INVESTIGATION OF TECHNIQUES TO INCREASE THE SCALABILITY OF GRAPH BASED DATA MINING ALGORITHMS

by

SRILATHA INAVOLU

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2006

ACKNOWLEDGEMENTS

I would like to thank Dr. Diane J. Cook for her invaluable guidance throughout my research. I am grateful to Dr. Lawrence B. Holder for his suggestions and providing me with the SUBDUE discovery system and the synthetic graph generator, subgen. I would also like to thank Dr.Alp Aslandogan for being on my committee.

I would like to express my gratitude to Jeff Coble and Joe Potts for their support and encouragement.

Finally I would like to thank my family and friends for their constant support and encouragement through out my academic career.

April 3, 2006

ABSTRACT

INVESTIGATION OF TECHNIQUES TO INCREASE THE SCALABILITY OF GRAPH BASED DATA MINING ALGORITHMS

Publication No. _____

Srilatha Inavolu, M.S.

The University of Texas at Arlington, 2006

Supervising Professor: Diane J. Cook

Frequent subgraph pattern recognition and graph-based relational learning have been an emerging area of data mining research with scientific and commercial applications. At the kernel of these algorithms are the computationally-expensive graph and subgraph isomorphism tests.

The graph isomorphism problem consists in deciding whether two graphs are isomorphic i.e., whether there is a one-one mapping between the vertices of the two graphs that respects the edge connections. Many graphs will be depicted quite differently but in actuality have the same inherent structure. This leads to the

isomorphism problem. The graph isomorphism problem belongs to the class of NP problems and has been conjectured intractable though probably not NP-complete.

We hypothesize that approximation algorithms can be developed for the graph and subgraph isomorphism problems, and that these algorithms can improve the runtime of data mining systems that rely on these capabilities. We analyze the validity of our hypothesis by implementing and testing three approaches to the problem: a genetic algorithm for subgraph isomorphism detection, canonical labeling of graphs for graph isomorphism testing and a technique that reduces the need for isomorphism tests in SUBDUE.

Canonical labeling is a technique that assigns a unique code to a graph that is invariant on the order of the vertices and the edges in the graph. As a result two graphs will have same canonical labels if they are isomorphic and vice versa. In cases where many isomorphism checks are required between same set of graphs, a better way of performing this task is to assign each graph a canonical label. Our research has considered canonical labeling technique in SUBDUE to reduce the number of calls to the graphMatch routine and also as an alternative to the graphMatch routine.

The subgraph isomorphism problem consists in deciding whether a graph is isomorphic to a subgraph of another graph. Subgraph isomorphism belongs to the class of NP-complete problems.

Genetic algorithms represent an approximation technique that runs for a certain number of generations, retaining the best chromosomes from the current generation to the next generation and producing new chromosomes using the genetic operators of

selection, crossover and mutation. This approach is inspired by the natural process of evolution. In our research, a genetic algorithm approach has been considered for subgraph isomorphism detection in SUBDUE to find the instances of the predefined substructures in the input graphs. Since it is an approximation technique, it is not guaranteed to find all the instances of the subgraph in the main graph.

Finally an approach taken by Potts is analyzed that reduces the number of calls to graphMatch by changing the order in which the instances are extended in SUBDUE.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

1.1 Motivation

The power of graphs to model complex data sets has been recognized by various researchers as it allows us to represent arbitrary relations among entities and solve problems that we could not previously solve. There are two types of settings, a graph transaction setting and a single transaction setting.

The problem of frequent pattern discovery in a graph transaction setting is formulated as that of discovering frequent subgraphs that occur over the entire set of graphs. The problem of frequent pattern discovery in a single transaction setting is formulated as that of discovering subgraphs that occur frequently over a single graph. FSG is an example of a frequent subgraph discovery algorithm that deals with graph transaction setting. SUBDUE is an example of a knowledge discovery algorithm that deals with both single transaction and graph transaction settings.

The problem of subgraph discovery involves computationally-expensive graph and subgraph isomorphism tests in both single transaction and graph transaction settings.

Subgraph isomorphism detection is an NP-complete problem. It is still an open question if graph isomorphism detection is an NP-complete problem; however there are no known polynomial time algorithms for graph isomorphism detection.

There are two basic approaches that past research has taken to address the problem of graph isomorphism. The first approach is based on group-theoretic concepts and the study of permutation groups. In Babai [8], it was shown that there exists a moderately exponential bound for the general graph isomorphism problem. Some of the known algorithms reduce the computational complexity of the matching process by imposing topological restrictions on the graphs. Algorithms for finding isomorphism between planar graphs [11], trees [22] or more generally bounded valence graphs [9] are known.

However, the major drawback of algorithms based on group-theoretic concepts is that there is usually a large overhead and consequently a large constant factor associated with the theoretical complexity.

The second approach to graph and subgraph isomorphism is more practically oriented and aims directly at developing an algorithmic procedure for isomorphism detection. Most of these algorithms are based on a state-space search with backtracking. A major improvement of the backtracking method was presented by Ullman, who introduced a refinement method which reduces the search space of the backtracking procedure remarkably [12]. More recent work is described in [13, 14] where the graph isomorphism problem was reduced to the problem of clique detection by constructing an association graph for all possible vertex mappings.

1.2 Hypothesis

We hypothesize that the scalability of graph-based data mining algorithms can be increased by making use of approximation techniques or by reducing the number of calls to the computationally-expensive graph match.

A genetic algorithm can be used as an approximation technique to perform subgraph isomorphism detection. Canonical labeling can be used to assign each graph a unique label that is invariant on the order of vertices and edges in the graph. To find out if two graphs are isomorphic, it is enough to compare their canonical labels. Hence the computationally-expensive graph matching is reduced to a polynomial-time string matching problem.

The number of calls to graphMatch can be reduced by changing the order of extending the instances of a substructure in SUBDUE. We validate our hypothesis by implementing and analyzing these three approaches on real and synthetic data.

1.3 Contributions

Our research has implemented a genetic algorithm for subgraph isomorphism detection and adapted a canonical labeling algorithm for use by SUBDUE. These techniques have been implemented on real and synthetic databases. A technique that reduces the number of calls to the graphMatch routine in SUBDUE has been described and analyzed.

Chapter 2 gives an overview of the graph-based knowledge discovery system "SUBDUE". Chapter 3 defines the concept of a genetic algorithm, describes the implementation of the genetic algorithm for subgraph isomorphism detection in

SUBDUE and shows the experimental results. Chapter 4 defines canonical labeling, describes the usage of the canonical labeling algorithm "nauty" in SUBDUE and shows the experimental results. Chapter 5 gives an overview of EE-SUBDUE, the efficiency-enhanced version of SUBDUE, obtained by reducing the number of calls to graphMatch. Finally, chapter 6 outlines the contributions of our research and suggests future work.

CHAPTER 2

SUBDUE

In this chapter we give an overview of SUBDUE. SUBDUE is a graph-based knowledge discovery system that has been used as the platform to implement and analyze the scalability-techniques proposed in the thesis.

2.1 Introduction

SUBDUE [1] is a graph-based knowledge discovery system that finds structural and relational patterns in data, representing entities and relationships. SUBDUE represents data using a labeled, directed/undirected graph in which entities are represented by labeled vertices or subgraphs, and relationships are represented by labeled edges between the entities. SUBDUE uses the minimum description length (MDL) principle, introduced by Rissanen [2] to identify patterns that minimize the number of bits needed to represent the input graph after being compressed by the pattern. SUBDUE can perform several learning tasks, including unsupervised learning, supervised learning, clustering and graph grammar learning. SUBDUE has been successfully employed in a number of areas including web structure mining, counter terrorism, social network analysis, aviation and geology.

1 .*Graph-based unsupervised learning (DISCOVERY)*

In discovery mode SUBDUE uses heuristic search guided by MDL to find patterns minimizing the description length of the entire graph compressed with the

pattern [3]. Once a pattern is found, SUBDUE can compress the graph using this pattern. It replaces the instances of the substructures with a node which is a pointer to the substructure. It can repeat the process on the compressed graph to look for more abstract patterns possibly defined in terms of previously discovered patterns.

2. *Graph-based Supervised learning*

If graphs depicting both positive and negative examples are given as input then SUBDUE enters supervised learning mode [4], searching for a pattern that compresses the positive graphs but not the negative graphs. For example, given positive graphs describing criminal networks and negative graphs describing benign social networks, SUBDUE can learn patterns distinguishing the two, and these patterns can be used as a predictive model to identify emerging criminal networks. When SUBDUE is given a new graph, it can see if this graph contains the patterns identified during its learning procedure and if yes, then labels the new graph as a criminal network.

An evaluation method called set-cover is used to identify the best substructures. In this method a substructure is identified that best covers the positive graphs but not the negative graphs. Positive graphs that are covered are removed and the process is repeated until there are no more positive graphs.

3. *Graph-based hierarchical Clustering*

The ability of SUBDUE to iteratively discover patterns and compress the input graph can be used to generate a clustering of the input graph [5]. Essentially, clustering mode forces SUBDUE to iterate until the input graph can be compressed no further. The resulting patterns form a cluster lattice, such that if a pattern *S* is defined in terms of

one or more previously-discovered patterns, then these patterns form the parents of *S* in the lattice.

4. *Graph Grammar Learning*

Graph grammars are similar to string grammars where the terminals and non-terminals represent arbitrary graphs. SUBDUE learns context-free, node-replacement graph grammars by looking for common connections between the instances of a substructure S [6].

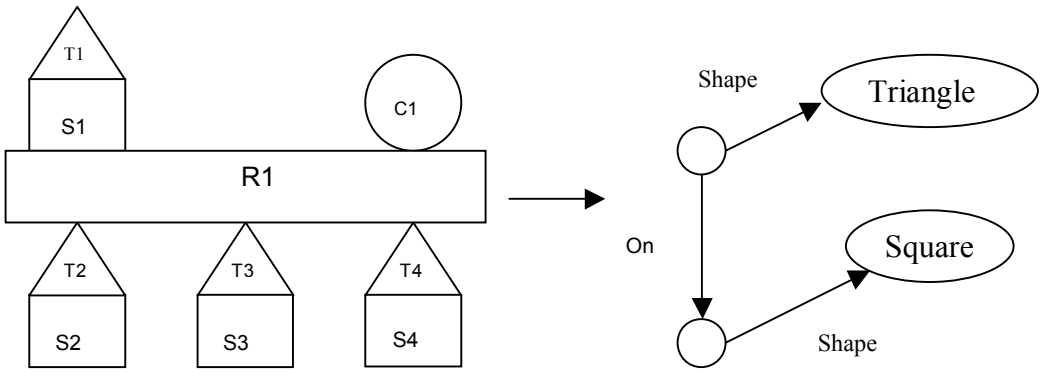


Figure 1: Example substructure in graph form.

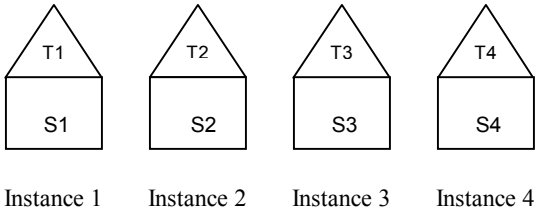


Figure 2: Instances of the substructure in the graph.

2.2 Algorithm

SUBDUE represents structured data in the form of a labeled graph. Objects in the data map to vertices or small subgraphs in the graph and the relationships between the objects map to the directed or undirected edges in the graph. A substructure is a connected subgraph that is part of a given input graph. A substructure instance is a set of edges and vertices in the input graph that is considered to match graph-theoretically to a given substructure.

SUBDUE performs a computationally-constrained beam search. The algorithm begins with the substructure matching a single vertex in the graph. During each iteration the best substructures are extended by one vertex and a connecting edge or one edge connecting vertices already in the substructure in all possible ways as guided by the example graphs, to generate candidate substructures. SUBDUE maintains the instances of substructures in examples and uses graph isomorphism to determine the instances of the candidate substructures in the examples. The substructures are then evaluated according to how well they compress the description length (*DL*) of the data set. The *DL* of the input dataset *G* using substructure *S* can be calculated using the formula, *DL(S) + DL(G|S),* where *S* is the substructure used to compress the graph dataset *G*. *DL(S)* and *DL(G|S)* represent the number of bits required to encode S and the dataset *G* after *G* has been compressed with *S*. The length of the beam represents the number of substructures left to be considered for further expansion. However if there are ties, all substructures of the last value are kept on the beam. The procedure stops after all

substructures are found or user-imposed computational constraints are exceeded. At the end of the procedure SUBDUE reports the best compressing substructures.

SUBDUE (*Graph, BeamWidth, MaxBest, MaxSubSize, Limit*)

1. *ParentList* = NULL;
2. *ChildList* = NULL;
3. *BestList* = NULL;
4. *ProceedSubs* = 0;
5. Create a substructure from each unique vertex label and its single-vertex instances;
6. Insert the resulting substructures in *ParentList;*
7. **while** *ProceedSubs* less than or equal to *Limit* and *ParentList* not empty
   - **a. do**
   - b. **while** *ParentList* not empty
     - **i. do**
       1. *Parent* = RemoveHead(*ParentList);*
       2. Extend each instance of *Parent* in all possible ways;
       3. Group the extended instances into *Child* substructures;
   - c. **for** each *child*
   - d. **do**
     - i. **if** Size of(*child)* less than *MaxSubSize*
     - **ii. then**
       1. Evaluate the child
       2. Insert *Child* in *ChildList* in order by value;
       3. if *BeamWidth* Less than Length(*ChildList*)
       4. **then**
          - **a.** destroy substructure at the end of *ChildList;*
   - **e.** Increment *ProceedSubs;*
   - **f.** Insert parent in *BestList* in order by value;
   - **g. if** *MaxBest* less than length(*BestList*)
   - **h. then**
     - **i.** destroy substructure at the end of *BestList;*
   - **i.** Switch *parentList* and *childList*;
8. **return** *BestList*

Figure 3: The SUBDUE algorithm.

The main processing loop of SUBDUE proceeds as follows. For the list of parents the while loop in line 7 (of Figure 3) is entered. A parent is removed from the list of parents and extended in all possible ways by one edge or one edge and one vertex. All of these extended instances are inserted into a sorted list, *childList,* whose size must be less than or equal to the *BeamWidth.*

Once all the children of a parent are processed, the parent is added to the *BestList.* When the current *ParentList* is emptied, the *ChildList* becomes the new *ParentList* and the loop is repeated until the number of substructure extensions exceeds *Limit* or there are no more extensions. The algorithm terminates and returns the *BestList* containing the best substructures found.

In addition, if the user has specified multiple iterations, then the graph is compressed with the best substructure and the process is repeated until there are no more positive graphs or the graph cannot be compressed further.

CHAPTER 3

GENETIC ALGORITHM

Genetic algorithms are used as approximation techniques for solving difficult problems. In this chapter a genetic algorithm has been considered to solve the NP-complete subgraph isomorphism detection problem. A genetic algorithm has been implemented in SUBDUE that performs an approximative subgraph isomorphism test to identify all the instances of the predefined substructures in the input graphs.

3.1 Definition of a genetic algorithm

Genetic algorithms were formally introduced by John Holland in the 1970s at the University of Michigan [7]. Genetic algorithms are adaptive heuristic search algorithms premised on the evolutionary ideas of natural selection and Darwin's principle of *Survival of the fittest*. They perform a parallel, non-comprehensive search for the global maximum of the graph. The search is not precise in that it does not guarantee that the global maximum will be found. We are in a stochastic system and a genetic pool may be too far from the solution, or a too-fast convergence may halt the process of evolution. Genetic algorithms have been successfully used to solve NP-complete problems like Traveling Salesman Problem and Knapsack problem. They are well suited for problems such as evolving the weights of a neural network.

A genetic algorithm is an iterative procedure that consists of a constant-size population of individuals, each one represented by a finite string of symbols, known as

chromosome, encoding a possible solution in a given problem space. This space, referred to as the search space, comprises all possible solutions to the problem at hand. Genetic algorithms are generally applied to problem spaces which are too large to be exhaustively searched.

3.2 Procedure

An initial population of individuals is generated either at random or heuristically. At every evolutionary step, the individuals in the current population are evaluated according to some fitness function. To form a new population, individuals are selected according to their fitness from the current generation. Selection does not introduce new individuals to the population. Hence selection alone is not sufficient to find new points in the search space. These are generated by the genetically-inspired operators, crossover and mutation. Thus selection is used in conjunction with mutation and crossover to generate new population. Crossover is performed with probability 'p_{cross}' between two selected individuals called parents and by exchanging parts of their genomes(chromosomes) to form two new individuals called offspring. This operator tends to move the evolutionary process towards more promising regions of the search space. The mutation operator is introduced to avoid premature convergence to local optima by randomly sampling new points in the search space. It is carried out by flipping bits at random with some probability 'p_{mut}'. Genetic algorithms are stochastic iterative processes that are not guaranteed to converge. The stop condition can be specified as a particular number of generations or as an attainment of a particular fitness level. Genetic algorithms tend to get slower than conventional searches by taking up

large runtime to find the solution. In order to speed up the algorithm some enhancements such as elitism can be used where a fraction of the best chromosomes from the current population is retained unaltered to the next generation. The algorithm is depicted by the flow chart in Figure 4.

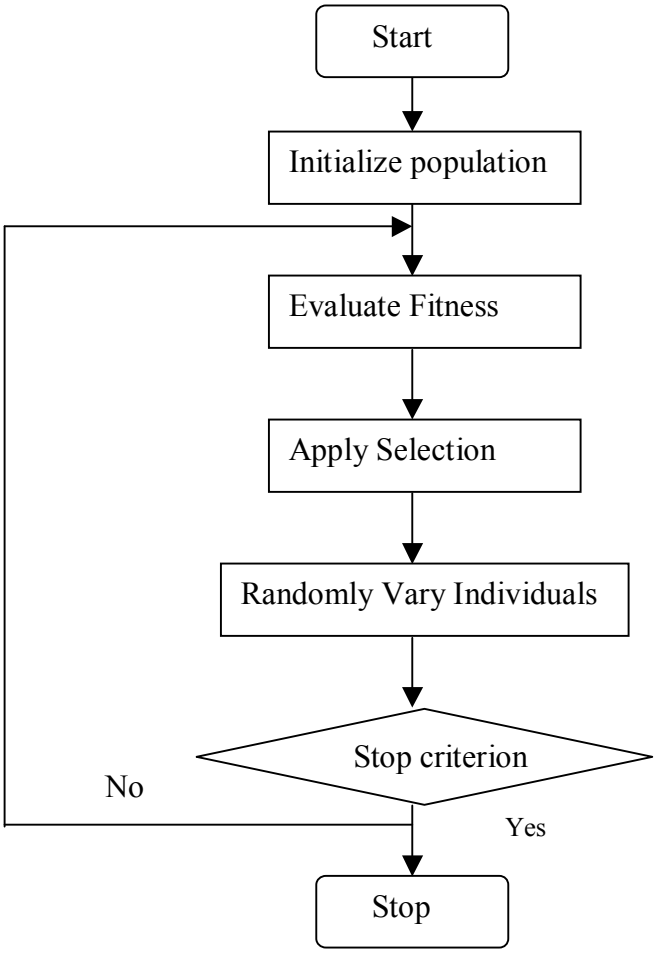
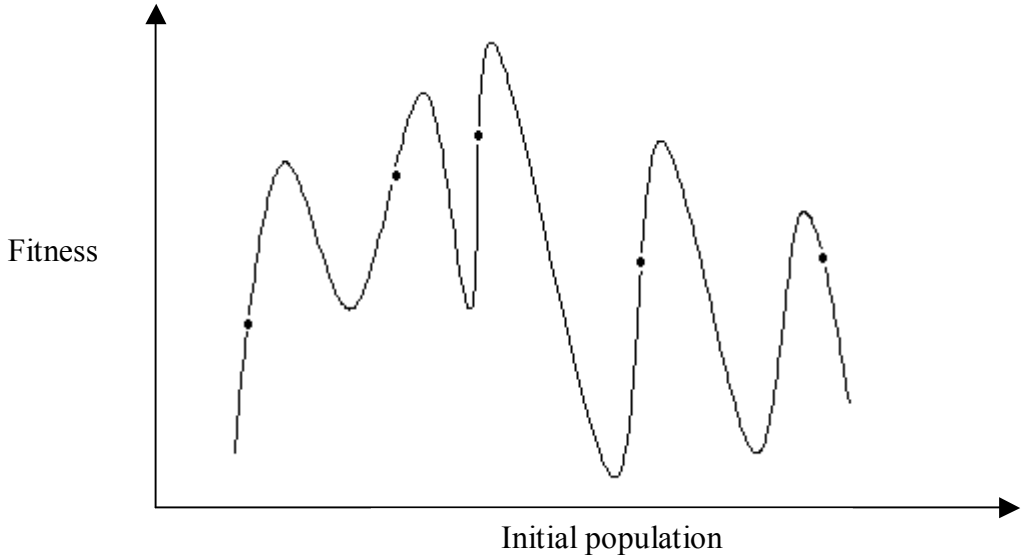
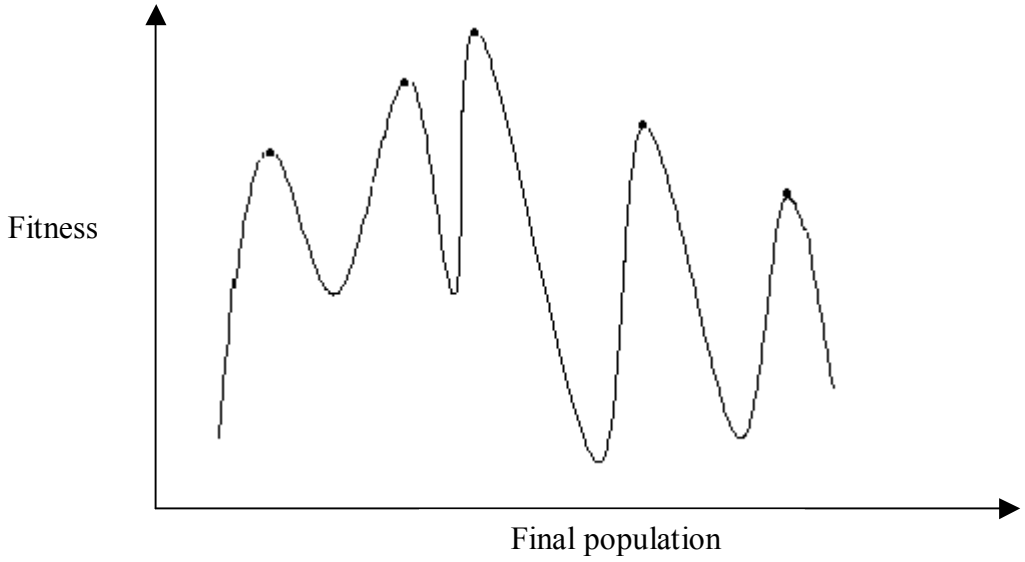


Figure 4: Structure of a genetic algorithm.



(a)



(b)

Figure 5: (a) Plot of Fitness vs. Initial population; (b) Plot of Fitness vs. Final population.

The above graphs show the fitness of the population initially and after a number of generations have been executed [19]. The curve shows the distribution of the fitness values of the chromosomes. The dots on the curve

show the current generation chromosomes. Figure 5(a) shows the initial distribution of the chromosomes and Figure 5(b) shows the distribution of the final generation population of chromosomes. As can be seen, the chromosomes reach the peaks of fitness after a number of generations have been executed. The chromosome with the highest fitness value is going to be the best chromosome.

3.3 Genetic Operators.

Encoding*:* Encoding is a process of mapping the knowledge domain to the solution space. The selection of an encoding scheme varies with the design decision and also depends on the problem to be solved. It will affect the selection of genetic operators. An improper encoding scheme will produce infeasible chromosomes generated by genetic operators. Chromosomes can be encoded in different ways, as described here

- *Binary Encoding*: One way of encoding chromosomes is as binary strings. Each bit in the chromosome can represent some characteristic or the presence or absence of a particular characteristic.

  Ex: chromosome = 10010011

- *Permutation Encoding*: Permutation encoding can be used in ordering problems such as the traveling salesman problem (TSP) or a task ordering problem. Every number in the chromosome represents a number in the sequence. In TSP each number represents a city to visit.

  Ex: chromosome = 1 3 5 2 6 4 8 7

- *Value Encoding*: Direct value encoding can be used in some problems where some complicated values such as real numbers are used and where binary encoding would not suffice. While value encoding is good for some problems, it is often necessary to develop some specific crossover and mutation operators for these chromosomes.

  Ex: chromosome = A C B C D E

  Here A could represent a task, B another and so on.

- *Tree Encoding*: Tree encoding is used to allow programs or expressions to evolve. In tree encoding, every chromosome is a tree containing objects, such as functions or commands in a programming language. LISP is often used to implement this type of encoding because programs in LISP can be represented in this form and then easily parsed as a tree.

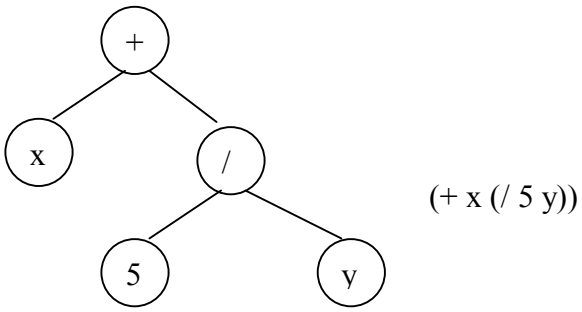


Figure 6: Example of tree encoding.

***Population Size**:* The population size affects both the ultimate performance and the efficiency of genetic algorithms. A small population size will reduce the evaluation cost in each generation. For a large population size, genetic algorithms can perform a more informed search because a large population is more likely to contain

representatives from a large number of hyperplanes. As a result, selecting the optimal population size is critical for the success of a genetic algorithm.

***Selection**:* The selection operator forms a new population by selecting chromosomes in the old population based on their fitness value. The rationale is that the chromosomes with higher fitness value should have a higher probability of surviving into next generation. There are many methods of selecting chromosomes for the next generation.

- *Proportional Selection*: Proportional Selection selects a candidate with probability proportional to the fitness of the candidate.
- *Roulette wheel selection*: It is a proportional selection where each individual is given a chance to be a parent in proportion to its fitness. The chances of selecting a parent can be seen as spinning a roulette wheel with the size of the slot for each parent being proportional to its fitness. Obviously those with larger fitness values have an increased probability of being chosen. Hence in roulette wheel selection there is a chance that some of the individuals dominate others and get selected a high proportion of the time.
- *Boltzmann selection*: This technique is similar to roulette wheel selection except that the area of each section on the wheel is give by

$$A_i = e^{-(F_m - F_i)/(kT)}$$

Where F_m is the fitness of the fittest solution and F_i is the fitness of the i^{th} chromosome. The term (kT) is the effective temperature; it can stay

constant or be reduced for each new generation. Selection pressure is slowly increased over evolutionary time to gradually focus the search.

- *Tournament selection*:

  Method1: Select a pair of individuals at random. Generate a random number, R between 0 and 1. If $R < r$, the first individual is used as the parent, otherwise if $R >= r$, the second individual is used as the parent. This is repeated to select the second parent. The variable r is a parameter to this method.

  Method2: Select a pair of individuals at random. Select the individual with larger fitness as a parent. Repeat this process to select a second parent.

- *Rank selection*: Rank selection ranks the population first and then every chromosome gets fitness proportional to its rank. The individual with the lowest rank gets a fitness of 1, the next lowest gets a fitness of 2 and the highest-ranked individual gets a fitness value of N, where N is the number of chromosomes. In this method, each chromosome has a significant chance of being selected. However, this method can lead to slower convergence since the best chromosomes do not differ much from others in terms of fitness.

- *Steady state selection*: In every generation a few good chromosomes are selected to create offspring, and then these offspring replace the bad chromosomes and the rest survive to the next generation.

Crossover*:* Selection redirects the search towards the best existing chromosomes but does not create new chromosomes. The crossover operator, on the other hand, takes valuable information from both parent chromosomes and then combines them to find highly fit chromosomes.

- *Single Point Crossover*: A random number is chosen between 0 and the length of the chromosomes. The chromosomes are split at that random point and merged. Two offspring are produced.

  Ex: **11001**011+11011**111** = **11001111 and 11011011**

- *Multi Point Crossover*: More than one random point is chosen between 0 and length of the chromosomes and the chromosomes are split at those points and merged. Two offspring are produced.

  Ex: **11**001**011** + 11**011**11 = **11011111 and 11001011**

- *Uniform Crossover*: For each gene, a gene value from the corresponding position in the parents is chosen at random. One offspring is produced.

  Ex: 1**10**01**011** + **1**10**111**01 = 11011111

- *Arithmetic Crossover*: Some arithmetic operation is performed to produce new offspring.

  Ex: 11001011 + 11011111 = 11001011 (AND)

If the crossover rate is too low, the search may stagnate due to a low exploration rate. The higher the crossover rate, the faster new chromosomes will be introduced into the population.

Mutation: Mutation maintains the diversity of the population.

- *Bit Inversion:* In a binary chromosome, a gene is randomly chosen and a bit inversion is done i.e. if the gene is 1 it is changed to 0 and vice versa.
- Other forms of mutation can be performed that are specific to the chromosomes and the genetic algorithm.

Mutation should be sparingly used because it is a random search operator. With high mutation rate, the algorithm would become little more than a random search.

Stop Condition: Another important design factor of a genetic algorithm is the stop condition, i.e. when should the evolution process be stopped. A number of options are possible such as the following

- A genetic algorithm can be run for a certain number of iterations and then stopped and the best chromosome is taken as the solution.
- A genetic algorithm can be stopped after the allocated computing time is used up.
- A genetic algorithm can be stopped after an individual is found that satisfies the required criteria
- A genetic algorithm can be terminated when a plateau is reached and the successive iterations do not produce better results
- Combinations of the above methods can be used as terminating conditions.

3.4 Implementing a genetic algorithm for subgraph isomorphism detection in SUBDUE.

The amount of effort expended by a brute force solution to graph isomorphism problem would typically be considered intractable, as there are *N!* possible node orders for a graph with *N* nodes. The subgraph isomorphism problem is even worse combinatorially, as the subset of the vertices in the main graph that are to be matched with the subgraph is unknown. In the worst case, all possible combinations of vertex subsets may be matched with a graph representing a candidate concept.

A method for subgraph isomorphism in polynomial time has been proposed by Messmer and Bunke [15]. This method constructs a decision tree with a number of modal graphs and then takes an input graph and gives a list of modal graphs which have this input graph as a subgraph. The time taken is said to be polynomial without taking into account the preprocessing step where the decision tree is constructed.

It is because of the NP-completeness of the subgraph isomorphism problem that an approximate solution was sought. "LeRP" is one such approximation technique for subgraph isomorphism detection [16]. It is based on counts of Length-R-Paths. The algorithm outputs if the given smaller graph is a subgraph of the larger graph. The worst-case time complexity is $O(N^3D^2R)$ where *N* is the number of nodes in the smaller graph, *D* is the mean degree and *R* is the highest power of the adjacency matrix used in processing. Another approximate technique has been proposed by David Eppstein [17] that uses a graph decomposition method similar to one used by Baker [18] to approximate various NP-complete problems on planar graphs. This method finds all the

instances of a smaller graph in the main graph with complexity $O\ (c^{w \log w}\ n + wk)$, where c is a constant, w is the number of vertices in the subgraph, n is the number of vertices in the main graph and k is the number of instances of the subgraph in the main graph.

Genetic algorithms are better than conventional search methods in that they are more robust [19]. They do not break easily even if the inputs change slightly or a reasonable amount of noise is present in the data. Also in searching a large state-space, a genetic algorithm may offer significant computation time benefits over conventional search techniques. Hence we consider here a genetic algorithm approach for subgraph isomorphism detection in SUBDUE. In SUBDUE, subgraph isomorphism is used to find all instances of predefined substructures in the input graphs.

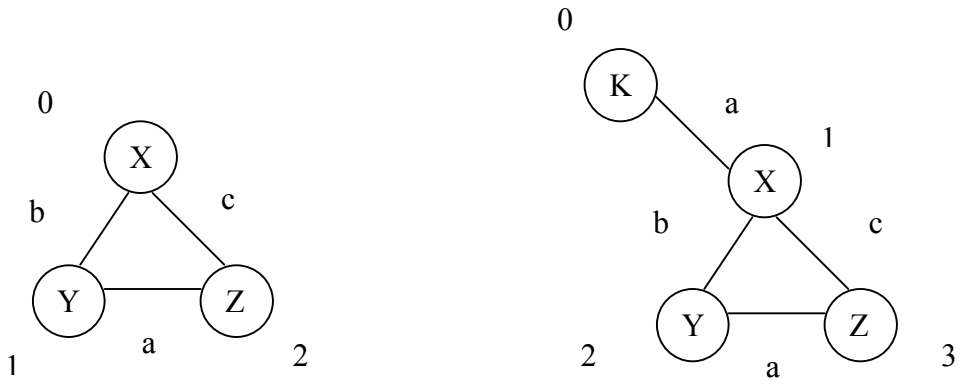
The three most important aspects of using genetic algorithms are: (1) representation of the chromosomes, (2) definition of the objective function, and (3) definition and implementation of the genetic operators.

The idea for the chromosome representation has been adopted from a paper written by Max Pesakhov and William Regli [19]. The paper searches for only one instance of the subgraph in the main graph. In contrast, our algorithm tries to list all the instances of the subgraph in the main graph. The fitness function used in their paper requires a significant amount of computational time. Hence, an alternative fitness function has been considered in our research. The size of the initial population and also the number of iterations were manually given in their paper whereas we present the

results of experimentation in which the size of the population and the number of iterations are both manually and dynamically initialized.

Initial Population: The chromosome used is a permutation chromosome. Each gene has two fields, *position* and *value*. *Position* refers to the number of the vertex in the subgraph and *value* represents the number of the vertex in the main graph to which the vertex in the subgraph is mapped.

Chromosomes are created by finding a mapping between the vertices in the subgraph and those in the super graph. Chromosomes are generated randomly by mapping a vertex in the subgraph to a vertex in the super graph. The mapping is constrained such that the label of both the vertices must be same and the degree of the vertex in the subgraph must be smaller than or equal to that of the degree of the vertex in the super graph.



Subgraph g1 Main graph g2

Figure 7: Example of creating a chromosome

Consider the example graphs shown in Figure 7, vertex 0 in graph g1 can be mapped to vertex 1 in graph g2, vertex 1 in graph g1 can be mapped to vertex 2 in graph

g2 and vertex 2 in graph g1 can be mapped to vertex 3 in graph g2. Hence a valid chromosome would be 123 which is represented as the following.

Value	1	2	3
Position	0	1	2

Size of the population: The size of the population is specified as *nv2* where *nv2* is the number of vertices in the main graph. This particular value is chosen because there will be at the most *nv2* instances of the subgraph in the main graph. However this is not necessarily an optimal population size. Our experimental results will show that an optimal runtime may sometimes be obtained when the size is varied to an alternative value.

Fitness: The fitness function gives each chromosome a fitness value which is a judgment of its surviving capability. Choosing and formulating an appropriate fitness function is crucial in obtaining an efficient solution.

In the genetic algorithm designed for SUBDUE, the fitness evaluation is performed as shown in Figure 8.

The fitness function does not give misleading results by being partial to either densely or sparsely connected graphs. This is because for all the graphs, the fitness function is normalized by the same value i.e. the number of vertices in the subgraph and for each vertex, it is normalized by the number of edges emanating from the corresponding vertex in the subgraph, which is again same for all the vertices in this position.

For each vertex v1 in the subgraph

For each edge e1 originating from v1

If there is an edge e2originating from v2, where v2 is a vertex in the main graph that is mapped to v1 in the subgraph and the destination vertices of the edges e1 and e2 match

Then increase fit by 1

Divide fit by the number of edges originating from of v1.

Add the quotient to the fitness

Divide fitness by number of vertices in the subgraph

Figure 8: Algorithm to calculate the fitness of a chromosome

The fitness calculation is illustrated through the following example.

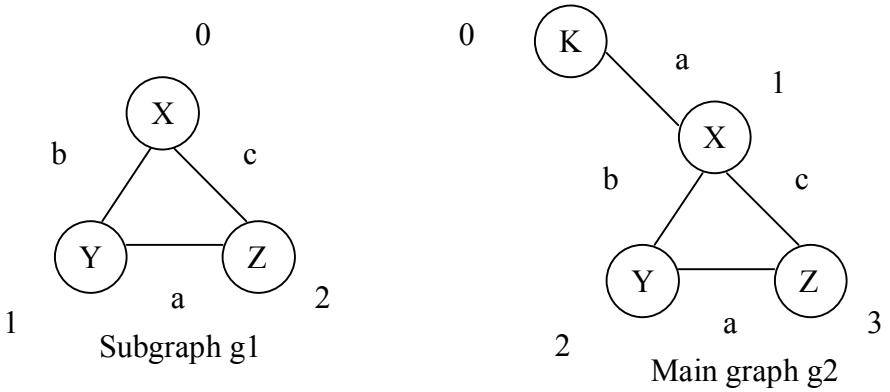


Figure 9: Example illustrating the calculation of fitness of a chromosome.

Consider the graphs shown in Figure 9. In this example, vertices 0, 1, 2 in the subgraph are mapped to vertices 1, 2, 3 in the main graph, respectively. Hence, the chromosome is 123.

The fitness of 123 is calculated as follows.

Step 1: The variables fit and fitness are initialized to 0. Consider the gene at position 0, with value 1. There are two edges originating from vertex 0 in g1. For the edge labeled b originating from vertex 0 in g1, there is a corresponding edge labeled b originating from vertex 1 in g2, and the destination vertices also match (i.e. Y). Hence the value of variable fit is incremented by one.

fit = 1;

For the edge labeled c originating from vertex 0 in g1, there is a corresponding edge originating from vertex 1 in g2. Hence the value of variable fit is further incremented by one.

fit = 2;

Now fit is divided by the degree of the vertex 0 in g1 and the value of the variable fitness is incremented by the quotient. Therefore value of fitness = fit/2;

fitness = 1;

Step 2: The variable fit is initialized to 0. Consider the gene at position 1(with a value of 2). Again there are 2 edges originating from vertex 1 in g1. For the edge labeled b originating from vertex 1 in g1 there is a corresponding edge labeled b originating from vertex 2 in g2. Hence fit is incremented by 1.

fit = 1;

Similarly for the other edge labeled c originating from vertex 1 in g1, there is an edge labeled c originating from vertex 2 in g2. Hence fit is incremented to 2. Now fit is divided by the degree of vertex 1 in g1 and added to fitness. Therefore the value of fitness is 2.

Step 3: The variable fit is initialized to 0. The value of gene at position 2 with value 3 is considered and the same operations as in the above two steps are performed. Now the value of fitness is 3. Finally fitness is divided by the number of vertices in the subgraph. Therefore the value of fitness is 1.

Since the value of fitness is 1, this is considered an exact match.

Selection: Rank selection is employed. Each chromosome is assigned a sector in the roulette wheel proportional to its rank. The wheel is spun N times, where N is the number of vertices in the subgraph and N chromosomes are selected for the next generation.

Elitism: 10% of the best chromosomes are retained from the current generation to the next generation.

Crossover: Uniform crossover is performed. After the parents are selected for crossover, a random number between 0 and 1 is generated and the crossover function is called only if the random number is less than or equal to 0.9. Within the crossover function, for every gene position, a random number between 0 and 1 is generated and the genes in this position are interchanged only if the random number is less than or equal to 0.2.

For example in Figure 10, the genes in positions 0 and 3 are exchanged producing offspring1 and offspring2. This procedure does not result in invalid chromosomes. Because genes in a particular position are interchanged, the mapping is still valid. For example if vertex 0 in the subgraph can be mapped to only vertices 1 and 3 in the main graph, an invalid mapping where gene at position 0 has a value other than 1 or 3 will not result using this procedure.

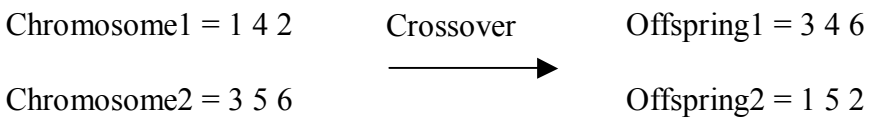


Figure 10: Example illustrating the crossover operation.

Mutation: Random mutation is performed. The value of a gene is replaced by a value from a list of vertices to which it can be mapped. A random number between 0 and 1 is generated and the mutation function is called only if the random number is less than or equal to 0.3. With in the mutation function, for every gene position, a random number between 0 and 1 is generated and the gene at that position is mutated only if the random number is less than or equal to 0.4. These rates of crossover and mutation have been decided based on the results of a number of trials using different values.

Number of Iterations: Deciding upon an optimal number of iterations is a difficult task for a genetic algorithm. The number of iterations for our algorithm is taken as $\prod_{1}^{mv1} Count(i)$/size of the population, where *Count(i)* is the number of vertices in the main graph to which the i^{th} vertex in the subgraph can be mapped. The product of *Count(i)* for all i gives the maximum number of possible instances of the subgraph in

the main graph. Hence by iterating $\prod_{1}^{nv1} Count(i)$/size number of times, all possible instances may be considered.

Stop Condition: The algorithm stops after running for a fixed number of iterations.

3.5 Subgraph isomorphism detection using sgiso

Algorithm sgiso, a utility provided with the SUBDUE code performs subgraph isomorphism detection by finding instances of the subgraph in the main graph. The function starts by finding all instances of a vertex of the subgraph in the main graph. It then extends all instances of this vertex by a single edge. The process of extending by a single edge is repeated until all the edges and vertices of the subgraph are covered. Finally the function performs a graph match to eliminate those instances that do not match the subgraph. This step also eliminates overlapped instances if they are not allowed by the user.

3.6 Experimental Results

Tables 1 through 3 show the number of instances found and runtime taken by sgiso and that taken by the genetic algorithm.

The graphs sample2.graph through sample10.graph are generated using subgen. Subgen is a synthetic graph generator that accepts as input the number of vertices and edges of a graph to be generated, the possible vertex and edge labels, the graph connectivity, a substructure to be embedded and the percentage of the final graph to be covered by the substructure as parameters and generates a graph consistent with the

specified parameters . The sizes of the graphs sample2.graph through sample10.graph vary from 12 vertices, 11 edges to 88 vertices, 100 edges. The graph shown in Figure 11 is the substructure, sample2sub.graph, that is embedded in all of these graphs.

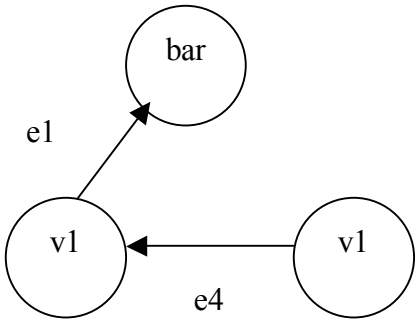


Figure 11: The substructure sample2sub.graph.

ttt_notwin is the graph representation of the database found in the UCI repository [36]. ttt_win is a graph with 5634 vertices and 10016 edges. ttt_notwin is a graph with 2988 vertices and 5312 edges. ttt_win is a graph representation of all possible winning board configurations of a tic-tac-toe game. ttt_notwin is a graph representation of all possible board configurations of a tic-tac-toe game with one blank position.

The desired results would be that the genetic algorithm finds all the instances of the subgraph in the main graph and also the time taken by the genetic algorithm is lower than that taken by sgiso.

Table1 shows the comparison results when the sizes of the population and numbers of iterations have been given manually for a number of trials. The table shows

that the genetic algorithm could find all the instances of the subgraph in the main graph. However the time taken by the genetic algorithm is a lot higher than that of sgiso.

Table 2 shows the comparison results when the size of the population has been calculated dynamically as equal to the number of vertices in the main graph and number of iterations has been given manually. Again the genetic algorithm could find all instances of the subgraph in the main graph but the time taken is higher than that of sgiso. The time taken by this approach is almost similar to the time taken in the above table.

Table 3 shows the comparison results when both the size of the population and the number of iterations have been dynamically calculated. The genetic algorithm could not find all the instances of the subgraph in the main graph in some of the experiments. The reason for this is that the genetic algorithm is stopped after too small a number of iterations. The GA would require more iterations to find all the instances. However, it could find more than 95% of the substructure instances. The time taken by the algorithm is less than in the above two methods.

The Figures 12 through 15 show the plots of the results shown in tables 1 through 3. The graphs are shown along the x-axis and the runtime or number of calls is shown along the y-axis. The labels s2 through s10 on the x-axis correspond to the graphs sample2.graph through sample10.graph respectively.

The sgiso algorithm takes less time compared to the GA. The reasons for this are the algorithm sgiso has a tab on the number of instances of the subgraph in the main graph which the GA does not have. This helps the algorithm in a big way when the

number of instances of the subgraph in the main graph is very small and the main graph is very large in which case the GA suffers. The GA is computationally intensive since it evaluates a large number of chromosomes which are probable solutions, although most of them are not actual solutions to the problem.

Table 1: Comparison of runtime of sgiso and the genetic algorithm with the size of the population and the number of iterations given manually.

Subgraph	**Main graph**	**No. of inst.s found by sgiso**	**Time taken by sgiso (secs)**	**No. of inst.s found by GA**	**Time taken by GA (secs)**	**Pop Size**	**No. of iterations**
Sample2sub	Sample2	3	0.00	3	0.00	22	20
Sample2sub	Sample3	22	0.00	22	0.7	91	1900
Sample2sub	Sample5	22	0.00	22	1.05	116	1800
Sample2sub	Sample6	10	0.00	10	0.04	27	900
Sample2sub	Sample7	10	0.00	10	0.04	27	800
Sample2sub	Sample8	10	0.00	10	0.02	27	600
Sample2sub	Sample9	10	0.00	10	0.03	27	600
Sample2sub	Sample10	11	0.00	11	0.03	31	500
ttt_notwin_sub	ttt_notwin	343	0.01	343	2348.43	4000	4000
ttt_win_sub	ttt_win	563	0.01	563	6000.32	5000	6000

Table 2: Comparison of runtime of sgiso and the genetic algorithm with the size of the population calculated dynamically.

Subgraph	**Main graph**	**Number of inst.s found by sgiso**	**Time taken by sgiso (secs)**	**No. of inst.s found by GA**	**Time taken by GA (secs)**	**Pop Size**	**No. of iterations**
Sample2sub	Sample2	3	0.00	3	0.00	12	50
Sample2sub	Sample3	22	0.00	22	1.02	88	3000
Sample2sub	Sample5	23	0.00	23	1.00	88	3000
Sample2sub	Sample6	10	0.00	10	0.02	36	200
Sample2sub	Sample7	10	0.00	10	0.02	36	200
Sample2sub	Sample8	10	0.00	10	0.05	36	600
Sample2sub	Sample9	10	0.00	10	0.05	36	600
Sample2sub	Sample10	11	0.00	11	0.07	44	600
ttt_notwin_sub	ttt_notwin	343	0.01	343	1631	2988	5000
ttt_win_sub	ttt_win	563	0.01	563	10000	5634	7500

Table 3: Comparison of runtime of sgiso and the genetic algorithm with the size of the population and the number of iterations calculated dynamically.

Subgraph	**Main graph**	**No. of inst.s found by sgiso**	**Time taken by sgiso (secs)**	**No. of inst.s found by GA**	**Time taken by GA (secs)**	**Pop Size**	**No. of iterations**
Sample2sub	Sample2	3	0.00	3	0.00	12	54
Sample2sub	Sample3	22	0.00	21	0.98	88	2904
Sample2sub	Sample5	23	0.00	23	0.99	88	2904
Sample2sub	Sample6	10	0.00	10	0.04	36	486
Sample2sub	Sample7	10	0.00	10	0.03	36	486
Sample2sub	Sample8	10	0.00	9	0.03	36	486
Sample2sub	Sample9	10	0.00	9	0.04	36	486
Sample2sub	Sample10	11	0.00	11	0.07	44	726
ttt_notwin_sub	ttt_notwin	343	0.01	336	927.18	2988	2833
ttt_win_sub	ttt_win	563	0.01	536	5515.66	5634	4740

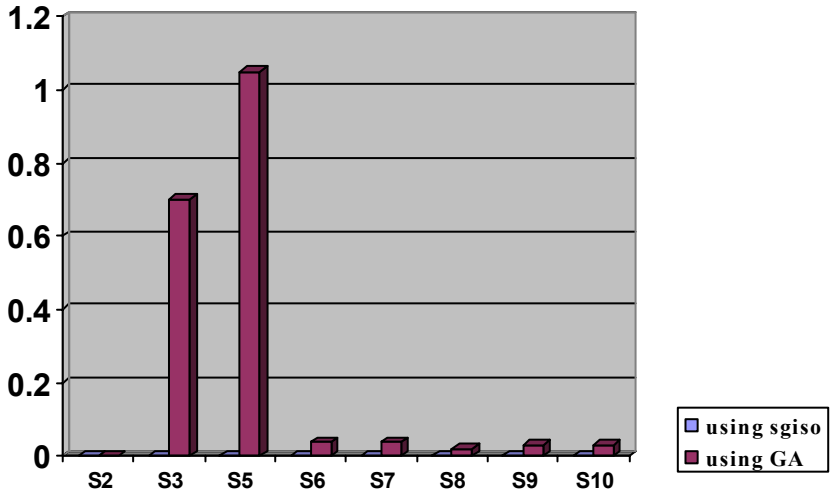


Figure 12: Plot of runtime of sgiso vs. runtime of the genetic algorithm with the size of the population and the number of iterations given manually.

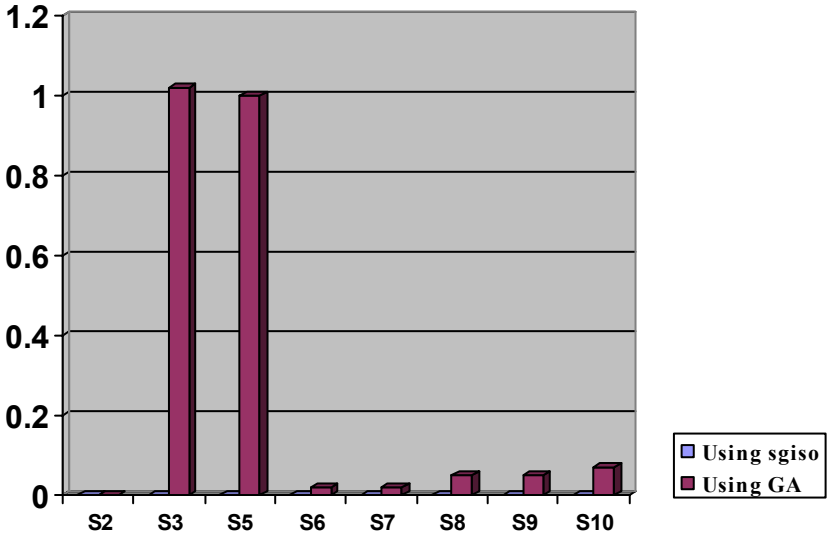


Figure 13: Plot of runtime of sgiso vs. runtime of the genetic algorithm with the size of the population calculated dynamically.

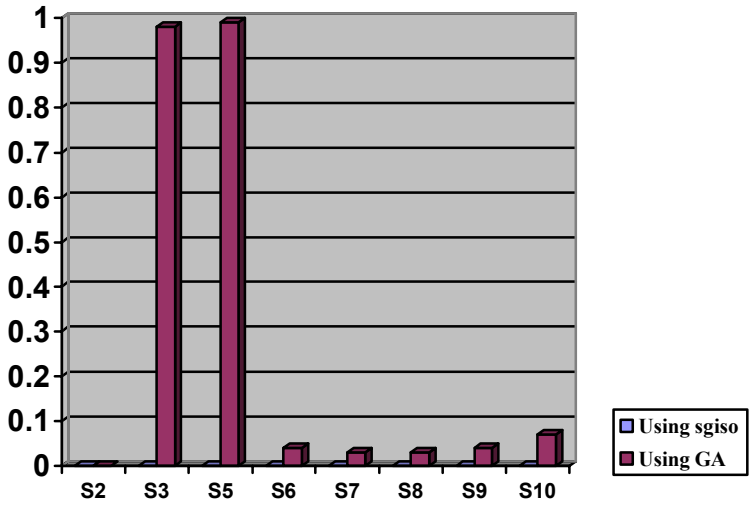


Figure 14: Plot of runtime of sgiso vs. runtime of the genetic algorithm with the size of the population and the number of iterations calculated dynamically.

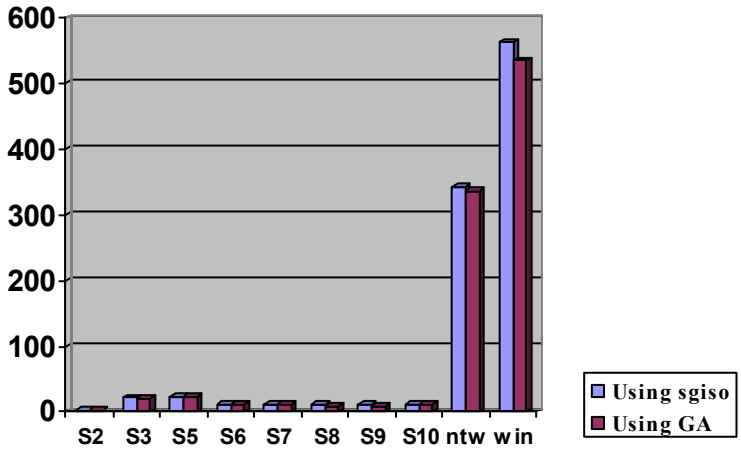


Figure 15: Plot of the number of instances found by sgiso vs. the number of instances found by the genetic algorithm with the size of the population and the number of instances calculated dynamically.

3.7 Conclusions

The genetic algorithm can be successfully used to perform subgraph isomorphism detection in order to determine if a given smaller graph is a subgraph of a larger graph. This approach is an alternative to generating all of the instances of the smaller graph in the larger graph. However the genetic algorithm is an approximative approach. We may not perfectly emulate the results of the exhaustive approach owing to the stochastic nature of the genetic algorithm.

The drawback of using a genetic algorithm to find all instances of a subgraph in a main graph is that since we do not know the number of instances beforehand, we cannot stop the algorithm even though all the instances may have been found and run the algorithm for a fixed number of iterations. Hence the time taken is high. Any other stop condition like the variance of the fitness of the population does not guarantee an optimal solution. On the other hand function sgiso, knows when all of the instances are found and stops after all of the instances have been found.

A genetic algorithm is an approximative technique and using it for an optimization problem would make it time consuming. Hence, usage of a genetic algorithm to find all instances of a predefined substructure in SUBDUE is not a good idea.

Genetic Algorithms give approximate solutions to difficult-to-solve problems. The main disadvantage of using Gas is that they are relatively slow, being very computationally intensive compared to other methods such as random search. As a general rule of thumb, genetic algorithms might be useful in problem domains that have

a complex fitness landscape as recombination is defined to move the population away from local minima in which a traditional hill climbing algorithm might get stuck.

3.8 Future Work

There are many factors that affect the performance of a genetic algorithm, such as the choice of fitness function. A good fitness function should be able to guide the search to more prominent regions in the search space. Our fitness function could be improved so that the GA performs better and the goal is reached more quickly.

Determining the size of the initial population is a major factor that affects the performance of the algorithm and there is no straightforward method to find an optimal size of the initial population. The best choice for the size of the initial population depends on the size of the graph. If the population size is too small, the algorithm does not find all of the instances of the subgraph in the main graph. On the other hand if the population size is too big, the run time taken is very high. However, the size of the population should be significant to find all the instances. Hence, deciding upon an optimal size of the initial population determines the success of the genetic algorithm. The population size used in our algorithm seems to be a good one though not necessarily an optimal one.

One more factor that affects the genetic algorithm is the choice of the number of generations to execute, which depends on the size of the input graphs and the size of the population. Hence, coming up with reasonable size for the initial population, number of iterations and a good fitness function may improve the results.

CHAPTER 4

CANONICAL LABELING

In this chapter, canonical labeling has been considered as an alternative technique for graph match. We give an overview of the nauty package, which has been adapted to perform canonical labeling in SUBDUE. Canonical labels have been assigned to the substructures and thereby the number of calls to the graphMatch routine has been reduced.

4.1 Introduction

Canonical labeling is an alternative technique to the graph match routine that assigns a unique code, or string, to each graph. Two graphs that are isomorphic have the same canonical labels. The graph match problem thereby reduces to matching the canonical labels. Hence, when a number of graph matches need to be performed among the same set of graphs, using canonical labels reduces the number of calls to the computationally expensive graph match routine and replaces them with polynomial-time string matches. Hence this technique has been considered to reduce the computational complexity of SUBDUE and thereby increase its scalability.

4.2 Definition of a canonical label

A canonical label is a unique code given to a graph that is invariant on the order of edges and vertices in the graph [20, 21]. Two graphs having the same canonical representation are isomorphic and vice versa.

4.3 Methods of canonical labeling

A simple way of finding a canonical label is by finding the lexicographically largest or smallest string obtained by concatenating the rows or columns in an adjacency matrix over all possible symmetric permutations of the adjacency matrix. This method is illustrated in Figure 16 that shows a graph G and the permutation of its adjacency matrix that leads to its canonical label *"aaazyx"* (taking the largest possible string as the canonical label). Any other permutation of G's adjacency matrix will lead to a code that is lexicographically smaller than *"aaazyx"*. The codes *"aaazyx"* and *"aaazxy"* are obtained by concatenating the columns in the upper triangular matrix prefixed by the vertex labels in the order they appear in the adjacency matrix. For a directed graph, however the entire columns should be concatenated. The time complexity of generating a canonical label using this method is $O(|V|!)$, where $|V|$ is the number of vertices in the graph [26].

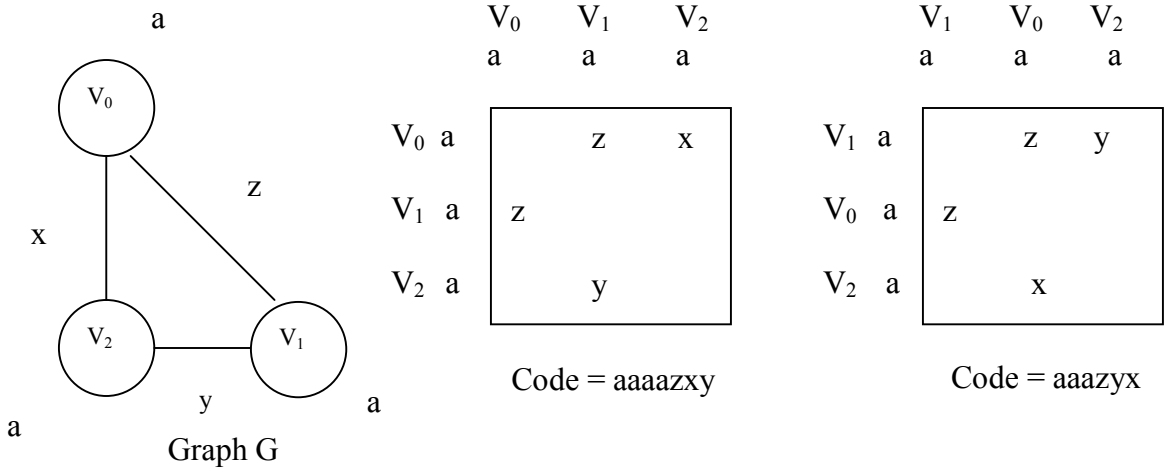


Figure 16: Example illustrating the calculation of the canonical label of a graph.

Liu and Klein have proposed a $O(N^3)$ canonical labeling algorithm [27]. The algorithm is based on the computation of the Eigen values of the graph adjacency matrix and is applicable to all types of graphs. However the proposed algorithm is not guaranteed to succeed especially for highly symmetric graphs (i.e. the canonical label assigned to the graph is not the appropriate one). The reason for this is computation of eigen values and eigen vectors is not sufficient for automorphism partitioning and hence computing a canonical label [28].

An algorithm for automorphism partitioning is proposed by Fortin [29] where vertices are partitioned according to their subspanning trees. The algorithm is efficient since the problem of automorphism partitioning is reduced to a tree isomorphism problem. However, this method succeeds in most instances but fails in specific cases [35].

The general characteristics of these methods is the use of vertex invariants to perform an initial vertex partitioning into equivalence classes, partitioning them into sets of vertices such that two vertices in different sets cannot possibly be mistaken for each other. Vertex invariants are some attributes or properties assigned to a vertex which do not change across isomorphism mappings. All vertices in each equivalence class must share the same value of some invariant that is independent of labeling. Possibilities for vertex invariants include:

- *Vertex degree* – The simplest way to partition vertices is based on their degree, the number of edges incident on the vertex. Clearly, two vertices

of different degree cannot be identical. This simple partition can often be a big win, but it will not do much for regular graphs, where each vertex has the same degree.

- *Shortest path matrix* – For each vertex *v,* the all-pairs shortest path matrix defines a multiset of *n-1* distances representing the distances between *v* and each of the other vertices. Any two vertices that are identical in isomorphic graphs will define the exact same multiset of distances, so we can partition the vertices into equivalence classes defining identical distance multisets.
- *Counting length-k paths* – Taking the adjacency matrix of G and raising it to the k^{th} power gives a matrix where $G^k[i,j]$ counts the number of paths from *i* to *j*. For each vertex and each *k*, this matrix defines a multiset of path-counts, which can be used for partitioning as with distances above. We could try all $1 \leq k \leq n$ or beyond and use any single deviation as an excuse to partition.

Using these invariants, it should be possible to partition the vertices of each graph into a large number of small equivalence classes. Because vertex invariants remain the same irrespective of the ordering of vertices and the edges, we can create the same partitions no matter how the vertices are ordered. Now the canonical label is generated by maximizing (or minimizing) over those permutations that keep the vertices in each partition together instead of maximizing (or minimizing) over all the permutations of the vertices. If the sizes of the equivalence classes of both graphs are

not identical, then the graphs cannot be isomorphic. It is harder to detect isomorphisms between graphs with high degrees of symmetry than it is for arbitrary graphs, because of the effectiveness of these equivalence-class partitioning heuristics. We may not have fine-grain partitioning of the vertices for symmetric graphs.

Nonetheless, since all vertices may have the same invariant, the upper bound of the time complexity for the exhaustive labeling generation scales exponentially with the number of vertices. While vertices with different invariants belong to different equivalent classes, the reverse is not necessarily true.

The complexity of finding a canonical label can be reduced by using various heuristics to narrow down the search space or by using alternate canonical labeling definitions that take advantage of special properties that may exist in a particular set of graphs [20, 21, and 29]. For example polynomial-time algorithms can be written to find the canonical labeling of chemical compounds [30].

4.4 nauty

nauty (no automorphism, yes?) is a set of very efficient C language procedures for determining the automorphism group of a vertex-colored graph developed by Brendan McKay [20, 21]. It is also able to produce a canonically-labeled isomorphism of a graph to assist in isomorphism testing. It is considered to be the fastest isomorphism detector in the world [31]. It is free for educational and research applications. The code may be obtained from http://cs.anu.edu.au/~bdm/nauty/.

Two graphs G and H are said to be *identical* (written $G = H$) if $V(G) = V(H)$, $E(G) = E(H)$ and $\psi_G = \psi_H$. Two graphs G and H are said to be *isomorphic* (written $G \cong$

H) if there are bijections $\theta : V(G) \rightarrow V(H)$ and $\phi : E(G) \rightarrow E(H)$ such that $\psi_G(e) = uv$ if and only if $\psi_H(\phi(e)) = \theta(u)\ \theta(v)$; such a pair$(\theta, \phi)$ of mappings is called an isomorphism between G and H. An automorphism of a graph is an isomorphism of the graph onto itself.

Let V be the vertex set of a graph with n vertices, γ a permutation of V, and $v \in V$, then v^{γ} is the image of v under γ, and G^{γ} is the graph in which vertices x^{γ} *and* y^{γ} are adjacent if and only if x and y are adjacent in G. An automorphism of a simple graph G can be regarded as a permutation of V that preserves adjacency. The automorphism group of a graph denoted *Aut (G)* is the set of all permutations $\gamma \in S_n$ (where S_n is the set of all permutations of n elements) such that $G^{\gamma} = G$.

Given a graph G with n vertices, a canonical label is a map C, such that $C(G) \cong G$, and $C(G^{\delta}) = C(G)$ for all permutations $\delta \in S_n$.

The vertex classification using vertex invariants is the basis of McKay's canonical labeling algorithm, which canonically colors an input graph and finds its automorphism group to compute its canonical form. However, in addition to vertex classification, this algorithm extensively utilizes the information of discovered automorphisms and hashes partial information of vertex labeling to keep search space from becoming impractically large. McKay's algorithm is based on a depth-first search through a tree whose nodes are stable vertex colorings. At each stage, a vertex is chosen and separated as a singleton color class by assigning a new color.

4.5 Making use of nauty in SUBDUE

In SUBDUE, a graph match is performed to see if a discovered substructure is already on a list of discovered substructures and to check if an instance is isomorphic to a substructure so that the instance can be added to the list of the substructure's instances. There are three routines where the routine graphMatch is called. They are *SubListInsert*, *MemberOfSubList* and *NewEdgeMatch*.

SubListInsert takes a substructure and a list of substructures as arguments and inserts the substructure into the list of substructures if the substructure is not already present in the list. To find out whether the substructure is present in the list or not, it performs a graphMatch between the substructure and all the substructures in the list.

MemberOfSubList takes a substructure and a list of substructures as arguments and returns True or False depending on the presence or absence of the substructure in the list.

NewEdgeMatch takes two instances of a substructure as arguments and finds out if the two instances are isomorphic. It performs graphMatch as a last resort to detect if the two instances are isomorphic.

Here we consider a canonical labeling approach to graph match. nauty has been used in SUBDUE to assign a canonical label to the substructures and the graphs and thus perform graph match by comparing the canonical labels. nauty does not consider edge labels. Because of this reason FSG has not used nauty inspite of nauty being the fastest isomorphism detector. They have developed their own canonical labeling technique. Our research has used nauty by converting the edges into vertices thereby

overcoming the limiting factor of nauty. However using this approach, the size of the graph increases and hence the time taken to calculate the canonical label may increase. This has been observed by creating few synthetic graphs with

Two methods of integrating canonical labels into SUBDUE using nauty have been considered. We describe these methods here.

Method 1: There are three data structures in SUBDUE that employ a graph representation. They are the input graph, substructure definitions and substructure instances. Canonical labeling has been assigned only to the substructures. The idea behind this is that graph match is being performed more than once only on a substructure to check if it is already present on the list. Hence the canonical labels are utilized completely in the case of substructures. The canonical labels have been used for graph match in two routines, namely, *SubListInsert* and *MemberOfSubList*.

SubListInsert: This routine inserts a substructure into a list of substructures if the substructure is not already there. To perform this operation the substructure is matched with every other substructure on the list. So by using canonical labels, the canonical labels of the substructures are compared instead of calling the graphMatch routine. A check is performed to see if both substructures have canonical labels assigned to them. If yes, then they are compared. If the labels match, then the substructures are possibly isomorphic. The time complexity of comparing the canonical labels is $O(n)$ where n is the number of vertices in the graph. Then a check is performed to see that both the graphs have same number

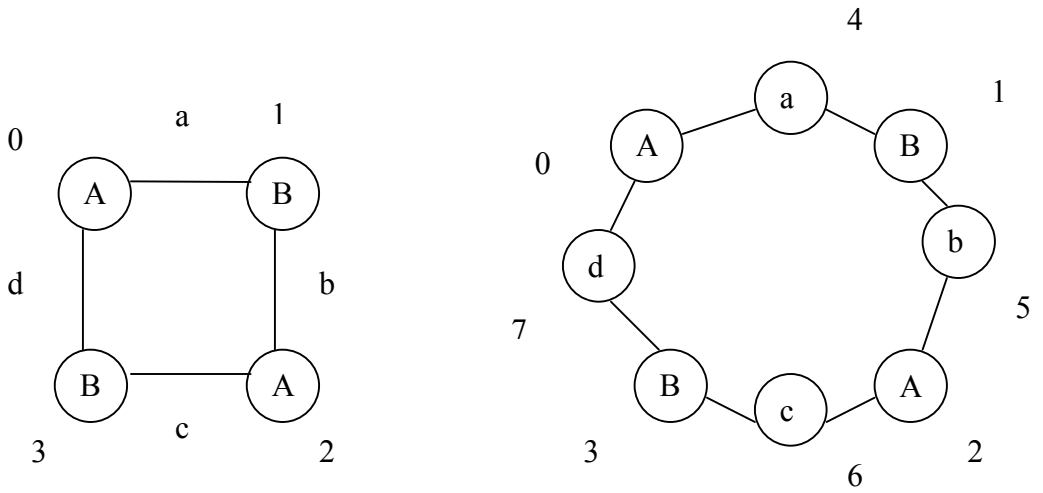
of vertices and edges. If yes, then the vertex labels of both the graphs are compared and if they too match, then the graphs are isomorphic. If either the canonical labels, the vertex labels or the number of vertices and edges do not match, then the substructures are not isomorphic. On the other hand if either of the substructures does not have a canonical label assigned to it, the graphMatch routine is called to perform graph isomorphism detection.

MemberOfSubList: This routine checks if the substructure's definition matches exactly with a substructure on the list. Using canonical labels, an operation similar to that performed in SubListInsert is performed here.

Method 2: nauty has been used to perform graph match instead of the graphMatch routine available in SUBDUE. Here canonical labels are assigned to both substructures and also graphs (that are created in the process). Whenever the graphMatch routine is called with graphs g1 and g2, a check is performed to see if g1 and g2 have been assigned a canonical label. If either of them has not been assigned one, then nauty is called to create a label for the graph and then the labels are compared to perform graph match.

Figure 17 illustrates the method of converting a graph g1 into graph g2 by converting the edges into vertices. Each edge in graph g1 has been converted into a vertex with the same label as that of the edge.

4



Graph g1 Graph g2

Figure 17: Example showing the conversion of edges into vertices.

With nauty, when two graphs are compared for isomorphism, we can give an initial partitioning of the vertices so that we can group all vertices with same label into one cell, thereby informing nauty that all these vertices have same label (there is no way of giving the vertex label information to nauty). For example, for the graph g2 in Figure 17, an initial partitioning of the vertices is {4|5|6|7|0, 2|1, 3}. This is to say that vertices 0 and 2 have the same label (A), vertices 1 and 3 have the same label (B) and so on.

The order of the cells in the partition is important in the sense that the same order must be maintained for all the graphs. The example shown in Figure 18 illustrates the way the order of the cells should appear in the vertex partitions of both the graphs which are tested for isomorphism.

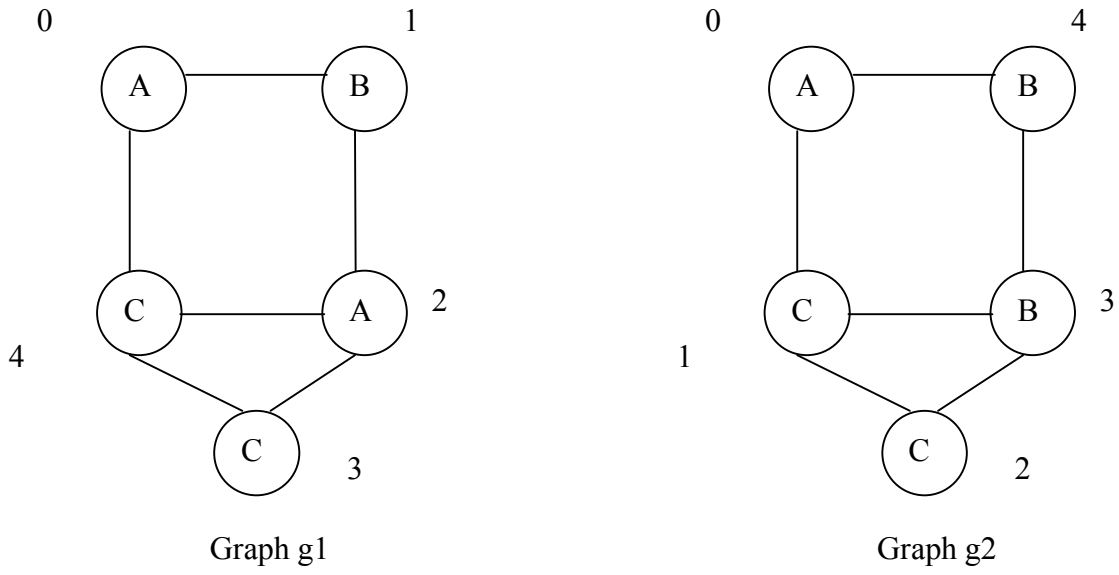


Figure 18: Giving initial vertex partitions to nauty.

For Graph g1, if the vertex partitioning is given as {0,2|3,4|1} i.e. the cell with vertices labeled A is followed by the cell with vertices labeled C and finally the cell with vertices labeled B, then the vertex partitioning of Graph g2 also must follow the same order. In this case, this means that the vertex partitioning of Graph g2 should be {0|1,2|4,3}.

4.6 Graph match in SUBDUE

Algorithm graphMatch is a utility provided in SUBDUE to perform a graph match. The function computes the minimum-cost transformation of g1 into an isomorphism of graph g2 (also known as computing the graph edit distance), but any match cost exceeding the given threshold is not considered. This algorithm returns true if the cost of transformation is less than the given threshold, otherwise it returns false.

4.7 Experimental Results

Table 4 shows the comparison of the runtime and the number of calls to graphMatch with and without using canonical labeling (as per method 1 discussed in section 4.5). Canonical labeling has been used only for the substructures and not their instances. By doing so, the number of calls to graphMatch from *SubListInsert* and *MemberOfSubList* can be reduced.

We conducted experiments using both real and synthetic datasets. The graphs sample3.graph through sample10.graph were generated using the synthetic graph generator, syngen. All the remaining graphs have been obtained from the UCI repository. The description of the graphs sample3.graph through sample10.graph, ttt_win and ttt_notwin can be found in section 3.5.

All the remaining graphs range from 1250 vertices, 1200 edges to 154812 vertices and 215587 edges. Graph chorales_bach and chorales_beethoven are single-line melodies of chorales. vote_d, vote_r, diabetes_0, diabetes_1, credit_1 and credit_2 are collections of disconnected subgraphs, where each subgraph has a vertex connected to other vertices in the form of a star. The chess_legal and chess_illegal graphs represent different chess board configurations in graph format.

As can be seen from table 4, the number of calls to graph match has been reduced when canonical labeling is used. However there is not a significant reduction in the runtime because the time saved in graphMatch is undone by the time spent in creating the canonical labels. The table also shows the time taken by SUBDUE with and without using nauty.

As can be seen from the table below, there is a reduction in the runtime for some inputs and there is an increase in the runtime for some others. The reason for this type of behavior is that a significant amount of time has been taken for creating the canonical labels. If these generated labels are not used many times, then the time saved in graphMatch is undone by the time taken to initially create the canonical labels.

Table 5 shows the comparison of the runtime taken by SUBDUE with and without using nauty to perform graphMatch (as per method 2 discussed in section 4.5). That is the graphMatch routine provided in SUBDUE has been replaced by a call to nauty to perform the graph match, just to observe the behavior of SUBDUE. As can be seen from the table below, the time taken when nauty is used is greater than the time taken when it is not used.

Figures 19 through 22 show the plots of the results shown in tables 4 and 5. The graphs are shown along the x-axis and the runtime or number of calls to graphMatch is shown along the y-axis. The labels cba, cbe, d0, d1, tnw, tw, vd, vr correspond to chorales_bach, chorales_beethoven, diabetes_0, diabetes_1, ttt_notwin, ttt_win, vote_d, vote_r respectively.

Table 4: Comparison of runtime of SUBDUE with and without using canonical labels to perform graphMatch for the substructures.

Graph	**No. of calls to graphMatch without using nauty**	**No. of calls to graphMatch using nauty (seconds)**	**Time Taken Without using nauty (seconds)**	**Time Taken using nauty (seconds)**
chess_illegal	72307782	67264157	16581.23	16513.23
chess_legal	33477481	31408046	3442.91	3435.14
chorales_bach	147097	141329	1.35	1.37
chorales_beethoven	100550	84402	1.01	0.98
credit_1	13915620	13830781	299.19	297.13
credit_2	5563057	5480263	58.34	57.68
diabetes_0	584691	580442	5.51	5.3
diabetes_1	317584	313118	2.3	2.27
ttt_notwin	593140	546937	7.94	7.20
ttt_win	1008916	956839	19.66	19.48
vote_d	2230530	2208469	25.92	25.67
vote_r	1639956	1619200	17.51	17.25
Sample10.graph	6697	4717	0.08	0.04
Sample9.graph	11449	7625	0.11	0.08
Sample8.graph	9802	6711	0.07	0.06
Sample7.graph	9057	6270	0.08	0.08
Sample6.graph	10679	7387	0.12	0.08
Sample5.graph	22460	16863	0.21	0.17
Sample3.graph	23442	17355	0.34	0.27

Table 5: Comparison of runtime of SUBDUE with and without using nauty to perform graphMatch.

Graph	**No. of calls to graphMatch without using nauty**	**No. of calls to graphMatch using nauty**	**Time Taken Without using nauty (seconds)**	**Time Taken using nauty (seconds)**
chess_illegal	72307782	67128945	16481.23	19835.23
chess_legal	33477481	31394393	3442.91	5747.03
chorales_bach	147097	141329	1.35	1.74
chorales_beethoven	100550	84402	1.01	1.08
credit_1	13915620	13830781	299.19	535.77
credit_2	5563057	5480263	58.34	118.39
diabetes_0	584691	580442	5.51	8.99
diabetes_1	317584	313118	2.3	3.22
ttt_notwin	593140	560449	7.94	12.58
ttt_win	1008916	950237	19.66	33.37
vote_d	2230530	2208469	25.92	45.97
vote_r	1639956	1619200	17.51	29.39
Sample10.graph	6697	4718	0.08	0.06
Sample9.graph	11449	7625	0.11	0.08
Sample8.graph	9802	6710	0.07	0.07
Sample7.graph	9057	6270	0.08	0.06
Sample6.graph	10679	7384	0.12	0.09
Sample5.graph	22460	16863	0.21	0.18
Sample3.graph	23442	17352	0.34	0.22

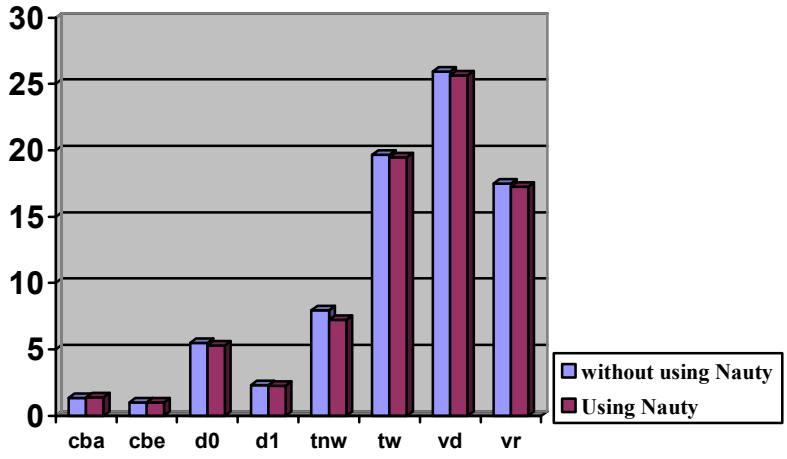


Figure19: Plot of runtime of SUBDUE with and without using canonical labels to perform graphMatch for the substructures.

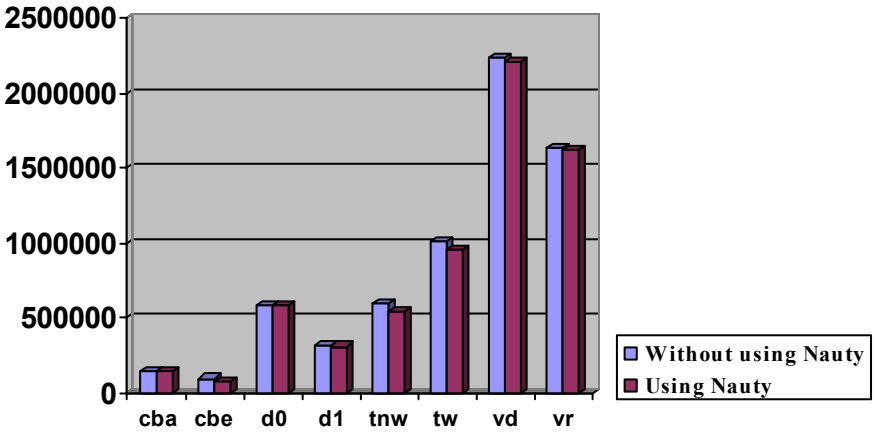


Figure 20: Plot of the number of calls to graphMatch with and without using canonical labels to perform graph Match for the substructures.

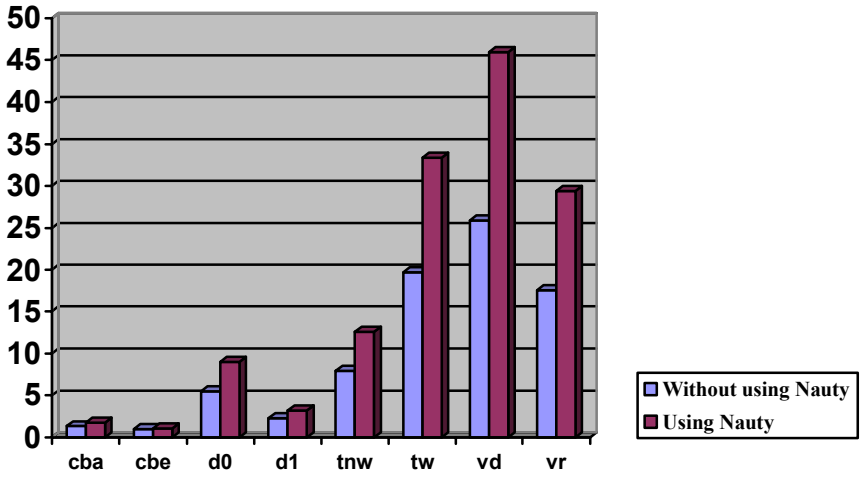


Figure 21: Plot of runtime of SUBDUE with and without using nauty to perform graphMatch.

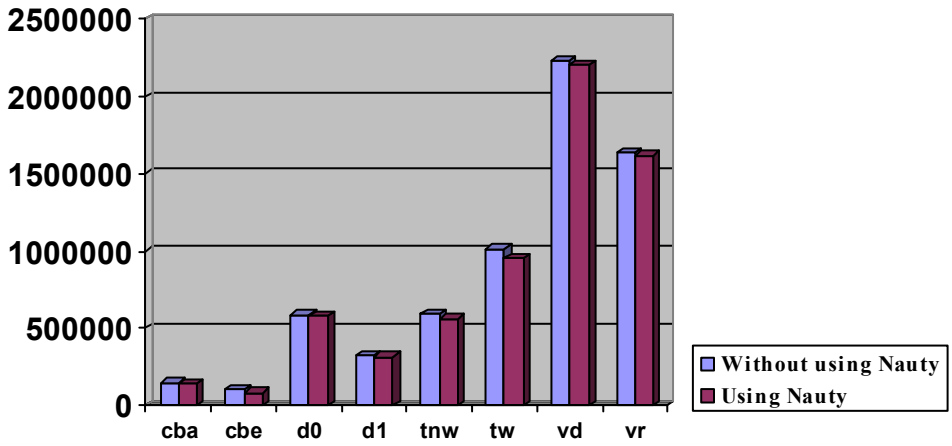


Figure 22: Plot of the number of calls to graphMatch with and without using nauty to perform graphMatch.

4.8 Conclusions

The idea of using canonical labeling as an alternative to graphMatch is more useful when graphMatch is performed among same set of graphs for a number of times. The experimental results show that there has not been a significant reduction in the runtime of SUBDUE when canonical labeling is used. The reasons for this type of behavior is there has not been a great reduction in the number of calls to graphMatch when canonical labeling is used, hence the time saved in graphMatch has been undone by the time taken to create canonical labels. There has been the highest reduction in the number of calls to graphMatch for chess_illegal with the reduction being 8%. For all the other graphs the reduction is less than 8%.

Also a lot of time is being spent to convert the graph representation in SUBDUE to be compatible with nauty (i.e., converting edges into vertices and giving an initial partitioning of the vertices of the graphs).

Even if nauty is considered to be the fastest graph isomorphism algorithm available, it has been shown that there are categories of graphs for which it employs exponential time in order to find an isomorphism [32].

However some studies suggest that canonical labeling and graph isomorphism detection are not equivalent and canonical labeling is harder than isomorphism [33].

As can be seen from the experimental results, the idea of using canonical labeling for SUBDUE does not seem appropriate. If a different canonical labeling algorithm is used, then the time taken for preprocessing is reduced and hence the computation time taken would probably be reduced. But still there would not be a

significant reduction owing to the fact that there has not been a great reduction in the number of calls to graphMatch when canonical labeling is used.

4.9 Future Work

One of the main reasons for the increase in the runtime is the time saved by reducing the number of calls to graphMatch has been undone by the time taken to initially create canonical labels. Other factors that contribute to the runtime are the time taken for preprocessing to make the graphs used in SUBDUE compatible with nauty and time taken to initially partition the vertices. The increase in the graph size by converting the edges into vertices increases the runtime. This has been validated by running experiments on the synthetic graphs created with one single edge label.

Hence if a canonical labeling algorithm can be devised that takes less time than nauty, then it would be a good option to try for SUBDUE.

CHAPTER 5

REDUCING THE NUMBER OF CALLS TO GRAPHMATCH

In this chapter, we provide the analysis of an approach proposed by Potts that reduces the number of calls to the graphMatch routine in SUBDUE.

5.1 Introduction

Graph match is computationally the most expensive routine in a graph-based data mining system. Hence, the computational complexity of a graph-based data mining system can be reduced by reducing the number of calls to graph match. Here we describe and analyze this alternative mechanism for scaling SUBDUE. The original algorithm enhancement has been designed by Potts [34]. Here we provide the analysis and assessment of its features in comparison with the improvement we have described for the graph match algorithm.

5.2 Graph match in SUBDUE

As discussed in the algorithm for SUBDUE in section 2.2, SUBDUE discovers substructures that best compress the input graph by growing the instances of the best substructures discovered. The process begins by discovering all instances of a single vertex and proceeds in a level-by-level fashion until the best substructure is found. SUBDUE discovers substructures by growing them level-by-level by one edge or an edge and a vertex in every level. This process of growing a substructure by an edge or an edge and a vertex is done by taking the instances of the substructure and extending

each one of the instances in all possible ways. The procedure *Extend* illustrates this process.

Procedure *Extend (Substructure Sub)*

1. *newInstanceList = ExtendInstances (sub → instances);*
2. Take an instance from *newInstanceList,* create a substructure from this instance and check if it is already present in the list of *extendedSubs (*call *MemberOfSubList* which performs a graphMatch between this substructure and all the substructures present in the list of *extendedSubs*)
   If no, then add this substructure to *extendedSubs* and add instances of this substructure to it, using *AddPosInstancesToSub* and *AddNegInstancesToSub*, both of which perform graphMatch using a call to *NewEdgeMatch*.
3. Repeat step 2 for all the instances in *newInstanceList.*
4. Return *extendedSubs.*

Procedure *ExtendInstances (instanceList)*

1. Take the first instance, *instance* from *instanceList.*
2. Extend *instance* in all possible ways by one edge and an already present vertex or one edge and one vertex and insert the extended instances into *newInstanceList.*
3. Repeat the above 2 steps for all instances in *instanceList.*
4. Return *newInstanceList.*

The above procedure does not take any advantage from the fact that all the instances of a substructure are isomorphic to each other and hence if they are extended in the same way, the resulting extended instances will be isomorphic and need not be compared by a graphMatch operation.

5.3 Reducing the number of calls to graphMatch (EE-SUBDUE)

The following are the observed features of SUBDUE that aid in reducing the number of calls to graphMatch. The instances of a substructure are always isomorphic to each other. As a result, if they are extended by the same edge, in the same direction, they are still isomorphic. Hence they need not be compared using graph match; instead a simple test can be performed on the last extension to decide if they are isomorphic.

The current version of SUBDUE extends an instance in all possible ways by one edge or one edge and one vertex and then performs an isomorphism test to see if an instance is isomorphic to a substructure. In the efficiency enhanced version of SUBDUE, all the instances of a substructure are extended at the same time by one edge or one edge and one vertex and hence they need not be compared for isomorphism. In this way the number of calls to graph isomorphism is reduced and hence the total runtime for SUBDUE is reduced.

The following are the revised functions corresponding to the *Extend* and *ExtendInstances* functions in the original version of SUBDUE.

Procedure *NewExtend (Substructure Sub)*

1. *newInstance = ExtendInstanceByEdgeOrVertex (sub → instance);*
2. Take the instance *newInstance,* create a substructure, *newSub* from this instance, if there is not already a substructure with this definition.
3. Add *newInstance* as an instance to *newSub.*
4. Repeat steps 2, 3 for all the instances of *Sub.*

Procedure *ExtendInstanceByEdgeOrVertex* extends the given instance by the given edge and vertex and returns the extended instance.

As seen from our function *NewExtend,* graphMatch is not being performed to check if a substructure is already present in the list of *extendedSubs.* In other words there are no calls to *MemberOfSubList*, which does a graphMatch to find if the substructure is already present in the list of *extendedSubs.* This method is taking advantage of the fact that all the instances of a substructure are isomorphic and hence they need not be tested for isomorphism by calling graphMatch but instead can be tested by comparing the last extension.

In addition, there are no calls to *AddPosInstancesToSub* and *AddNegInstancesToSub,* both of which make a call to *NewEdgeMatch* that performs graphMatch. Hence there is a reduction in number of calls to graphMatch here also.

On the whole, no calls are made to *MemberOfSubList, AddPosInstancesToSub* and *AddNegInstancesToSub* and in turn there are no calls to graphMatch from these routines. As a result the number of calls to graphMatch is drastically reduced and hence the runtime is improved.

5.4 Experimental Results

Table 6 shows the comparison of the original SUBDUE code (version 5.1.2) and the efficiency enhanced version of SUBDUE (which we will refer to as EE-SUBDUE). The experiments are conducted with the following parameter values. MDL is used as the substructure evaluation method. The value of threshold is 0, so exact graph match is being performed. SUBDUE's beam width is set to 4.

Table 6 shows the number of calls to graphMatch and the time taken by the two versions. As can be seen from the table, there has been a significant reduction in the number of calls to the graphMatch routine.

Figures 23 and 24 show the plot of the results shown in table 6. The graphs are shown along the x-axis and the runtime or number of calls to graphMatch is shown along the y-axis. The labels cba, cbe, d0 ,d1, tnw, tw, vd and vr correspond to chorales_bach, chorales_beethoven, diabetes_0, diabetes_1, ttt_notwin, ttt_win, vote_d and vote_r respectively.

The reduction in the number of calls to graphMatch is a lot more than the reduction obtained by making use of the canonical labeling technique. The reason for this is when canonical labeling is used, there is a reduction in the calls to graphMatch from the two routines *SubListInsert* and *MemberOfSubList*. But a maximum number of calls to the graphMatch routine are made from the routine *NewEdgeMatch.* With the approach proposed in EE-SUBDUE, there are no calls to the graphMatch routine from *NewEdgeMatch.*

Table 6: Comparison of runtime of SUBDUE-5.1.2 and EE-SUBDUE.

Graph	**No. of calls to graphMatch in SUBDUE**	**No. of calls to graphMatch in EE-SUBDUE**	**Time taken by SUBDUE (seconds)**	**Time taken by EE-SUBDUE (seconds)**
chess_illegal	72307782	789	16481.23	588
chess_legal	33477481	836	3442.91	230
Chorales_bach	147097	1075	2.44	1.00
Chorales_beethoven	100550	1634	1.89	1.00
ttt_notwin	587676	586	7.33	2.00
ttt_win	1009523	684	17.82	4.00
vote_d	2230530	669	41.34	5.00
vote_r	1639956	764	32.16	3.00
diabetes_0	584691	418	5.51	1.00
diabetes_1	317584	372	2.25	0.00
credit_1	13915620	3107	299.19	49
credit_2	5563057	3079	58.34	14
Sample10.graph	6697	465	0.08	0.0
Sample9.graph	11449	527	0.11	0.0
Sample8.graph	9802	411	0.07	0.0
Sample7.graph	9057	465	0.08	0.0
Sample6.graph	10679	476	0.12	0.0
Sample5.graph	22460	742	0.21	0.0
Sample3.graph	23442	1176	0.34	0.0

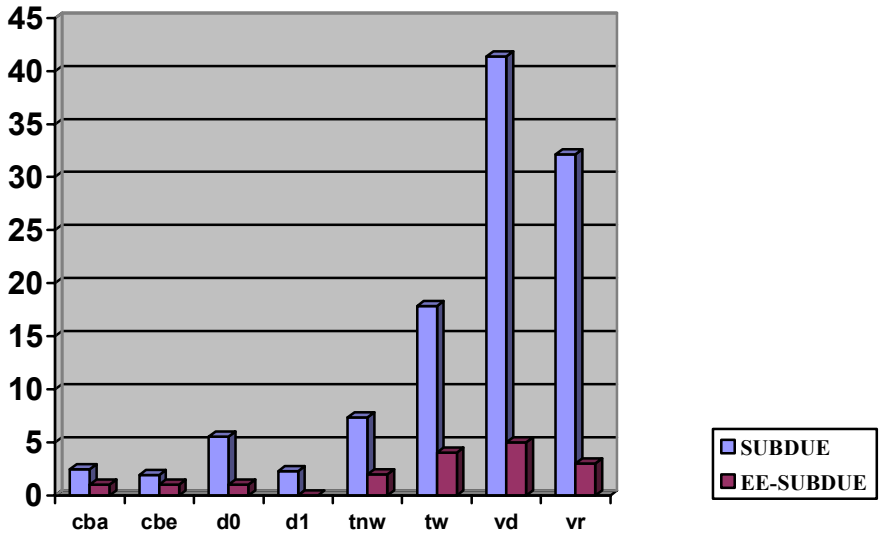


Figure 23: Plot of runtime of SUBDUE Vs. runtime of EE-SUBDUE.

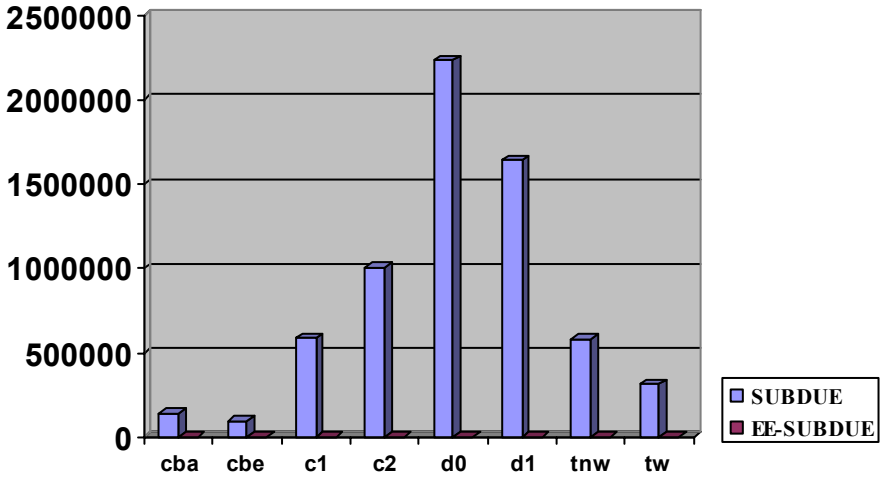


Figure 24: Plot of the number of calls to graphMatch in SUBDUE vs. the number of calls to graphMatch in EE-SUBDUE.

5.5 Conclusions

As seen from the experimental results, a significant reduction in the runtime has been obtained by reducing the number of calls to graphMatch. The experiments showed that there were no calls to graphMatch from *MemberOfSubList* and *AddPostInstancesToSub* and *AddNegInstancesToSub*. All calls to graphMatch are from *SubListInsert.*

5.6 Future Work

The above approach works for only exact graph match i.e. only if the threshold is 0.0. An approach that works for inexact graph match should be considered.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this thesis, we have looked at methods to increase the scalability of graph-based data mining systems. Methods such as approximation techniques using genetic algorithms and alternative techniques like canonical labeling are discussed. An overview of an approach proposed by Potts that reduces the number of calls to graphMatch has been given. All of these techniques have been implemented and analyzed on a variety of databases.

The use of a genetic algorithm for subgraph isomorphism detection suffers from the fact that there is no appropriate stop condition for the algorithm. This is due to the fact that the number of instances of a subgraph in the main graph is not known before hand and the algorithm in turn has to run for a fixed number of generations, which increases the runtime of the algorithm.

On the other hand, the canonical labeling technique for graph isomorphism detection suffers when the time saved by reducing the number of calls to graphMatch is undone by the time taken to create the canonical labels. As a result the canonical labeling approach is useful when graphMatch is being performed among the same set of graphs multiple times.

Of the three approaches we analyze, the approach that reduces the number of calls to graphMatch is the most successful. While all of the techniques offer some benefit for graph-based data mining, reducing the need for graph isomorphism and subgraph isomorphism tests, appears to be the most effective scaling mechanism.

6.2 Future Work

Our reasons for investigating approximation algorithms for subgraph isomorphism and graph isomorphism are twofold. First, the runtime cost of these algorithms is intractable, making application of data mining algorithms to graph data impractical. Second, approximation approaches have been suggested in the literature, thus we sought to determine if they would benefit a complex discovery algorithm such as SUBDUE.

Our research results indicate that while approximation algorithms do improve the runtime of isomorphism tests, their benefits are limited. There are many directions for future research that could further improve these results. First, the genetic algorithm may be modified by changing the fitness function. In addition the size of the population and the number of generations may be parameterized. A new canonical labeling algorithm may be considered that takes less time to calculate the canonical labels. A canonical labeling algorithm may be developed for SUBDUE as was done for FSG. Other techniques that increase the scalability of the graph-mining algorithms by pruning the search space should be considered. Our research has looked at techniques to improve the scalability of graph-based data mining systems by reducing the time-complexity of the graph and subgraph detection algorithms or by reducing the number

of calls made to these algorithms. Future work may be focused on other techniques to improve the scalability such as pruning the search space.

REFERENCES

[1]. D.J.Cook and L.B.Holder. Substructure Discovery Using Minimum Description Length and Background Knowledge. In *Journal of Artificial Intelligence Research,* Volume 1, pages 231-255, 1994.

[2]. Rissanen, J., Modelling the shortest data description, In *Automatica*,(14), pp 465-471 1978.

[3].L.B.Holder, D.J.Cook and S.Djoko. Substructure Discovery in the SUBDUE system. In *Proceedings of the AAAI workshop on knowledge discovery in Databases,* pages 169-180, 1994.

[4]. J.Gonzalez, L.B.Holder, and D.J.Cook, Graph-Based Concept Learning, *Proceedings of the Florida Artificial Intelligence Research Symposium,* 2001.

[5]. I.Jonyer, L.B.Holder and D.J.Cook, Hierarchical Conceptual Structural Clustering, *International Journal on Artificial Intelligence Tools,* 10(1-2) pages 107-136, 2001.

[6] I.Jonyer, L.B.Holder and D.J.Cook, MDL-Based Context-Free Graph Grammar Induction, *Proceedings of the sixteenth International Conference of the Florida AI Research Society,* May 2003.

[7]. http://www.econ.iastate.edu/tesfatsi/holland.GAIntro.htm

[8]. L.Babai. Moderately exponential bound for graph isomorphism. In F. Gecseg, editor, Lecture Notes in Computer Science: *Fundamentals of Computation Theory*, pages 34-50. Springer Verlag, 1981.

[9]. E.M.Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, pages 42-65, 1982.

[10]. C.M.Hoffman. *Group-theoretic Algorithms and Graph Isomorphism*. Springer Verlag, 1982.

[11]. J.E. Hopcroft and J.K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Annual ACM Symposium on Theory of Computing*, pages 172-184, 1974.

[12]. J.R. Ullman. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31-42, 1976.

[13]. S.H. Myaeng and A. Lopez-Lopez. Conceptual graph matching: a flexible algorithm and experiments. *Journal of Experimental and Theoretical Artificial Intelligence*, 4:107-126, April 1992.

[14]. B. Falkenhainer, K.D. Forbus, and D. Gentner. The structure-mapping engine: Algorithms and examples. *Artificial Intelligence*, 41:1-63, 1989/90.

[15]. Subgraph Isomorphism in Polynomial Time by B.T.Messmer and H.Bunke.

[16]. An algorithm Using Length-R Paths to Approximate Subgraph Isomorphism by Fred DePiero and David Krout.

[17]. Subgraph Isomorphism in Planar Graphs and Related Problems by David Eppstein.

[18]. B.S.Baker. Approximation algorithms for NP-complete problems on planar graphs. J.Assoc. Comput.March, 41:153-180, 1994. Preliminary version in 24th *IEEE Symp*. *Foundations of Computer Science*, 1983, pp. 265-273.

[19]. Genetic Algorithms for the Subgraph Isomorphism problem by Max Pesakhov and William Regli.

[20]. B.D.McKay, nauty users guide. http://cs.anu.edu.au/~bdm/nauty/.

[21]. B.D.McKay, Practical Graph isomorphism. *Congressus Numerantium,* 30:45-87, 1981.

[22]. A.Aho, L.Hopcroft, and J.Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading MA, 1974.

[23].D.G.Corneil, C.C. Gotlieb, An efficient algorithm for graph isomorphism, *Journal of the Association for Computing Machinery,* 17,pp. 51-64, 1970.

[24]. R.Mathon, Sample graphs for isomorphism testing, *Congressus Numerantium, 21,* pp. 499-517, 1988.

[25]. D.C.Schmidt, L.E.Druffel, A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices, *Journal of the Association for Computing Machinery,* 23, pp.433-445, 1976.

[26]. An efficient algorithm for discovering frequent subgraphs by Karypis and Kuramochi.

[27]. Liu, X.; Klien, D.J. The graph isomorphism problem. *J.Comput. Chem.* 1991, 12, 1243-1251.

[28]. Collatz, L.; Sinogowitz, U. Spektren endlichen Graphen. *Abh. Math. Sem. Univ. Hamburg* 1957, 21, 63-77.

[29]. S.Fortin, The graph isomorphism problem. Technical Report TR96-20, Department of Computing Science, University of Alberta, 1996.

[30]. Fan, B.T.; Barbu, A.; Panaye, A.; DOucet, J.P. Detection of constitutionally equivalent sites from a connection table.*J.Chem.Inf.Comput.Sci.*1996, 36,654-659.

[31]. http://www.cs.sunysb.edu/~algorith/implement/nauty/implement.shtml

[32]. T.Miyazaki, The complexity of McKay's canonical labeling algorithm, in *Groups and Computation, II* (L.Finkelstein and W.M.Kantor,eds), *Amer.Math.Soc*, Providence, RI, pp.239-256, 1997.

[33]. Babai, L.; Kucera, L.; Canonical labeling of graphs in linear average time. In *Foundations of Computer Science*, Proceedings of the 20th IEEE symposium, 1979; pp-39-46.

[34]. J.Potts, Supervised Learning in Embedded Subgraphs .

[35]. Isomorphism, Automorphism Partitioning and Canonical Labeling can be Solved in Polynomial-Time for molecular graphs. In *J. Chem. Inf. Comput. Sci.* **1998,** 38, 432-444.

[36]. Murphy,~P.~M., \& Aha,~D.~W. (1994). {\it UCI Repository of machine learning databases} [http://www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, Department of Information and Computer Science.

BIOGRAPHICAL INFORMATION

Srilatha Inavolu received her Bachelor of Technology degree in Computer Science and Engineering from SriNidhi Institute of Science and Technology affiliated to Jawaharlal Nehru Technological University, Hyderabad, India in 2004. She received her Master of Science degree in Computer Science and Engineering from The University of Texas at Arlington in May 2006.