공학박사학위논문

An Incremental Genetic Algorithm for Graph Optimization Problems

그래프 최적화 문제를 위한 점진적 유전 알고리즘

2016년 8월

서울대학교 대학원
전기·컴퓨터공학부
김 진 현

An Incremental Genetic Algorithm for Graph Optimization Problems

by

Jinhyun Kim

Department of Electrical Engineering & Computer Science
Seoul National University
2016

Abstract

A combinatorial optimization problem is an optimization problem having a discrete solution space. Lots of the graph problems belong to this category as graphs are discrete objects. Graphs are widely used in the various field and there are lots of real world combinatorial optimization problems which take the graphs as their input. For some of these problems, the magnitude of the solution space is exponential to the size of the problem, and thereby efficient space search algorithms are required to deal with them.

Genetic algorithms are widely used to solve combinatorial optimization problems, and incremental genetic algorithms could be used to efficiently solve graph optimization problems. We define subproblems and solve them step by step instead of tackling the problems directly. A subproblem solved by an incremental genetic algorithm deals with a restriction of the original graph structure. The subproblems are solved in the intermediate steps and the size of the subproblem is gradually increased. We apply the same genetic algorithm to each subproblem, and it is initialized with the evolved population of the previous step.

We propose incremental genetic algorithms for two different combinatorial optimization problems; the subgraph isomorphism problem and graph cut optimization problem. We devise an optimal substructure on the subproblem sequence and explain how it is related to the optimality of the process, along with other related factors. We present graph expansion methodologies and vertex reordering schemes to define an appropriate sequence of subproblems. We combine the proposed incremental approach with a hybrid genetic algorithm for the subgraph isomorphism problem, and the algorithm was further developed for nearly perfect results. Based on our analysis, we also propose an incremental genetic algorithm to solve graph cut optimization problems. We tested the implementation of the algorithm on benchmark graph instances for the graph partitioning problem and the maximum cut problem. Through experiments, we investigate and analyze how the sequence of subproblems affects the search space landscape. The performance of a genetic algorithm makes an improvement when the incremental approach is applied with respect to an appropriate sequence of subproblems.

Keywords : Incremental genetic algorithm, graph optimization problems, subgraph isomorphism problem, graph partitioning problem, maximum cut problem

Student Number : 2008–20860

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A combinatorial optimization problem is an optimization problem that has a discrete, or sometimes even a finite solution space [CCPS98, Sch03]. Permutations and/or sets of the objects are commonly used to represent the elements of such spaces. Graphs are representative discrete data structures and most of the graph optimization problems belong to combinatorial optimization problems. For the graph problems that are classified as NP-hard problems, we need to examine an extreme number of combinations to find a good solution. Even worse, the discreteness of the solution space makes it hard to escape from bad local optima. Genetic algorithms (GAs), and other stochastic approaches are widely used to deal with this difficulty.

In contrast to NP-hard graph problems, there exist polynomial time algorithms for some of the graph problems in P class. Minimum spanning tree problems and shortest path problems are well-known examples frequently appearing in algorithm textbooks [CSRL01, HH13]. The algorithms for these problems are based on either a greedy method or a dynamic programming algorithm. Both of them rely on the optimal substructure of the problem, which is a property that optimal solutions of subproblems could be extended to an optimal solution of the original problem.

The idea of utilizing an optimal substructure could be applied to solve NP-hard graph problems, by means of an incremental genetic algorithm.

An incremental genetic algorithm is an evolutionary computation model to solve a problem that dynamically changes over time [MAEf06, WYJ$^+$04]. The algorithm initializes the population only once at the beginning. And it reuses the population even when the problem has been changed, by evolving the population with respect to the new problem.

To solve a graph optimization problem with an incremental genetic algorithm, it begins with solving a small subproblem and the problem is gradually expanded to the original problem. Lots of graph optimization problems could be tackled in this manner if the structure of the subproblems is carefully designed. When the subproblems are expanded in an appropriate way, which is the way that best reflects the optimal substructure of the problem, the incremental genetic algorithm could bring a significant performance improvement upon the ones without the incremental approach.

This thesis investigates the mechanism of an incremental genetic algorithm for graph optimization problems through empirical analysis. We first formalize the underlying structure and the process of the algorithm. Then we analyze the property of the subproblems dealt during the incremental process, and discuss how it is related to the overall space search behavior. Based on our analysis, we propose an IGA applied to two kinds of graph problems; one of them is the subgraph isomorphism problem [CYM12], and the other is the graph cut optimization problem [KHKM11, WWL15]. We use the natural subproblem structure defined by the subset of the vertices or the subset of the edges, and we seek methodologies to build up a fine sequence of subproblems. The proposed algorithms are implemented and tested through various experiments. Results of the experiments to verify our analysis are provided with discussions. For both of the problems, using an incremental approach in an appropriate way has brought a performance improvement. It was driven by solving the problem with respect to a sequence of subproblems which has an optimal substructure and controls the difficulty of the space search accordingly, which is defined in terms of the number of optimal solutions of the subproblem.

The main contributions of this thesis are as follows.

- *We have applied an incremental genetic algorithm to solve graph optimization problems.*

  An incremental genetic algorithm is an evolutionary approach to specific problems. We have applied the algorithm to solve seemingly unrelated graph problems, by introducing a notion of subproblem. We formally define the process in terms of subproblems, and a sequence of subproblems. The methodologies and schemes to define the sequence of subproblems are systemically organized as well. Although there are a tremendous number of possible subproblem sequences, only a few of them are actually suitable for an incremental approach. We inspect this property in terms of optimal substructure and provide a guideline for defining the subproblem structure that works well.

- *We propose a high-performance incremental genetic algorithm dedicated to solving the subgraph isomorphism problem.*

  Several evolutionary approaches have been proposed to tackle the subgraph isomorphism problem. However, even the best known one fails to solve the problem in specific cases. We had applied an incremental approach to this problem in various ways, and some of the algorithms outperformed previous ones. We have further improved the performance with two more operators.

- *We provide experimental results on an incremental genetic algorithm for graph cut optimization problems.*

  We propose an incremental genetic algorithm for graph cut optimization problems and analyze the properties through experiments. The structures of these problems are non-trivial and optimal substructures are only found in a few specific subproblem sequences. We seek for useful substructures and investigate how these structures are related to the behavior of the search algorithm.

- *We introduce the related applications of the incremental genetic algorithm.*

  Even if we deal with specific problems in our empirical study, the incremental approach is not restricted to these problems. We introduce an application of the subgraph isomorphism problem, an incremental genetic algorithm with an approximate fitness evaluation for a different problem.

Some portions of the work discussed in this thesis have been presented in [CKM14, KCYM16, KM14, KYM16].

The rest of this thesis is organized as follows. In the next chapter, we first review the incremental genetic algorithm and formally define the process to solve the graph optimization problems. We also provide the properties of the algorithm. We then propose incremental genetic algorithms for the subgraph isomorphism problem and graph cut optimization problems in Chapters 3 and 4, respectively. The result of an analysis on their structures, including experimental results is provided as well. Chapter 5 introduces some related applications, and we make conclusions in Chapter 6.

Chapter 2

Incremental Genetic Algorithm

2.1 Overview and Traditional Applications

A genetic algorithm (GA) is an optimization algorithm inspired by evolution process in nature. It is a search algorithm which tries to evolve a set of solutions, called *population*, to solve an optimization problem [BBM93]. At each iteration, called *generation*, two *parent* solutions are chosen from the population. The *crossover* operator recombines the properties of two parents and creates a new solution called *offspring*, and the *mutation* operator slightly modifies the offspring. The offspring *replaces* one of the solution in the population based on a replacement strategy. For example, the worst solution in the population could be replaced. Through an enough number of generations, the solutions in the population evolve and converge to certain solutions. The algorithm returns the best solution found during the evolutionary process.

When solving an optimization problem, GAs and other evolutionary approaches use one single fixed evaluation function. This function is also called a fitness function and it evaluates how good the solution is. The direction of the evolution is to maximize the fitness of the solutions in the population. However, there are lots of real world optimization problems which use a dynamic fitness function [BKSS00]. The value of a solution is not fixed and it may change over time. These problems are called dynamic optimization problems. For these problems, the fitness is defined to be a function that depends not only on the solution itself, but also on time [CGP11]. The problem becomes much more difficult as we cannot control the time parameter.

Even if other optimization algorithms are not directly applicable to these problems, evolutionary approaches are still effective on them. As evolution in nature occurs without explicitly being aware of the dynamic environment, the algorithms follow the similar property as well and are known to be effective [NYB12]. Lots of algorithms have been proposed and tested on real world problems [CGP11, JB05].

Maintaining a population also plays a key role in solving a dynamic optimization problem. A good solution or partial schemata of it may be found in the populations of past generations, and it is more likely to happen when there are more solutions in the population. The simplest way is to use the same population, and this property is a rationale behind population reusing. Some of the algorithms also maintain an archive of good solutions and reuse them afterward [JB05]. Controlling the diversity of the solutions is another issue. The algorithms maintain the diversity by inserting newly generated solutions (random immigrants), by slowing down the convergence, and by using classical approaches, such as sharing and/or crowding [JB05, NYB12]. Maintaining more than one subpopulation is another approach, and they are called multipopulation approaches [BKSS00, JB05].

An incremental genetic algorithm (IGA) is one example of an algorithm for dynamic optimization. It is an adaptive GA dealing with the situation when the problem has been changed during the evolutionary process. GAs usually solve the same problem from the beginning to the end of the algorithm. However, the problem instance, the characteristic of the problem, or even the problem itself might be changed during evolution. In such situation, the IGA uses the same GA, with the same operators and parameters, to solve the changed problem. Moreover, instead of re-initializing the population at that moment, the IGA reuses the solutions evolved during solving the prior problem. When the alteration in the problem is minor, the saved solutions significantly reduced the running time of the algorithm [MAEf06]. It is known that for a GA, the quality of the solutions in the initial population greatly affects the quality of the final solution [SL12]. The reused solutions help the re-optimization in this respect.

Some previous works intentionally changed the problem to guide the space search in a certain direction. In an incremental approach to multiprocessor scheduling problem, the length of the solutions to be rewarded was increased over the generations [WYJ$^+$04]. Allowing shorter valid sequences in the earlier generations helped with finding longer valid sequences.

For GA based classification models, updating the model is required after a large number of new data are added [BBK11]. The update is essential when the current model fails to reflect the characteristics of the new data [VN]. Variants of incremental approaches are applied to this case as well, and they are called incremental learning algorithms [NYB12].

2.2 Application on Graph Optimization Problems

2.2.1 Formalization of the Incremental Process

In this section, we propose an IGA for graph optimization problems. Basically, graph optimization problems are not dynamic and thus an IGA is not directly applicable to them. For an efficient space search, we intentionally define the subproblems of the given graph optimization problem and solve them sequentially with an IGA. To the best of our knowledge, it is the first application of an IGA on graph optimization problem in this context.

We first define the subproblem of the target problem in terms of the substructure of the target graph. Then we solve the problem with increasing the size of the subproblem step by step. The graph for which we solve is gradually expanded from the empty structure to the entire one. We apply a GA to each subproblem, initialized with the evolved population of the previous step. Note that the actual problem we are dealing with does not change over time; we instead define virtual subproblems and solve them step by step. Also, as the last subproblem equals the original problem, the final solutions found by an IGA are the solutions of the original problem as well.

We then formalize the IGA for graph problems. We define a graph problem to be an optimization problem which takes a graph $G = (V, E)$ as the input. Since graph is a discrete structure, we can easily obtain a substructure of G by taking the subset of V or E. The former one is called a *subgraph*, and the latter one is called a *spanning graph*. When the input

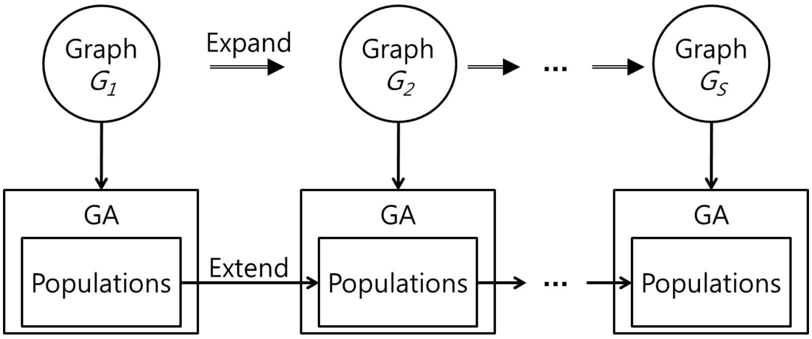


Figure 1: An overview of an incremental genetic algorithm for a graph problem

graph of the same problem is changed to the one having a substructure, we say that the new problem is a *subproblem* of the original one. Note that a subproblem is a kind of a restricted problem, as the input graph consists of a partial structure of the original graph.

To solve a graph problem more efficiently, we use an IGA to consecutively solve the subproblems. As the actual problem to be solved stays the same, we identify each subproblem with its input graph. Consider a finite sequence of subproblems $\{G_1, G_2, \ldots, G_S\}$, where G_i is a subproblem of the original problem G. We must set G_S to be as same as G, and we may set G_1 to be a sufficiently small graph. The rest of the graphs are obtained by expanding the structure of the previous graph, which means that G_i is a subproblem of G_{i+1}.

We use an IGA to solve the subproblems in S steps, as illustrated in Figure 1. The same evolutionary process is applied to solve each of the subproblems, and the evolved population of the i^{th} step is used as the initial population for the next $(i+1)^{\text{th}}$ step. The graph is expanded at each step, and the solutions are extended before the beginning of each step to become solutions of the changed problem. Graph optimization problems are usually to find an optimal permutation or an optimal subset. The extension is to change the solution to a permutation with more elements, or to a subset of a set with more elements, respectively. Furthermore, recalculation of the fitness is required after the extension.

The rationale behind the incremental process is that high-quality solutions of the previous step may provide good starting points of the current step. Previous works on IGA commonly suggest similar properties [BBK11, MAEf06, VN, WYJ$^+$04], and they are called memory-based approaches to dynamic optimization problem [JB05]. Furthermore, it is easier to solve the subproblem as G_i is smaller than G_{i+1}. Sufficiently good solutions for the easier previous subproblem are likely to be found by the algorithm.

Providing good initial solutions to a GA is widely used technique, particularly in a context of a memetic algorithm [VS02, SL12]. A memetic algorithm is a variant of a GA which locally optimizes the solution whenever a new one is generated. It searches the space consisting of local optima, instead of searching the solution space directly. Furthermore, as there are a large number of local optima for most of the problems, the population of the solutions of a memetic algorithm is likely to be diverse. An IGA is another algorithm which starts the process with good solutions, and it uses relatively small computational cost compared to a memetic algorithm. And it is more flexible so that other techniques, including local optimization, are applicable to IGAs. One example of such hybridization will be discussed in Chapter 3.

2.2.2 Theoretical Background

The key notion of a greedy method or a dynamic programming algorithm is an optimal substructure. We say that the graph optimization problem has an optimal substructure, if optimal solutions of the subproblems could be combined into an optimal solution of the original problem [CSRL01]. In other words, it is the case when an optimal solution contains optimal solutions of the subproblems inside.

Suppose that we are to find a shortest path to a vertex v in a graph G. Dijkstra's algorithm and Bellman-Ford's algorithm are well known algorithms for this problem, and each of them is a greedy method and a dynamic programming algorithm, respectively. The algorithms search for shortest paths to intermediate vertices. When a path to an intermediate vertex u is found, the algorithms combine the path with an edge (u, v), or another path from u to v to construct a new path to v. They are relying on an optimal substructure of the problem, which is a property that a shortest path to v contains a shortest path to u. Note that the number of shortest paths could be greater than one, and any one of them could be chosen by the algorithms. Furthermore, the numbers of the subproblems are polynomial functions in the size of the graph.

The generalization of this notion could be applied to an NP-hard graph optimization problem. Some of the graph problems are solvable by a greedy method or a dynamic programming algorithm. For example, there exists a dynamic programming algorithm for the traveling salesman problem (TSP), the best known NP-hard problem [HH13]. However, these algorithms are much more computationally expensive than the ones for P problems. They have to either maintain a large number of optimal solutions to each of the subproblems, or solve a large number of subproblems. The amount of the computation is usually exponential to the size of the problem.

We can say that an IGA has an optimal substructure. More precisely, when a sequence of subproblems is given, we say that the sequence has an optimal substructure, in NP-hard perspective, if one of the optimal solutions of each subproblem could be extended to an optimal solution of the next subproblem. Existence of an optimal substructure conceptually explains that we can solve the original problem by enumerating all of the optimal solutions for each of the subproblems. We have to remark that the number of optimal solutions could be exponential to the size of the graph. However, as GA maintains multiple solutions but not a single solution, the algorithm may find an extendable solution with a higher chance. We expect the utilization of this optimal substructure in an IGA to be highly effective.

However, existence of an optimal substructure does not guarantee that an IGA can always find an optimal solution; it only suggests that it is highly probable. Not every sequence having an optimal substructure leads to a good solution. The IGA expects the subproblem of the previous step is completely solved beforehand, but it is not guaranteed if there exist a large number of optimal solutions for intermediate subproblems, and only a few of them are extendable to optimal solutions for the original problem. Then, intermediate GAs will hardly find promising solutions. This suggests that we need to deliberately set the difficulty of some earlier subproblems to be high, in order to reduce the search space by reducing the number of optimal solutions.

For the SIP tackled by an IGA in Chapter 3, any sequence of the subgraphs has an optimal substructure because the subgraph relation is transitive, i.e., if G_A is a subgraph of G_B and G_B is a subgraph of G_C, then G_A is a subgraph of G_C. But an experimental result shows that only degree based vertex reordering scheme is actually effective. Consecutive subproblems should be highly related in order to promote the utilization of the previous population. In addition, selecting a vertex with the highest degree adds the largest number of edges to the graph. This makes a strong constraint which reduces the number of intermediate optimal solutions as well.

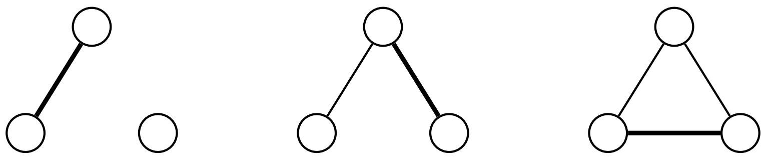
Even worse, the IGA is more likely to fail if there exists no subproblem sequence with an optimal substructure. This property does not hold for some specific problems, and the traveling salesman problem (TSP), the problem to find a shortest tour of the given graph, is one of the examples. When we choose a vertex and remove it from the graph to obtain a subproblem, the solution to the subproblem could be entirely different. An IGA may find a reasonable solution through the evolutionary process in the last step, but the initial solutions may not play any role in an efficient search. Note that the TSP is to find an optimal permutation of vertices. For the problems of finding an optimal subset of vertices, an IGA is more likely to work well on the problem. We will examine two example problems in Chapters 3 and 4.

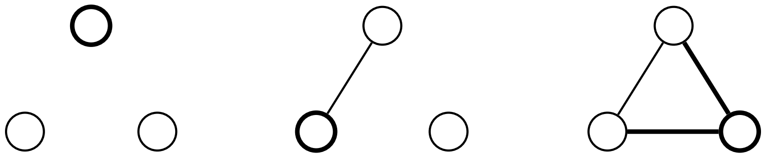
Using an incremental approach has another advantage. When calculating the fitness function value for a subproblem, only part of the original graph is used and it approximates the original fitness. When an approximate fitness evaluation is used with a GA, it helps to prevent the GA from falling in bad local optima. Another previous work used similar mechanism, and it increased the probability of finding a valid solution [WYJ+04]. For graph problems, some part of the graph may mislead the space search [HKY15]. We could avoid this situation by solving a subproblem which does not have a misleading part in earlier steps. This presents another perspective in constructing the sequence of subproblems.

2.2.3 Sequence of Subproblems

As described in the previous subsection, an IGA is defined when a sequence of subproblems is built. We first set the virtual 0th subproblem to be a graph having no edges. The rest of the subproblems, from the first one to the last one, are obtained by adding some edges to the previous graph. The subproblems are determined by the number of the steps, the number of the edges to be added at each step, and the order of the edges to be added. The first two determine '*how*' the edges are added, and the third one determines '*which*' edges are added. In this thesis, we propose three different graph expansion methods in determining the '*how*' part, and propose four different reordering schemes in determining the '*which*' part. Note that the best method and scheme depend on the problem, and choosing the most suitable ones is an important design issue. The methods and schemes will be briefly outlined, and then a design guideline will be provided.

The three graph expansion methods are illustrated in Figure 2 for an example case. All of the methods expand the graph through three steps, and the steps are presented from left to right. The original graph consists of three vertices and three edges, as shown in the rightmost part of the figure. The edges to be added at each step, and the vertices considered at each step are


(a) Edge-wise expansion


(b) Vertex-wise expansion

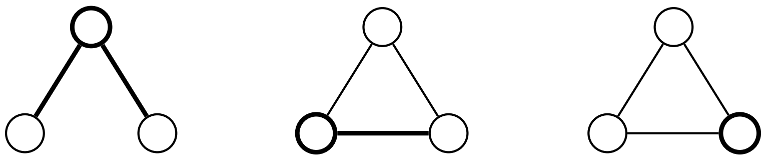

(c) Mixed expansion

Figure 2: Three methods of expanding a graph

drawn with thicker lines. Note that the number of vertices or edges to be added at each step may differ. The details are as follows.

- **Edge-wise expansion**: As shown in Figure 2(a), an edge is added to the previous graph at each step. The number of the steps equals the number of the edges.

- **Vertex-wise expansion**: As shown in Figure 2(b), a vertex is considered at each step. The incident edges connected to a vertex that has already been considered in the previous step are added to the graph. This method is conceptually identical to adding the considered vertex at each step. This method may either actually or conceptually add a vertex to the previous graph. When vertices are added conceptually, the set of the vertices remains the same and corresponding edges are added. The number of the steps equals the number of the vertices.

- **Mixed expansion**: As shown in Figure 2(c), a vertex is considered at each step and all of the incident edges are added to the previous graph. The philosophies of the two above methods are mixed in this one. The number of the steps equals the number of the vertices.

We call IGAs based on each of the three graph expansion methods an edge-wise IGA (E-IGA), a vertex-wise IGA (V-IGA), and a mixed IGA (M-IGA), respectively.

We also propose various reordering schemes to find a good sequence of subproblems. As optimal substructures of the problems may differ, the suitable reordering scheme differs as well. Therefore, we provide the details of the schemes in Chapters 3 and 4, as well as the structural overview of the corresponding problem.

To find one of the best reordering schemes for the given problem, we first have to examine the optimal substructure of the problem. For an IGA to be effective, the subproblem sequence should have an optimal substructure. To determine whether a subproblem sequence has the property, it is recommended to check whether an optimal solution to the original problem retains its optimality on the subproblems. If this property holds, it means that there exists an extendable optimal solution, which is the one we have checked, for all of the subproblems. The problem is that we may conceptually check the optimality of the solution, but we cannot find the actual solution before solving the problem. If the subproblem sequence is hardly induced without knowing an optimal solution a priori, then we have to find the sequence which indirectly reflects the optimal substructure. Using a greedy based approach would be helpful in those cases.

The second principle is to reduce the number of optimal solutions of intermediate subproblems. If more than one subproblem sequences are expected to have an optimal, or a near-optimal substructure, then using the one having relatively easy subproblems is better than the others. Adding more number of edges or vertices in the earlier generation is also a useful strategy in this context. Even if a large number of elements are added, the entire size of the graph is small and it is easier to solve such a problem.

More details on selecting an appropriate reordering scheme will be provided in Chapters 3 and 4.

Chapter 3

Subgraph Isomorphism Problem

3.1 Introduction

Graphs are useful, universal and pervasive data representation models in various fields. There are lots of interesting problems defined in terms of graphs including the subgraph isomorphism problem. Finding a common structure in two given graphs is an important and general form of pattern matching, and a common structures is often established by an isomorphism or a subgraph isomorphism. These problems arise in a number of real world applications such as pattern recognition [RP94], computer-aided design [OEGS93], image processing [LL01], bioinformatics [BB02], and cheminfomatics [IWM00].

Given two graphs G_1 and G_2, the subgraph isomorphism problem is to determine whether G_2 contains a subgraph that is isomorphic to G_1. It is a generalization of the maximum clique problem, and is a well-known NP-hard problem [Coo71]. The generalization of the subgraph isomorphism problem is the maximum common subgraph isomorphism problem, which is to find the largest subgraph of two given graphs that are isomorphic to each other. This problem is also known to be NP-hard [GJ90].

Suppose that two graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ are given as the input graphs of the problem to determine whether G_1 is isomorphic to a subgraph of G_2. Instead of tackling the problem directly, an optimization version of the problem is often solved. This problem asks to find a subgraph of G_2 which has the least number of different edges with G_1, or a subgraph which has the most number of common edges with G_1 [BJWG94, KM10, ZWL$^+$11]. If the graph found is the same as G_1, then it is a solution to the decision version of the problem. In this thesis, we use the decision version and the optimization version of the problem interchangeably.

Many algorithms have been proposed for the subgraph isomorphism problem. But usually, these algorithms can solve the problem only for small-sized graphs or for those with notable restrictions [Epp95, Luk82, RWH$^+$10, Ull76]. The recursive backtracking algorithm proposed by Ullmann [Ull76] is one of the most commonly used for exact graph matching, which has exponential time complexity in general. Some approaches reduced the overall computational complexity by setting some limitations on the graphs [Epp95, Luk82] or adopting domain specific knowledge [RWH$^+$10]. Messmer and Bunke [MB00] studied a modified problem that detects subgraph isomorphisms from a number of a priori known graphs, so-called model graphs. They matched the decomposed model graphs onto the given input, and the subgraph isomorphisms for the complete graphs are obtained by recombining this results.

For real world applications, genetic algorithms were also used to solve the problem. Brown et al. [BJWG94] used GA in 2D chemical structure matching. Zhong et al. [ZWL$^+$11] applied GA for the subgraph isomorphism problem to compute resource assignment in real time digital simulators. Kim and Moon [KM10] proposed a malware detection system using a hybrid GA. They represented a malware as a directed dependency graph and transformed the malware detection problem into the subgraph isomorphism problem.

Some previous works used evolutionary approaches for the problem [BJWG94, KM10, ZWL+11]. Among them, the one using a multi-objective approach proposed by Choi et al. showed notable performance [CYM12]. They suggested a multi-objective GA with a local search heuristic for this problem. Comparing the degrees of each vertex of two graphs that are mapped, they counted the number of mismatched vertices. Mapping a vertex v of G_1 to w of G_2 is mismatched if either ingoing or outgoing degree of v is greater than that of w. They combined it with a commonly used fitness function. The function counts the number of mismatched edges, which equals the number of edges in G_1 but not in G_2 plus the number of edges in G_2 but not in G_1. The combined fitness function was $f = 0.1 \cdot f_1 + 0.9 \cdot f_2$ where f_1 and f_2 denote the number of mismatched edges and vertices, respectively. They showed that the new fitness function is more globally convex than that of previous studies and thereby improved the performance and efficiency.

3.2 The Proposed Algorithm

3.2.1 The Structure of the Incremental Genetic Algorithm

The subgraph isomorphism problem has a huge problem space, as it is an NP-hard problem. However, this problem has a good property which could be utilized for an efficient search. If a graph $G_1 = (V_1, E_1)$ is isomorphic to a subgraph of $G_2 = (V_2, E_2)$, then any subgraph of G_1 is also isomorphic to a subgraph of G_2. This means that if we first search for a small subgraph of G_1 in G_2, then the solutions could be used in finding G_1 in G_2. A solution of a large size problem contains those of subproblems, and a solution of a subproblem may contain many good components which could be evolved to a high-quality solution to the bigger problem.

The IGA takes advantage of this structure. We start from a subproblem of small size in the first step of the IGA and gradually expands the problem size. In each step, the results of smaller problems in the previous step constitute an initial population of a hybrid GA in the current step. Through applying this step over and over, the final result, an isomorphic subgraph of the original size, is obtained.

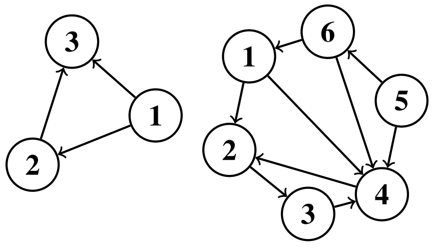
Algorithm 1: Incremental Approach for Subgraph Isomorphism Problem (SIP)

```
Input: $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$
Output: A injective function $g : V_1 \to V_2$
 1: $n \leftarrow$ the number of steps
 2: for $i \leftarrow 1$ to $n$ do
 3:     $m_i \leftarrow$ expansion size of $i$-th step
 4: end for
 5: $V' \leftarrow$ Reordering($V_1$)
 6: $G_{1,0} \leftarrow \varnothing$
 7: $P_0 \leftarrow$ random initial solutions of SIP($G_{1,0}, G_2$)
 8: for $i \leftarrow 1$ to $n$ do
 9:     $V_{curr} \leftarrow \{V'_1, \ldots, V'_{m_i}\}$
10:     $V' \leftarrow V' - V_{curr}$
11:     $G_{1,i} \leftarrow G_{1,i-1} \cup V_{curr}$    //adding $m_i$ vertices
12:     $P_i \leftarrow$ initial solutions generated by $P_{i-1}$
13:     $P_i \leftarrow$ hybrid GA($P_i$)
14: end for
15: return the best in $P_n$
```

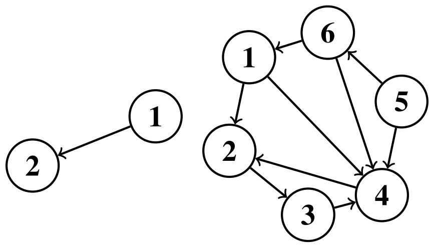
Algorithm 1 presents the IGA for the subgraph isomorphism problem. First, the number of steps n ($1 \leq i \leq n$) and the problem expansion size m_i ($\sum_i m_i = |V_1|$) are determined. Then the vertices in G_1 are rearranged to decide the order of problem expansion. The incremental approach starts from the problem SIP($G_{1,0}$, G_2) with an empty graph $G_{1,0} = \varnothing$. For every i^{th} step, the graph $G_{1,i}$ to be matched is made up from expanding the graph of the previous step $G_{1,i-1}$ by adding m_i vertices. The problem to be solved at this moment is to find a subgraph of G_2 which is isomorphic to $G_{1,i}$. As vertices are added, $G_{1,i-1}$ is a subgraph of $G_{1,i}$, and this suggests that the solutions of the previous problem SIP($G_{1,i-1}$, G_2) could efficiently evolve to the solutions of the current problem SIP($G_{1,i}$, G_2). The results of the previous step are expanded to compose an initial population, and a hybrid genetic algorithm, with a help of a local optimization algorithm, evolves this population over generations for matching $G_{1,i}$ to G_2. This process is repeated for n steps.

Figure 3 shows how the IGA works. Consider two graphs in Figure 3(a). In this case, the number of steps n is 3 and the expansion size for every step m_i is 1. Figure 3(b) describes the state after running the 2nd step. In this step, the incremental GA solves the subproblem SIP($G_{1,2}$, G_2) and gets solutions P_2. Figure 3(c) shows the beginning state of the 3rd step. Graph $G_{1,2}$ expands into $G_{1,3}$ and the initial population of this step P_3 is initialized by P_2 and extra mapping for a newly added node 3. After expansion, the GA evolves P_3 to solve SIP($G_{1,3}$, G_2).

We use this incremental GA framework to solve the optimization version of the subgraph isomorphism problem. Even if the problem is to find a

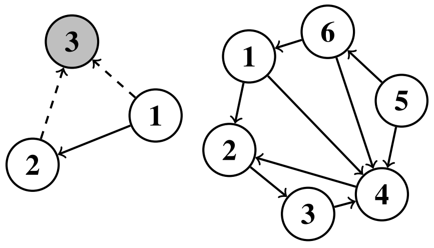


(a)



P_2^1	P_2^2	P_2^3	$\cdots$
$1 \to 2$	$1 \to 6$	$1 \to 4$	$\cdots$
$2 \to 3$	$2 \to 1$	$2 \to 2$	

(b)



P_3^1	P_3^2	P_3^3	$\cdots$
$1 \to 2$	$1 \to 6$	$1 \to 4$	
$2 \to 3$	$2 \to 1$	$2 \to 2$	$\cdots$
$\mathbf{3 \to 4}$	$\mathbf{3 \to 5}$	$\mathbf{3 \to 3}$	

(c)

Figure 3: An example of the incremental genetic algorithm process

similar subgraph, but not exactly the same one, the subgraph structure still could be used for an efficient evolution.

3.2.2 Design Issues

As adding vertices in any order guarantees the optimal substructure, we decided to use a V-IGA in our design. And for a faster computation, we actually add the vertices to the previous graph. The V-IGA thereby solves the subproblems with a different number of vertices at each step. We met four design issues in this algorithm. The primary design issue was to reorder the vertices to build a sequence of subproblems, as explained in Section 2.2.3. The other three issues are the population inheritance, the stopping criterion for each step, and the expansion size of each step. The details of them are explained below.

Vertex Reordering

As any reordering scheme leads to a subproblem sequence with an optimal substructure, the key point is to use the one that provides diversity and proper search direction. The selection of vertices for the problem determines the search direction of the next step and affects the connection between solutions of each step. If vertices far away from the current subgraph are selected, the previous results are not very useful in the next step. Thus it makes sense to select vertices highly related to the current subgraph.

A graph search algorithm and vertex adjacency were used to measure the relation [HKM06]. In addition, selecting an appropriate vertex in earlier steps may prune unnecessary searches. The degree of vertex plays a key role in pruning the search space of the problem [CG70].

We applied three different schemes for reordering the vertices of G_1, as well as a randomized reordering which provides a baseline. The details of the reordering schemes are as follows:

- Max-degree reordering.

  We sort the vertices in non-increasing order of degree. The degree of a vertex is the sum of both ingoing and outgoing degrees.

- BFS reordering.

  We randomly select a starting vertex, and then run the breadth-first search on G_1. When the graph is disconnected and not all of the vertices are visited, we randomly choose another unvisited vertex and continue the procedure.

- Max-adjacency reordering.

  We randomly select a starting vertex, and repeatedly select one of the most attractive vertex in a greedy manner. The attractiveness of a vertex v is the number of adjacent vertices that are already ordered. Two vertices are adjacent to each other if there is an edge in any direction.

Population Inheritance

At the beginning of each step of the IGA, we reuse the population from the previous step. In some previous works on IGA, population reusing does not necessarily mean 100% inheritance. They only copied a certain portion of the population and fill the rest with randomly initialized solutions. This is mainly due to the characteristic of the problem in previous works; the old and new problems may not share similar structures. However, as we use the subproblems which are highly related to their previous subproblem, reusing the entire population would be helpful. We tested IGAs which randomly initialize some chromosomes to verify this property.

Stopping criterion

Basically, we used a fixed number of generations for all of the steps. But this may lead to an excessive number of generations in earlier steps, because hybrid GA may converge very fast for relatively simple graphs. Moreover, keeping some solutions that are not converged in the population may preserve solution diversity. Both the quality and the diversity of solutions in the previous step have a decisive effect on the next step.

We terminate each step if the population is sufficiently converged. We regard the number of the generations as the unit of time and distributed totally 100 generations equally to each step. Before starting each step, we redistributed the remaining generations equally to the remaining steps. By this procedure, more time is assigned to the later steps.

Expansion size

The number of vertices to be added at each step also determines the subproblem of each step. A naive way is to add a single vertex at each step. But it is a waste of time to run a GA when the expanded graph is too simple, for example, when an isolated vertex is added. Adding more vertices at that time enables efficient space search, but an immoderate expansion size may generate an excessively complicated graph. It is required to strike a balance between efficiency and difficulty by selecting a moderate expansion size.

3.2.3 Genetic Framework

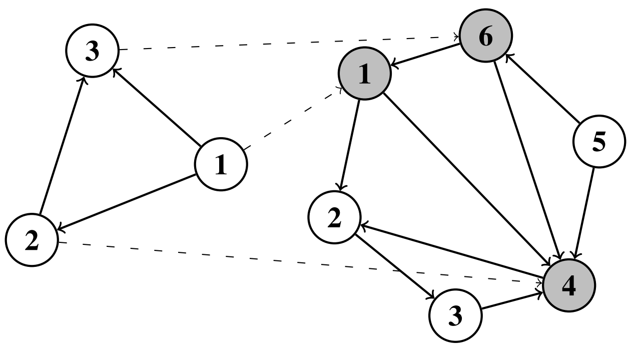
The hybrid genetic algorithm we used in the incremental approach for the subgraph isomorphism problem is described below.

- Representation

  Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ where $|V_1| \leq |V_2|$, a chromosome represents a permutation of V_2 as an integer array. A subgraph isomorphism $g : V_1 \to V_2$, a solution of SGIP(V_1, V_2), is decoded by first $|V_1|$ genes in the chromosome. A vertex $v_{1,i} \in V_1$ is mapped to $v_{2,p[i]} \in V_2$ and an edge $(v_{1,i}, v_{1,j}) \in E_1$ is mapped to $(v_{2,p[i]}, v_{2,p[j]}) \in E_2$. Figure 4 shows an example. Each vertex i in G_1 is mapped by a vertex $p[i]$ in G_2, drawn by dashed lines. The main advantage of this representation is the flexibility toward the problem size expansion. Since a chromosome already has a full permutation of V_2, we can easily extend the mapping at each step without changing values of the genes.

- Fitness Function

  We use the function introduced in [CYM12]. It is defined to be $f = 0.1 \cdot f_1 + 0.9 \cdot f_2$, where f_1 denotes the number of mismatched edges and f_2 denotes the number of mismatched vertices.



i	1	2	3	4	5	6
P[i]	**1**	**4**	**6**	**3**	**5**	**2**

Figure 4: Representation of a chromosome

- Population

  The population size in each step of the incremental approach is 100. When GA starts, it takes the population evolved in the previous step as an initial population of the current step.

- Selection

  The tournament selection is used. We pick two chromosomes randomly and return the better one with 80 percent of a chance, and otherwise return the worse one.

- Crossover

  We used cycle crossover [OSH87].

- Mutation

  We select a number of genes to shuffle them in random order. Each of the genes independently has 40 percent of a mutational chance.

- Local optimization algorithm

  We hybridize the local optimization algorithm with the incremental GA, by applying it to the initial population at the beginning, and to the offspring after crossover and mutation. We randomly swap two vertices when there is an improvement; this is repeated until there is no way to improve. The details are described in Algorithm 2.

Algorithm 2: Vertex swap local optimization algorithm

```
Input: A chromosome $C$ of SGIP($G_1$, $G_2$)
1:  $L \leftarrow \{(i,j) \mid 1 \leq i \leq |V_1|,\ i < j \leq |V_2|\}$
2:  repeat
3:      $flag \leftarrow false$
4:      for all $(i,j) \in L$ in random order do
5:          $swap\ (C[i], C[j])$
6:          calculate the difference
7:          if improved then
8:              $flag \leftarrow true$
9:          else
10:             $swap\,(C[i], C[j])$ // cancel
11:         end if
12:     end for
13: until flag
```

- Replacement

  We generate 50 offspring per generation and take 100 best solutions out of the existing solutions and the offspring.

- Stopping Criterion

  The hybrid GA stops when a certain ratio of the population becomes the optimal solutions to the subproblem of the i^{th} step. We use the ratio values of 1%, 50% and 100%. In the last step, when the subproblem is the same as the original one, the algorithm stops if an optimal solution is found. The fitness value of an optimal solution in the last step is always zero.

3.3 Experimental Results

3.3.1 Dataset and Evaluation

We randomly generated 200 pairs of graphs by following a widely-used graph generation process [CYM12, CFV07, FSV01]. We first generated a larger graph G_2. Exactly $\eta|V_2|^2$ directed edges are randomly generated without any other constraint, where η denotes the edge density. The graph may contain self-loops, and there may be two edges in both directions between two vertices. The smaller graph G_1 is generated from a subgraph of G_2. We randomly selected $|V_1|$ vertices from G_2, and the induced subgraph is taken as G_1. This means that there is always a subgraph isomorphism from G_1 to G_2 and the optimal fitness function value is always zero.

We used 4 different values for η and 5 different values for $|V_1|$ to generate 20 classes of graphs. For each of 20 classes, 10 pairs of graphs are independently generated. This means that we used 200 pairs of graph instances in our experiments. We used 0.01, 0.05, 0.1, and 0.5 for η, and 10, 30, 50, 70, and 90 for $|V_1|$. The number of vertices in the larger graph $|V_2|$ is fixed to 100. These are the parameters used in previous work [CYM12].

We conducted 1,000 runs for each of 200 pairs of graphs to test the algorithms, which means 200,000 runs in total. For each class, we averaged the results of 10 pairs of graphs. This is 10,000 runs for each of the class.

Table 1: Results of a traditional hybrid genetic algorithm

$\|V_1\|$	η	f average	Ratio
10	0.01	0.0000	100.00%
	0.05	0.0000	100.00%
	0.1	**0.0005**	**99.46%**
	0.2	**0.0566**	**62.02%**
30	0.01	0.0000	100.00%
	0.05	**0.4995**	**50.06%**
	0.1	**0.7271**	**84.14%**
	0.2	**0.0499**	**99.56%**
50	0.01	**0.0088**	**91.52%**
	0.05	**0.0050**	**99.93%**
	0.1	0.0000	100.00%
	0.2	0.0000	100.00%
70	0.01	**0.0079**	**92.66%**
	0.05	0.0000	100.00%
	0.1	0.0000	100.00%
	0.2	0.0000	100.00%
90	0.01	0.0000	100.00%
	0.05	0.0000	100.00%
	0.1	0.0000	100.00%
	0.2	0.0000	100.00%

We measured the average fitness function value, the average running time, and the proportion of runs where an optimal solution was found. We wrote the program in C++ language and compiled it using g++ 4.8.4 with an O3 option. We executed the program on servers with Intel Xeon CPU E5-2660 v3 @ 2.60GHz and 1GB memory. We measured only the real running time of GA part of the program.

3.3.2 Results and Discussions

Baseline Results

We tested a traditional hybrid genetic algorithm without using an incremental approach, to obtain a baseline result. Table 1 shows the average fitness value and the ratio of finding an optimal solution in percentage.

The baseline algorithm is an improved version of the multi-objective GA proposed by Choi et al. [CYM12]. We modified some of the genetic operators and parameters in order to improve the performance. The operators and parameters we used are explained in Section 3.2.3. When compared to experimental results in [CYM12], the ratio of finding an optimal solution slightly decreased for 2 out of twenty classes, and considerably increased for 8 classes. It did not change for the rest 10 classes.

For more than half of the classes, the GA found an optimal solution at every trial. Only 8 out of 20 classes were relatively difficult, and the results of these classes are marked in bold. Most of the difficult classes were when $|V_1|$ was not too small and not too big. For the later experiments, only these 8 classes will be used.

Effect of Vertex Reordering

Table 2 shows the average fitness function value of hybrid IGA with four different reordering schemes mentioned. The expansion size of each step was 1, which means that a single vertex was added at each step. The schemes are random reordering (RAND), Max-degree reordering (MD), BFS reordering (BFS), and Max-adjacency reordering (MA). The result of

Table 2: Results of vertex reordering

$\lvert V_1 \rvert$	η	BASE	RAND	MD	BFS	MA
10	0.1	0.0005	0.0027	**0.0001**	0.0010	0.0006
10	0.2	0.0566	0.0865	**0.0434**	0.0644	0.0575
30	0.05	0.4995	0.8999	**0.2160**	**0.3992**	**0.2876**
30	0.1	0.7271	1.2843	**0.4205**	0.9276	**0.4124**
30	0.2	0.0499	0.4251	**0.0376**	0.4164	0.1353
50	0.01	0.0088	0.1145	**0.0053**	0.0104	0.0099
50	0.05	0.0050	0.2398	**0.0036**	0.0990	0.0291
70	0.01	0.0079	0.1242	**0.0049**	0.0121	0.0123

the traditional hybrid GA is also shown as the baseline (BASE). We highlighted the results when there was an improvement compared to the baseline. Adding an incremental approach to the hybrid GA with randomized vertex reordering degraded the performance in all of the 8 classes. But this could be overcome by using proper vertex reordering schemes in most of the cases. The incremental approach using the most appropriate scheme, namely max-degree reordering (MD), even showed a better performance than that of the baseline. We therefore use this reordering scheme in the rest of our experiments.

Effect of Partially Random Initialization

At the beginning of each step, we randomly initialized a certain number of chromosomes in the population. We used 0, 10, 20, 30, 40, and 50 for the values. Table 3 shows the ratio of finding an optimal solution in percentage. For most of the cases, random initialization degraded the quality of solutions, and the quality decreased more when there was more randomness. The results suggest that we used an adequate sequence of subproblems. We

Table 3: Results of partially random initialization

$\lvert V_1 \rvert, \eta$	Ratio					
	0	10	20	30	40	50
10, 0.1	99.92%	99.79%	99.91%	99.87%	99.88%	99.79%
10, 0.2	71.54%	71.29%	72.04%	70.60%	69.74%	68.26%
30, 0.05	71.39%	63.87%	54.40%	40.40%	27.79%	21.85%
30, 0.1	91.05%	91.52%	89.53%	85.55%	80.62%	74.18%
30, 0.2	99.67%	99.78%	99.70%	99.45%	99.26%	98.63%
50, 0.01	94.79%	83.61%	70.77%	58.95%	46.81%	35.16%
50, 0.05	99.95%	99.79%	98.20%	95.09%	92.00%	88.56%
70, 0.01	96.14%	83.24%	59.10%	28.41%	16.09%	10.95%

therefore decided to reuse all of the solutions in the population from the previous step.

Effect of Changing Stopping Criterion

Table 4 shows the average fitness function value of the hybrid incremental GAs with different stopping criteria for each step. Each of the step is terminated when the count of optimal solutions in the population reaches a certain threshold. We used 100, 50, and 1 for the threshold value. The stopping criterion is applied to the algorithm with max-degree reordering and the algorithm with random reordering. For each reordering scheme, we also showed the result of an algorithm without stopping criterion, which runs for a fixed number of generations for each step, as the baseline. We marked the best result for each case in bold.

In general, reducing the threshold value gave better results for both schemes. In the case with max-degree reordering, using the lowest threshold value of one gave the best result in only three out of eight classes. However,

Table 4: Results of changing stopping criterion

$\lvert V_1 \rvert$	η	With max-degree reordering				Random reordering			
		Base	100	50	1	Base	100	50	1
10	0.1	0.0001	**0.0000**	**0.0000**	0.0001	0.0027	0.0092	0.0073	**0.0048**
10	0.2	0.0434	0.0364	**0.0319**	0.0336	0.0865	0.7295	0.6649	**0.5550**
30	0.05	0.2160	0.1712	0.1622	**0.1464**	0.8999	7.8436	7.5126	**6.9422**
30	0.1	0.4205	0.3909	0.3875	**0.3385**	1.2843	12.8211	12.5180	**11.6318**
30	0.2	0.0376	0.0380	**0.0350**	0.0362	0.4251	4.8571	**4.5777**	4.8345
50	0.01	0.0053	**0.0038**	0.0044	0.0056	0.1145	0.6808	0.6049	**0.3715**
50	0.05	0.0036	0.0057	0.0051	**0.0048**	0.2398	1.5404	1.3950	**1.3229**
70	0.01	0.0049	0.0053	**0.0050**	0.0050	0.1242	1.2539	1.2585	**1.0344**

the overall results from this threshold value were most similar to the best results. And for the case with random reordering, only one case was exceptional. Since reducing the threshold value increases population diversity, focusing on exploration in intermediate steps seems to be more helpful than focusing on exploitation. Instead of evolving from a population full of local optima, it was better to evolve from a diverse population where only one of the solutions is locally optimal.

Effect of Changing Expansion Size

Table 5 shows the average fitness function value and the average running time of hybrid incremental GAs with different expansion size values. Since there are classes of graphs with $|V_1| = 10$, we used expansion sizes less than or equal to five. We used max-degree reordering and stopping criterion with a threshold value of one. Correlation coefficients between the averages and the expansion size are also calculated and presented. Among the five different cases, the best result for each class is marked in bold.

Generally, increasing the number of vertices to be added at each step was better in terms of fitness function value. However, the values do not have a consistent and strong correlation with the expansion size. By measuring the relative error, we found that the expansion size of 4 is the best. Large expansion size was also good in terms of average running time, and there was a consistent and strong correlation. Changing the expansion size from 1 to 2 had the largest gap of running time, and the gap decreased afterward. Therefore, it seems better to use a large expansion size since increasing the size reduces the running time without losing the solution quality.

Table 5: Results of changing expansion size

$\lvert V_1 \rvert$	η	Average fitness function value						Average running time in seconds					
		1	2	3	4	5	Corr.	1	2	3	4	5	Corr.
10	0.1	**0.0001**	0.0002	0.0002	0.0002	0.0004	0.9514	0.18	0.11	0.12	0.10	0.11	-0.7629
10	0.2	**0.0336**	0.0389	0.0405	0.0438	0.0512	0.9729	0.75	0.78	0.83	0.84	1.02	0.8998
30	0.05	0.1464	**0.1380**	0.1400	0.1393	0.1428	-0.2769	4.11	3.12	2.87	2.99	2.89	-0.7802
30	0.1	0.3385	0.3127	0.2935	0.3055	**0.2686**	-0.9063	4.04	2.65	2.16	2.13	1.77	-0.8991
30	0.2	0.0362	0.0306	0.0296	**0.0237**	0.0263	-0.8917	4.50	3.00	2.36	2.23	1.78	-0.9269
50	0.01	0.0056	**0.0053**	0.0056	0.0059	0.0054	0.1719	8.05	5.06	4.30	3.93	3.43	-0.8962
50	0.05	0.0048	0.0025	0.0013	**0.0007**	0.0023	-0.6760	10.75	5.96	4.64	4.01	3.28	-0.8978
70	0.01	0.0050	**0.0033**	0.0034	0.0035	0.0035	-0.6489	20.79	12.27	10.16	8.88	7.87	-0.8914

3.3.3 Overall Results

We combined the best choices from the previous subsections in building our hybrid IGA. We used max-degree reordering, the stopping criterion with a threshold value of one, and the expansion size value of four. All of the 20 classes of graphs were tested by this algorithm and the result was compared to that of the traditional hybrid GA without the incremental approach.

Table 6 shows the overall results. The average fitness function value, the standard deviation (SD) of the values, the average running time in seconds, and the proportion of runs in which an optimal solution has been found, are shown in the table, respectively. The table also contains *p*-values for each class computed by Welch's *t*-test. The null hypothesis is that the performance of the traditional algorithm is the same as that of the incremental algorithm, where the performance is measured by the average fitness function value. Therefore, the *p*-value roughly denotes the probability that the incremental approach makes no improvement in its performance. Eleven out of twenty classes are marked as NA, which means that both algorithms always found an optimal solution and thus the *p*-value could not be defined. For the eight relatively difficult classes, the performance was significantly improved by the incremental approach. The minimum ratio of finding an optimal solution was dramatically increased from 50.06% to 69.40%. The result shows that using the incremental approach is significantly helpful to improve the performance of a hybrid genetic algorithm. There was one exceptional case when $|V_1| = 50$ and $\eta = 0.1$. The average fitness function

Table 6: Overall results of the proposed hybrid incremental genetic algorithm compared to a traditional hybrid genetic algorithm

$\|V_1\|$	η	Traditional hybrid GA				Hybrid incremental GA				p-value
		f average	f SD	Time	Ratio	f average	f SD	Time	Ratio	
10	0.01	0.0000	0.0000	0.00	100.00%	0.0000	0.0000	0.05	100.00%	NA
	0.05	0.0000	0.0000	0.01	100.00%	0.0000	0.0000	0.07	100.00%	NA
	0.1	0.0005	0.0073	0.10	99.46%	0.0002	0.0047	0.10	99.78%	0.0002
	0.2	0.0566	0.0806	1.10	62.02%	0.0438	0.0729	0.84	69.40%	<0.0001
30	0.01	0.0000	0.0000	0.09	100.00%	0.0000	0.0000	0.61	100.00%	NA
	0.05	0.4995	0.5907	7.58	50.06%	0.1393	0.3263	2.99	80.25%	<0.0001
	0.1	0.7271	1.6799	6.49	84.14%	0.3055	1.1553	2.13	93.43%	<0.0001
	0.2	0.0499	0.7522	2.47	99.56%	0.0237	0.5172	2.23	99.79%	0.0041
50	0.01	0.0088	0.0295	7.44	91.52%	0.0059	0.0236	3.93	94.18%	<0.0001
	0.05	0.0050	0.1898	3.60	99.93%	0.0007	0.0720	4.01	99.99%	0.0342
	0.1	0.0000	0.0000	2.94	100.00%	0.0328	0.8450	5.28	99.85%	0.0001
	0.2	0.0000	0.0000	2.49	100.00%	0.0000	0.0000	8.36	100.00%	NA
70	0.01	0.0079	0.0292	15.80	92.66%	0.0035	0.0243	8.88	97.26%	<0.0001
	0.05	0.0000	0.0000	0.08	100.00%	0.0000	0.0000	7.71	100.00%	NA
	0.1	0.0000	0.0000	0.20	100.00%	0.0000	0.0000	9.37	100.00%	NA
	0.2	0.0000	0.0000	0.93	100.00%	0.0000	0.0000	13.32	100.00%	NA
90	0.01	0.0000	0.0000	4.50	100.00%	0.0000	0.0000	12.40	100.00%	NA
	0.05	0.0000	0.0000	0.03	100.00%	0.0000	0.0000	14.94	100.00%	NA
	0.1	0.0000	0.0000	0.04	100.00%	0.0000	0.0000	18.18	100.00%	NA
	0.2	0.0000	0.0000	0.08	100.00%	0.0000	0.0000	23.93	100.00%	NA

value was increased and the corresponding p-value was 0.0001. This is the case when the traditional hybrid GA could find an optimal solution in all of the 10,000 runs. However, the incremental algorithm has failed in 15 out of 10,000 runs and recorded higher function value.

In an ideal case, the IGA should spend less time than the traditional GA. It solves smaller problems in earlier steps which require less running time. It might spend additional time in expanding the problem and extending the solutions, which are usually not dominating factors. However, the average running time was not decreased by using the incremental approach as shown in Table 6. We observed the running time decrease in only seven classes. For relatively easy classes, the traditional GA found an optimal solution in earlier generations and terminated. However, the incremental GA must run for at least one generation for each step. The increase in running time seemed to be caused by this basic cost of the IGA, which also made the difference in the number of generations executed by the algorithms.

In Table 7, the average number of generations executed by two genetic algorithms is presented. These are the algorithms with threshold (TH) value of one. They are terminated when an optimal solution of zero cost is found. As shown in the table, the incremental GA tends to run for more number of generations for relatively easy classes, and less number of generations for relatively difficult classes. We also tested the algorithms which use an infinite threshold value. These are the algorithms which run for a fixed number of generations. The results are presented in Table 7 as well, and the running time of the IGA was shorter than the traditional GA for cases with $|V_1| \leq 70$. The cases with $|V_1| = 90$ are extremely easy cases and the populations of the

Table 7: Running time analysis of the two GAs

$\lvert V_1 \rvert$	η	Generations (TH= 1)		Time (TH= ∞)	
		GA	IGA	GA	IGA
10	0.01	0.0000	3.0000	0.88	0.10
	0.05	0.1484	3.0073	1.18	0.26
	0.1	5.8462	4.8632	1.44	0.62
	0.2	53.2016	43.5417	2.02	1.32
30	0.01	0.8870	8.0004	5.69	2.58
	0.05	78.1283	51.2670	9.96	6.56
	0.1	41.4299	26.9124	13.50	7.85
	0.2	7.7737	14.7546	16.36	8.12
50	0.01	42.9123	30.7370	17.55	11.63
	0.05	10.2969	22.0814	27.21	21.06
	0.1	4.2882	19.6819	32.23	20.67
	0.2	1.4239	19.2067	43.97	25.21
70	0.01	42.6622	33.6521	38.73	28.82
	0.05	0.0000	18.0239	47.58	45.48
	0.1	0.0002	18.1230	55.59	48.68
	0.2	0.0180	18.5602	77.48	60.66
90	0.01	4.7247	23.0131	69.39	71.13
	0.05	0.0000	23.0000	73.21	90.23
	0.1	0.0000	23.0000	88.88	98.96
	0.2	0.0000	23.0000	117.50	127.50

GAs are converged after a very few number of generations. The overhead of the incremental approach was dominant for these cases. Nonetheless, the results suggest that the IGA is basically an efficient approach.

3.4 Further Improvement

As shown in Table 6, the hybrid incremental approach with appropriate schemes outperformed the previous works. However, for some of the graphs, such as the ones with $\lvert V_1 \rvert = 10$ and $\eta = 0.2$, the success ratio of $\approx 70\%$ is not satisfactory. We noticed that the ones recording relatively bad performance were the ones with sparse G_1. The algorithm has to be further developed to deal with these cases. We devised new operators and tested them.

3.4.1 New Operators

Improved Reordering Scheme

Experimental results show that reordering the vertices in an appropriate order plays a key role in an incremental GA. The best method was to sort the vertices in the decreasing order of degree. However, this does not make much sense for sparse graphs. Lots of the vertices have the same degree and a tie-breaking rule is needed. We modify the reordering scheme; in a case of a tie, the vertex having more adjacency to the previously ordered vertices comes earlier in the ordering. This is a combination of the best and the second best reordering schemes proposed in Section 3.2.3, and we expect a synergy effect. We name this new vertex reordering scheme Max-degree-adjacency (MDA) reordering.

Improved Local Optimization

The algorithms using an evolutionary computation in previous works hybridized a genetic algorithm with a local optimization algorithm [CYM12, KM10]. Two vertices are chosen at random, and they are interchanged if it has a gain in fitness. But for sparse graphs, the local move is mostly likely to produce a solution with the same fitness. This is called plateau phenomenon, and it makes the search algorithm less effective.

Algorithm 3: GDA-like local optimization algorithm

```
Input: A solution $X$ of SIP($G_1$, $G_2$)
1:  $L \leftarrow \{(i,j) \mid 1 \leq i \leq |V_1|,\ i < j \leq |V_2|\}$
2:  $Level \leftarrow C(X)$
3:  repeat
4:      $flag \leftarrow false$
5:      for all $(i,j) \in L$ in random order do
6:          $swap\ (X[i], X[j])$
7:          evaluate the new cost $C(X)$
8:          if $C(X) < Level$ then
9:              $flag \leftarrow true$
10:             $Level \leftarrow Level - \Delta$
11:         else
12:             $swap\ (X[i], X[j])$ // cancel
13:         end if
14:     end for
15: until flag
```

To escape from a plateau in a search space, we use a variant which permits some local moves that do not improve the fitness. We adopt an idea from the great deluge algorithm (GDA), which was reported to be effective [Due93]. Any local move that makes the new fitness be above a certain *level* is allowed, and the *level* is increased over time. The local optimization algorithm we used is described in Algorithm 3. Note that $C(X)$ denotes the error correction cost of a solution X, but not the fitness of the solution.

For Δ, the amount of change in *level*, we use the cost of the initial solution divided by $0.1 \times |L|$. Here, $|L|$ denotes the number of neighbors of a solution. This means that the number of the local moves made is at most $0.1 \times |L|$. The running time is increased when Δ is smaller, and the quality of the solution is degraded when Δ is bigger. The value of Δ is set to make a good balance between the running time and the solution quality.

Table 8: Results of improved operators

$\lvert V_1 \rvert$	η	Incremental GA		Local optimization	
		MD	MDA	Prev. one	New one
10	0.1	**0.0001**	**0.0001**	0.3179	**0.2531**
10	0.2	**0.0434**	0.0461	0.7216	**0.6152**
30	0.05	0.2160	**0.1670**	2.8218	**2.6332**
30	0.1	**0.4205**	0.4484	6.8694	**6.5598**
30	0.2	0.0376	**0.0297**	14.7564	**14.6842**
50	0.01	0.0053	**0.0021**	1.4857	**1.4143**
50	0.05	**0.0036**	0.0069	11.1004	**10.7590**
70	0.01	0.0049	**0.0022**	3.6902	**3.4166**

3.4.2 Improvements by New Operators

We first compare the new operators with the previous ones. The proposed vertex reordering scheme and local optimization algorithm are tested on the same graph instances from the previous subsection. We mainly used the eight relatively difficult classes in our experiments. We compare the new reordering scheme, Max-degree-adjacency reordering (MDA), with the Max-degree reordering (MD). The details of the incremental GA framework is the same as the ones in Section 3.2.3, for fair comparison. The new local optimization algorithm is compared to the previous vertex-swap local optimization algorithm used in [CYM12, KM10] and Section 3.2.3. We conducted 100 runs for each of the algorithms and averaged the results. Table 8 shows the average cost function values, and the better ones are marked in bold. We compare the two IGAs using different vertex reordering methods, and compare the old and the new local optimization algorithms.

For five out of eight classes, the performance of MDA was better than or equal to MD. This result suggests that the combination of the two vertex reordering schemes could bring a synergy effect. In addition, the new local optimization was better than the previous one for all of the classes. Both of the two proposed operators turned out to be effective. We used both of them in the following experiments.

3.4.3 Overall Result

As both of the new operators showed performance improvement, we combined them to build an improved hybrid IGA. Since two core parts of the algorithm have been modified, we also tried to change other parameters. For the stopping criterion, the threshold value of one was still the best. However, for the expansion size, we found that the algorithm using a smaller value produces the better results. We tested the expansion sizes ranging from one to five again, and found that the value of one was the best. Table 9 shows the result of the improved hybrid IGAs with five different expansion size values. With expansion size of one, the success ratio was maximized for seven out of eight classes. All of them recorded over 96%, and compared to the result in Table 6, they were better than the previous hybrid IGA. The average fitness function values were also smaller. In particular, for graphs with $|V_1| = 10$ and $\eta = 0.2$, the success ratio was increased by more than 28%, from 69.40% to 97.93%.

The hybrid IGA with expansion size one was tested on all of the 20 classes of graphs. The overall results are presented in Table 10. The average fitness function value, the standard deviation (SD) of the values, the average running time in seconds, and the proportion of runs in which an optimal solution has been found, are shown in the table, respectively. The results were

Table 9: Results of the improved hybrid incremental genetic algorithm with various expansion size

$\|V_1\|$	η	Average fitness function value					Average running time in seconds				
		1	2	3	4	5	1	2	3	4	5
10	0.1	0.0000	0.0000	0.0000	0.0000	0.0000	100.00%	100.00%	100.00%	100.00%	100.00%
10	0.2	0.0022	0.0046	0.0040	0.0029	0.0040	97.93%	95.75%	96.29%	97.21%	96.36%
30	0.05	0.0103	0.1148	0.0634	0.0506	0.0258	96.22%	77.42%	85.85%	87.09%	91.58%
30	0.1	0.0018	0.0032	0.0018	0.0009	0.0018	99.96%	99.93%	99.96%	99.98%	99.96%
30	0.2	0.0000	0.0000	0.0000	0.0000	0.0000	100.00%	100.00%	100.00%	100.00%	100.00%
50	0.01	0.0005	0.0357	0.0220	0.0144	0.0054	99.46%	70.50%	79.95%	86.18%	94.69%
50	0.05	0.0000	0.0000	0.0000	0.0000	0.0000	100.00%	100.00%	100.00%	100.00%	100.00%
70	0.01	0.0000	0.0217	0.0090	0.0034	0.0004	100.00%	85.85%	93.07%	97.00%	99.57%

Table 10: Overall results of the improved hybrid incremental genetic algorithm

$\|V_1\|$	η	f average	f SD	Time	Ratio	p-value
10	0.01	0.0000	0.0000	0.05	100.00%	NA
	0.05	0.0000	0.0000	0.07	100.00%	NA
	0.1	0.0000	0.0000	0.20	100.00%	<0.0001
	0.2	0.0022	0.0153	0.83	97.93%	<0.0001
30	0.01	0.0000	0.0000	1.34	100.00%	NA
	0.05	0.0103	0.0592	5.35	96.22%	<0.0001
	0.1	0.0018	0.0875	5.19	99.96%	<0.0001
	0.2	0.0000	0.0000	4.63	100.00%	<0.0001
50	0.01	0.0005	0.0073	11.78	99.46%	<0.0001
	0.05	0.0000	0.0000	15.34	100.00%	0.3173
	0.1	0.0000	0.0000	12.13	100.00%	0.0001
	0.2	0.0000	0.0000	13.97	100.00%	NA
70	0.01	0.0000	0.0000	39.16	100.00%	<0.0001
	0.05	0.0000	0.0000	35.24	100.00%	NA
	0.1	0.0000	0.0000	29.79	100.00%	NA
	0.2	0.0000	0.0000	35.35	100.00%	NA
90	0.01	0.0000	0.0000	114.82	100.00%	NA
	0.05	0.0000	0.0000	59.39	100.00%	NA
	0.1	0.0000	0.0000	60.77	100.00%	NA
	0.2	0.0000	0.0000	79.01	100.00%	NA

compared to previous hybrid IGA in Section 3.2.3 and p-values computed by Welch's t-test are shown in the table. The hybrid IGA combined with carefully designed operators and appropriate process showed an almost perfect result. The new IGA was effective for all of the cases and there was a statistically significant improvement.

Chapter 4

Graph Cut Optimization Problems

4.1 Introduction

Given an undirected graph $G = (V, E)$, we want to divide the vertices into groups. Each group is called a *partition*, and the number of the groups is usually set to be two. An edge connecting vertices from two different partitions is called a *cut edge*, and a *cut size* of the partitioning is the number of cut edges. The graph cut optimization problem is to find a partitioning that either minimizes or maximizes the cut size.

The minimization version of the problem is called the graph partitioning problem (GPP) [KHKM11]. There is an additional constraint in this problem; the sizes of the partitions have to be balanced and should not differ by more than one. The problem becomes easier without this constraint and is solvable by a maximum flow algorithm [CSRL01]. GPP has a number of applications in parallel computing, image processing, sparse matrix factorization, and VLSI design [BMS$^+$13, KHKM11]. Both exact algorithms [DFG$^+$15] and metaheuristics [CBM07] are used to solve this NP-hard problem [GJ90]. Various genetic algorithms have been proposed as well [BM96, HKY15, KHKM11, KM04]. Vertex reordering schemes [BM94], normalization techniques [Las91], and hybridizations with local optimization [vLM91] are relevant issues of this problem.

There is no constraint on the partition size for the maximization problem, and the problem is called the maximum cut problem (MCP). Though this problem looks similar to the GPP, both of them have a distinctive characteristic. The MCP is also an NP-hard problem [GJ90], and the application domain includes VLSI design and statistical physics [SL12]. A Tabu-search based metaheuristic [YHM14], a genetic algorithm based approach [SL12], and a hybrid [WWL15] have been proposed.

4.2 The Proposed Algorithm

4.2.1 Subproblem Structure

The fitness function of the graph cut optimization problem is defined in terms of the cut size. Since the size of a cut is defined to be the number of edges satisfying a constraint, it is natural to construct the subproblem by removing the edges from the original graph, or in other words, by gradually adding the edges to an empty graph. However, the graph cut optimization problem has more complicated subproblem structure compared to the subgraph isomorphism problem, and only a few particular sequences have the optimal substructure.

We will explain the details with the GPP, the minimization version of the problem. Let x^* be an optimal solution of the original problem, and $E_{cut}(x^*)$ be the set of the cut edges formed by the solution x^*. Then, the following propositions hold.

Proposition 1. *If some of the cut edges in $E_{cut}(x^*)$ are removed from the graph, then x^* is still an optimal solution for the corresponding subproblem.*

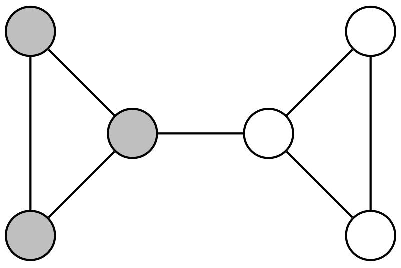
Proof. Suppose that there exists a better solution x' with less cut size. Then, by adding up the number of the removed edges to the cut size of x' with respect to the subproblem, we obtain a value that is less than $|E_{cut}(x^*)|$. This contradicts the assumption that x^* is an optimal solution. □

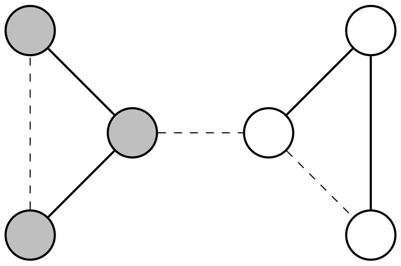
Proposition 2. *If all of the cut edges in $E_{cut}(x^*)$, and other non-cut edges are removed from the graph, then x^* is still an optimal solution for the corresponding subproblem.*

Proof. For this subproblem, x^* is a solution with cut size zero, as all of the cut edges are removed. Since only non-cut edges are left in the graph, x^* will stay as an optimal solution with cut size zero. □

Therefore, from the above propositions, it follows that there exists a sequence of subproblems having an optimal substructure. The sequence is obtained by adding the non-cut edges before the cut edges. Note that as the MCP is opposite to the GPP, a sequence of subproblems having an optimal substructure for the MCP could be obtained by adding the cut edges before the non-cut edges.

Figure 5 shows an example of an original problem, and two examples of subproblems. In figure 5(a), the original graph G and the optimal solution with one cut edge are shown. The vertices with the same color constitute each partition. In figure 5(b), the cut edge and two non-cut edges are removed from G. The removed edges are drawn with dashed lines. As shown in Proposition 1 and 2, we can see that the optimal solution is the same, and it also has cut size zero. However, in figure 5(c), the subproblem has a different optimal solution of cut size zero. In this case, only four non-cut edges


(a) A sample graph G


(b) A spanning graph of G which has the same optimial solution as that of G

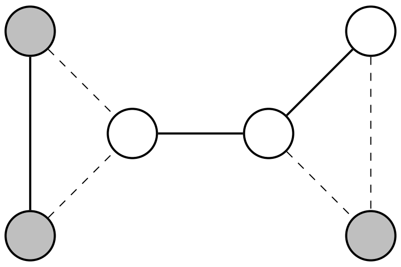

(c) A spanning graph of G which has a different optimial solution from that of G

Figure 5: The original problem and two subproblems of graph partitioning problem

are removed from the original graph. The above solution for G has one cut edge, and thus it is not even one of the optimal solutions. If an IGA faces this subproblem in the intermediate step, then the optimal solutions may not be useful to find the optimal solution for the original problem.

The problem is that we cannot know the optimal solution prior to actually solving it. Even if we know the cut edges formed by the optimal solution, the order of the non-cut edges, the order of the cut edges, and the number of edges to be added at each step have to be determined as well. Therefore, to find a good sequence of subproblems, we have to investigate a heuristic method which indirectly reflects the optimal substructure. This issue is discussed in detail in the next subsection.

4.2.2 Reordering Schemes

We also use three different ordering schemes to find a good sequence of subproblems. As discussed in Section 2.2.3, the order of the edges is the key part of an IGA. In an ideal ordering of the edges, the non-cut edges of the optimal solution are needed to be placed earlier for the optimal substructure, and cut edges could be placed in between non-cut edges to reduce the difficulty of the space search. The three schemes are chosen to approximately satisfy this property. The details of each scheme and rationales behind them are as follows.

- **Randomized ordering**: The entire edges are randomly ordered for an E-IGA, and the vertices are considered in a randomized order for a V-IGA and an M-IGA. It provides a baseline result.

- **Degree based ordering**: We sorted the edges by the degree of their incident vertices for an E-IGA, and sorted the vertices by their degree for a V-IGA and an M-IGA. We use a decreasing order and in a case of a tie, we randomly choose one of the objects. The vertex with a high degree value is likely to have more non-cut edges as its incident edges than the one with a low degree value. Also, placing such vertex in an earlier step may help to draw up an outline of the search space landscape, as it complicates the search space when it is added to the graph.

- **BFS ordering**: We randomly choose one vertex and run a breadth-first-search (BFS) starting from that vertex. The adjacent vertices are added to the queue in random order. For an E-IGA, we iterate the vertices in BFS order and list the incident edges. And for a V-IGA and an M-IGA, the vertices are considered in BFS order. We choose this scheme to utilize the cluster inside the graph [BM96, HKM06, HKY15]. If the vertices and edges in a cluster are added to the graph almost consecutively, those vertices are likely to be assigned to the same partition. Note that edges inside a cluster are non-cut edges, and they are likely to be added earlier when this scheme is used.

4.2.3 Genetic Framework

As the algorithm for GPP and MCP are similar, we describe the algorithm only for the GPP. We use a typical GA with and without the incremental approach in our experiment. Note that the proposed incremental approach could be applied in combination with any other operators and techniques, such as gene reordering [BM94], normalization [Las91], greedy repairing schemes [BM96, HKM06], and local optimization algorithms [vLM91]. We decide not to combine them with the incremental algorithm in our experiment, to thoroughly observe the effect of using the incremental approach. Instead, we use the common operators that have been widely used [KHKM11]. The operators and parameters are as follows.

- **Population management**: The size of the population is 100. Twenty new offspring are generated in each generation. The best 100 out of 120 chromosomes survive. The population is randomly initialized in the first step of the IGA, and is reused in the rest of the steps. As we keep the same set of vertices and add the edges only, we reuse the chromosomes without extending them.

- **Representation**: We use a binary representation. If a chromosome has a different number of 0s and 1s, we randomly repair it.

- **Selection**: We randomly select eight chromosomes and run a tournament. The better chromosome wins the match with probability 80%. The best two chromosomes are finally selected.

- **Crossover and Mutation**: We use a uniform crossover and a random mutation. Based on Hamming distance, we first normalize the chromosomes before crossover. Each gene is inherited from one of the two parents with equal probability, and is toggled afterward with 0.5% of a chance.

- **Stopping criterion**: For a fair comparison, we use a fixed number of generations. We use 10^5 for a traditional GA, and it is evenly distributed to each step for an IGA.

4.3 Experimental Results

4.3.1 Dataset and Evaluation

To test an IGA for the GPP, we use random graphs (G*n.d* graphs) and random geometric graphs (U*n.d* graphs) [JAMS89], which have been widely used in literature [BM96, CBM07, DFG+15, HKY15, KM04]. They were randomly generated to have *n* vertices and to have an average vertex degree of *d*. The value of *n* is either 500 or 1000. For the values of *d*, 2.5, 5, 10, and 20 are used for G*n.d* graphs, and 5, 10, 20, and 40 are used for U*n.d* graphs. Note that *d* is represented in percent. As one may infer from the name of the graph, U*n.d* graphs have a geometric property; the vertices are laid on a 2D Euclidean plane and two vertices are connected by an edge if the Euclidean distance between them is under a certain threshold value. And to test an IGA for the MCP, we use the graphs so-called G-set [HR00], which have been widely used in literature [WWL15, YHM14]. These are a variety of randomly generated graphs, and some of them have weights on the edges. As we are considering a problem regarding the cut size, we only use the ones without edge weights. And we could use the same incremental framework for both of the problems if such graphs are selected. Eighteen graphs are used in total, and they are G1 to G5, G14 to G17, G43 to G47, and G51 to G54.

Each of the algorithms was performed 1000 times for each of the graph instances, and the running environment was the same as the ones in previous Chapter 3. The best cut sizes found by the algorithms in each run are averaged. To compare two algorithms, we have calculated the *p*-value of Welch's *t*-test.

4.3.2 Results on Graph Partitioning Problem

Performance of Incremental Genetic Algorithm

We implemented a traditional GA, and the result can be considered as a baseline. There are nine combinations out of three graph expansion methods and three reordering schemes; we tested all of them. Table 11 shows the average cut size found by GA and IGAs. The value is marked in bold if it is better than the traditional GA. Among the results of nine algorithms, the minimum cut size for each graph is parenthesized by a square bracket.

Among the three graph expansion methods, the worst one is the edge-wise expansion which results in relatively high cut size. An E-IGA adds one edge to the graph at each step, and a required number of steps is equal to the number of edges in the graph. As the number of steps is large compared to other two kinds of IGAs, E-IGAs run for a relatively small number of generations in each step.

On the other hand, the overall best method is mixed expansion method used in M-IGAs. It shows performance improvement compared to the result of traditional GA, for almost all cases except three ones. Moreover, M-IGAs recorded the best results for 15 instances out of 16. The V-IGA, which also

Table 11: Performance of the tested GA and IGAs for the GPP

Graph	GA	E-IGA			V-IGA			M-IGA		
		Rand.	Deg.	BFS	Rand.	Deg.	BFS	Rand.	Deg.	BFS
G500.2.5	80.59	83.09	**77.27**	**72.76**	**76.61**	**74.14**	**73.09**	**77.52**	**73.87**	[**71.04**]
G500.05	267.92	272.79	**264.49**	**262.52**	**264.90**	270.52	**260.46**	**264.28**	**259.08**	[**258.94**]
G500.10	690.47	695.22	**685.59**	**684.99**	692.06	**690.24**	**685.34**	**683.59**	[**678.32**]	**680.38**
G500.20	1836.88	1839.58	**1829.66**	**1828.83**	1837.91	1840.99	**1832.27**	**1822.14**	[**1817.65**]	**1819.87**
G1000.2.5	167.85	185.10	169.46	**145.15**	170.42	**145.13**	[**140.55**]	**165.51**	**155.14**	**141.47**
G1000.05	552.74	584.30	565.58	**550.55**	565.22	562.16	**545.12**	**551.89**	**539.88**	[**535.18**]
G1000.10	1516.47	1566.87	1540.64	1529.95	1560.67	1559.07	1533.24	**1516.45**	[**1502.49**]	**1504.77**
G1000.20	3597.18	3683.09	3643.41	3637.21	3680.18	3682.27	3653.99	3606.00	[**3591.97**]	**3596.96**
U500.05	60.05	**58.35**	**37.00**	**13.88**	**49.23**	**35.85**	**13.64**	**55.12**	**33.95**	[**12.82**]
U500.10	151.15	**145.39**	**116.33**	**63.37**	**134.58**	**119.20**	**67.84**	**139.97**	**108.55**	[**61.87**]
U500.20	350.70	**333.83**	**279.18**	**241.81**	**322.13**	**293.83**	**243.14**	**332.88**	**256.95**	[**238.71**]
U500.40	673.18	**655.60**	**653.96**	**530.77**	**593.89**	**525.69**	**538.39**	**660.84**	**548.98**	[**523.08**]
U1000.05	132.12	150.98	**111.03**	**38.02**	133.90	**89.44**	**46.17**	**128.24**	**82.76**	[**28.31**]
U1000.10	323.25	358.10	**257.07**	**105.79**	334.14	**245.65**	**134.56**	**321.38**	**211.90**	[**101.64**]
U1000.20	727.88	797.12	**600.04**	**376.02**	764.55	**702.54**	**402.38**	762.64	**524.85**	[**372.45**]
U1000.40	1442.98	1523.60	**1439.13**	**1054.63**	1574.49	1583.88	**1092.58**	1522.31	**1380.04**	[**1053.74**]

adds multiple edges in expanding the graph, is not as effective as an M-IGA. The main difference between them is that an M-IGA adds a number of edges in early steps, and adds a few edges in later steps. Adding more edges to the graph can be considered as adding more constraints on the solution, and it reduces the number of optimal solutions. As a result, the number of optimal solutions for the first few subproblems is small, and it has an effect of pruning the search space in early steps. The result suggests that solving a subproblem with adequate complexity in early steps is an essential part in a successful space search.

Using a proper ordering scheme is another crucial factor. Among the three vertex ordering schemes proposed, the best one was the BFS ordering while the worst one was the randomized ordering. Most of the best IGAs for each graph instances use BFS ordering. In particular, there were prominent performance improvements for U*n*.*d* graphs. These graphs are known to contain a cluster which heavily interrupts efficient space search [HKY15]. The edges inside a cluster are added with small time difference, and an M-IGA effectively assigns them in the same partition. For G*n*.*d* graphs, both of degree based ordering and BFS ordering brought similar performance improvement, because these graphs have less cluster-related properties than U*n*.*d* graphs.

We also measured the running time of the algorithms in seconds as shown in Table 12. The IGAs were faster than a GA, and the overhead of an incremental process was not critical for these algorithms. Among the three graph expansion methods, the fastest and slowest ones were V-IGAs and E-IGAs, respectively. And the fastest and slowest reordering schemes were

Table 12: Running time of the tested GA and IGAs for the GPP

Graph	GA	E-IGA			V-IGA			M-IGA		
		Rand.	Deg.	BFS	Rand.	Deg.	BFS	Rand.	Deg.	BFS
G500.2.5	19.39	17.85	17.89	17.66	17.50	17.99	17.84	18.21	18.58	18.04
G500.05	22.05	19.35	19.31	18.94	18.33	18.90	18.64	19.89	20.49	19.73
G500.10	28.04	23.06	22.78	22.23	19.98	20.73	20.23	24.03	25.08	23.86
G500.20	53.52	37.78	37.10	35.60	25.79	27.02	26.03	40.87	43.59	40.74
G1000.2.5	39.09	35.60	35.71	35.24	34.65	35.85	35.40	36.52	37.38	36.09
G1000.05	46.76	40.35	40.39	39.48	37.05	38.54	37.80	42.04	43.70	41.61
G1000.10	69.04	55.36	54.92	53.68	43.26	45.35	43.91	58.26	61.66	58.06
G1000.20	124.41	100.28	99.35	97.08	58.01	60.34	58.62	97.85	103.91	97.82
U500.05	21.47	18.85	18.64	18.45	18.19	18.99	18.53	19.32	19.28	18.44
U500.10	25.77	21.26	20.65	20.09	19.57	20.55	19.99	21.76	21.56	19.98
U500.20	34.17	26.90	25.52	23.85	22.30	24.16	23.02	26.79	25.97	23.31
U500.40	49.62	40.62	37.43	33.63	27.19	29.33	28.65	36.26	34.46	30.15
U1000.05	42.45	37.84	37.12	36.37	35.96	37.37	36.61	38.50	38.38	36.26
U1000.10	52.17	44.66	42.25	40.32	39.21	40.96	39.68	44.41	43.21	39.44
U1000.20	71.73	62.33	55.66	50.58	45.78	49.23	46.01	57.08	53.34	46.40
U1000.40	108.40	108.57	95.42	80.54	57.93	64.73	58.80	81.76	76.56	62.10

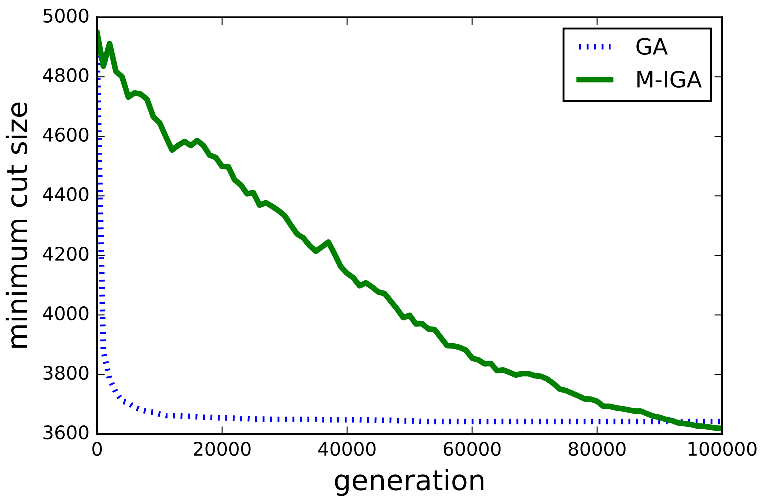
BFS based one and degree based one, respectively. When compared to the traditional GA, relative running time and relative performance of the M-IGA was 78.94% and 71.74%, respectively.

Prevention of Premature Convergence

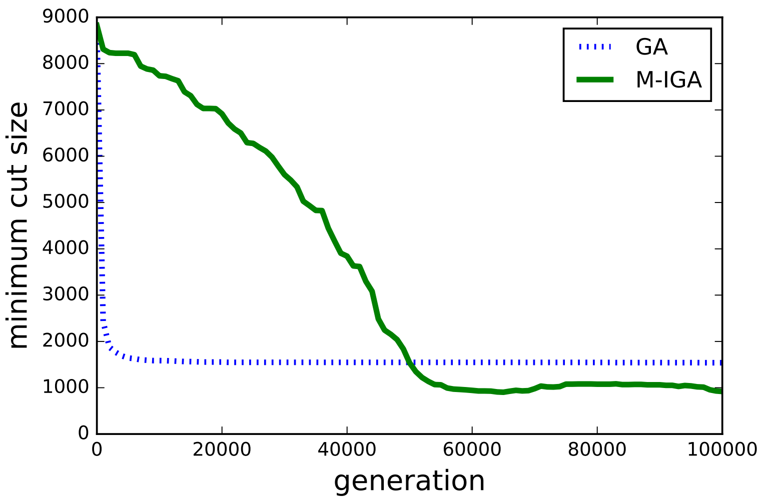
Experimental result showed that combining an M-IGA with a BFS ordering was the best choice. The M-IGA narrows the search space in early steps, and BFS ordering effectively separates cut edges from non-cut edges. Moreover, the M-IGA has an ability to prevent premature convergence and escape from bad local optima. Figure 6 shows the fitness of the best solution found by the GA and the M-IGA in a single representative run. Figure 6(a) is the result on the largest G*n.d* graph, G1000.20, and Figure 6(b) is the result on the largest U*n.d* graph, U1000.40. Note that for the M-IGA, we calculate the cut size with respect to the original graph.

As shown in the figure, the GA converges to a bad local optimum after few tens of thousands of generations. In contrast, the cut size found by an M-IGA decreases slowly and consistently, and it is even increased at a few points. In intermediate steps of the M-IGA, only part of the edges are used to define the subproblem and it makes it easier to escape from bad local optima. This is a key ability of the proposed incremental approach, and these figures provide the evidence.

We also ran a traditional GA, and an M-IGA, which showed the best result, for more number of generations. We first calculated the relative cut size $|E_{cut}(x_{new})|/|E_{cut}(x_{prev})|$ for each graph instance and averaged them over 1,000 runs. We then averaged the values over 16 graph instances. Table 13



(a) G1000.20



(b) U1000.40

Figure 6: The minimum cut size found by the algorithms at each generation for two large graph instances

Table 13: Relative performance of a GA and an M-IGA with BFS ordering which runs for more number of generaions

#Generations	GA	M-IGA (BFS)
2×10^5	0.9923	0.9616
3×10^5	0.9889	0.9524
4×10^5	0.9870	0.9452
5×10^5	0.9879	0.9408

shows the result. In the result of the M-IGA, the improvement of cut size was larger than in the result of GA. If more number of generations is given, that is, more running time is available, an M-IGA with BFS ordering tends to find better solutions, whereas the traditional GA tends to find similar solutions. This also suggests that using a suitable incremental approach helps GA not to converge to, and not to stay in a bad local optimum.

Changing Number of Generations

One drawback of the proposed IGA is that the number of generations is evenly distributed to each step. IGA might waste time in early steps, when the subproblems are trivial and easy to solve. Spending more time in the later part of the process will help solve corresponding challenging and large subproblems. We tested two IGAs using a strategy to put more generations on later steps. One of them stops the intermediate GA when an optimal solution of cut size zero is found. The remaining generations are evenly distributed to the remaining steps. Another strategy is to linearly increase the number of generations of each step. We use a parameter m, which denotes the ratio of the number of generations of the last S^{th} step to the number of

Table 14: Performance of an M-IGA using a BFS ordering and a strategy to dynamically change the number of generations

Graph	Early break		Linear ($m = 5$)	
	Result	p-value	Result	p-value
G500.2.5	**70.89**	0.4653	71.12	0.7088
G500.05	**258.32**	0.0718	**258.20**	**0.0314**
G500.10	**680.13**	0.6079	**679.21**	**0.0136**
G500.20	1820.99	0.1249	**1819.03**	0.2345
G1000.2.5	**140.68**	**0.0070**	**140.69**	**0.0074**
G1000.05	535.59	0.4312	**533.56**	**0.0016**
G1000.10	**1503.92**	0.2741	**1496.40**	<**0.0001**
G1000.20	3598.07	0.3278	**3584.50**	<**0.0001**
U500.05	13.55	**0.0012**	13.08	0.2252
U500.10	**60.66**	0.1265	**61.50**	0.6317
U500.20	**238.39**	0.8668	**238.65**	0.9773
U500.40	**522.89**	0.9594	524.79	0.6578
U1000.05	36.69	<**0.0001**	**25.18**	<**0.0001**
U1000.10	**99.67**	0.1400	**100.51**	0.3753
U1000.20	**367.39**	0.1064	**368.14**	0.1692
U1000.40	**1041.43**	0.1013	**1029.10**	**0.0005**

generations of the 1st step. We applied the two strategies to an M-IGA with BFS ordering. We used $m = 5$ in our experiment.

Table 14 shows the average cut size and the p-value. The p-values in the table are that of a comparison to the M-IGA without the strategy. We call the two strategies ‘Early break’ and ‘Linear’, respectively. The results are compared to the result of the M-IGA with BFS ordering in Table 11. We marked the cut size in bold when it is decreased, and marked the p-value in bold when it is less than 0.05.

In overall, the performance of the ‘Early break’ strategy was not significantly different from the original M-IGA with BFS ordering, but slightly improved for most of the cases. When this strategy is used, the evolutionary process is immediately terminated even when only one optimal solution of the subproblem is found. This degrades the reusability of the populations in the next step. On the other hand, the performance of the 'Linear' strategy was significantly improved for most of the G*n.d* graphs. This strategy not only spends more time on solving harder subproblems, but also provides sufficient amount of time to find diverse optimal solutions for easier subproblems. It was particularly effective for G*n.d* graphs.

4.3.3 Results on Maximum Cut Problem

We tested the algorithms with the MCP as well. As this problem is to maximize the cut size, the role of the cut edges and non-cut edges is interchanged. Starting from an empty graph, we first have to add the non-cut edges to the graph and then add the cut edges in order to obtain a sequence of subproblems having the optimal substructure. As both of the roles of the edges and the fitness function are just reversed, we still can use the same ordering schemes. Therefore, we used the same genetic framework as for the GPP in our experiment, except the repairing scheme which is not necessary for the MCP.

Table 15 shows the average cut size found by the algorithms. The average cut size is marked in bold if it is greater than the result of the traditional GA. And for each of the graph instances, we parenthesized the value by a square bracket if it is the best result.

Unlike the result on GPP, E-IGAs and M-IGAs were not effective for almost all of the cases regardless of the ordering scheme used. It was reported that falling in a bad local optimum is critical for the MCP, and most

Table 15: Performance of the tested GA and IGAs for the MCP

G	GA	E-IGA			V-IGA			M-IGA		
		Random	Degree	BFS	Random	Degree	BFS	Random	Degree	BFS
1	11411.00	11373.20	11290.10	11283.90	11404.40	[**11411.90**]	11408.90	11366.40	11400.60	11384.70
2	11416.90	11379.00	11249.30	11290.20	11409.60	[**11417.10**]	11415.10	11371.50	11392.20	11389.20
3	11412.10	11375.60	11325.10	11286.50	11404.90	11410.10	11409.50	11365.70	[**11435.30**]	11385.30
4	11425.70	11386.00	11271.50	11299.00	11423.20	[**11442.40**]	**11425.80**	11380.10	11413.90	11398.50
5	11417.00	11380.40	11258.10	11291.00	11414.40	[**11450.10**]	**11418.70**	11374.50	11379.60	11391.20
14	2983.84	2964.97	2964.66	2962.03	**2989.07**	**2991.89**	[**2998.08**]	2971.87	2978.91	2982.76
15	2964.75	2946.12	2944.23	2944.41	**2970.86**	**2973.65**	[**2979.85**]	2953.22	2954.62	2961.28
16	2969.65	2951.15	2946.89	2944.18	**2974.30**	**2981.32**	[**2983.94**]	2957.30	2961.99	2965.21
17	2965.15	2946.83	2941.05	2943.74	**2971.31**	**2973.53**	[**2980.29**]	2954.07	2958.31	2962.56
43	6466.47	6410.77	6379.89	6375.71	6453.73	[**6474.56**]	6465.18	6423.30	6435.21	6443.29
44	6463.19	6408.30	6379.05	6370.58	6450.29	[**6470.49**]	6461.40	6418.64	6457.53	6439.86
45	6462.90	6406.78	6376.12	6370.45	6448.72	[**6469.92**]	6461.25	6419.64	6435.69	6440.85
46	6465.33	6410.04	6379.09	6372.30	6451.85	[**6475.64**]	6462.22	6422.36	6441.94	6443.28
47	6470.45	6414.29	6376.46	6376.91	6457.16	[**6475.07**]	6467.70	6426.59	6430.64	6448.11
51	3742.54	3710.64	3713.08	3698.30	**3743.31**	**3750.27**	[**3757.44**]	3720.95	3717.56	3727.72
52	3746.41	3715.03	3715.55	3706.69	**3747.10**	**3749.51**	[**3761.92**]	3725.74	3730.79	3737.34
53	3745.21	3712.13	3716.28	3704.69	**3745.72**	**3748.51**	[**3758.96**]	3723.85	3729.02	3733.04
54	3745.05	3712.70	3710.18	3707.08	3744.94	**3745.10**	[**3758.62**]	3724.93	3736.54	3735.34

of the near state-of-the-art algorithms have a routine to avoid such situation [WWL15, YHM14]. Expanding the graph in an edge-wise way and in a mixed way as proposed seem to reinforce this situation.

Moreover, there was no universally notable ordering scheme, and the characteristic of the graph decides which scheme works best on that graph. G1 to G5, and G43 to G47 are random graphs, and G14 to G17 and G51 to G54 are random planar graphs [HR00]. Basically, the vertex degree was a key factor for random graphs, as degree based ordering showed the best result. For the random planar graphs having geometric property, the BFS ordering scheme seems to capture this property as it did on U*n.d* graphs of the GPP. When used with an appropriate ordering scheme, the incremental approach was effective for the MCP as well.

Table 16 shows the running time of the algorithms, which is measured in second. The average cut size is marked in bold if it is greater than the result of the traditional GA. The fastest and slowest ones were the same as the result in Table 12. V-IGAs were the fastest ones, and their performance was the best as well. Among the nine IGAs, the fasted one was the V-IGA with randomized reordering. The relative running time was 44.30% of that of a GA. However, the performance of this algorithm was worse than a GA for about half of the graph instances. The better ones, V-IGAs with degree based reordering and BFS based reordering, recorded 50.03% and 46.81% relative running time, respectively. The IGA turned out to be effective in terms of the cut size, and efficient in terms of the running time.

Note that for the MCP, only the vertex-wise expansion has brought a performance improvement. If the edges are added to the subgraph by fol-

Table 16: Running time of the tested GA and IGAs for the MCP

G	GA	E-IGA			V-IGA			M-IGA		
		Random	Degree	BFS	Random	Degree	BFS	Random	Degree	BFS
1	227.18	210.76	164.79	115.31	86.77	95.38	88.40	111.22	119.23	108.46
2	227.21	210.79	166.50	115.42	86.73	95.79	88.41	111.11	121.19	108.53
3	227.23	210.62	164.88	115.38	86.71	95.27	88.48	111.00	119.06	108.54
4	226.76	210.58	166.25	115.25	86.63	95.76	88.37	111.14	119.86	108.51
5	226.85	210.63	164.14	115.42	86.62	95.12	88.44	111.11	119.15	108.57
14	62.87	43.55	42.04	39.03	34.22	40.08	38.15	43.41	49.80	46.44
15	62.49	43.32	41.76	38.59	34.12	39.92	38.04	43.19	49.04	46.20
16	62.74	43.42	42.12	38.85	34.15	40.08	38.12	43.22	49.31	46.24
17	62.63	43.37	41.64	38.62	34.14	39.98	38.15	43.15	48.43	46.16
43	120.73	90.85	86.73	68.81	56.25	62.86	58.33	78.57	86.40	75.90
44	121.15	90.85	86.20	68.77	56.24	62.80	58.29	78.58	85.28	75.85
45	120.91	90.84	86.31	68.75	56.21	62.46	58.24	78.70	84.91	75.79
46	120.77	90.80	85.46	68.71	56.20	62.37	58.14	78.62	85.24	75.68
47	120.93	90.80	86.55	68.74	56.23	62.53	58.18	78.65	85.62	75.81
51	82.89	57.95	56.52	51.12	44.45	52.86	50.39	56.93	64.30	61.46
52	83.00	58.06	56.01	51.79	44.40	52.83	50.28	57.06	64.42	61.54
53	82.89	58.02	56.15	51.59	44.43	52.81	50.35	56.96	65.58	61.64
54	83.02	58.05	55.39	51.46	44.45	52.97	50.38	57.05	65.72	61.57

lowing this expansion method, the process is similar to adding a vertex to the subgraph at each step. Moreover, there is no balance constraint on the size of the partition for the MCP. With a help of these two properties, we can build a subgraph only containing the vertices that have been selected, and solve the subproblems defined on subgraphs. As the number of the vertices of a subproblem in an intermediate step is less than before, the incremental algorithm could be easily hybridized with a local optimization algorithm.

4.3.4 Results on Problem Variants

The problem we considered in this chapter is to divide the vertices of an unweighted graph into two partitions. This is the most common version of the problem, which has been studied most widely [KHKM11]. The general version of the problem is to divide the vertices into multiple groups, and to optimize the summation of the weights of the cut edges. These problem variants often arise in real world applications [MKYM07, WWL15]. We tested the IGA on two representative variants; the 4-way graph partitioning problem and the weighted maximum cut problem. The same dataset in Section 4.3.1 are used in our experiment.

4-way Graph Partitioning Problem

A k-way graph partitioning problem is a variant of GPP. The problem is to divide the vertices into k partitions which minimize the cut size. The most widely used value of k is two, as discussed in this chapter. When k is larger than two, then the problem is called a multi-way GPP. The powers of two up to 128 are often used for the value of k [MKYM07, SWC04]. The optimal cut size and the difficulty of the problem tend to increase as k increases. We use $k = 4$ in our experiment.

For the incremental approach on a multi-way GPP to be effective, we have to find an ordering of the edges which leads to a subproblem sequence with an optimal substructure. The same argument on 2-way GPP could be applied to the cases when $k > 2$. If we order the non-cut edges before the cut edges, then an optimal solution to the original problem will stay as an optimal solution to each of the subproblems. The same strategies could be applied as well and we decided to use the M-IGA with BFS ordering scheme.

Table 17: Results on the 4-way graph partitioning problem

Graph	GA	M-IGA
G500.2.5	153.10	**126.48**
G500.05	463.32	**440.70**
G500.10	1127.06	**1106.26**
G500.20	2895.99	**2879.88**
G1000.2.5	314.99	**261.22**
G1000.05	949.43	**913.57**
G1000.10	2465.43	**2455.63**
G1000.20	**5705.93**	5722.02
U500.05	104.51	**30.89**
U500.10	258.70	**143.27**
U500.20	640.33	**485.05**
U500.40	1344.74	**1160.31**
U1000.05	235.30	**136.02**
U1000.10	571.48	**356.87**
U1000.20	1347.45	**935.65**
U1000.40	2752.43	**2267.20**

The average cut sizes found by GAs without and with the incremental approach are shown in Table 17. The smaller values for each graph instances are marked in bold. The values are bigger than that of the 2-way GPP, and the values obtained by M-IGAs were smaller in general. The *p*-values of Welch's *t*-test have been calculated, but they are presented in the table as all of them was much smaller than 0.0001. Only one of the instances, G1000.20, was an exceptional case, and the amount of the improvement for U*n.d* graphs was greater than that of G*n.d* graphs. This suggests that the IGA is effective on multi-way GPP as well.

Weighted Maximum Cut Problem

The weighted MCP is a general version of MCP. In this case, the cut size is defined to be the summation of the weights of the cut edges. The unweighted version of the problem is a special case when all of the edges have the same weight of one. As explained in Section 4.3.1, we used unweighted graph instances from the G-set [HR00] for the experiments on the unweighted MCP. The dataset consists of weighted graphs as well, and we used twelve graph instances from the dataset in our experiment. They are G6 to G10, which are similar to G1 to G5, G11 to G13, which are toroidal graphs, and G18 to G21, which are similar to G14 to G17.

As V-IGAs were the only effective IGAs for unweighted MCP, we used them in our experiment. We exclude randomized ordering as it only provides a baseline. Instead, we tried another reordering scheme based on the sum of the weights sum. For each vertex in the graph, We had calculated the summation of the weights of incident edges and reordered the vertices with decreasing order of the weights sum. This is a generalization of vertex degree and it is likely to be suited for the weighted version of the problem.

Table 18: Results on weighted maximum cut problem

G	GA	V-IGA		
		Degree	BFS	WS
6	1956.06	**1959.17**	[**1959.23**]	1952.62
7	1793.39	[**1804.64**]	1793.18	1783.90
8	1802.97	[**1813.10**]	1801.70	1793.29
9	1837.49	[**1850.16**]	**1840.97**	1829.99
10	1791.52	[**1802.71**]	1790.55	1776.41
11	498.92	**508.75**	[**527.29**]	**510.35**
12	491.89	**502.19**	[**521.65**]	**502.99**
13	515.20	**525.99**	[**546.72**]	**526.88**
18	887.43	**912.66**	[**915.61**]	874.47
19	800.22	**820.83**	[**831.85**]	796.81
20	828.98	**858.47**	[**860.33**]	**848.42**
21	824.13	**849.59**	[**854.17**]	810.55

Table 18 shows the average cut sizes found by the algorithms. The improved ones are in bold, and the best one is parenthesized as before. The incremental approach worked successfully on the weighted MCP, and the characteristic of the instances influenced the result similarly to the result on the unweighted problem. For random graphs (G6 to G10) and planar graphs (G18 to G21), the best reordering scheme was degree based one and BFS based one, respectively. They were the best schemes for the unweighted problem, as shown in Table 15. Toroidal graphs (G11 to G13) are the new ones and they are almost planar graphs. Like the result on other planar graphs, the BFS reordering scheme worked well on them.

Surprisingly, the V-IGA which reorders the vertices according to the weights sum (WS) was not effective for most of the instances. It was even worse than the one with degree based reordering. This result seems to be caused by the characteristic of the dataset. For all of the weighted graphs in the G-set, the weights are either -1 or 1, which means that the graphs have edges with negative weights. If all of the edge weights are non-negative, then we could apply the same argument on the unweighted MCP to build up a subproblem sequence with an optimal substructure. But this does not hold if there exists an edge with a negative weight. More specifically, suppose that there exists an edge with negative weight in the cut edges of an optimal solution of the original problem. Then an optimal solution of the subproblem with this edge only is not to use it as a cut edge, as it will change the cut size from a negative value to zero. Therefore, even if the weights sum based reordering is expected to work well on the weighted problems, it does not fit in with this dataset.

Chapter 5

Related Applications

5.1 Measuring Source Code Similarity with an Incremental Genetic Algorithm

5.1.1 Introduction

The source code similarity is an important concept in the software engineering field. It is related to determining whether two codes share a similar property, or to finding duplicated code fragments in a large program [WERC^{+}07]. There are a huge number of real world applications based on code similarity, including code plagiarism detection, code clone detection, and malware detection.

A lot of methods for measuring the similarity have been proposed and the methods could be categorized by the types of the data they mainly handle with. Among text based methods, metric based methods, and graph based methods, the best approach is known to be the graph based methods. They transform a given program into a graph, and measuring code similarity is modeled as a combinatorial optimization problem on these graphs.

One of the widely used graph structure is a program dependence graph (PDG). A PDG is a graph generated from the source code which demonstrates the flow of data and control [ASU86]. The logical connections between the code statements are also illustrated in PDGs. Among the graph structures, PDGs are known to best reflect the logical structure of the code, and most of the structural characteristics of the graph survive even when the disguise techniques are applied to the code [RC07]. However, most of the graph algorithms require a large amount of time in comparing PDGs. Several heuristics to reduce the running time have been proposed and experimental results show that PDG based methods are more effective than the other ones [KM10, Kri01, LCHY06].

To avoid being detected by a code similarity detection tools, diverse ways of disguise techniques are possible on the source code [Ayc06, KM10, LCHY06]. While the semantics or functionality of the program is being preserved, the appearance of the program is modified by these techniques. The examples of the techniques include format alteration, identifier renaming, code replacement, code reordering, code insertion, and subroutine inlining/outlining.

5.1.2 The Proposed System

A New Cost Function

First, we formulate the problem as follows. We measure the similarity between two codes by solving an error correcting subgraph isomorphism problem. Given two codes, we transform each of them into PDGs, namely G_1 and G_2. For the problem to be well-defined, we choose the smaller one that has a fewer number of vertices as G_1. The task of the problem is to find a subgraph of G_2 which requires the least amount of error correction cost to transform the subgraph into G_1. In other words, it is to find a subgraph of G_2 that is the most similar to G_1. The difference between two codes is defined to be the minimum error correction cost.

The error correction cost of the problem could be defined in terms of both vertices and edges [Bun00, KM10]. Empirical studies showed that using the weighted sum of error correction costs for edges and vertices makes the search space to be more globally convex, and thereby makes it easier to find a better solution [CYM12]. However, as the error correction cost for vertices used in the previous work is based on the degree condition of two graphs, it could be only used to find an exactly matched subgraph. Therefore, a different cost function should be investigated for the error correcting subgraph isomorphism problem.

Let C_v and C_e denote the error correction costs for vertices and edges, respectively. For C_v, we use the number of pairs of vertices that match even when their colors are different. The color of the vertex represents the attribute of the corresponding component of the source code. Thus matching two vertices with different colors might not be good, and it is required to minimize this case. For C_e, we use the number of edges which need to be either added to or removed from the subgraph. For the error correction cost C for the entire graph, we use the weighted sum of C_v and C_e. We use $C = 0.9\,C_v + 0.1\,C_e$ in our system.

We define the similarity of two graphs to be $(1 - 10\,C)/|E_1|$. E_1 is the set of the edges of the smaller graph G_1. This value roughly denotes the portion of the common edges to the total number of edges. This value could be negative, which means that the two graphs are totally different.

PDG Modification

A PDG of a source code is a data structure that reflects the logical structure of the code [ASU86]. The statements in the code are transformed into vertices of the graph. If a statement has a logical dependency on another statement, they are connected with a directed edge. Both the flow of data and flow of control are used to determine the dependency. Before trying to match the PDGs, we preprocess the graphs by slightly modifying the structure.

We first color the vertices, according to the type of the corresponding statement. If two statements are of different type, they are less likely to be matched. The coloring technique was widely used in previous works in order to prevent unnecessary computation [KKM$^+$06, LCHY06].

We then remove some of the vertices from the graph. This is the technique used in a previous work to reduce the computational cost [KM10]. Four different kinds of reduction rules were introduced, but using all of them runs a risk of destroying the distinctive structure of the code. We use only two of them; we remove the vertices which have no connected edges, and vertices that are used for variable declaration. We remove the edges connected to these vertices as well.

GA Operators and Parameters

We use a genetic algorithm hybridized with the proposed local optimization algorithm in each step of the incremental approach. For the genetic operators and parameters, we use the best ones explained in Chapter 3.

PDG Generation

We tested the proposed system with the source codes written in C and VBS (Visual Basic Script) language. The detailed information of the dataset is presented in the next subsection with the experimental results.

For codes in C, we use Frama-C[1] to generate PDGs. This program performs a static value analysis first, and then generates the PDG for each functional procedure in the source code. Some vertices indicating the start and the end of the procedure are added to the graph. We found that Frama-C adds these kinds of vertices at each time when a function call is made, and it occasionally adds too many vertices for some procedures. These nodes are removed from the graph, as they do not properly reflect the logical structure inside the procedure. We use all the three types of dependency edges generated by Frama-C, including address, control, and data dependency edges. The statements are classified by their types, and the corresponding vertices are colored according to the type. The types used are goto labels, control statements (`goto`, `break`, and `continue`), loops, assignment statements, logical expressions, and the other kinds of statements.

For VBS language, we could not find an appropriate program. We used a tokenizer and generated PDG-like graphs by ourselves instead. Each line of the code, but not each statement, is represented as a vertex. If a variable is used in a line, it is connected to the next line where the same variable is used. And if there is a function call in a line, it is connected to the first line of that functional procedure. The vertices are classified by the set of *keywords*

[1]`http://frama-c.com/`

used in the corresponding line of the code, and only the vertices with the same *keywords* set are to be matched. The *keywords* are the reserved words (`dim`, `and`, `next`, etc.) and the name of API and library functions.

5.1.3 Experimental Results

GPLAG Dataset

GPLAG, one of the best PDG based software plagiarism detection tool, was tested on a real world source codes written in C language [LCHY06]. The source code of an open source Linux program `join` was plagiarized by applying a number of plagiarism disguise techniques. Since both GPLAG and the plagiarized code are not open in public, we applied the same set of plagiarism operators to generate a modified code. For fairness, the codes were modified independently.

We used the source code of version 8.23, and modified the functional procedures listed in the previous work [LCHY06]. Since one of the procedures was not found in the version we used, we instead used a procedure with sufficient complexity. The procedures we plagiarized are add_field_list, get_line, join, keycmp, prjoin, and xfield. The size of the code and the PDG are listed in Table 19. We name this GPLAG dataset. The lines of code (LOC), the number of vertices $|V|$, and the number of edges $|E|$ are listed in the table.

We conducted an experiment of plagiarism detection on the GPLAG dataset. Every pair of modified and original codes is compared and the similarities between them are measured by the proposed system. Then, for each

Table 19: The size of the codes in the GPLAG dataset

Procedure	Original code			Plagiarized code		
	LOC	$\|V\|$	$\|E\|$	LOC	$\|V\|$	$\|E\|$
add_field_list	17	9	31	18	9	31
get_line	27	17	26	27	17	26
join	132	112	1097	143	116	1134
keycmp	47	38	188	47	38	188
prjoin	58	26	96	57	26	96
xfield	33	44	602	43	39	510

Table 20: Results of the plagiarism detection experiment

Modified procedure	1st match	2nd match
mod_add_field_list	add_field_list 100.00%	join 93.55%
mod_get_line	get_line 100.00%	keycmp 69.23%
mod_keycmp	keycmp 100.00%	get_line 69.23%
mod_prjoin	prjoin 100.00%	add_field_list 90.32%
mod_join	join 96.72%	add_field_list 93.55%
mod_xfields	xfields 99.80%	join 79.41%

of the modified codes, the best match with the highest similarity is found. If the similarity is above a certain threshold value, then the two codes are reported to be plagiarized. The best and the second best matched results are listed in Table 20. The names of the matched procedure and the similarity are shown in the table. For clarity, we used the prefix mod_ to denote that the procedure is a modified, plagiarized one.

All of the six modified source codes are matched to the original versions of them with the highest similarity. This suggests that the proposed system well measures the similarities between the code and its plagiarized version, even when the disguised techniques are applied. The results of the four codes recorded 100% similarity, and it means that the proposed system found no difference between the matched codes. The other two codes recorded the similarity over 95%. The codes were also tested on another well-known plagiarism detection tool MOSS[2]. This tool reported that the most similar pair has the similarity value of 25%, which is much lower than the proposed system. The proposed system was able to cope with the disguise techniques, whereas MOSS could not.

Since all of the codes were matched to their original version in the best match, the second best match shows the similarity between two distinct procedures. These values are thus expected to be much lower than the previous cases, but it was above 90% in half of the cases. As shown in Table 19 and Table 20, these are the cases when there is a huge gap between the sizes of the graphs. When the sizes differ greatly, it is more likely that G_2 has a subgraph that is isomorphic to G_1, even if the two programs are not logically similar. We could avoid this by filtering these cases. If we skip the smallest and biggest procedures, then the highest similarity dropped to 69.23% and a negative similarity is obtained in more than half of the cases. The results suggest that the proposed system comfortably detects the similarity; it is high when two codes are similar, and low if they are distinctively different, unless the sizes of two codes differ too much.

[2]`http://theory.stanford.edu/~aiken/moss/`

Table 21: Similarity between the graph algorithms

Code 1	Code 2	Similarity
Dijkstra	Prim	96.71%
BF	Dijkstra	87.27%
FW	Kruskal	85.95%
BF	Prim	82.91%
BF	Kruskal	74.18%
BF	FW	72.00%
FW	Prim	70.57%
Dijkstra	FW	67.22%
Dijkstra	Kruskal	53.73%
Kruskal	Prim	52.95%

Graph Algorithms Dataset

The dataset used in the previous subsection was the codes intentionally generated to be similar to each other. To test that the proposed system is capable of measuring the general similarity between the codes, we implemented five graph algorithms in C language and measured the similarity between them. The algorithms are Kruskal's and Prim's algorithm for the minimum spanning tree problem, and Bellman-Ford's (BF), Dijkstra's and Floyd-Warshall's (FW) algorithms for the shortest path problem. Table 21 shows the similarity between all the pairs of the codes measured by our system, listed in decreasing order of the similarity. A routine for generating an adjacent matrix was used in all of the codes, and this seems to make the similarity be above 50%.

Interestingly, the most similar pair was turned out to be <Dijkstra, Prim>. Even though they are for different problems, both of the two codes have a routine for finding a vertex with the least distance, and it makes the overall structures of the algorithms be similar. <BF, Dijkstra>, the pair of the algorithms having a relaxation operator in common, and <FW, Kruskal>, the pair of the algorithms transforming the given adjacent matrix into another data structure, are the next two similar pairs. The similarity measured by the proposed system seems to catch all these structural features of the algorithms.

Malware Dataset

We also tested our system with the dataset used in malware detection [KM12]. The dataset includes original malicious codes, the variants of the malware, and harmless codes all written in VBS language. We used the variants that are generated by applying disguise techniques (generated codes), and the harmless codes (benign codes). The total numbers of codes in each set are 56 and 24, respectively. The purpose of the former codes is to see if the similarity between two similar malware measured by the proposed system is high, and the purpose of the latter codes is the opposite.

Similarly to the plagiarism detection experiment, each of the codes is compared to all of the malware and the most similar match is found. If the similarity is higher than a threshold, then the code is reported to be malicious. Since there exists a great deal of original malware in the dataset, we used 13 out of 359 codes in our experiment. The names of the malware are `Agent.ay`, `Fuss`, `Ham`, `Hello`, `Jadra`, `Marata`, `Neves.a`, `Neves.b`, `Quest`, `Sheep`, `Sock`, `Stress`, `Yabuka.a`. We name the set to be malware database. The selected malware samples are the ones used in the generation of variants in the dataset.

Table 22: Histogram of similarity between the generated codes and codes in malware database

Similarity	1st match	2nd match
100%	19	0
80% ∼ 100%	21	9
60% ∼ 80%	10	3
40% ∼ 60%	2	6
20% ∼ 40%	4	9
below 20%	0	29

As we did in experiments on GPLAG dataset, we found the best and the second best matches for each of the generated code. Table 22 shows the summary of the results. The results of the best and the second best matches are shown. Since the number of cases is big, we present the result as a histogram. For the best match, 50 out of 56 cases recorded a similarity over 80%. Most of the generated variants were reported to be similar to their original malware. For the second best match, the similarity dropped sharply. Moreover, for 17 out of 18 cases with 2nd similarity higher than 40%, the proposed system reported `Neves.a` and `Neves.b` to be the best and the second best match. These two malware are similar to themselves and their similarity was measured to be 82.14%. If they are compared to any other malware, the similarity value was much lower than these cases. The measured similarity seems to be fairly reasonable for these source codes.

To find a suitable threshold value, we examined the five codes with the lowest similarity for generated codes, and the five codes with the highest similarity for the benign codes. The result is shown in Table 23. For generated codes, the five codes with the lowest similarity are listed. And for benign codes, the five codes with the highest similarity are listed. The name of

Table 23: the five cases with the highest chance of error for each set of codes

(a) Generated codes

Origin	Matched to	Similarity
Stress	Sock	22.81%
Stress	Sock	25.00%
Stress	Stress	29.03%
Sock	Sock	32.14%
Hello	Hello	48.39%

(b) Benign codes

Matched to	Similarity
Yabuka.a	28.57%
Yabuka.a	25.00%
Yabuka.a	22.73%
Fuss	22.22%
Yabuka.a	17.24%

the matched malware and the similarity are presented. For generated codes, the name of the original malware is presented as well.

For the generated codes, the two cases with the lowest similarity were matched to a wrong malware. These were the only two mismatched cases in our experiment, and the correct malware was in their second best match. We found that all of these 5 variants were generated by applying subroutine outlining with other disguise techniques. If these techniques are applied, the structure of the PDG is altered and this results in a decrease in similarity. This technique was applied in 11 variants, and the average similarity to their origins was 55.19%. Compared to the average of the other 45 cases, which is 91.63%, a notable decrease in similarity is found if subroutine outlining is applied. The structure of the graph seems to be revised to deal with these disguise techniques.

For the benign codes, all of them were matched to either Yabuka.a or Fuss. These two malware are relatively short and have relatively simple logical structure. Due to the characteristics of the script languages, most of the benign codes have simple, or even linear logical flow as well. And

Table 24: Accuracy of our malware detection system with two different threshold values, a system from a previous work, and known anti-virus programs

Method	Generated	Benign
Threshold = 30%	94.64%	100.00%
Threshold = 25%	98.18%	95.83%
Previous work [KM12]	100.00%	95.83%
Anti-viruses [KM12]	43.34%	98.91%

this makes it harder to distinguish them. It seems that the relatively high similarity was obtained by this property.

As shown in Table 23, the threshold values of malware detection system have to be near 22.81% or 28.57%. We choose 25% and 30% as threshold values, and the accuracy of the detection system is shown in Table 24. The system was compared to results presented in [KM12], which includes the detection system proposed in the previous work, and known anti-virus programs. Our system missed two cases for both of the threshold value, whereas the system in previous work misses only one case. Nevertheless, it was much better than known anti-viruses. Note that we use a general code similarity measurement system and the system in previous work uses domain specific knowledge. We may build a better system if we adopt problem-specific knowledge on malware.

Among the three different kinds of dataset, the malware dataset consists of the case which requires the longest running time. Measured on Intel Core i7-3820 CPU @ 3.60GHz system, it took 141.23 seconds. For the entire dataset used in our experiment, about half of the cases required less than a second, and ninety percent of the cases took less than ten seconds.

This shows that the proposed system measures the code similarity in fairly reasonable time, for most of the cases.

5.1.4 Discussion

The proposed system reported that some of the two unlike codes were similar. This happens when there is a huge difference between the sizes of the graphs. The false alarm could be avoided by only comparing the graphs with a comparable size. However, this makes the system vulnerable to the code insertion disguise technique. We may use a domain specific knowledge or hybridize the system with other methods to handle these cases. We leave it as future work.

The two missed cases are related to subroutine outline. If this disguise technique is applied to a source code, the structure of the dependence graph is altered and the similarity between the code and its original form is decreased. The PDGs hardly reflects the dependency if a function call is made in the code. A better graph structure needs to be investigated to deal with this case properly.

5.2 Linear Ordering Problem and an Approximate Fitness Evaluation

5.2.1 Introduction

Let $G = (V,E)$ be a directed graph with N vertices and M edges, and $w(u,v)$ be the weight of an edge from vertex u to v. The linear ordering problem (LOP) is to find a linear ordering of vertices $\langle v_1, v_2, \ldots, v_N\rangle$ which maximizes the fitness function

$$f(\langle v_1, v_2, \ldots, v_N \rangle) = \sum_{i=1}^{N} \sum_{j=i+1}^{N} w(v_i, v_j).$$

If the graph is acyclic, then a topological ordering is an optimal solution; if not, the problem is equivalent to the feedback arc set problem (FASP) which tries to remove minimal edges from the graph to make it acyclic.

LOP, FASP, and several other related problems have been studied for a long time [GJR84]. These problems have a number of applications in various fields [SS05], and they are known to be NP-hard problems. Many meta-heuristic approaches, including evolutionary approaches, have been proposed to solve the problem efficiently [MRD12]. It is known that genetic algorithms are suitable for solving combinatorial optimization problems. But usually they are slow since a large number of candidate solutions are generated and evaluated during the evolutionary process.

In this section, we propose a genetic algorithm for LOP which evaluates an approximate fitness value. In the early generation of the GA, only a portion of the edges are used to evaluate the fitness. We increase the number of used edges from 0 to M over time. The proposed algorithm was tested on well-known instances. Experimental results show that the proposed approach reduces the running time without losing the quality of the solutions.

5.2.2 The Proposed Method

We use a typical generational GA to solve the problem. The operators and parameters are as follows.

- **Population**: The population consists of 100 permutations, each representing a linear ordering of the vertices. We generate 20 offspring in each generation. Among the 120 solutions, the best 100 solutions form the population of the next generation.

- **Selection**: We randomly pick two solutions and return the better one with probability 80%, and the worse one with probability 20%.

- **Crossover**: We use order based crossover [SS05]. First, each gene is copied from one of the two parents. Then we pick some genes and reorder them with respect to their order in the other parent. The probability for each gene to be picked is 50%.

- **Mutation**: Each gene has 1% chance of mutation, and the selected genes are randomly shuffled.

- **Stopping Criterion**: The GA runs for a fixed number of generations and stops. The number of generations K depends on the characteristic of the instance.

To compute the fitness directly, all N^2 pairs of vertices have to be considered. Buy by considering the edges of the graph, the fitness can be rewritten as

$$\sum_{(u,v)\in E} w(u,v)R(u,v).$$

R is a binary relation on V and $R(u,v)$ is defined to be 1 if u comes before v in the ordering, and 0 otherwise. The running time of the evaluation is linear to the number of the edges.

To approximate the fitness, we use the subset $E' \subseteq E$ which has M' edges, and compute the summation only for $(u,v) \in E'$. The accuracy of the approximation is controlled by the size of E'; it is more accurate if the size is close to M. We set the size to be 0 in the first generation, and to be M in the half-way point. During the first half of the GA, the size is gradually and linearly increased as the generation progresses. In the second half, the GA is the same as the one using exact fitness evaluation. The GA travels the search space almost randomly in the earlier generations, and it gradually finds the right direction.

The edges in E' need to be changed at each time of the fitness evaluation. GA could easily be stuck in a local optimum, if E' has only few edges in it and it is being kept similar for a while. However, it is inefficient to randomly choose M' edges all the time. It takes time to generate a random number, and shuffling the set of edges diminishes the cache utilization. Instead, we only change some part of the elements in E'. We first randomly shuffle the list of edges, and the first M' edges form the subset E'. We exchange $0.005 \times M$ random pairs of edges at each time of evaluation; the i^{th} and the j^{th} edges (i and j are randomly picked from $\{1,2,\ldots,M'\}$ and $\{1,2,\ldots,M\}, respectively$).

5.2.3 Experimental Results

The proposed algorithm was tested on a widely used benchmark library LOLIB [MRD12]. There are 485 instances, comprising both real world instances and randomly generated ones. Among them, we selected the instances with sizes $N = 50, 100, 150, 200$, and 250, since there exist a suf-

Table 25: Comparing the performance of the algorithms in terms of fitness and time

N	APPROX-K to EXACT-K		APPROX-2K to EXACT-2K	
	fitness	time	fitness	time
50	0.9990	0.9819	1.0001	0.9821
100	1.0001	0.9426	1.0004	0.9446
150	0.9989	0.9595	0.9998	0.9595
200	1.0015	0.9346	1.0013	0.9342
250	0.9966	0.9905	0.9983	0.9907

ficient number of instances having those sizes. An instance having an error, N-t65f11xx_150, was excluded.

The number of generations *K* was selected in order to guarantee that the population is converged after *K* generations. We assume that the population is converged if no change is made to the population for 100 generations. An appropriate value of *K* for each instance was chosen after a sufficient number of experiments.

We conducted experiments with 4 different kinds of GAs. Two of them are the GA using exact fitness (EXACT-K) and the proposed GA using an approximate fitness (APPROX-K). The other two GAs run for 2*K* generations, and each of them uses either exact (EXACT-2K) or approximate fitness (APPROX-2K). We measured the fitness value of an optimal solution found and the running time for each run. We conducted 100 runs for each instance and each algorithm, and computed the average result.

Table 25 shows the comparison between APPROX-$\star$ algorithms and EXACT-$\star$ algorithms. APPROX-K algorithms were compared to EXACT-K algorithms, and APPROX-2K algorithms were compared to EXACT-2K algorithms. For each instance, we compute the relative fitness and time. The relative fitness is the ratio of the average fitness value obtained by the algorithm with approximation to the average fitness value obtained by the algorithm without approximation. The relative time is computed similarly. Then, we averaged the relative values over the instances with the same size.

By using the approximate fitness evaluation, the running time was reduced for both of the cases with K and $2K$. However, the quality of an optimal solution found did not change significantly by using the approximation scheme. In fact, it was even better for some of the instances.

The relative performance of `APPROX-2K` algorithms was slightly better than that of `APPROX-K` algorithms. Note that the genetic algorithm was converged in K generations. This suggests that, when running a genetic algorithm with sufficiently large number of generations, the approximation scheme helps to escape from a local optimum. Using the approximate fitness evaluation gives a chance of accepting a worse solution, and the probability gradually decreases during the space search. The same idea is often used in some metaheuristics, and the representative one of them is simulated annealing.

Chapter 6

Conclusions

In this thesis, we proposed a new genetic algorithm with an incremental approach for graph optimization problems, including the subgraph isomorphism problem and graph cut optimization problems. The incremental approach starts from solving a small-sized subproblem and it gradually expands the size of the problem. The prior results were used for an initial population of a hybrid genetic algorithm for the following step.

For the subgraph isomorphism problem, we met several design issues for the new algorithms. We observed that stopping criterion also had an influence on the performance. If we focus on exploration in intermediate steps, it was better to evolve from a diverse population with few local optima. The size of problem expansion did not have an effect on the performance as much as a vertex reordering and stopping criterion did. However, increasing the expansion size of each step showed a tendency to improve the running time. By hybridizing a local optimization algorithm and maintaining moderate diversity, the hybrid incremental approach with appropriate schemes outperformed previous work in the experimental results. The performance was further improved by newly designed operators, including a new vertex ordering scheme and a new local optimization algorithm.

For the graph cut optimization problems, We first formally defined the process of an incremental genetic algorithm for a graph problem in terms of a subproblem sequence. The algorithm solves the subproblems according to this sequence. To obtain a sequence having an optimal substructure, appropriate vertices or edges are needed to be selected at an adequate step of the process. Graph expansion methods, reordering schemes, and their combinations were proposed and tested. The incremental approach has brought performance improvements for these problems as well.

Although we traced the design issues for the incremental approach, there are still other issues that have to be figured out for a better performance. Better genetic operators, and a local optimization algorithm can improve the performance of the proposed algorithm. As memetic algorithms spend large time in local optimization, an efficient graph expansion method has to be developed as well. Moreover, the generalization of the incremental approach on other graph optimization problems have to investigated as well.

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Ayc06] John Aycock. *Computer Viruses and Malware*. Springer, 2006.

[BB02] C. Borgelt and M.R. Berthold. Mining molecular fragments: finding relevant substructures of molecules. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 51–58, 2002.

[BBK11] Gozde Bakirli, Derya Birant, and Alp Kut. An incremental genetic algorithm for classification and sensitivity analysis of its parameters. *Expert Systems with Applications*, 38(3):2609 – 2620, 2011.

[BBM93] D Beasley, D R Bull, and R R Martin. An overview of genetic algorithms: Part 1, fundamentals. *University Computing*, 15(2):58–69, 1993.

[BJWG94] Robert D. Brown, Gareth Jones, Peter Willett, and Robert C. Glen. Matching two-dimensional chemical graphs using genetic algorithms. *Journal of Chemical Information and Computer Sciences*, pages 63–70, 1994.

[BKSS00] Jürgen Branke, Thomas Kaussler, Christian Smidt, and Hartmut Schmeck. *A Multi-population Approach to Dynamic Optimization Problems*, pages 299–307. Springer London, London, 2000.

[BM94] T. N. Bui and B.-R. Moon. A genetic algorithm for a special class of the quadratic assignment problem. In *the Quadratic*

Assignment and Related Problems, volume 16 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 99–116. 1194.

[BM96] Thang Nguyen Bui and Byung Ro Moon. Genetic algorithm and graph partitioning. *Computers, IEEE Transactions on*, 45(7):841–855, Jul 1996.

[BMS$^+$13] A. Buluc, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. *ArXiv e-prints*, November 2013.

[Bun00] Horst Bunke. Graph Matching: Theoretical Foundations, Algorithms, and Applications. In *International Conference on Vision Interface*, pages 82–84, May 2000.

[CBM07] Pierre Chardaire, Musbah Barake, and Geoff P. McKeown. A probe-based heuristic for graph partitioning. *IEEE Trans. Comput.*, 56(12):1707–1720, December 2007.

[CCPS98] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[CFV07] Donatello Conte, Pasquale Foggia, and Mario Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms and Applications*, 11(1):99–143, 2007.

[CG70] D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *J. ACM*, 17(1):51–64, January 1970.

[CGP11] Carlos Cruz, Juan R. González, and David A. Pelta. Optimization in dynamic environments: a survey on problems, methods and measures. *Soft Computing*, 15(7):1427–1448, 2011.

[CKM14] HyukGeun Choi, JinHyun Kim, and Byung-Ro Moon. A hybrid incremental genetic algorithm for subgraph isomorphism problem. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 445–452, New York, NY, USA, 2014. ACM.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[CYM12] Jaeun Choi, Yourim Yoon, and Byung-Ro Moon. An efficient genetic algorithm for subgraph isomorphism. In *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference*, GECCO '12, pages 361–368, 2012.

[DFG$^+$15] Daniel Delling, Daniel Fleischman, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. An exact combinatorial algorithm for minimum graph bisection. *Math. Program.*, 153(2):417–458, November 2015.

[Due93] Gunter Dueck. New optimization heuristics: The great deluge algorithm and the record-to-record travel. *Journal of Computational Physics*, 104(1):86 – 92, 1993.

[Epp95] David Eppstein. Subgraph isomorphism in planar graphs and related problems. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, pages 632–640, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[FSV01] Pasquale Foggia, Carlo Sansone, and Mario Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarking. In *Proc. of the 3rd IAPR TC-15 International Workshop on Graph-based Representations*, pages 176–187, 2001.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[GJR84] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32(6):1195–1220, 1984.

[HH13] S. Halim and F. Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests*. Number V. 3. Lulu.com, 2013.

[HKM06] Inwook Hwang, Yong-Hyuk Kim, and Byung-Ro Moon. Multi-attractor gene reordering for graph bisection. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1209–1216, New York, NY, USA, 2006. ACM.

[HKY15] Inwook Hwang, Yong-Hyuk Kim, and Yourim Yoon. Moving clusters within a memetic algorithm for graph partitioning. *Mathematical Problems in Engineering*, 2015, 2015.

[HR00] C. Helmberg and F. Rendl. A spectral bundle method for semidefinite programming. *SIAM Journal on Optimization*, 10(3):673–696, 2000.

[IWM00] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In DjamelA. Zighed, Jan Komorowski, and

Jan Żytkow, editors, *Principles of Data Mining and Knowledge Discovery*, volume 1910 of *Lecture Notes in Computer Science*, pages 13–23. Springer Berlin Heidelberg, 2000.

[JAMS89] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation. part i, graph partitioning. *Oper. Res.*, 37(6):865–892, October 1989.

[JB05] Yaochu Jin and J. Branke. Evolutionary optimization in uncertain environments-a survey. *IEEE Transactions on Evolutionary Computation*, 9(3):303–317, June 2005.

[KCYM16] Jinhyun Kim, HyukGeun Choi, Hansang Yun, and Byung-Ro Moon. Measuring source code similarity by finding similar subgraph with an incremental genetic algorithm. In *Proceedings of the 2016 Annual Conference on Genetic and Evolutionary Computation*, GECCO '16, pages 925–932, New York, NY, USA, 2016. ACM.

[KHKM11] Jin Kim, Inwook Hwang, Yong-Hyuk Kim, and Byung-Ro Moon. Genetic approaches for graph partitioning: A survey. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 473–480, New York, NY, USA, 2011. ACM.

[KKM$^+$06] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, pages 207–226, Berlin, Heidelberg, 2006. Springer-Verlag.

[KM04] Yong-Hyuk Kim and Byung-Ro Moon. Lock-gain based graph partitioning. *Journal of Heuristics*, 10(1):37–57, January 2004.

[KM10] Keehyung Kim and Byung-Ro Moon. Malware detection based on dependency graph using hybrid genetic algorithm. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 1211–1218, 2010.

[KM12] Jinhyun Kim and Byung-Ro Moon. New malware detection system using metric-based method and hybrid genetic algorithm. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, pages 1527–1528, 2012.

[KM14] Jinhyun Kim and Byung-Ro Moon. A genetic algorithm for linear ordering problem using an approximate fitness evaluation. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1461–1462, New York, NY, USA, 2014. ACM.

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 301–, Washington, DC, USA, 2001. IEEE Computer Society.

[KYM16] Jinhyun Kim, Yourim Yoon, and Byung-Ro Moon. Solving maximum cut problem with an incremental genetic algorithm. In *Proceedings of the Companion Publication of the 2016 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '16, pages 49–50, New York, NY, USA, 2016. ACM.

[Las91] Gregor Von Laszewski. Intelligent structural operators for the k-way graph partitioning problem, 1991.

[LCHY06] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD ’06, pages 872–881, 2006.

[LL01] Jianzhuang Liu and Yong Tsui Lee. A graph-based method for face identification from a single 2D line drawing. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(10):1106–1119, October 2001.

[Luk82] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25(1):42 – 65, 1982.

[MAEf06] Nashat Mansour, Mohamad Awad, and Khaled El-fakih. Incremental genetic algorithm. *The International Arab Journal of Information Technology*, 3(1):42–47, 2006.

[MB00] B.T. Messmer and H. Bunke. Efficient subgraph isomorphism detection: a decomposition approach. *Knowledge and Data Engineering, IEEE Transactions on*, 12(2):307–323, 2000.

[MKYM07] Alberto Moraglio, Yong-Hyuk Kim, Yourim Yoon, and Byung-Ro Moon. Geometric crossovers for multiway graph partitioning. *Evol. Comput.*, 15(4):445–474, December 2007.

[MRD12] Rafael Martí, Gerhard Reinelt, and Abraham Duarte. A benchmark library and a comparison of heuristic methods for the linear ordering problem. *Comput. Optim. Appl.*, 51(3):1297–1317, April 2012.

[NYB12] Trung Thanh Nguyen, Shengxiang Yang, and Juergen Branke. Evolutionary dynamic optimization: A survey of the state of the art. *Swarm and Evolutionary Computation*, 6:1 – 24, 2012.

[OEGS93] Miles Ohlrich, Carl Ebeling, Eka Ginting, and Lisa Sather. Subgemini: Identifying subcircuits using a fast subgraph isomorphism algorithm. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, pages 31–37, New York, NY, USA, 1993. ACM.

[OSH87] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, pages 224–230, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[RC07] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. Technical Report 2007-541, School of Computing, Queen's University at Kingston, Ontario, Canada, 2007.

[RP94] J. Rocha and T. Pavlidis. A shape analysis model with applications to a character recognition system. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(4):393–404, April 1994.

[RWH$^+$10] J. H. Rutgers, P. T. Wolkotte, P. K. F. Holzenspies, J. Kuper, and G. J. M. Smit. An approximate maximum common subgraph algorithm for large digital circuits. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 699–705, Sept 2010.

[Sch03] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Number V. 1 in Algorithms and Combinatorics. Springer, 2003.

[SL12] Bo Song and Victor Li. A hybridization between memetic algorithm and semidefinite relaxation for the max-cut problem. In *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference*, GECCO '12, pages 425–432, New York, NY, USA, 2012. ACM.

[SS05] Tommaso Schiavinotto and Thomas Stützle. The linear ordering problem: Instances, search space analysis and algorithms. *Journal of Mathematical Modelling and Algorithms*, 3(4):367–402, 2005.

[SWC04] A.J. Soper, C. Walshaw, and M. Cross. A combined evolutionary search and multilevel optimisation approach to graph-partitioning. *Journal of Global Optimization*, 29(2):225–241, 2004.

[Ull76] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[vLM91] Gregor von Laszewski and Heinz Muhlenbein. Partitioning a graph with a parallel genetic algorithm. In Hans-Paul Schwefel and Reinhard Manner, editors, *Parallel Problem Solving from Nature*, volume 496 of *Lecture Notes in Computer Science*, pages 165–169. Springer Berlin Heidelberg, 1991.

[VN] P. Vivekanandan and D. R. Nedunchezhian. A new incremental genetic algorithm based classification model to mine data with concept drift.

[VS02] Jorge Valenzuela and Alice E. Smith. A seeded memetic algorithm for large unit commitment problems. *Journal of Heuristics*, 8(2):173–195, 2002.

[WERC$^+$07] Andrew Walenstein, Mohammad El-Ramly, James R. Cordy, William S. Evans, Kiarash Mahdavi, Markus Pizka, Ganesan Ramalingam, and Jürgen Wolff von Gudenberg. Similarity in programs. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[WWL15] Qinghua Wu, Yang Wang, and Zhipeng Lü. A tabu search based hybrid evolutionary algorithm for the max-cut problem. *Appl. Soft Comput.*, 34(C):827–837, September 2015.

[WYJ+04] A.S. Wu, H. Yu, S. Jin, Kuo-Chi Lin, and G. Schiavone. An incremental genetic algorithm approach to multiprocessor scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 15(9):824–834, 2004.

[YHM14] Hansang Yun, MyoungHoon Ha, and RobertIan McKay. Vlr: A memory-based optimization heuristic. In Thomas Bartz-Beielstein, Jurgen Branke, Bogdan Filipie, and Jim Smith, editors, *Parallel Problem Solving from Nature(PPSN) XIII*, volume 8672 of *Lecture Notes in Computer Science*, pages 151–160. Springer International Publishing, 2014.

[ZWL+11] Qing Zhong, Zhigang Wu, Lingxue Lin, Yao Zhang, and Yao Zhang. Computing resources assignment in rtds simulators with subgraph isomorphism based on genetic algorithm. In *Electric Utility Deregulation and Restructuring and Power Technologies (DRPT), 2011 4th International Conference on*, pages 1144–1149, 2011.

국문 초록

조합 최적화 문제는 최적화 문제 중에서 그 해공간이 이산적인 경우를 말한다. 그래프 객체는 그 구조가 이산적이기 때문에 다수의 그래프 문제를 조합 최적화 문제로 분류할 수 있다. 그래프가 다양한 분야에서 널리 활용되고 있는 자료 구조이다 보니 실세계에는 그래프를 입력으로 하는 다양한 조합 최적화 문제가 존재한다. 그중 일부 문제는 해공간의 크기가 문제의 크기에 대해 지수적이며, 따라서 그러한 문제를 풀 때는 보다 효과적인 공간 탐색 방법을 사용해야 한다.

유전 알고리즘은 조합 최적화 문제를 풀 때 널리 사용되는 방법이며, 특히 점진적 유전 알고리즘을 이용하면 그래프 최적화 문제를 효율적으로 풀 수 있다. 이를 위해서는 주어진 문제를 직접 푸는 대신에 부분 문제들을 정의한 후에 이들을 단계적으로 풀어야 한다. 부분 문제는 원래 주어진 그래프의 부분적인 구조를 택하여 정의하고, 이들을 풀 때에는 점진적 유전 알고리즘을 사용한다. 그리고 알고리즘의 중간 단계에서 사용되는 부분 문제의 크기를 점차 증가시킨다. 개별 부분 문제를 풀 때에는 유전 알고리즘이 사용되고, 유전 알고리즘의 초기 해집단으로는 이전 단계에서 진화된 해집단이 이용된다.

본 논문은 두 가지 조합 최적화 문제를 위한 점진적 유전 알고리즘을 제시한다. 각각의 문제는 부분 그래프 동형 문제와 그래프 컷 최적화 문제다. 또한, 본 논문에서는 부분 문제 수열에 대한 최적 부분 구조라는 개념을 고안하였으며, 이러한 개념 및 관련된 다른 요소들이 알고리즘의 진행 과정에 어떻게 영향을 끼치는지를 설명한다. 또한, 올바른 부분 문제

수열을 구축하려는 방법의 일환으로 몇 가지의 그래프 확장 방법과 정점 재배열 방식을 제시한다. 제안된 점진적 기법을 혼합형 유전 알고리즘과 결합하여 부분 그래프 동형 문제를 효과적으로 풀 수 있었으며, 이 알고리즘을 추가로 개선하여 거의 완벽한 결과를 거둘 수 있었다. 또한, 점진적 유전 알고리즘에 대한 분석 결과를 바탕으로 그래프 컷 최적화 문제를 푸는 효과적인 알고리즘을 고안하고 구현하였다. 그리고 그 알고리즘을 그래프 분할 문제 및 최대 컷 문제에 대한 벤치마크 그래프 인스턴스에 대해 수행하였으며, 실험을 통해서 부분 문제의 수열이 탐색할 공간의 지형도에 어떻게 영향을 끼치는지도 조사하였다. 적합한 부분 문제 수열을 이용한 점진적 유전 알고리즘을 이용하면 기존에 비해 향상된 결과를 얻을 수 있었다.