmann, VF2 starts with the first vertex and selects a vertex connected from the already matched query vertices. For the process of refining the candidates, VF2 uses three pruning rules to prune out data vertex candidates: (1) Prune out any vertex v in $\Phi(u)$ such that v is not connected from already matched data vertices; (2) Let Mq and Mg be a set of matched query vertices and a set of matched data vertices, respectively. Let Cq and Cg be a set of adjacent and not yet-matched query vertices connected from Mq and a set of adjacent and not-yet-matched data vertices connected fromMg, respectively. Let $adj(u)$ be a set of adjacent vertices to a vertex u. Then, prune out any vertex v in $\Phi(u)$ such that $Cq \cap adj(u) > Cg \cap adj(v)$; (3) prune out any vertex v in $\Phi(u)$ such that $adj(u)\backslash Cq\backslash Mq > adj(v)\backslash Cg\backslash Mg$.

2.3.3 GraphQL

GraphQL[20] is another algorithm for subgraph isomorphism and it tries to solve this problem with a combination of two techniques: Neighborhood signature based pruning and the pseudo subgraph isomorphism test based pruning. Neighborhood signature of a vertex v, denoted as $sigGraphQL(v)$ is a multiset of labels of $adj(v)$. The neighborhood signature based pruning prunes out a candidate vertex v if $sigGraphQL(u) \not\subset sigGraphQL(v)$ For example, if vertex u has three children with labels A, A and C; then $sigGraphQL(u)$ is $\{A, A, C\}$. If the multiset of label of v is $\{A, B, D\}$; then $sigGraphQL(v)$ is $\{A, B, D\}$. Then, v is pruned since $sigGraphQL(u) \not\subset sigGraphQL(v)$. The Pseudo isomorphism test is an iterative algorithm using the depth as a parameter. At first iteration, we obtain two breadth first search trees Tu and Tv for u and v respectively, where their depth is 1. Then, we can prune out v if Tu is not contained in Tv. We can iterate this process by increasing depth by one until depth $= r$, where r is called the refinement level. The process of selecting the next query vertex first selects a query vertex u with the smallest candidate set size $\|\Phi(u)\|$. In the subsequent calls, GraphqL selects a query vertex u that is connected already matched query vertices and that makes the smallest size of intermediate results.

2.3.4 Dual Iso

DualIso[1] is known as Dual based Isomorphism. This algorithm is faster than VF2 and GraphQL and similar to Ullmann's subgraph isomorphism algorithm but provides more effective pruning based on Dual Simulation. It initially finds all the feasible matches of each vertex in the query. Given a query vertex u, $\Phi(u)$ is created which contains all vertices of the data graph with the same label as u. Then dual simulation is applied to prune the vertices from the data graph. It then uses a search algorithm to recursively find the matches in a depth-first manner. The most important feature of this algorithm is that it can refine both conditions at the same time. DualIso is able to operate without requiring any parent list. Because of this feature the runtime is reduced significantly.

2.3.5 Dual Simulation

Dual Simulation is an extension of graph simulation model, but has some additional features. It takes into account not only the children of the query vertex but also its parents. Query Graph $Q(Vq, Eq, Lq, lq)$ matches data graph

$G(V, E, L, l)$, if

A. a match between $u \in Vq$ and $u' \in V$ is accepted if for each vertex v in $child(u)$ there is a vertex in V that is present in $child(u')$
B. a match beween $u \in Vq$ and $u' \in V$ is accepted if for each vetex w in $parent(u)$ there is a vertex in V that is present in $parent(u')$

2.3.6 Strict Simulation

Strict simulation is an extension of Strong simulation introduced by Ma et al. [13]. Here they add a locality property to dual simulation. They introduce the concept of a ball to define locality. A ball b in $G(V, E)$, denoted by $G'(v, r)$ is a subgraph of G such that it contains all the vertices that are not more distant than a given radius r from a center $v \in V$ and the radius is acquired from the query graph which is the diameter of the query graph. A size of a ball is the number of vertices it has. In strict simulation, balls are created from the dual match set rather than the original data graph which reduces the solution.

2.3.7 Tight Simulation

Tight simulation [6] is an advanced modified version of Strict simulation. A further condition is added to reduce the number of balls. First, before applying dual simulation on G to find the dual simulation match set, they find the vertex candidate in Q which decreases the size of the balls and also reduces the number of balls. In addition to this, the radius of the ball is equal to the radius of the query graph, not the diameter, of the query graph as in strict simulation.

2.3.8 Cardinality Restriction Simulation

Cardinality restricted or CAR-dual simulation, introduces a condition 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. A graph G is said to be the cardinality match of graph Q if the number of matches of children and parents with the same label in G is not less than that of the number of children and parents matches with same labels in Q. Cardinality restriction improves the accuracy and the quality of the result set. By this condition, experiment results show that many vertices have been pruned to match the query graph and this is a step towards subgraph isomorphism

2.4 Graph Database Systems

Relational databases store highly structured data in tables with predetermined columns of certain types and many rows of the same type of information. In relational databases, references to other rows and tables are indicated by referring to their primary key attributes via foreign-key columns. This is enforceable with constraints. Joins are computed at query time by matching primary and foreign-keys of the many rows of the to-be-joined tables. In order to reduce these costly operations graph databases were proposed where data is stored in the form of vertices and their relationship are represented in the form of edges. Relationships are first-class citizens of the graph data model, unlike relational databases, which require us to infer connections between entities using foreign keys. By assembling the simple abstractions of nodes and relationships into connected structures, graph databases enable us to build sophisticated models for almost any domain problem.

2.4.1 NEO4J

Neo4j [1] is one of the most popular open source graph databases implemented in Java. It uses Cypher query language and is a transactional database. In Neo4j, everything is stored in the form of either an edge, a node, or an attribute. Each node and edge can have any number of attributes. Both the nodes and edges can be labelled.

2.4.2 ORIENT DB

OrientDB [2] is an open source NoSQL database management system written in Java. It is a multi-model database, supporting graph, document, key/value, and object model. It supports schema-less, schema-full and schema-mixed modes. It supports querying with Gremlin along with SQL extended for graph traversal. OrientDB uses several indexing mechanisms based on B-tree and Extendible hashing.

[1]http://neo4j.com/developer/graph-database/#_what_is_neo4j
[2]http://http://orientdb.com/orientdb/

2.4.3 Query Processing on Graph Databases

Now that we have reviewed some of the graph pattern matching algorithms, we can discuss how query processing works on graph databases. Query processing is a problem of finding small patterns or subgraphs on a graph database. Because processing graph data is a complex task, it requires efficient algorithms that can find query graphs on large graph databases. Graph databases can be grouped into two types. In the first type, the graph database can have very large graphs like Web of Data, social networks, etc. Query processing for this kind of data base would be finding the optimum path between the vertices or finding a subgraph in the data graph that is similar to the query graph. The second type of database consists of a large set of smaller graphs. An example of this type would be in the field of bio-informatics. Filtering and verification are two important steps in query processing [14]. In the filtering phase, the query graph is decomposed into features and these features are later searched using an index. Each feature is searched to get a set of graphs. The set of graphs are intersected to get candidate sets. In the verification phase, the set of graphs are matched using a subgraph isomorphism algorithm to obtain final result set.

2.5 QUERY LANGUAGES.

2.5.1 SPARQL

SPARQL [3] stands for SPARQL Protocol and RDF(Resource Description Framework) Query Language. It is a graph query language that is used to query data that is stored in RDF format. Queries are in the form of a triple Subject-Predicate-Object and queries RDF graph using pattern matching.

2.5.2 CYPHER

Cypher [4] is the query language used by Neo4j database. Cypher allows creation, deletion and updating of a database, which is applicable to nodes and relationships. This language is powerful because it allows to execute complex queries with ease. Cypher uses an SQL like structure where certain keywords are being reused and expressions for pattern matching are inspired by SPARQL.

[3]https://www.w3.org/TR/rdf-sparql-query/
[4]https://neo4j.com/developer/cypher/

2.5.3 GREMLIN

Developed by Apache Tinkerpop, Gremlin [5] is the graph traversal language and has been implemented in Java and is open source. Gremlin works for both OLTP-based graph databases as well as OLAP-based graph processors. Gremlin is a functional language that enables users to succinctly express complex traversals on (or queries of) their application's property graph. Gremlin will work for any framework or graph database that implements the Blueprints data model. Blueprints is something similar to JDBC but intended for graph databases.

[5] http://gremlindocs.spmallette.documentup.com/

Chapter 3

Dual Cardinality Algorithms

3.1 Dual Cardinality Simulation

Our algorithm is divided in three parts: Preprocessing the graph and query graph, Retrieval of feasible matches and Pruning algorithm. This algorithm can be categorized into tree search base algorithm. The most important difference is in the preprocessing of the graph G and query graph Q and also the pruning technique based on count sets; this gives the algorithm an increase in its performance. For each node and different query label we have two lists: one representing the children and another representing the parents.

3.1.1 Preprocessing the graph and query graph

Algorithm 1 takes the the array of child (adjacency) vertex sets named ch as input. Our algorithm starts by mapping each vertex and query label into two lists: one representing the children and another representing the parents . In this manner for any vertex v and label l, we can know which are the children or parents of vertex v with label l. Since this is kept in memory we can access it in a fast way when applying the pruning technique.

$labelCAR$ contains all the different labels from the query graph Q. For each vertex from the query or data graph and each label l from $labelCAR$ we obtain two lists. One representing the children of vertex v with label l; and second list representing the parents of vertex v with label l. These two lists are added to a map with vertex v and label l as keys and lists $listCh$ and $listPa$ as values.

Algorithm 1 Preprocessing G

```
1: procedure BUILDMAP($ch$)
2:     $map \leftarrow Map[(Int, Label), (List[Int]List[Int]))$
3:     for $v \leftarrow ch$ do
4:         for $l \leftarrow labelCAR$ do
5:             $listCh := getListOfChild(v, l, ch)$
6:             $listPa := getListOfParent(v, l, ch)$
7:             $map$ :=map + $[(v, l) \rightarrow (listCh, listPa)]$
8:         end for
9:     end for
10:    return $map$
11: end procedure
```

3.1.1.1 Profiling based on count sets and bit sets

In order to decrease the number of possible matches for each vertex u in $\Phi(u)$ we have introduced two similar techniques: Count sets for dual cardinality simulation and bit sets for dual simulation.

Count sets pruning technique is based on the number of children and parent and their labels. For example if a vertex v has one child with label C and five parents, two with labels A and three with label B, vertex v will be profiled as [2,3,1]; this is the count set for labels A, B and C. We refer to this as the count set for vertex v.

A BitSet represents a collection of small integers as the bits of a larger integer. For example, the bit set containing 3, 2, and 0 would be represented as the integer 1101 in binary, which is 13 in decimal. Dual simulation does not take into account the cardinality so if a vertex v does not have any vertex with label A as a children or parent, 2 parents with label B and 3 children with label C; the bit set for vertex v is a BitSet containing 1 and 2 that is represented as 110. Second and third bit is equal to 1 because we have at least one vertex with label B and C respectively.

3.1.2 Retrieval of feasible matches

The algorithm starts by obtaining feasible matches. Given a query vertex u, $\Phi(u)$ is created which contains all vertices of the data graph with the same label as u. Second step is to prune $\Phi(u)$ based on the count sets of each vertex. For example if $\Phi(u)$=v, we will have to check the cardinality restriction for the count set of vertex v and vertex u. If this cardinality restriction fails we remove v from $\Phi(u)$.

3.1.3 Pruning Algorithm

3.1.3.1 Pruning based on Simple Simulation

The concept of graph simulation [12], [16] is adapted for the pruning procedure for our algorithm. The basic version of graph simulation, simple simulation, is similar to the procedure used by Ullmann's algorithm for pruning. The simple simulation alone fails to reduce large graphs considerably to be used effectively in obtaining all subgraph isomorphic matches.

3.1.3.2 Pruning on Dual Sim

We adapted the algorithm of Dual Simulation, an extension of simple simulation above [12], as a pruning technique in our algorithm. While simple simulation only preserves the child relationship, dual simulation preserves both child and parent relationship. We used this condition for our cardinality restriction that says "A subgraph G is said to be the cardinality match of graph Q if the number of matches of children and parents with the same label in G is not less than that of the number of children and parents matches with same labels in Q".

Algorithm 2 takes one input: funtion Φ. This algorithm starts by looping through each vertex u of the query graph Q and each element that is present in $\Phi(u)$. The cardinality check is performed in the includeIn procedure. If this conditions fails vertex v needs to be removed from $\Phi(u)$. The procedure ends when there is no alteration after one loop.

Algorithm 3 take three inputs: vertex u, vertex v and function Φ. This procedure $includeIn$ starts by making a list L of children/parents of v_c. In order to get this list first we loop through each child/parent of vertex u_c from query graph Q and each child/parent of vertex v_c from data graph G. In every iteration we check the condition where v_c(child/parent of v) is an element of $\Phi(u_c)$; if this condition succedes we add v_c to the list L. Lines 10 to 20 describes the algorithm when the list L is non-empty. In this part we used what we stored in memory

Algorithm 2 Pseudocode for the cardinality dual simulation

```
1: procedure DualSimCAR($\Phi$)
2:     $alter := true$
3:     while $alter$ do
4:         $alter := false$
5:         for $u \leftarrow qRange$ do
6:             for $v \leftarrow \Phi$(u) do
7:                 $res1 := includedIn(u, v, \Phi)$
8:                 if  $res1 = false$ then
9:                     remove $v from \Phi$(u)
10:                    if  $\Phi = \emptyset$ then
11:                        return empty $\Phi$
12:                    end if
13:                    $alter := true$
14:                end if
15:            end for
16:        end for
17:    end while
18:    return $\Phi$
19: end procedure
```

Algorithm 3 Pseudocode for *includedIn* procedure

```
1: procedure includedIn($u$,$v$,$\Phi$)
2:     $L := \emptyset$
3:     for $u_c \leftarrow Q.Adj(u)$ do
4:         for $v_c \leftarrow G.Adj(v)$ do
5:             if  $\Phi(u_c)$ contains $v_c$ then $L := L+ v_c$
6:             end if
7:         end for
8:     end for
9:     if  $L \neq \emptyset$ then
10:        for $uc \leftarrow Q.Adj(u)$ do
11:            $l := Label(uc)$
12:            $size := Map(u, l).size$
13:            $elemts := Map(v, l)$
14:            $intersect := elemts$ & $L$
15:            if  $intersect = \emptyset$ then
16:                return $false$
17:            end if
18:            if $size > elemts$.size then
19:                return $false$
20:            end if
21:        end for
22:        return $true$
23:    else
24:        return $false$
25:    end if
26: end procedure
```

during the preprocessing phase. For every child/parents of uc from the query graph Q, we get the label l using Label(uc) . Given this label l and u; we can get the size of elements with u and l as keys. For the data graph G, we can get the elements *elemts*, children/parents of v that have label l as its label. The first condition that we check is if there is some intersection between the list L and *elemts*. The second condition checks the size of children/parents of u from data graph and children/parent of v from query graph with label l. These two conditions verify cardinality restriction 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.

3.2 Dual Cardinality Isomorphism

Algorithm 4 Pseudocode for FIND ISOMORPHISM

```
1: procedure FINDISOMORPHISM($G$,$Q$)
2:     $matches := \emptyset$
3:     $\Phi := FEASIBLEMATCHES(G, Q)$
4:     $\Phi := DUALSIMCAR(G, Q, \Phi)$
5:     $Search(G, Q, \Phi, 0)$
6: end procedure
7: procedure SEARCH($G, Q, \Phi, depth$)
8:     if $depth = Q.size$ then
9:         $matches := matches + \Phi$
10:    else
11:        for $v \leftarrow \Phi(depth)$ do
12:            if $v \notin \Phi(0)......\Phi(depth)$ then
13:                $\Phi'$ := copy of $\Phi$
14:                $\Phi'(depth) := \{v\}$
15:                $\Phi'$ := DUALSIMCAR($G, Q, \Phi'$)
16:                if $\Phi'$ is not $\emptyset$ then
17:                    Search($G, Q, \Phi, depth + 1$)
18:                end if
19:            end if
20:        end for
21:    end if
22: end procedure
```

Dual cardinality simulation algorithm results can contain vertices that are not in any isomorphic match. In figure 3.1 we have a query graph Q and data graph G. Query graph Q has a cycle for vertices 0 and 1 with labels A and B, respectively. We can find many vertices in the data graph where the dual cardinality constraint is satisfied but not isomorphism; for example in the data graph G we have a vertex 1 with label A with one child: vertex 2 with label B and one parent: vertex 0 with label B. Vertex 2 with label B has a child: vertex 3 with label A and one parent: vertex 1 with label A. In this example query vertices 0 and 1 map to data vertices 1 and 2, respectively using Dual Cardinality. Moreover, we have to consider another situation where one vertex in the data graph maps to more than one vertex in the query graph. FIND ISOMORPHISM procedure, presented in Algorithm 4, covers these situations to present only results in isomorphic match.

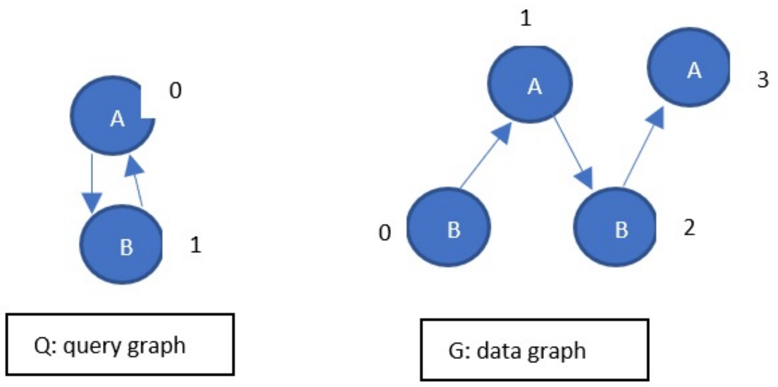

FIGURE 3.1: Cardinality and Isomorphism

FIND ISOMORPHISM procedure uses recursive method called search that works the same way as in Ullmann's and DualIso algorithm. The first step is to invoke $feasibleMatches$. After that, $dualSimCAR$ procedure is executed for the first time. After this first time, the algorithm will prune most of the unwanted vertices. After that, search method is invoked with graph G, query graph Q, the first result of dualSimCAR and depth equals to zero as input. First, a copy $\Phi(0)$ is made of Φ and a vertex v in $\Phi(0)$ is isolated and treated as if it were the only vertex to match query vertex 0. $dualSimCAR$ is then performed on $\Phi(0)$, which necessarily removes all vertices in $\Phi(1), ..., \Phi(|Vq|$ - 1) that are not contained in an isomorphic match with $f(0) = v$. If Φ is not empty we invoke again the refine method with a new depth(depth+1). We continue collecting matches until depth is equal to the query size.

Chapter 4

Experimentation

4.1 Introduction

In this chapter we are going to discuss the experimental results to compare the different subgraph isomorphism that we present in chapter 2 like DualIso, GraphQL,VF2 and our implementation of the dual cardinality isomorphism algorithm. We also compared the precision and runtime of the different simulations: Dual cardinality simulation, Dual Simulation, Tight simulation and Strong Simulation

The runtime of the different subgraph isomorphism algorithms can be affected by various factors: the number of vertices in the data graph, the number of vertices in the query graph, the number of distinct labels and the density of the graph. Since subgraph isomorphism can be too strict for emerging applications we have compared the precision and runtime of the different simulations: Dual Cardinality Simulation, Dual Simulation, Tight simulation and Strict Simulation.

We have used synthetic and real-world data graphs. For synthetic data we have used up to 3 millions of vertices in the data graph, different number of vertices in the query graph(4-80) and different numbers of unique labels(10-200). We have divided synthetic data in uniform graphs and power law graphs. Power law graphs contain a few vertices with a high out degree and many vertices with a smaller out degree. For uniform and power law graphs we have used $\alpha = 1.2$ as used by others [12], [14s], where $E= V^{\alpha}$. Query graphs are generated by using BFS search process. It performs a Breadth-First Search until the required number of specified vertices are found. In this way there is at least one match in the data graph. For real-world datasets, we have used amazon-2008 which has 735,323 vertices and 5,158,388 edges and uk-2014-tpd with 1766010 vertices and 18244650 edges.

GraphQL, DualCARIso, DualIso, DualCARSim,TightSim and StrictSim were written in Scala version 2.12. For the VF2 algorithm, the implementation provided by JGraphT library was used due to its implementation in Java and because Java and Scala have very close performance. All experiments were run on a machine with 128GB RAM, AMD Opteron, IB interconnect having 48 cores.

4.2 Effects that impact the runtime

In chapter 3 we have introduced the concept of count sets for the dual car simulation and bit sets for the dual simulation. Here we are going to present the impact of those concepts on the runtime in the DualCARIso algorithm and the DualIso algorithm. In this part we call DualCARIsoWithSets the algorithm that implements count sets and DualIsoWithSets the algorithm that implements bit sets.

4.2.1 Effect of data graph size:

This experiment will test the performance of the different algorithms with up to 3 million vertices and 60 million edges. As we can see in figure 4.1 the algorithm scale well as the graph size increases. We keep the number of labels(100) and the query size constant(10) constant and $\alpha = 1.2$. In the experiment shown, DualIsoCARWithSets is in average 280 and 7 times faster than VF2 and GraphQL respectively. The behavior of the runtime is very similar for uniform and power law graphs. This experiment also shows that there is a good impact in the performance when count sets are implemented in DualIsoCAR. The same does not happens for DualIso, here bit sets implementation does not decrease the runtime.

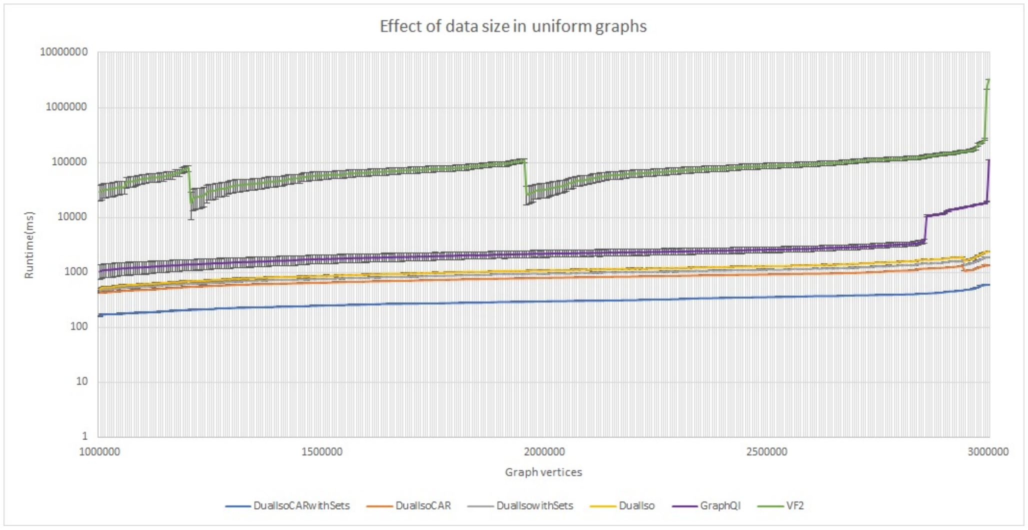

FIGURE 4.1: data size effect for uniform graphs

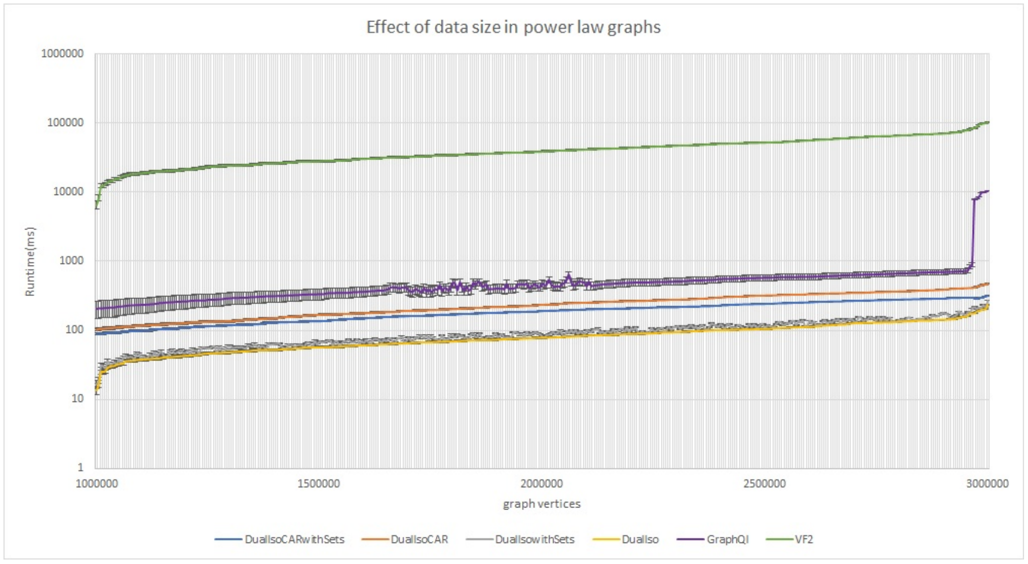

FIGURE 4.2: data size effect for power law graphs

4.2.2 Effect of query size

In this experiment we keep the data graph size constant to 1 million vertices , the number of labels to 100 and $\alpha = 1.2$. As we expect, Figure 4.3 shows that the runtime increases when the query size increases. We did not limit the number of matches. Uniform and power law graphs have the same behavior of the runtime.

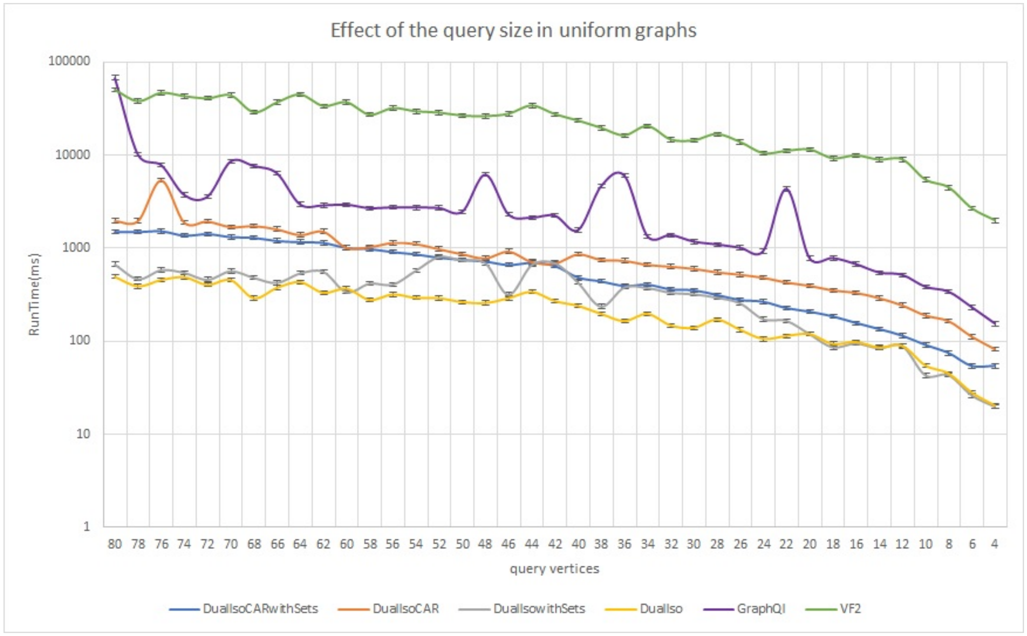

FIGURE 4.3: query size effect for uniform graphs

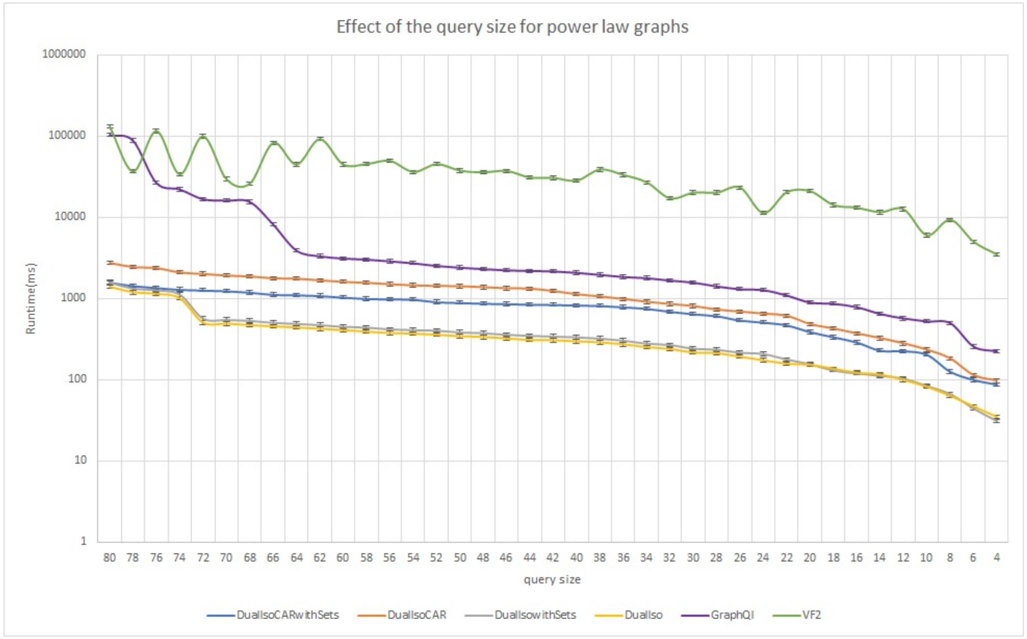

FIGURE 4.4: query size effect for power law graphs

4.2.3 Effect of the number of labels

In the previous section we described the method that gets the feasible matches. This method works based on the labels on the data graph, so if we have more unique number of labels in the data graph, we will get fewer numbers of vertices for each query and when this happens fewer paths will be traversed. Figure 4.5 and figure 4.6 show that behavior where the number of labels have a impact on the runtime. For uniform and power law graphs 1 million vertices was used for data graph size, 10 for query size and $\alpha = 1.2$.

Figure 4.7 shows the experiment using the amazon-2008 data sets. This dataset is a symmetric graph that describes the similarity among books. Query size equals to 10 and $\alpha = 1.2$ have been used in this experiment.

Figure 4.8 represents the runtimes for the different algorithms using 200 as the number of labels and query size equal to 10 and $\alpha = 1.2$. This experimentation was performed on the uk-2014 top private domains data set graph.

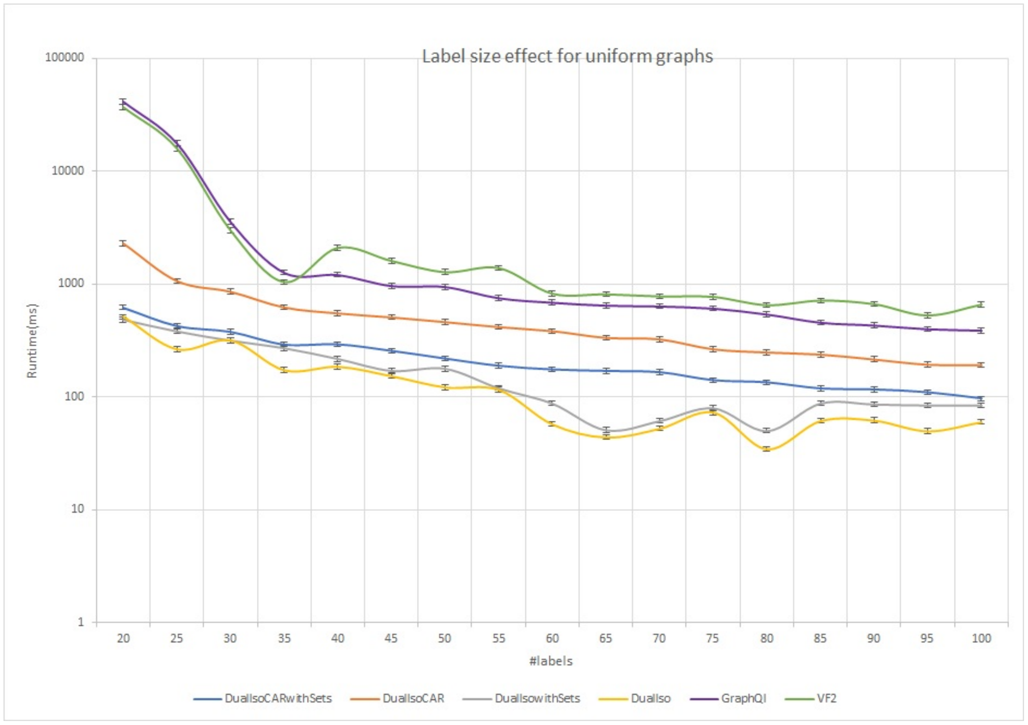

FIGURE 4.5: label size effect for uniform graphs

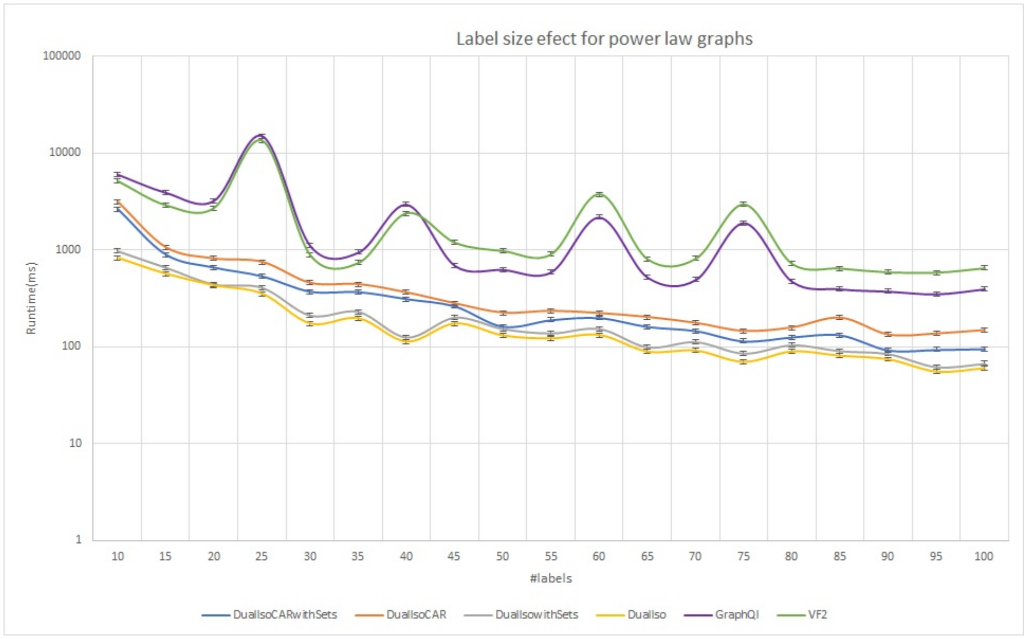

FIGURE 4.6: label size effect for power law graphs

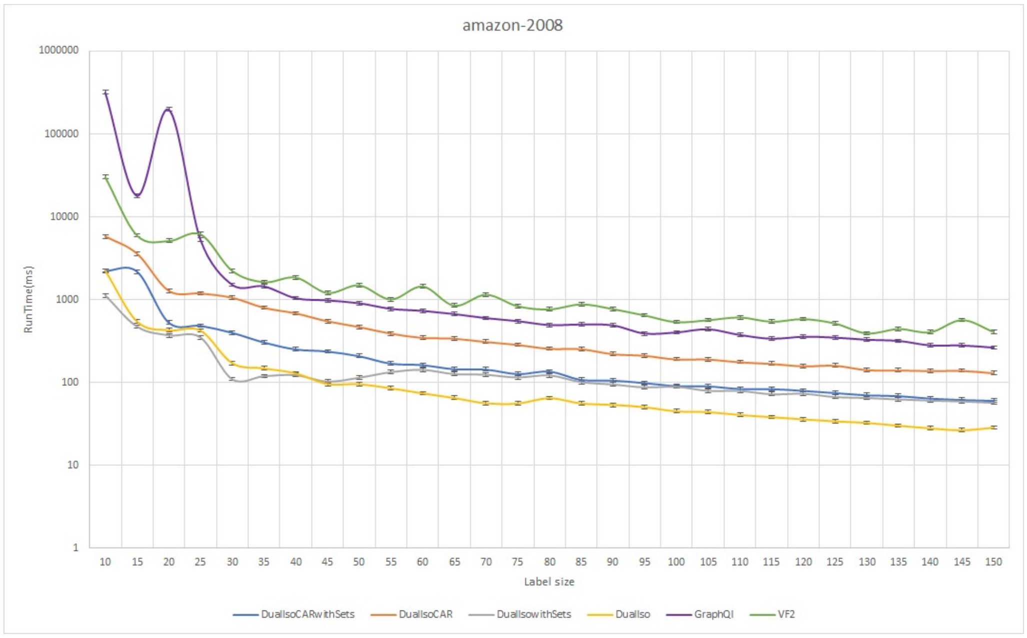

FIGURE 4.7: label size effect for amazon-2008 dataset

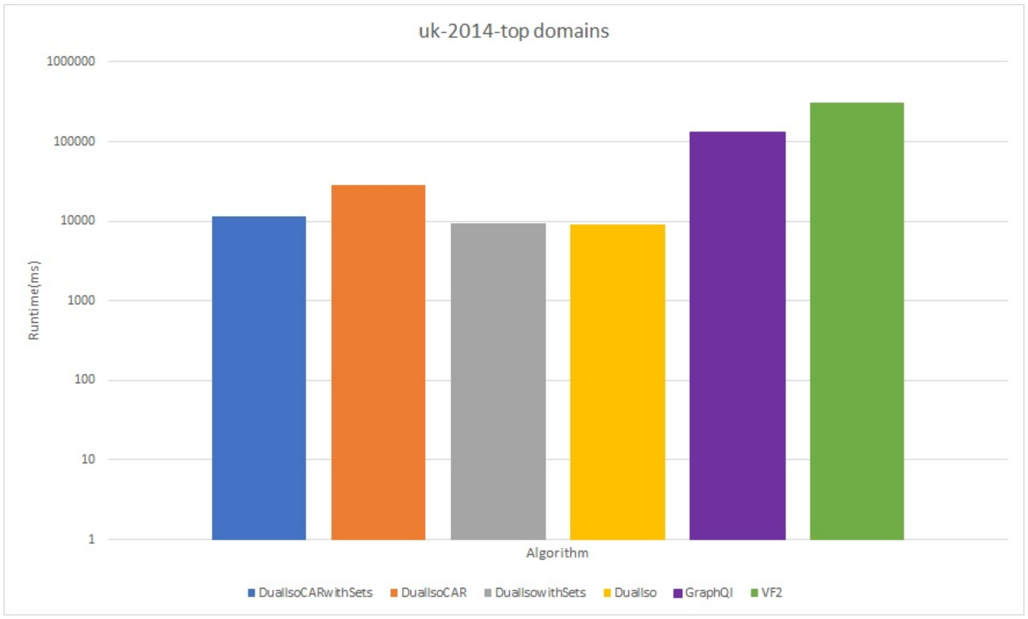

FIGURE 4.8: uk-2014 data set for label=200

4.2.4 Effect of the data graph density

Figure 4.9 describes the effect of graph density on query response time. The number of vertices in the data graph and the number of label was 1 million and 100, respectively. We will have more matches as the density increases. This will affect the runtime.

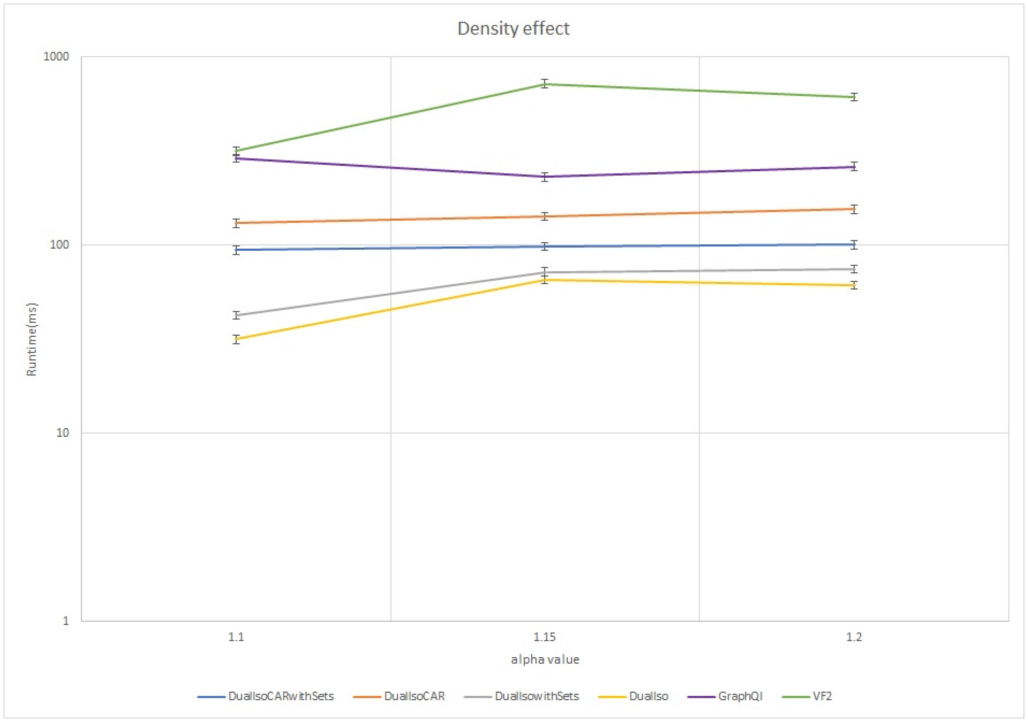

FIGURE 4.9: data graph density effect

4.3 Precision and runtime

In this experimentation we test the precision and runtime of the simulations algorithms presented in chapter 2. Using subgraph isomorphism, particularly DualIso, we defined precision as the number of vertices in the result set of DualIso divided by the number of vertices in the result set of each simulation. We consider DualIso with unlimited matches to have the perfect match; so its precision will always be 1.

We have run this experiments with different number of vertices in the graph size, different number of vertices in the query size and different number of labels for uniform, power law and real graphs. We calculate the precision mean and runtime mean after 10 iterations for each different parameter value(graph size, query size and label size). We have also shown error bars in the graph.

In figure 4.10 we can see that dual cardinality simulation and the CAR version of tight simulation have high precision as it checks for cardinality restriction constraints. Although Tight simulation cardinality has the highest precision, Figure 4.11 shows it has the worst runtime among all the simulations. Figures 4.12 an 4.13 show the same behavior for power law graphs. Figures 4.14 and 4.15 show the query size effect on the simulations precision and runtime respectively. Figures 4.16 and 4.17 show the precision and runtime of the different simulations in the amazon-2008 dataset with different label size. Figures 4.18 and 4.19 show precision and runtime, respectively for graphs with different density.

Precision and runtime are expressed between 0 and 1. Simulations closer to 1 are faster and more precise. In figure 4.20 we can see that dual cardinality simulation offers good precision with a good enough performance for uniform graphs. The same behavior is shown in figure 4.21 for power law graphs. In these two graphs we have used different graph size (1 million, 2 million, 3 million). Figure 4.22 represents the precision and runtime for power law graphs with different query size values (60, 80, 100). Figure 4.23 show the precision and runtime for amazon-2008 dataset with different label size (40, 60, 80). Runtime decreases as the number of labels increases. Figure 4.24 uses graphs with different density (1.1, 1.15 ,1.20) to express precision and runtime. In these experiments where we have tested precision and runtime, dual simulation has the best runtime and the lowest precision, tight cardinality simulation has the highest precision and worst runtime and dual cardinality simulation has the second-best performance and the second highest precision.

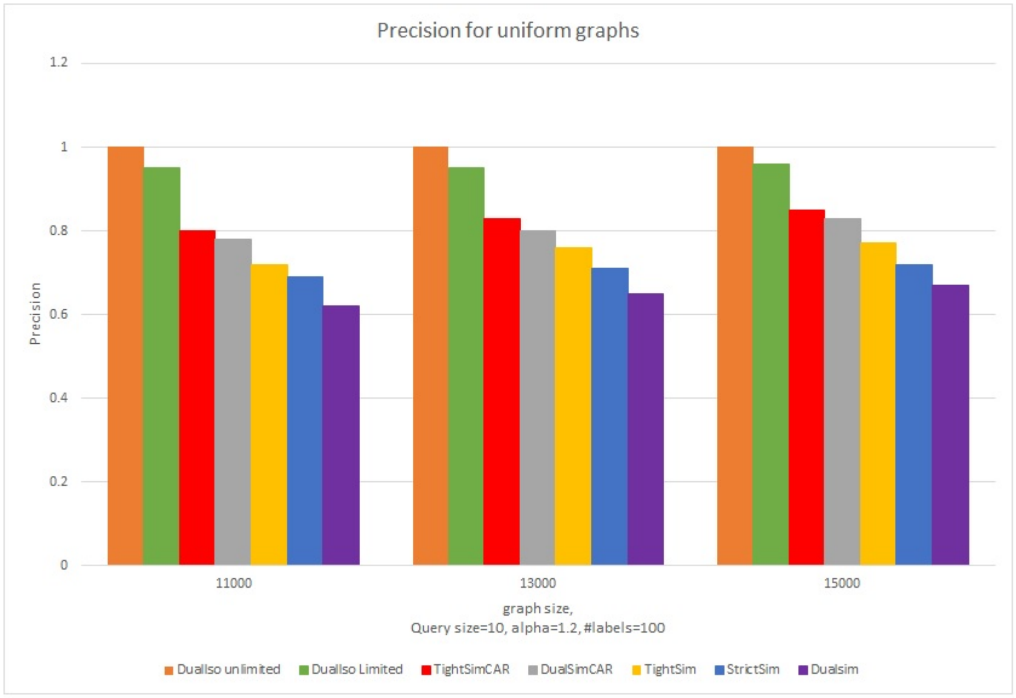

FIGURE 4.10: Precision for uniform graphs

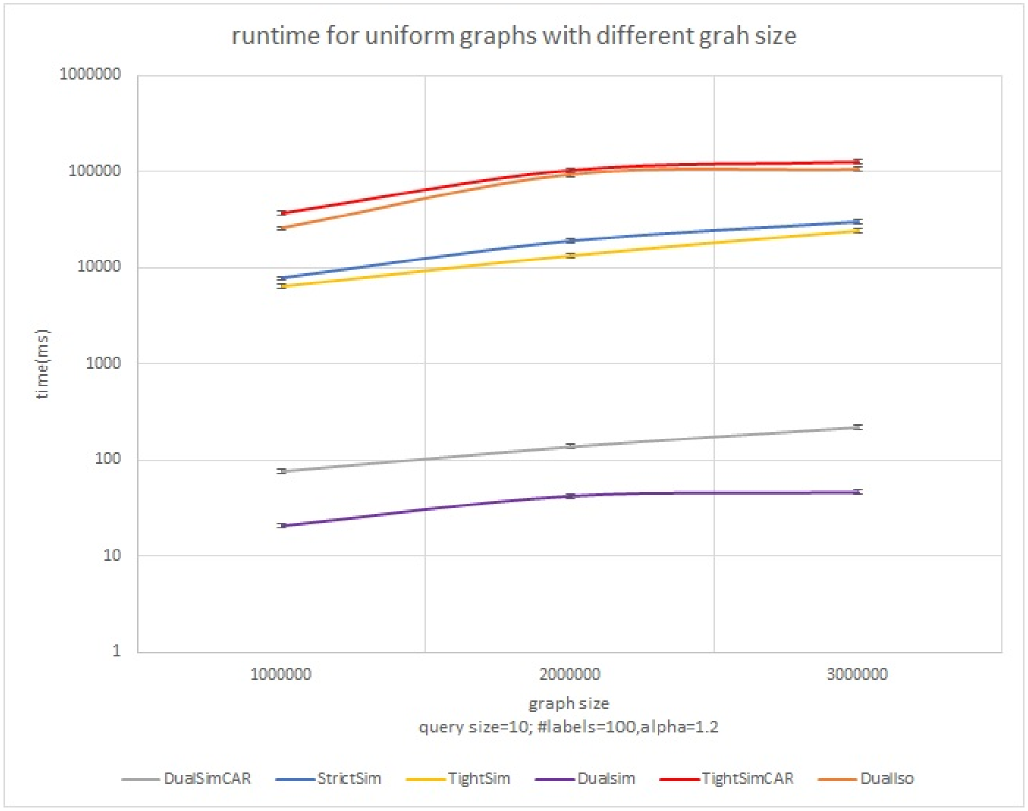

FIGURE 4.11: runtime for uniform graphs

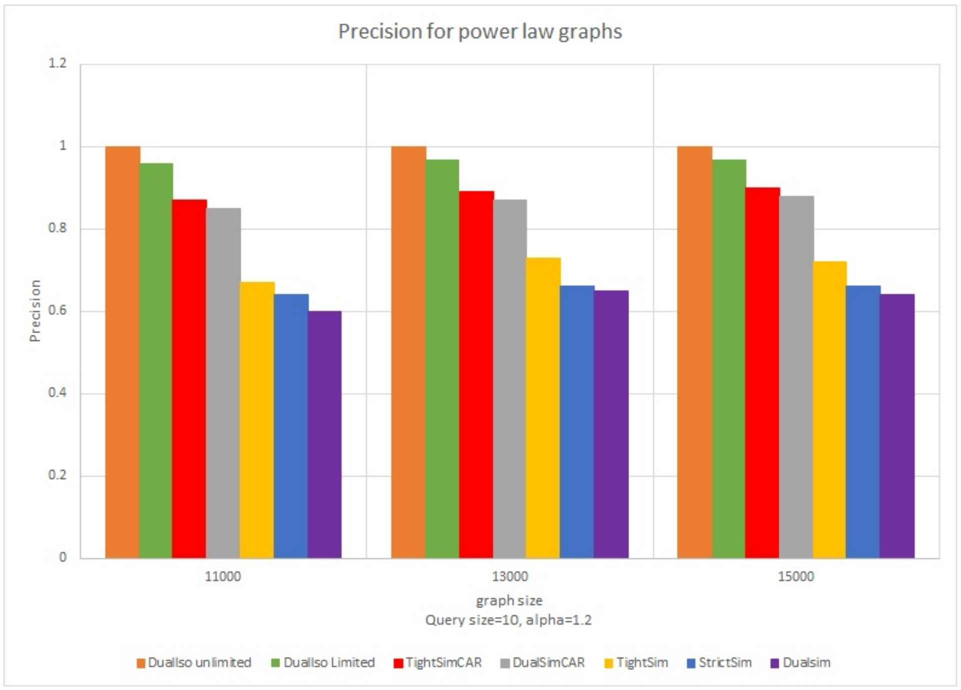

FIGURE 4.12: Precision for power law graphs

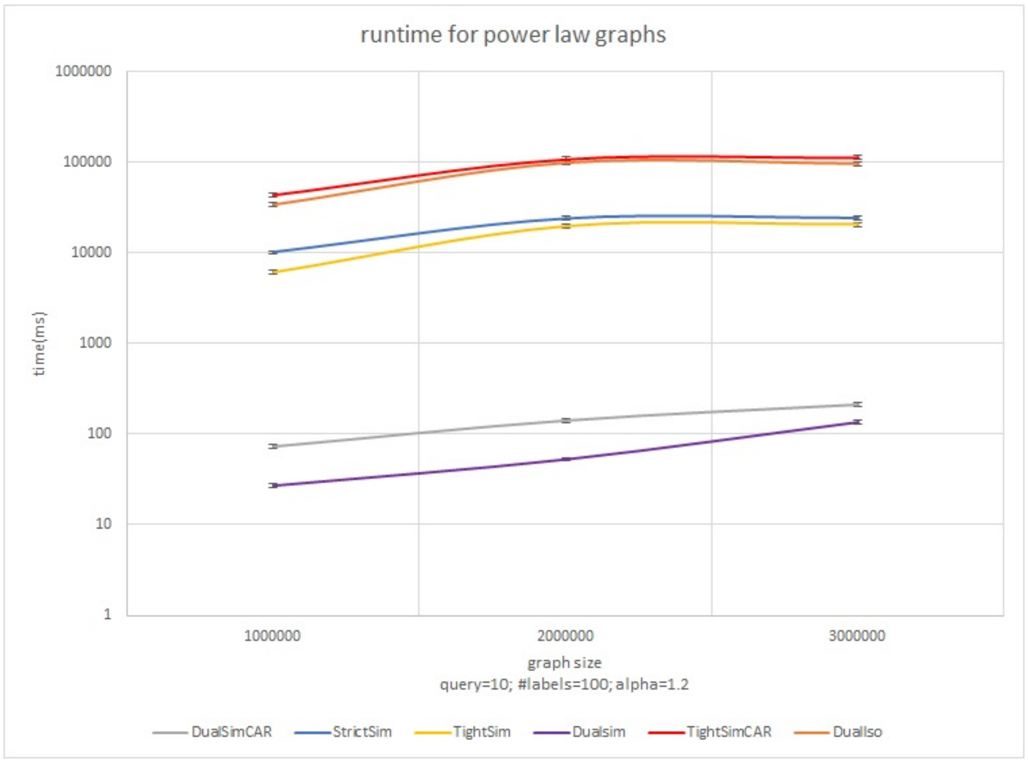

FIGURE 4.13: runtime for power law graphs

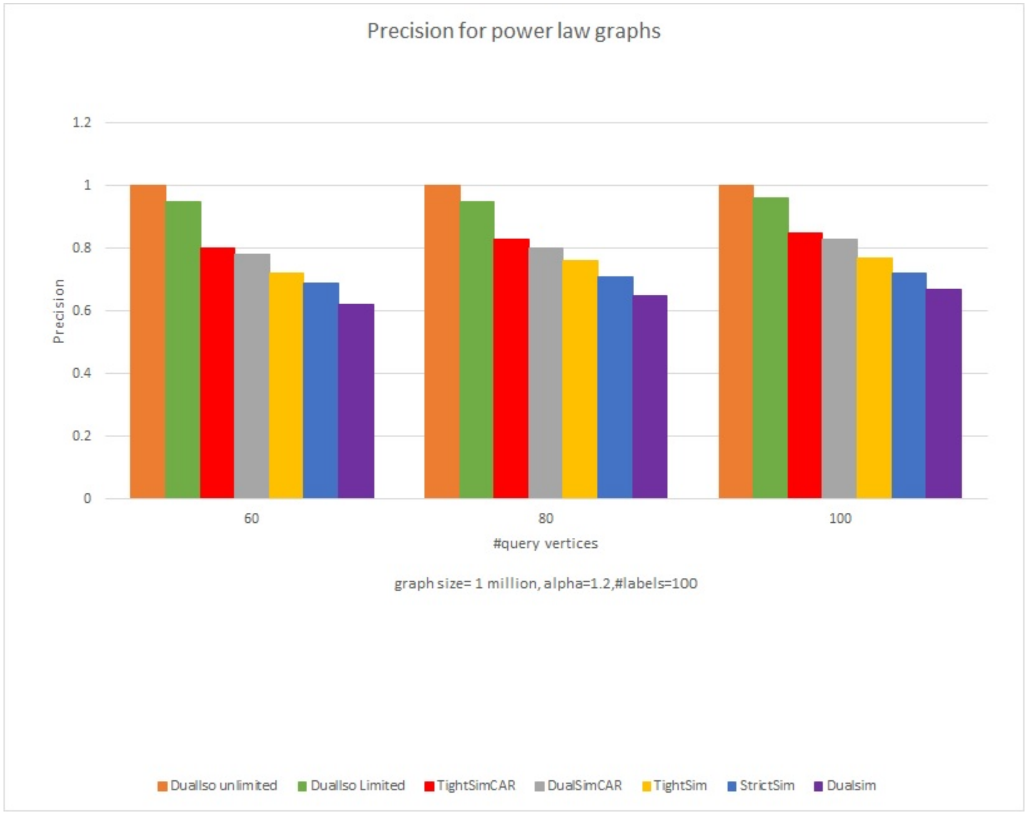

FIGURE 4.14: Precision for power law graphs with different query size

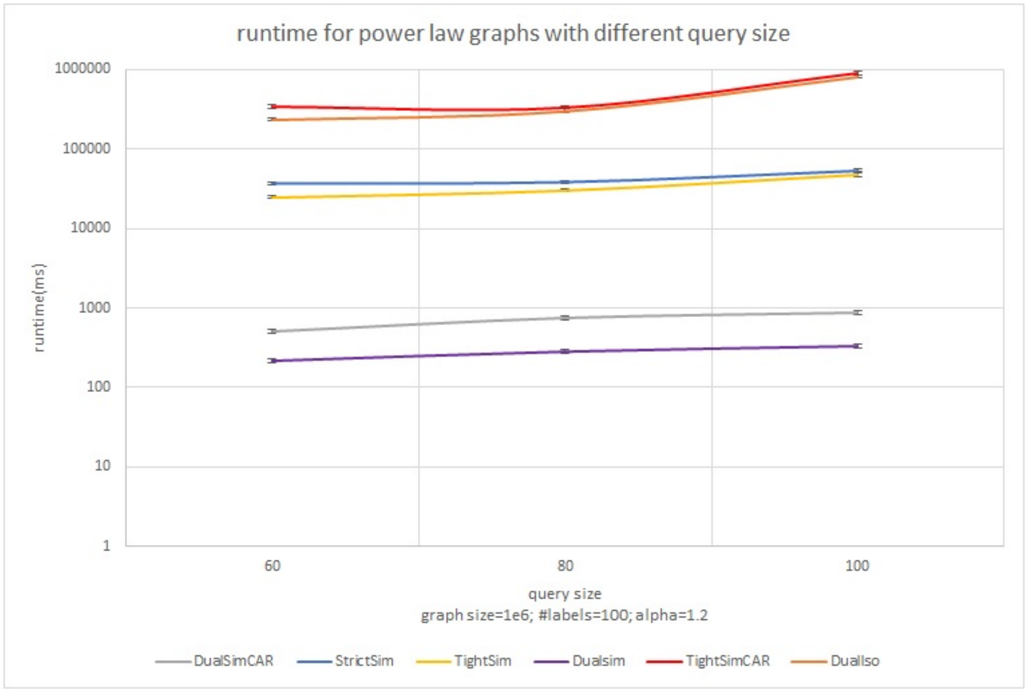

FIGURE 4.15: runtime for power law graphs with different query size

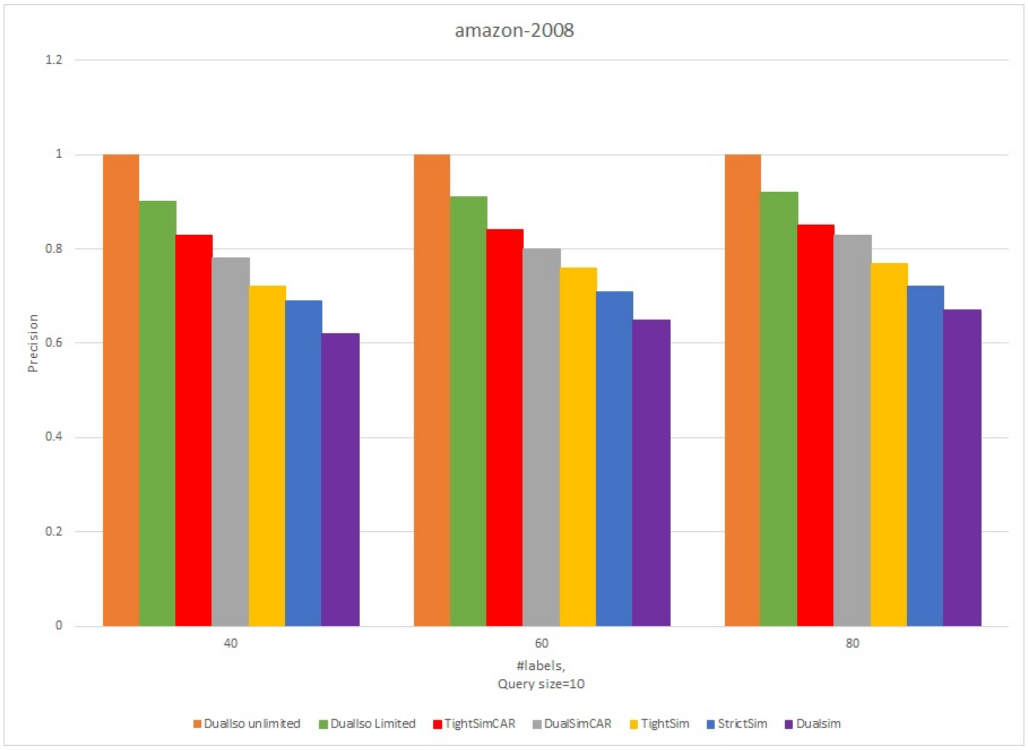

FIGURE 4.16: Precision for amazon-2008 dataset

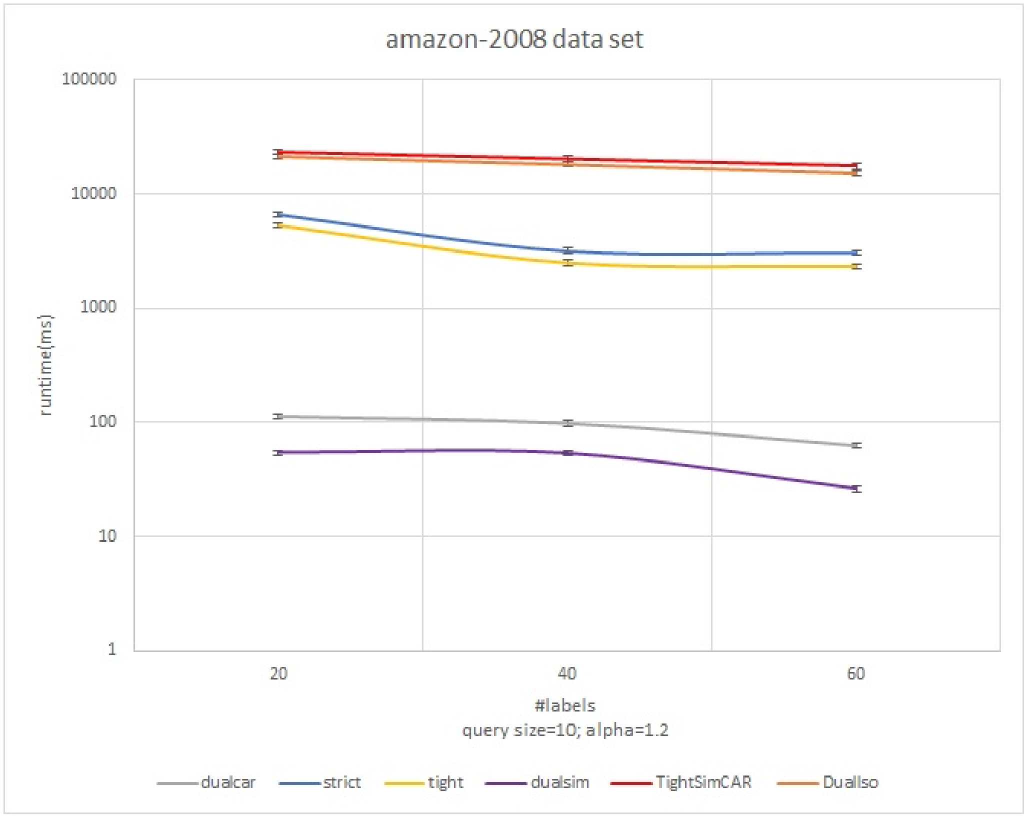

FIGURE 4.17: runtime for amazon-2008 dataset

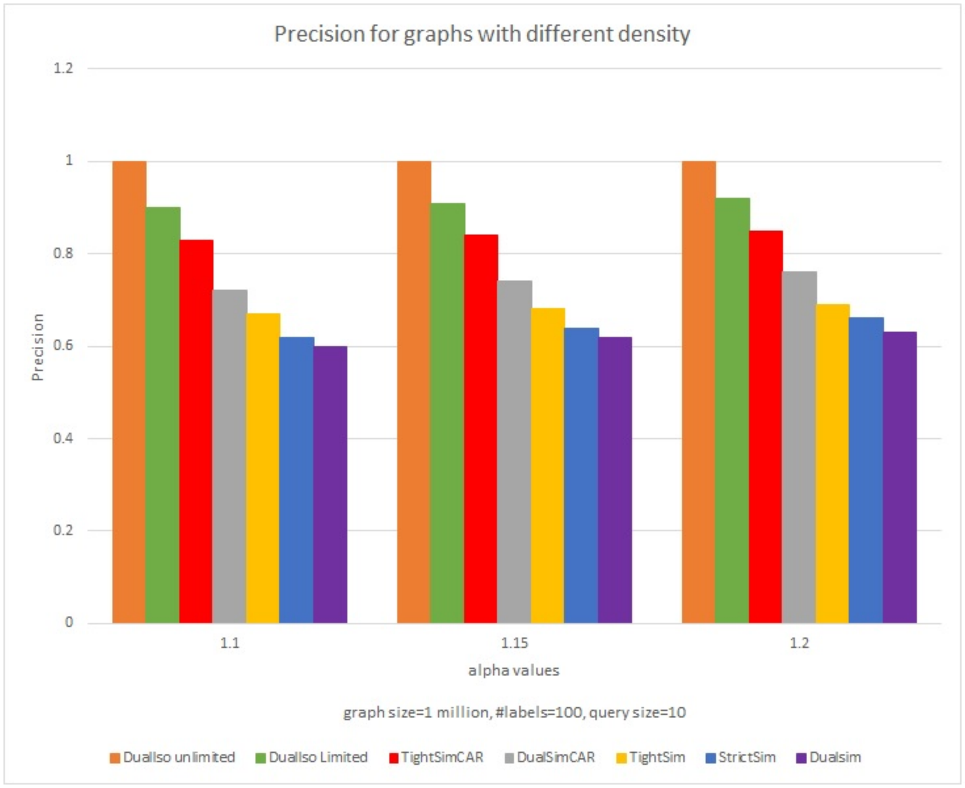

FIGURE 4.18: Precision for graphs with different density

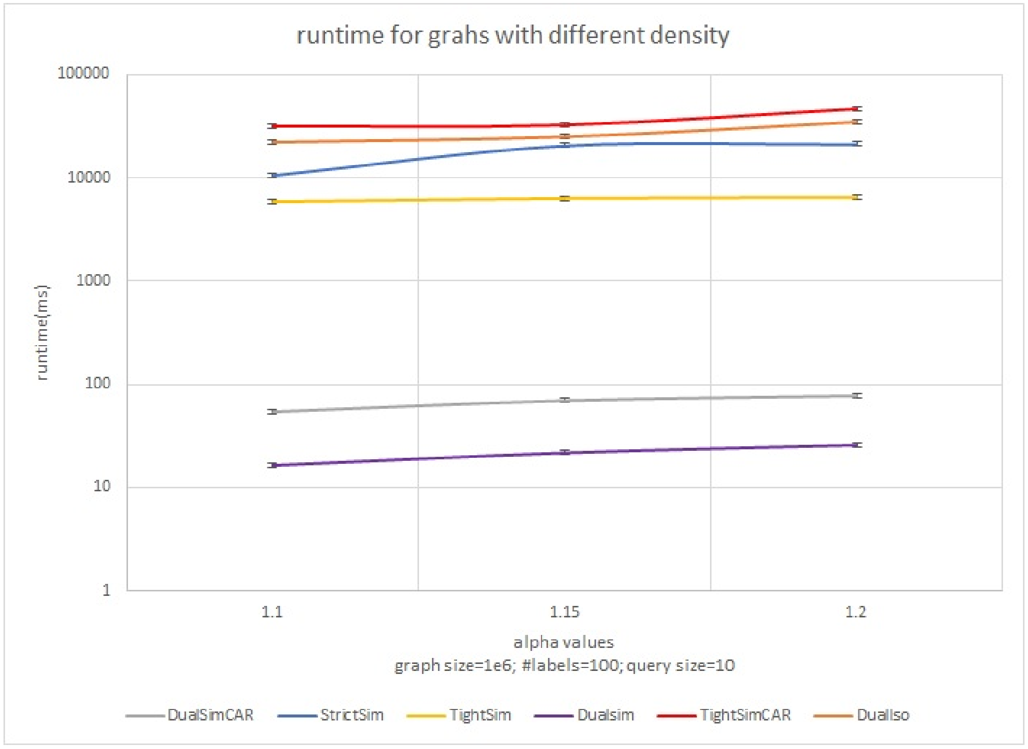

FIGURE 4.19: runtime for graphs with different density

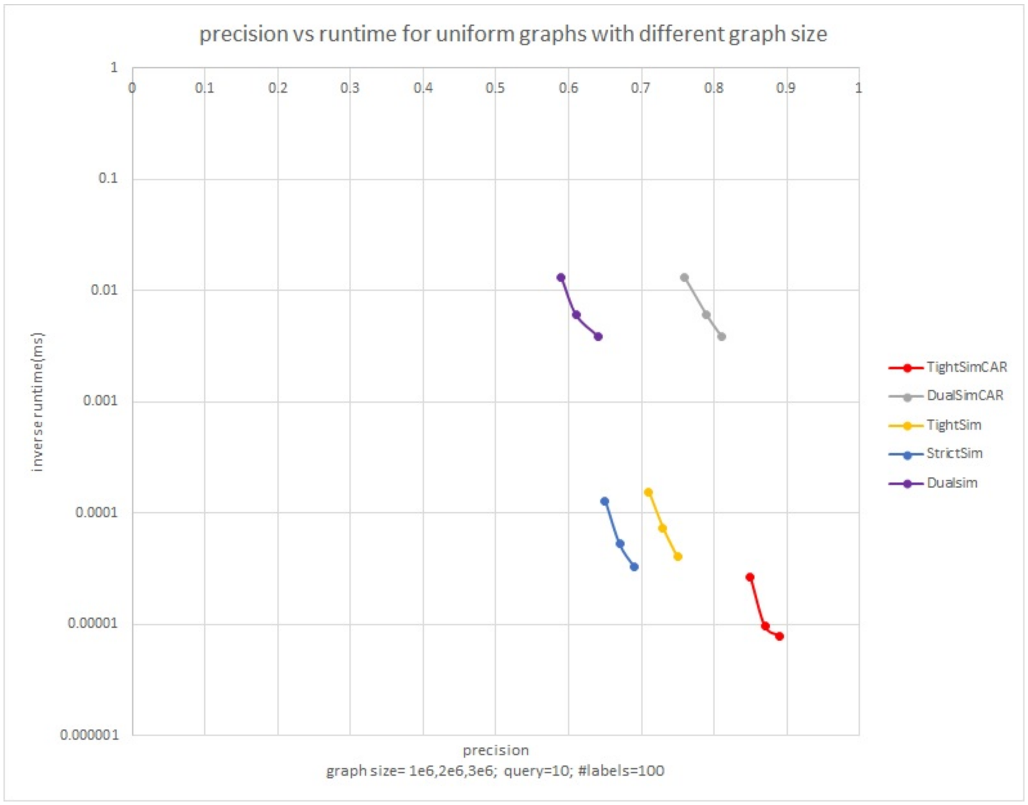

FIGURE 4.20: Precision vs runtime for uniform graphs

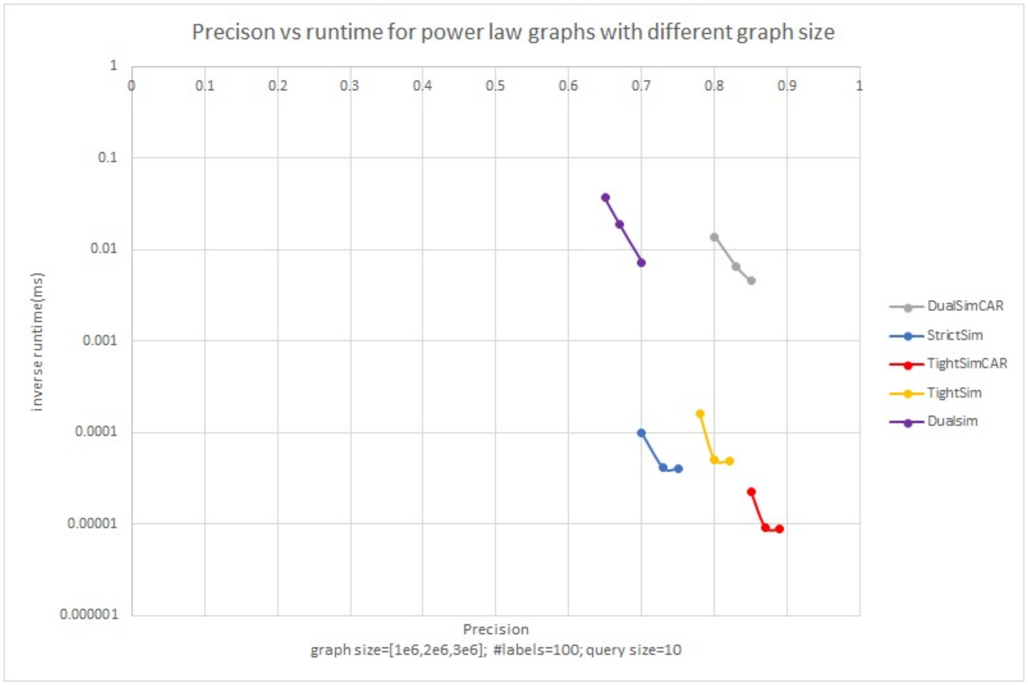

FIGURE 4.21: Precision vs runtime for power law graphs

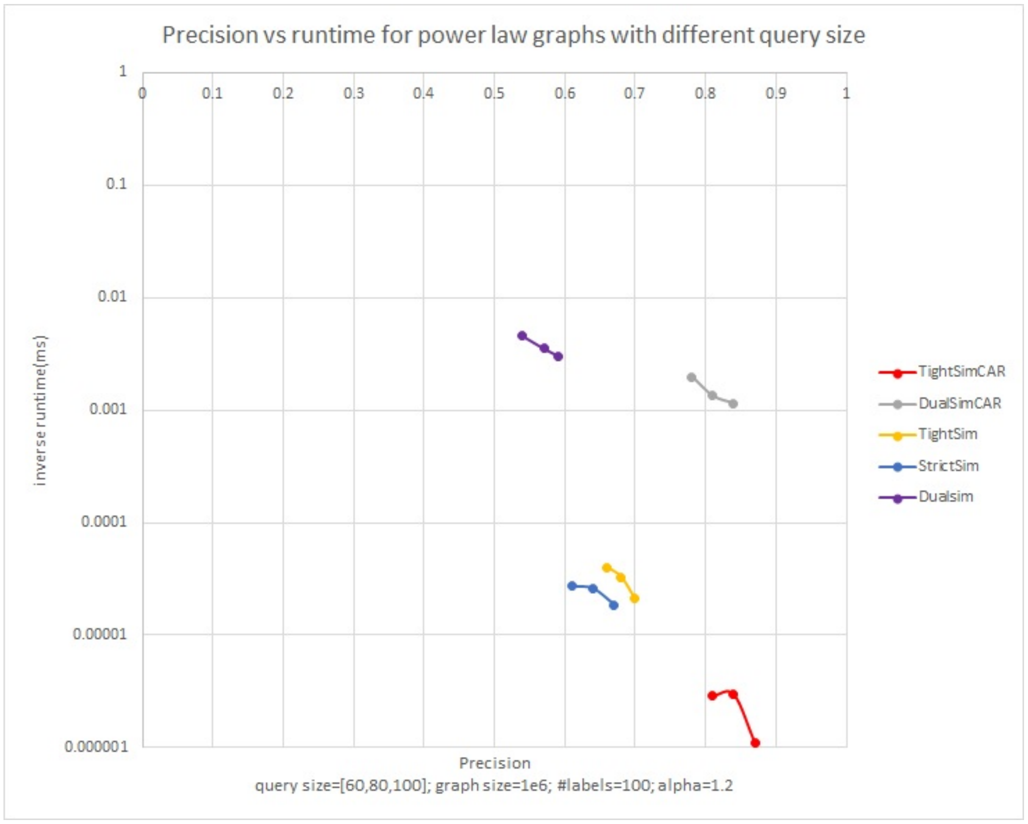

FIGURE 4.22: Precision vs runtime for power law graphs with different query size

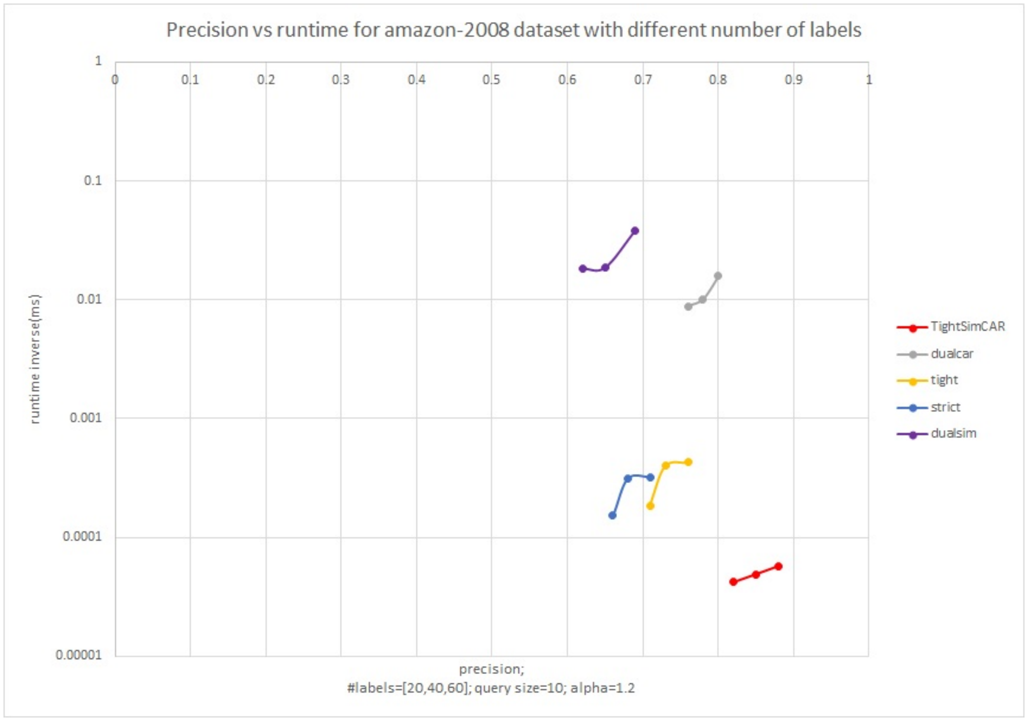

FIGURE 4.23: Precision vs runtime for amazon-2008 dataset with different label size

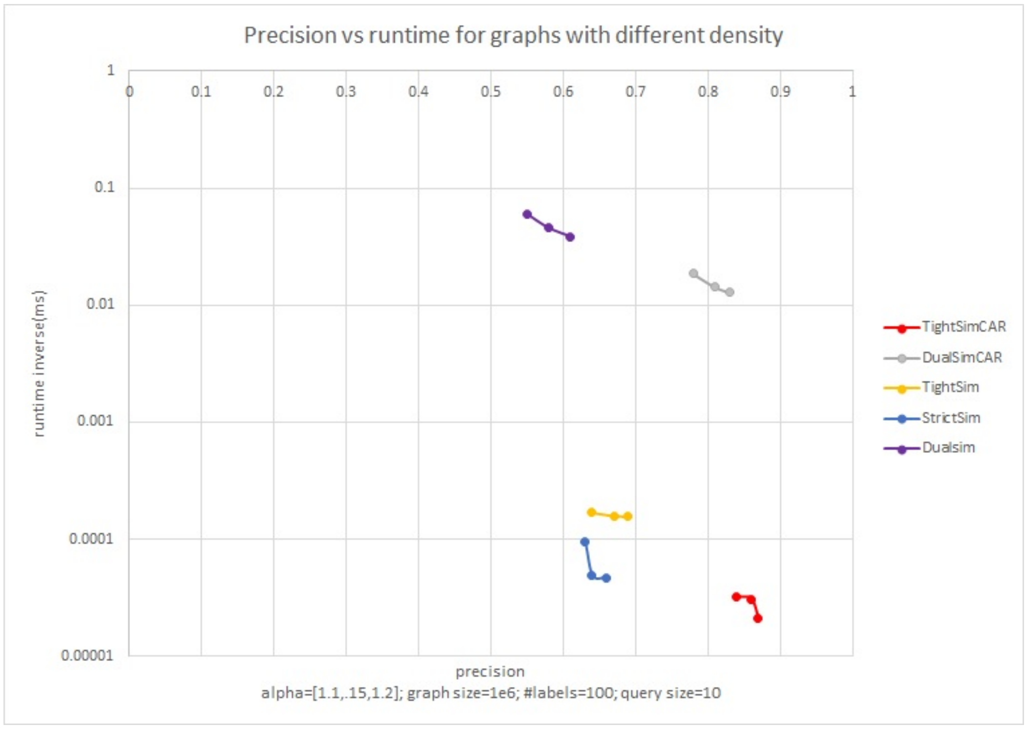

FIGURE 4.24: Precision vs runtime for graphs with different density

4.3.1 Edge Labeled Graphs

In this part we check the performance of the graph simulations with edge labeled graphs. Simulations supporting edge labeled graphs are represented with dashed lines. As we expected the performance of all the simulations supporting edge labeled graphs decreased because it has to check the edge labels. Figure 4.25 and Figure 4.26 show the performance with different graph size. Figure 4.27 and figure 4.28 show the effect of edge label size on the runtime. We can see that as the number of edge label increases the runtime decreases.

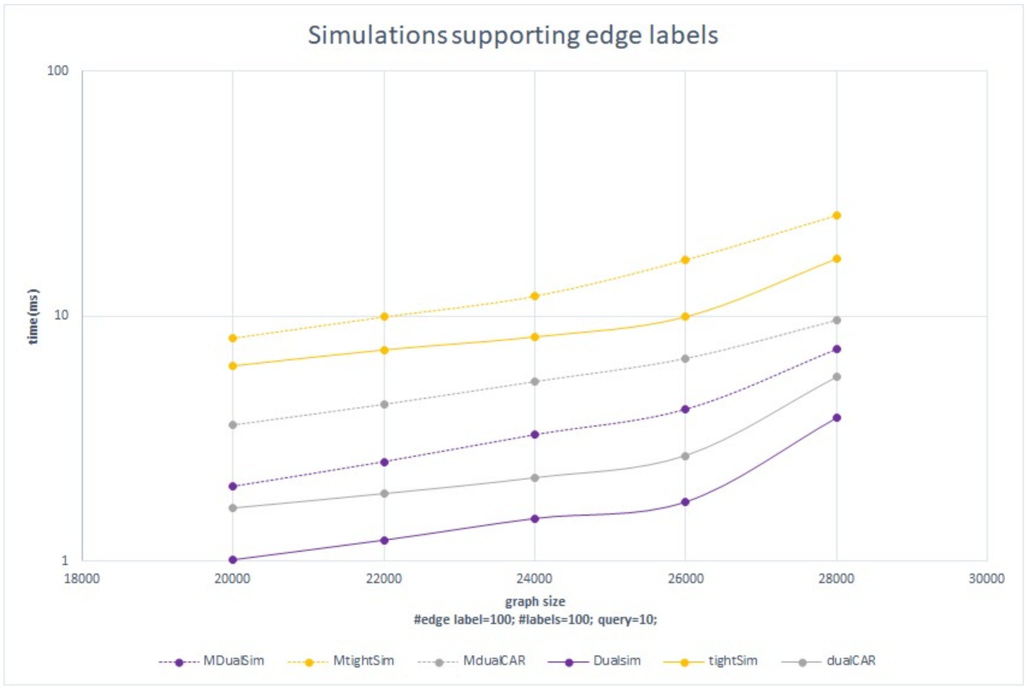

FIGURE 4.25: runtime for edge labeled power law graphs with different graph size

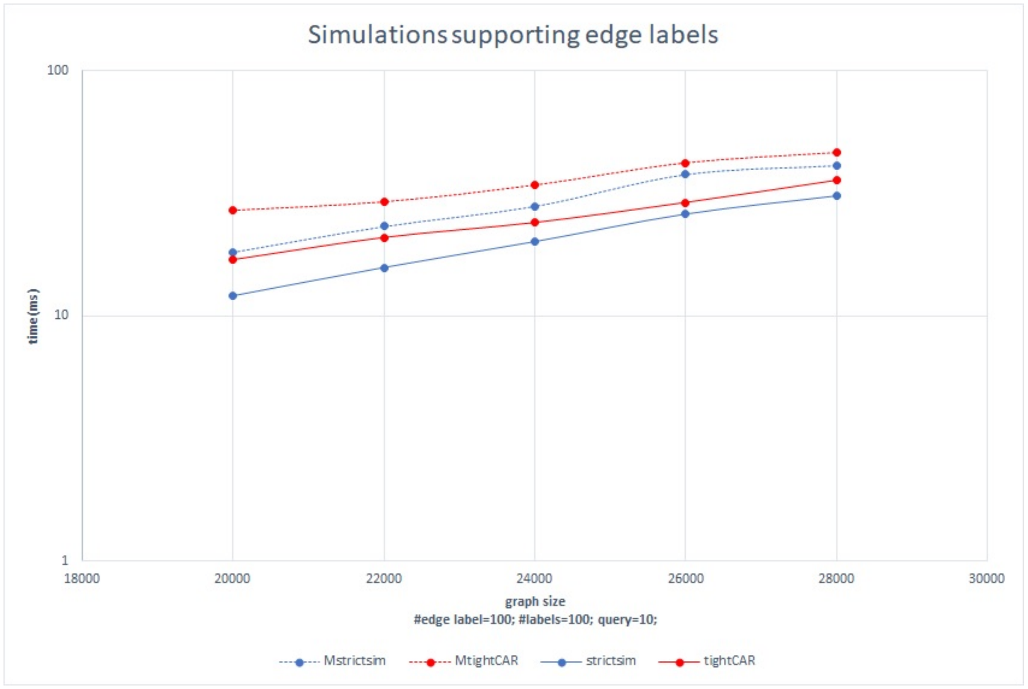

FIGURE 4.26: runtime for edge labeled power law graphs with different graph size

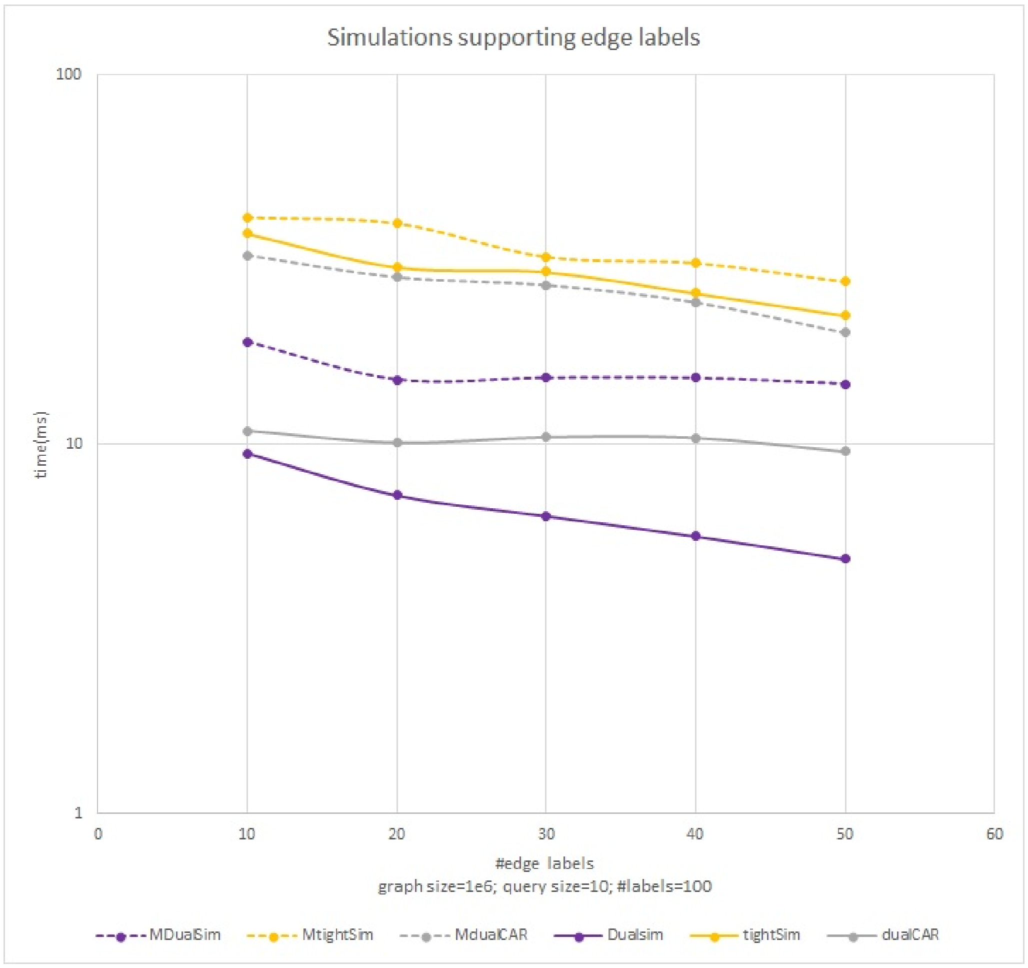

FIGURE 4.27: runtime for edge labeled power law graphs with different edge label size

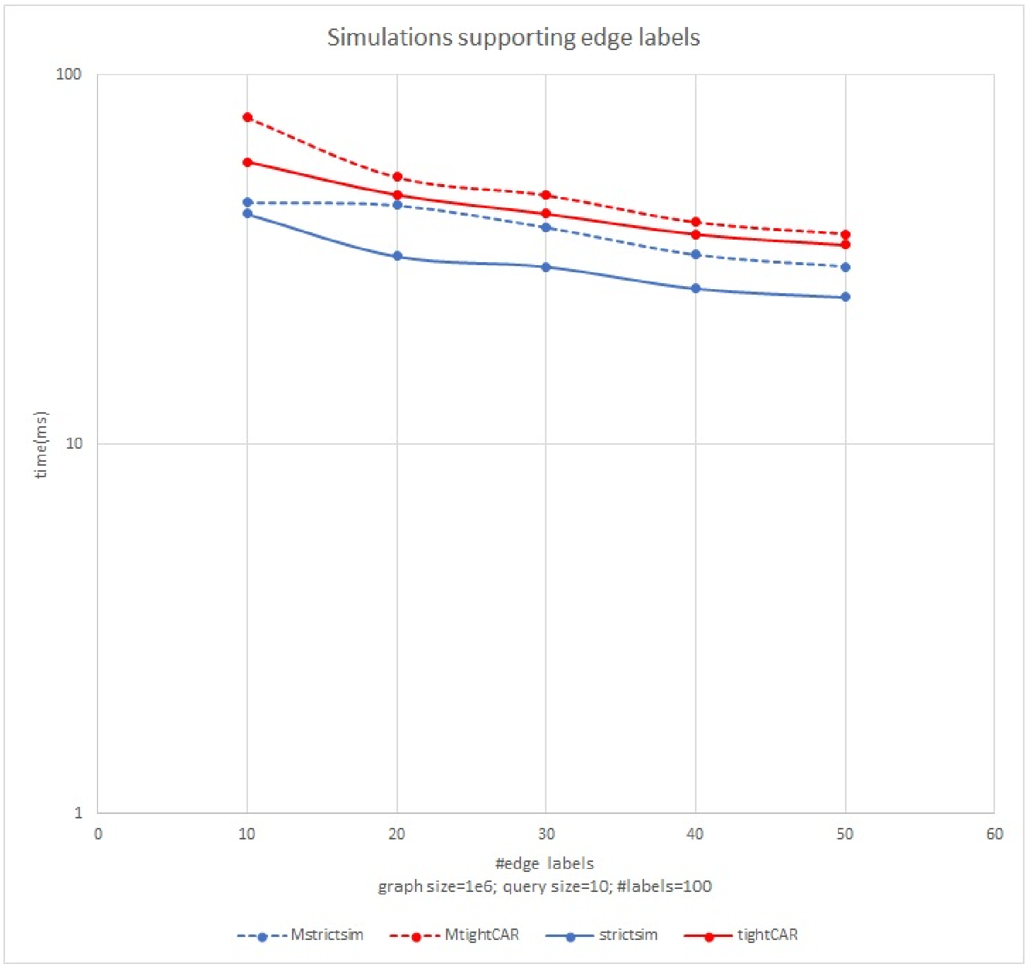

FIGURE 4.28: runtime for edge labeled power law graphs with different edge label size

4.3.2 Mutable vs Immutable

ScalaTion has two versions of each simulation; one works with mutable collections and the other uses immutable collections. A mutable collection can be updated or extended in place while a immutable collection never change. As expected simulations that work with mutable collections have a better performance than simulations using immutable colllections. This can be seen in Figure 4.29 and Figure 4.30. 100,000 vertices was used for data graph size, 10 for query size and $\alpha = 1.2$.

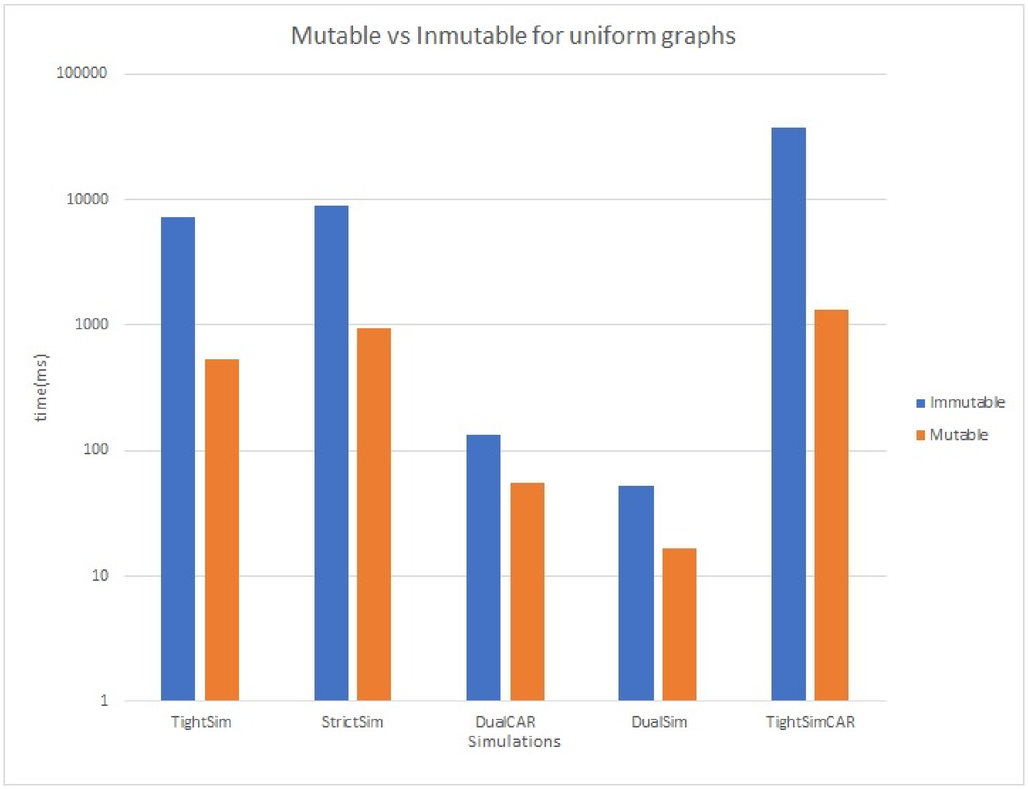

FIGURE 4.29: Mutable vs Immutable for uniform graphs

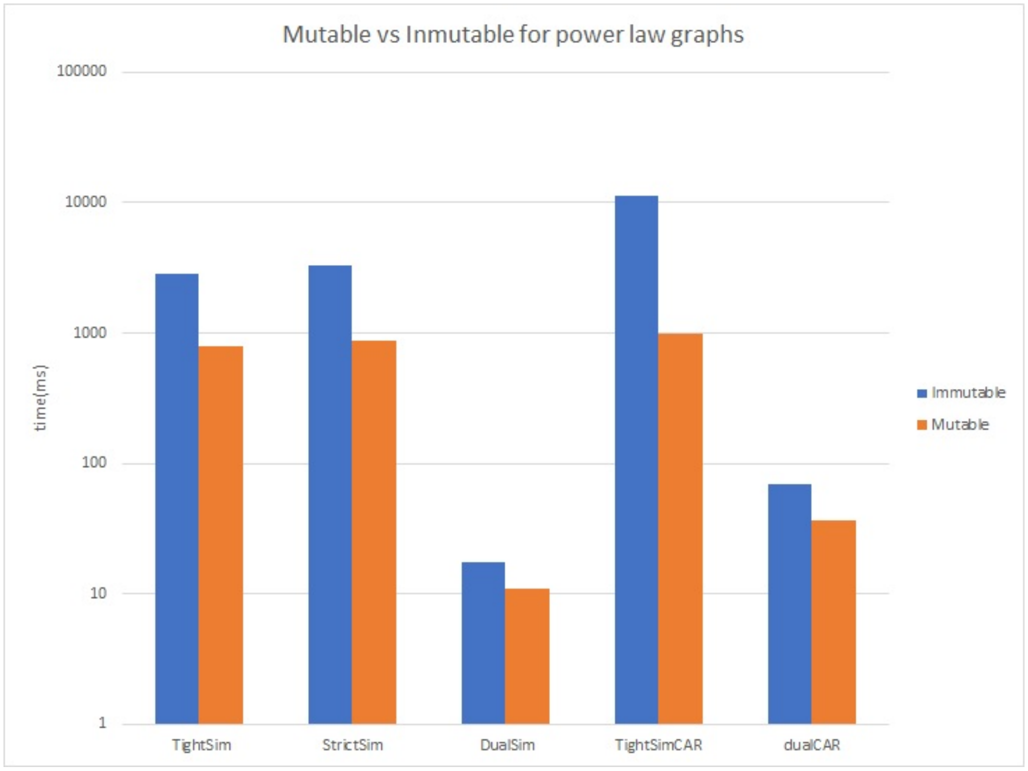

FIGURE 4.30: Mutable vs Immutable for power law graphs

4.4 Cardinality in graphs

Dual simulation does not consider cardinality of vertices in the parent or child relationships. In many real world graphs like online purchasing and social networks, many vertices may have the same label. According to the dynamics of viral marketing[18], the amazon purchasing network is a graph where if a product a is frequently purchased with a product b, there is an edge between a and b. For example we can have a children's book and two music cds with the same label. In this case we will have a children's book A connected with two music cds that share the same label B. Using this idea we have modified the way uniform and power law graphs are generated in order to benefit the cardinality restriction. For each data graph, every vertex have at least two children with the same label and two parents with the same label.

In these experiments we are comparing DualIso that uses DualSim simulation and DualIsoCAR that uses DualSimCAR simulation. We refer as version 1 to the regular method to generate graphs and version 2 to the new method to generate graphs where the cardinality restriction can be applied to any vertex in the data graph.

In figures 4.31 and 4.32 we have used different data sizes for uniform and power law graphs respectively. Query size effect is tested in figures 4.33 and 4.34. Finally, figures 4.35 and 4.36 show the effect of label size for version 1 and version 2. These experiments tell us that there is average improvement of 33 % in the runtime for DualIsoCAR when we generate graphs using the version 2 method.

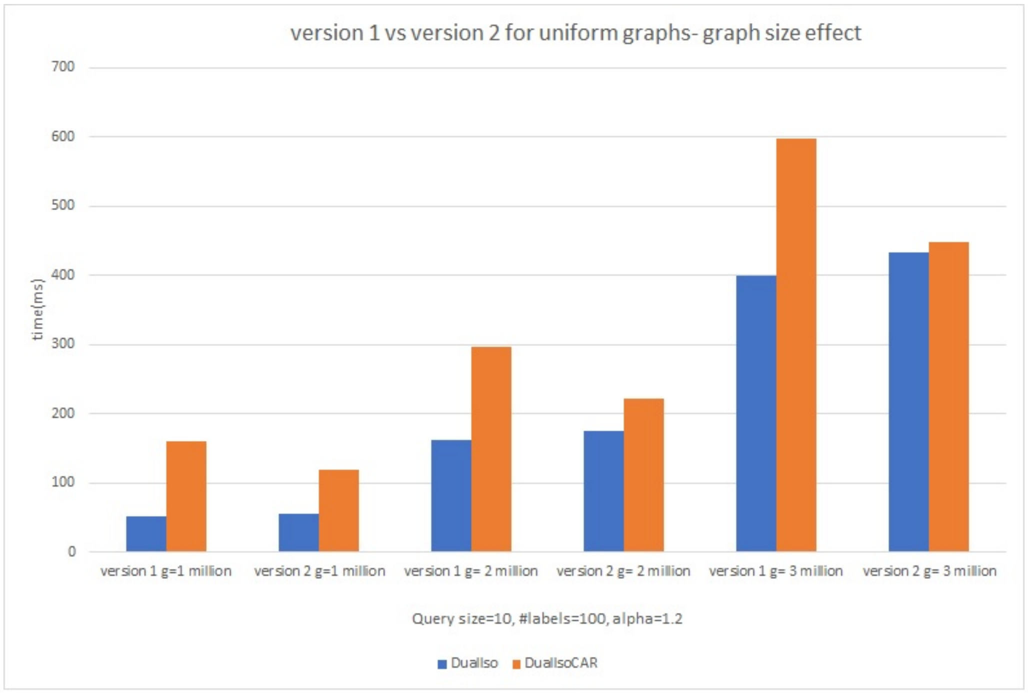

FIGURE 4.31: version 1 vs vesion 2 for uniform graphs- graph size effect

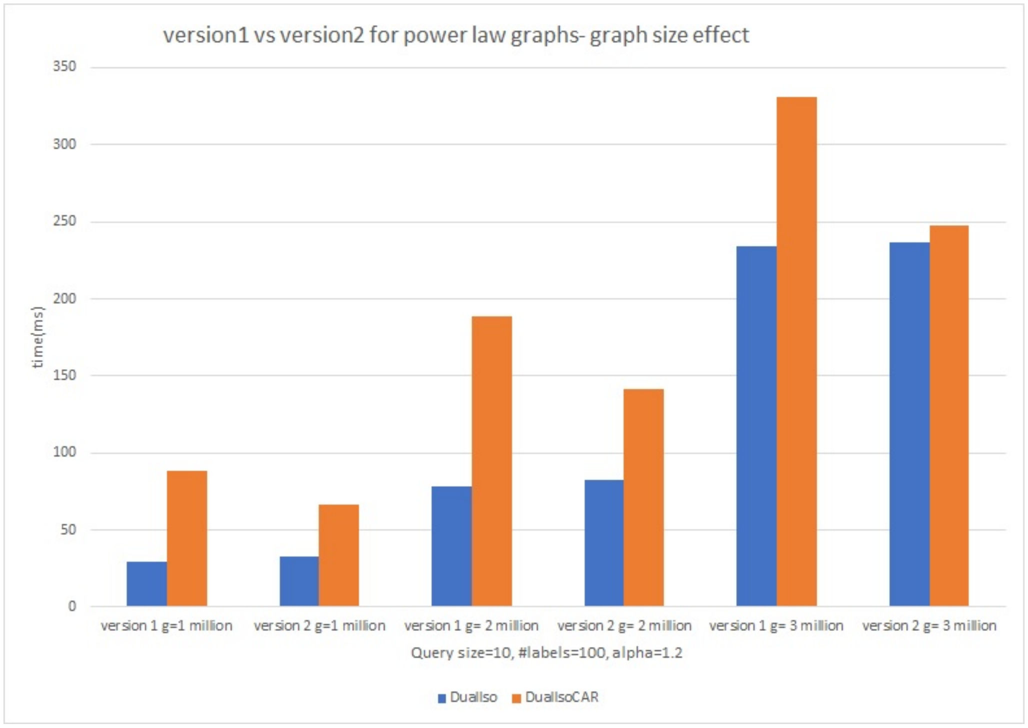

FIGURE 4.32: version 1 vs vesion 2 for power law graphs- graph size effect

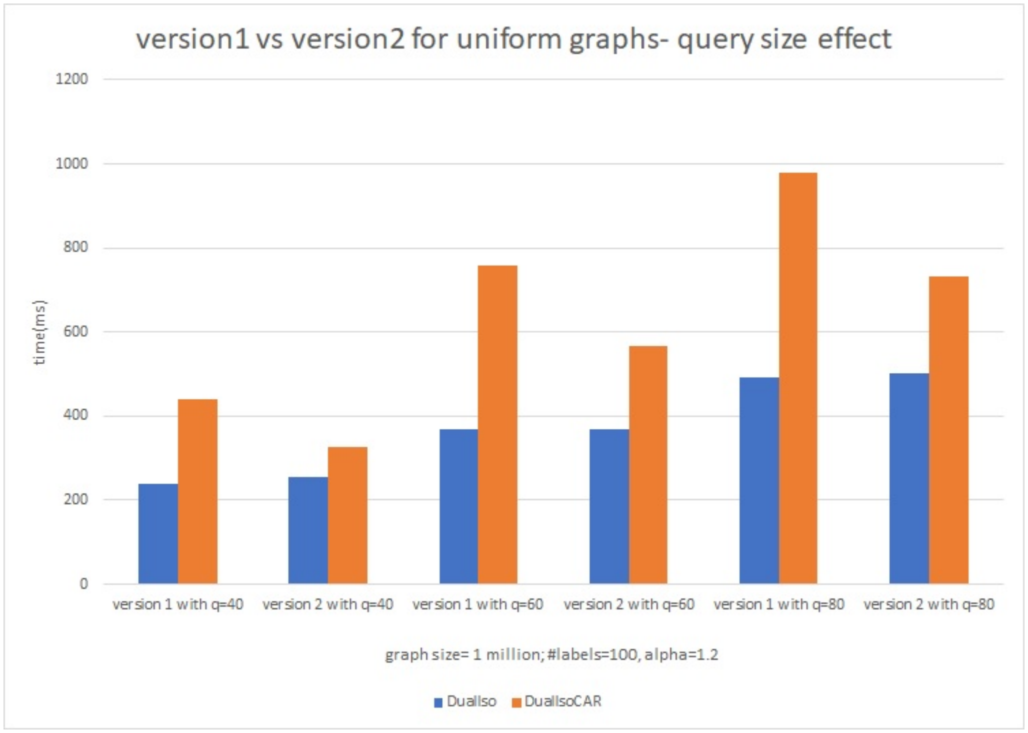

FIGURE 4.33: version 1 vs vesion 2 for uniform graphs- query size effect

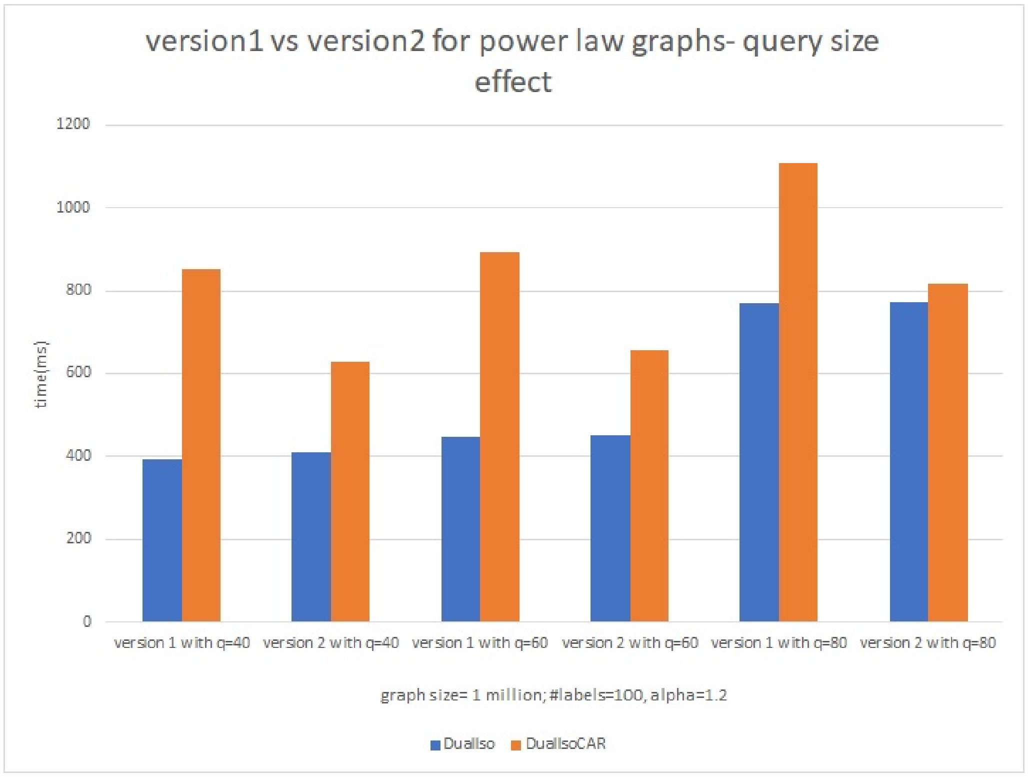

FIGURE 4.34: version 1 vs vesion 2 for power law graphs- query size effect

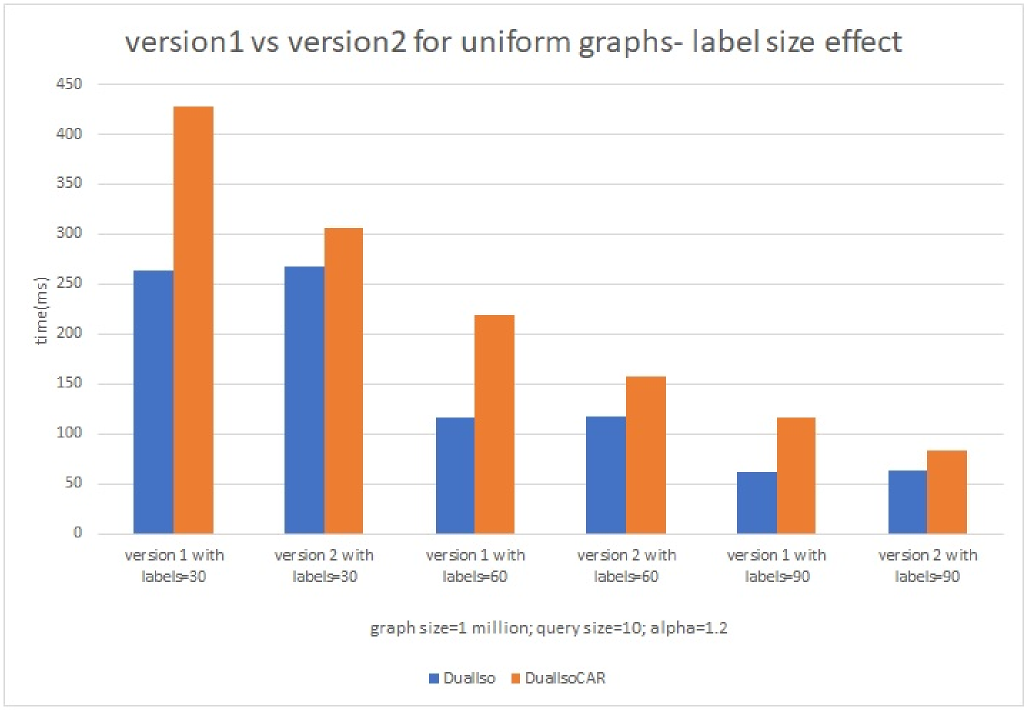

FIGURE 4.35: version 1 vs vesion 2 for uniform graphs- label size effect

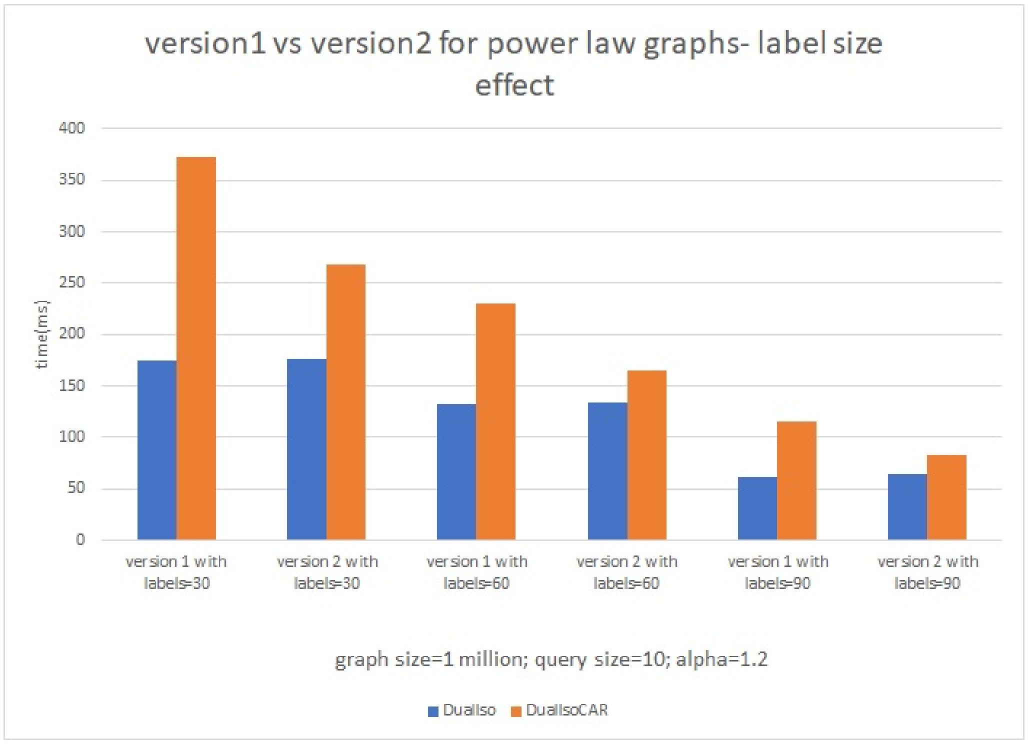

FIGURE 4.36: version 1 vs vesion 2 for power law graphs- label size effect

Chapter 5

Conclusion and Future Work

In this research project we have implemented the dual cardinality simulation expecting it to solve the problem of subgraph isomorphism strictness. We have introduced a new technique called count sets to reduce the number of possible matches.

The experimentation results show that dual cardinality simulation to be stable and have demonstrated on both synthetic and real world graphs. Although Dual-IsoCAR , that uses cardinality simulation algorithm, is not the fastest, it shows a good average performance better than known algorithms like VF2 and GraphQL. Moreover, dual cardinality simulation proves it has the best precision and is much faster than Tight and Strict simulation. Our last experimentation results show that, when cardinality restriction simulation is applied to graphs where many vertices have the same label(like the amazon recommendations graph) and there is always a child or parent relationship for any vertex in the data graph , there is a improvement of 33 % in the algorithm performance.

Future work can be extended to find an advanced implementation of optimized search order, extending the algorithm to work with multiple vertex attributes, multiple edges and edge attributes. Because the bottleneck of our algorithm is when it checks for the labels cardinality in parents and children, faster ways to check this restriction would benefit the runtime of the algorithm.

Another interesting future study can be experimenting with real-life datasets in various domains, to identify areas in which Dual Cardinlity Simulation is most effective.

Precision was one of the main experimentation we have done. Finding ways to improve precision can be beneficial in some areas where high precision is required. There may be some other areas where high precision is not required, so implementing Dual Cardinality Simulation with precision as a variable can be an innovative future work.

Combine other students research projects like graph algebra operations and Regular expressions for vertex and edge labels with Dual Cardinality Simulation can be an important future research direction.

Bibliography

[1] M. Saltz, A. Jain, A. Kothari, A. Fard, J. A. Miller, and L. 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

[2] 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.

[3] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. IEEE transactions on pattern analysis and machine intelligence, 26(10):1367-1372, 2004.

[4] U. degli studi di Milano. The laboratory for web algorithmics - law.

[5] A. Fard, S. Manda, L. Ramaswamy, and J. A. Miller. Effective caching techniques for accelerating pattern matching queries. In Big Data (Big Data), 2014 IEEE International Conference on, pages 491-499. IEEE, 2014.

[6] A. Fard, M. U. Nisar, J. A. Miller, and L. Ramaswamy. Distributed and scalable graph pattern matching: Models and algorithms. International Journal of Big Data (IJBD), 1(1):1-14, 2014.

[7] A. 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. IEEE, 2013

[8] A. J. Z. Fard. Subgraph Pattern Matching: Models, Algorithms, and Techniques. PhD thesis, University of Georgia, Athens, GA, 2014.

[9] W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In Proceedings of

the 2013 ACM SIGMOD International Conference on Management of Data, pages 337-348. ACM, 2013.

[10] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on, pages 453-462. IEEE, 1995.

[11] A. Jain. Parallel Algorithms for Subgraph Pattern Matching. Masters thesis, University of Georgia, Athens, GA, 2014.

[12] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Capturing topology in graph pattern matching. Proceedings of the VLDB Endowment, 5(4):310-321, 2011.

[13] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. ACM Transactions on Database Systems (TODS), 39(1):4, 2014.

[14] M. U. Nisar, A. Fard, and J. A. Miller. Techniques for graph analytics on big data. In 2013 IEEE International Congress on Big Data, pages 255-262. IEEE, 2013.

[15] S. Zhang, S. Li, and J. Yang, Summa: subgraph matching in massive graphs, in Proceedings of the 19th ACM international conference on Information and knowledge management, ser. CIKM 10. New York, NY, USA: ACM, 2010, pp. 12851288. [Online]. Available: http://doi.acm.org/10.1145/1871437.1871602

[16] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke, Computing simulations on finite and infinite graphs, in Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on. IEEE, 1995, pp. 453-462.

[17] J. R. Ullmann, An algorithm for subgraph isomorphism, Journal of the ACM (JACM), vol. 23, no. 1, pp. 31-42, 1976.

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

[19] John Miller. Big Data Simulation using ScalaTion, 2014.

[20] Huahai He, Ambuj K. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases, 2008