LEBANESE AMERICAN UNIVERSITY

IMPROVED SEARCH-TREE ALGORITHMS FOR THE CLUSTER EDIT PROBLEM

By

ALI KASSEM GHRAYEB

A Thesis submitted in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science

School of Art and Science

May 2011

Lebanese American University
School of Arts and Sciences

Thesis Approval Form

Student Name: Ali Kassem Ghrayeb I.D. #: 200803926

Thesis Title :

Improved Search-Tree Algorithms for

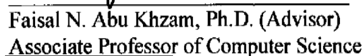
the Cluster Edit Problem

Program : MS in Computer Science

Division/Dept : Computer Science and Mathematics

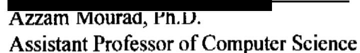
School : Arts and Sciences

Approved by :



Faisal N. Abu Khzam, Ph.D. (Advisor)
Associate Professor of Computer Science

Haidar Harmanani, Ph.D.
Professor of Computer Science



Azzam Mourad, Ph.D.
Assistant Professor of Computer Science

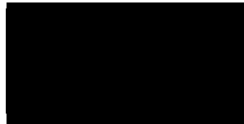
Date : 30/5/2011

THESIS PROJECT COPYRIGHT RELEASE FORM

LEBANESE AMERICAN UNIVERSITY

By signing and submitting this license, I (the author(s) or copyright owner) grant the Lebanese American University (LAU) the non-exclusive right to reproduce, translate (as defined below), and/or distribute my submission (including the abstract) worldwide in print and electronic format and in any medium, including but not limited to audio or video. I agree that LAU may, without changing the content, translate the submission to any medium or format for the purpose of preservation. I also agree that LAU may keep more than one copy of this submission for purposes of security, backup and preservation. I represent that the submission is my original work, and that I have the right to grant the rights contained in this license. I also represent that my submission does not, to the best of my knowledge, infringe upon anyone's copyright. If the submission contains material for which I do not hold copyright, I represent that I have obtained the unrestricted permission of the copyright owner to grant LAU the rights required by this license, and that such third-party owned material is clearly identified and acknowledged within the text or content of the submission. IF THE SUBMISSION IS BASED UPON WORK THAT HAS BEEN SPONSORED OR SUPPORTED BY AN AGENCY OR ORGANIZATION OTHER THAN LAU, I REPRESENT THAT I HAVE FULFILLED ANY RIGHT OF REVIEW OR OTHER OBLIGATIONS REQUIRED BY SUCH CONTRACT OR AGREEMENT. LAU will clearly identify my name(s) as the author(s) or owner(s) of the submission, and will not make any alteration, other than as allowed by this license, to my submission.

Name: Ali Ghrayeb

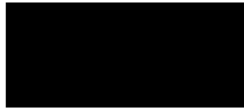
Signature:  Date: May 2011

PLAGIARISM POLICY COMPLIANCE STATEMENT

I certify that I have read and understood LAU's Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.

This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Ali Ghrayeb

Signature:  Date: May 2011

ACKNOWLEDGMENTS

This research would not have been possible without the help and assistance of many persons. First I would like to express my gratitude to my supervisor Dr. Faisal Abu-Khzam. He has been very supportive, guiding and patient throughout the entire work of my thesis. I would also like to thank Dr. Haidar Harmanani and Dr. Azzam Mourad for giving me the honor of being part of my thesis committee. A special thanks to the President of the Lebanese American University, Dr. Joseph Jabbra along with the entire faculty of the Department of Computer Science and Mathematics at LAU.

DEDICATION

To my family,

I dedicate this work to you as a bashful notice for your love and support. Thank you for being what you are, my family!

To Christine Markarian,

You are a friend who motivated me the most to complete my masters studies and made me a better person.

IMPROVED SEARCH-TREE ALGORITHMS FOR THE CLUSTER EDIT PROBLEM

ALI KASSEM GHRAYEB

Abstract

In the Cluster Edit problem, we are asked to transform a given graph into a transitive graph, via edge deletion or addition operations, to make sure that the vertices are partitioned into a disjoint union of cliques. Cluster Edit finds application in a number of domains, including computational biology and social networks. When parameterized by the number of permitted edge-edit operation (k), the problem can be solved in O (3^k) time via a search-tree backtracking strategy. The current fastest worst-case fixed-parameter algorithm described in [7] adopts the same strategy and solves Cluster Edit in O (1.82^k).

This thesis presents new techniques to enhance any search tree- based algorithm for the Cluster Edit problem. These techniques, which include new heuristics and impose bounds on allowable edge operations per vertex, cause effective pruning of search-trees and yield noticeable improvements in experimental running times on almost all types of input instances.

Keywords: Cluster Edit, Capacitated Cluster Edit, Edit to Clique, Bounded Search-Tree, Fixed Parameterized Tractability, Clique.

TABLE OF CONTENTS

I – INTRODUCTION 1

1.1 The Cluster Edit Problem 1

1.2 Graph Theoretic Background and Terminology 2

1.3 Existing Algorithms for the Cluster Edit Problem 3

1.3.1 A Simple Search-Tree Algorithm 3

1.3.2 An O (2.62^k) Cluster Edit Algorithm 4

1.3.3 An O (2^k) Cluster Edit Algorithm 4

1.3.4 Cluster Edit in O (1.82^k) 5

1.4 Thesis Outline 6

II - A Pareto Annotation: The Add-Delete Capacities 7

2.1 Bounds on Capacities 8

2.2 Pruning Constraints 8

2.3 Pareto Annotation Rules 9

2.4 Pseudo-code of the New Algorithm 11

III - Guided Branching Using Dynamic Edge-Existence Probabilities 14

3.1 Dynamic Edge-Existence Probabilities 14

3.2 Niedermeier's Algorithm after applying the above dynamic edge-existence heuristic. 15

IV - Experimental Results 17

4.1 Implementation Strategy 17

4.2 Data Sets 17

4.3 Experimental Running Time for the Annotated Version 17

4.4 Experimental Running Time for the Edge-Existence Heuristic 19

4.5 The Hybrid Algorithm 20

V – Conclusion 24

References: 25

LIST OF TABLES

Table 1-Generic vs. Capacitated Set1 18
Table 2-Generic vs. Capacitated Set2 18
Table 3-Generic vs. Edge-Existence Heuristic Set1 19
Table 4-Generic vs. Edge-Existence Heuristic Set2 20
Table 5-Generic vs. Hybrid Set1 21
Table 6-Generic vs. Hybrid Set2 21
Table 7-Generic vs. Hybrid Set3 22

LIST OF FIGURES

Figure 1-Before applying reduction rules 10
Figure 2-After applying reduction rules.. 10

I – INTRODUCTION

1.1 The Cluster Edit Problem

Given an undirected loopless graph G = (V, E) and an integer $k \geq 0$, the Cluster Edit Problem asks to transform G into a disjoint union of cliques by deleting/adding at most k edges. Cluster Edit is NP-complete [7]. It has been in the graph theory literature since the 1980's [2].

Cluster Edit received considerable attention recently due to its highly relevant applications in many scientific areas. The recent parameterized algorithms for the problem started with a simple recursive backtracking algorithm of worst-case time complexity in $O(3^k)$ [2]. The same algorithmic techniques were used in [1] with an improved running time of $O(2.62^k)$. An $O(2^k)$ algorithm was introduced later by Bocker in [7]. Theoretically, the best published algorithm, in terms of worst-case running time, runs in $O(1.82^k)$ [7]. This latter algorithm is based on the previous $O(2^k)$ algorithm of [7], but uses more sophisticated search techniques.

The weighted version of Cluster Edit assumes a cost for deleting an edge or inserting a new one. In other words, the input graph is given with a function that maps each pair of vertices to a number that indicates the said cost. Several heuristics were developed for Weighted Cluster Edit (see [8], [9], [10], and [11] for a sequence of known heuristic techniques). The problem is APX-hard, and has a constant-factor approximation of 2.5 [12]. This thesis does not consider approximation strategies, but rather focuses on exact algorithmic techniques that work well in practice.

Cluster Edit has been a subject of interest to many applications, including computational biology, social networks, marketing, machine learning, data mining,

document clustering and chemistry. In computational biology, the problem can be used for finding families of orthologous genes, or analyzing gene expression data for tissue classification [1]. Cluster Edit has been efficiently used to group customers with similar behavior into different types for efficient marketing (Arabie and Hubert, 1994). It has been also used to group services delivery engagements for workforce management and planning. In 2004, Bansal [2] explored Cluster Edit as a way to solve the Document Clustering Problem which aims to arrange documents in groups according to a function that describes the similarity of the documents. Finally, one of the most recent areas of interest, which is highly relevant to Cluster Edit, is Social Networks: a cluster represents a collection of individuals with dense friendship patterns internally and sparse friendship patterns externally. In such networks, Cluster Edit can be used to recognize communities within large groups of closely related people and to draw other meaningful information.

1.2 Graph Theoretic Background and Terminology

A **graph** G is a pair of sets G = (V, E). V is the set of **vertices** and the number of vertices n = |V| is the **order** of the graph. The set E contains the **edges** of the graph. In an undirected graph, each edge is an unordered pair {v, w}. The vertices v and w are called the **endpoints** of the edge. The term **non-edge is used to designate** a pair of vertices that are not connected by an edge. A **path** is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence. Two vertices are **connected** if there is a path between them. A graph is connected if any two of its vertices are connected. A connected subgraph (V',E') is maximally connected if none of the vertices in V-V' is connected to a vertex in V'. If a graph is not connected,

then its maximally connected subgraphs are its ''connected components.'' A set of connected vertices where each vertex has edges to exactly two other vertices is called a **circle**. Two vertices are called **adjacent** if they share a common edge. The **neighborhood** of a vertex v is the set of vertices adjacent to v and is denoted as $N(v)$. The **degree** of a vertex v is the total number of vertices adjacent to v and is denoted as d (v). A **clique** in a graph is a set of pair-wise adjacent vertices. In the simplest case a clique consists of only one vertex. A graph consisting of a union of cliques, where a vertex can only be a member of exactly one clique is called a **cluster graph**. In other words, a cluster graph is a graph whose connected components are cliques.

In addition to the above common graph theoretic terminology, we shall use some terms that help us describe our methods. In this thesis, an edge that is not allowed to be in the cluster graph is called a **forbidden edge.** On the other hand, an edge that is decided to be in the solution graph is called a **permanent edge**. To **cliquify** means to transform a set of vertices into a clique by adding/removing edges. A set of three vertices connected by exactly two edges is called a **conflict triple**. A conflict triple can never be a part of a cluster graph.

1.3 Existing Algorithms for the Cluster Edit Problem

1.3.1 A Simple Search-Tree Algorithm

The current best-known implemented algorithm for Cluster Edit runs in O (3^k) time. The algorithm finds a conflict triple in the input graph and resolves it by exploring two cases per input instance: either by inserting a missing edge and proceeding recursively; or deleting one of the two existing edges of the triple. In the sequel, this latter behavior is denoted by branching on a conflict triple. This means that

a number of decision choices are recursively explored, based on all the possible choices for resolving the conflict.

In this simple algorithm, the cost of deleting or inserting an edge is added to the accumulated edit cost for the current branch of the search tree. The algorithm continues to branch until all conflict triples have been resolved or there is no branch where the accumulated edit cost is less than or equal to k, the allowable edit budget. This yields to an algorithm with a search tree of size O (3^k) (three choices per triple).

1.3.2 An O (2.62^k) Cluster Edit Algorithm

The O (2.62^k) algorithm improves only the worst-case behavior of the previous one. Again, the algorithm finds a conflict triple and arbitrarily selects one of the two edges in the conflict triple to branch on. The branching occurs in two ways, either (a) the edge is marked as forbidden or (b) the vertices of the edge are merged. In either way, the edit cost of the operation is added to the accumulated total edit cost. The algorithm continues to branch until either all conflict triples have been resolved or no branch with an edit cost of at most k remains.

1.3.3 An O (2^k) Cluster Edit Algorithm

The O (2^k) algorithm improves again the worst-case behavior only. It finds the edge in the graph with the lowest "branching number" and branches on it. This is done by either following (a) the branch where the selected edge has been deleted or (b) the branch where the Cluster Edit vertices of the selected edge have been merged. In both cases, the edit cost of the deletion (a) or merge (b) is added to the accumulated total edit cost of the whole graph.

The algorithm continues to branch using this approach until either all branching numbers are infinite or there is no branch that resolves the input graph with an accumulated edit cost of at most k. If all the branching numbers are infinite, a solution is found. This is because in a cluster graph, all pairs of vertices will either have zero cost for merging the vertices (if they are in the same cluster) or zero cost for forbidding the edge between the vertices (if they are in different clusters). The costs for forbidding and merging two vertices are the components of the branching vector. If one of these two is zero, the branching vector gives an infinite branching number.

What distinguishes the O (2^k) algorithm from its predecessor is that the former examines the whole graph while the latter only considers the first observed conflict triple.

1.3.4 Cluster Edit in O (1.82^k)

The current fastest published Cluster Edit algorithm runs in O (1.82^k). It is based partly on the previous algorithm but uses a larger set of more specific branching rules. While the previous algorithm only branches on the edge with the lowest branching number, the O (1.82^k) algorithm applies five more complex branching rules. Branching stops when each connected component of the graph is any of the following: path, cycle, a graph of size at most four, weak clique or a clique minus one edge. The algorithm stops (and possibly backtracks) if neither of the branching rules can be applied further to the remaining isolated subgraphs. The clustering is complete in the case of a weak clique. In the case of a path or a circle, the cost of resolving the remaining graph is found via dynamic programming. In the case of a clique minus an edge, the cost is determined by the minimum cost of adding the missing edge or the

minimum cost of splitting the almost-clique in two, with the vertices of the missing edge on either side of the splitting. To find the splitting cost, a min-cut algorithm is used. Brute force is used for graphs consisting of four vertices.

1.4 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 presents our first technique: "Pareto Annotation: the add-delete capacities." Chapter 3 presents the second technique "Branch-Guiding using dynamic edge-existence probabilities". Chapter 4 discusses experimental results that show the effectiveness of our techniques. Chapter 5 is devoted for the conclusion.

II - A Pareto Annotation: The Add-Delete Capacities

In almost all real applications of Cluster Edit, an input instance may result from imprecise data collection. The fact that a Cluster edit input is not exactly a cluster graph is often attributed to noise. The amount, or rate, of noise differs from one application to the other. Since we do not delete vertices, each vertex is assumed to be part of a clique-cluster. The number of edges to add or delete per vertex is controlled by the amount of expected noise or expected effect of noise on each vertex. To model the expected noise, we introduce a more general version of Cluster Edit, dubbed Capacitated Cluster Edit, in which vertices are annotated with their add and delete capacities. These capacities bound the number of edge modifications a vertex can experience during the search for a solution. For a vertex u, the add capacity $c+ (u)$ represents the maximum allowed number of edges that can be added to u. Similarly, the delete capacity $c- (u)$ represents the maximum number of edges that can be deleted from u. These capacities represent the maximum number of false positives (wrong edge to be deleted) and false negatives (missing edge) that could be attributed to noise.

In addition to representing noise, the add-delete capacities can be used to provide a Cluster Edit instance with the knowledge about the data. Examples include power, error tolerance, ability, similarity, domination and other social characteristics depending on the clustering problem.

In the rest of this chapter, we will illustrate how the introduced annotation technique works as we discuss implicit bounds on the capacities, *pruning constraints*, the constraints that help find a no-instance quickly, and *reduction rules* that reduce the parameter k without branching.

2.1 Bounds on Capacities

Since k is the maximum number of edge modifications, $c+(u)$ can have a value between 0 and Min(k, N-d(u)) where N is the number of vertices in the input graph and d(u) is the number of neighbors of u. $c-(u)$ can have a value between 0 and d(u).

2.2 Pruning Constraints

The pruning constraints improve the running time because, as the name suggests, they prune the search tree by (sometimes early) detection of failure. The constraints adopted in this work are:

Pruning constraint 1: If u and v are two non-adjacent vertices such that $c+(u) = 0$ or $c+(v) = 0$, then: if the algorithm decides in one of its branches to add edge uv, then failure is reported. *Backtrack.*

The soundness of this constraint is obvious since u and v cannot be in the same clique because edge uv cannot be added.

Pruning constraint 2: If u and v are two adjacent vertices and $c-(u) = 0$ or $c-(v) = 0$, then: if the algorithm decides to delete u and v, report failure. *Backtrack.*

The soundness of this constraint is obvious since u and v cannot be isolated because edge uv cannot be deleted.

Pruning constraint 3: Let d(u) and d(v) represent the current degree of u and v respectively. If u and v are not adjacent and $d(u) - c-(u) > d(v) + c+(v)$ or $d(v) - c-(v) > d(u) + c+(u)$, then: if the algorithm decides to add edge uv, report failure. *Backtrack.*

Soundness: Two vertices cannot be in the same clique unless they have (eventually) the same degree. If c-(u) edges (the maximum permitted) were deleted from u, then the number of edges connected to u in the cluster graph is bounded below by $d(u) - c-(u)$.

If $c+(v)$ edges were added to v, then the number of edges connected to v would be bounded above by $d(v) + c+(v)$. If $d(u) – c-(u) > d(v) + c+(v)$ or $d(v) – c-(v) > d(u) + c+(u)$, then u and v have different degrees in the target solution, which makes it impossible for them to be in the same clique.

2.3 Pareto Annotation Rules

The following four reduction rules are due to our introduced annotation. They help reduce the parameter k without branching:

Rule 1 (PAR 1)

Let uv and uw be two *permanent* edges, then edge vw must be *permanent*.

Rule 2 (PAR 2)

Let edge uv be a *permanent* edge and uw be a *forbidden* edge, then edge vw must be *forbidden*

Rule 3 (PAR 3)

Let u be a vertex satisfying $c-(u) = 0$, then cliquify the neighborhood of u. This is equivalent to setting every edge incident on u as permanent and applying PAR1.

This rule is sound since u and all its neighbors must be in the same cluster.

Rule 4 (PAR 4)

Let u be a vertex with $c+(u) = 0$, then for every non-neighbor vertex v of u, set edge uv as *forbidden.*

This rule is sound since u will not be joined to any other vertex.

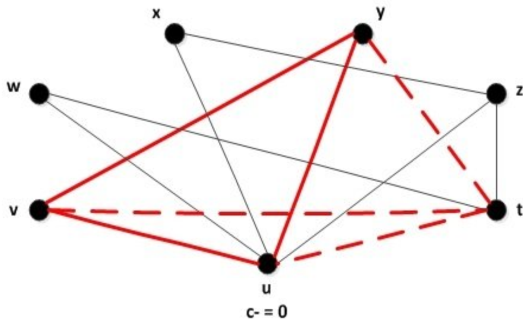


Figure 1-Before applying reduction rules

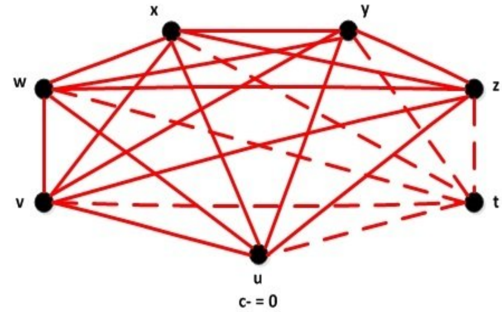


Figure 2-After applying reduction rules

Figure 1 shows six connected vertices where deletion capacity of vertex *u* is zero.(c-(u) =0). The "black" edge represents a *normal* edge that connects two vertices; the "bold red" edge represents a *permanent* one; and the "dashed bold red" edge represents a *forbidden* edge between two vertices.

By applying **"PAR 3"**, we end up with a cluster of five vertices *u, v, x, y*, and *z.* This is done by setting every "black" edge connected to *u* as *permanent,* applying **"PAR 3"** to the vertices of the cluster, then applying **"PAR 4"** between vertex *t* and all the vertices of the cluster. The overall process of edge modification will result in reducing the edge modification parameter by 15!

Figure 2 shows the graph which results after applying **"PAR 3"** to *u* where vertices *u, v, x, y* and *z* are *permanently* connected forming one cluster and *t* is *forbiddingly* isolated from the cluster.

Branch on conflicts that have Permanent and forbidden edges.

As shown in the above two examples, applying the reduction rules on vertices that have zero capacities significantly reduces the edge modification parameter without branching. In order to have such vertices more frequently in our graph while branching and to benefit from the cascading property that is ascertained by rule 1 and 2 while

modifying edges, we always branch on conflict triples that have one of their edges either permanent or forbidden.

2.4 Pseudo-code of the New Algorithm

The following pseudo-code shows our modifications to Niedermeier's algorithm due to introducing the add-delete capacities and applying the corresponding rules

PseudoCode

```
Array of vertices ZeroAdditionCapacities
Array of vertices ZeroDeletionCapacities
Graph G
Vertices u, v, w, x, y, z.
Int K

ClusterEdit(G,K)
If K>= 0
    Find a conflict (u,v,w). Assume uv be the missing edge.
    Add (u,v)
    ClusterEdit (G,K)
    BacktrackAddition (u,v)
    Delete (w,u)
    ClusterEdit (G,K)
    BacktrackDeletion(w,u)
    Delete (w,v)
    ClusterEdit (G,K)
    BacktrackDeletion(w,v)

Add(u,v)
    If c+(u) = 0 or c+(v) = 0
        Return
    If d(u) – c-(u) > d(v) + c+(v) or d(v) – c-(v) > d(u) + c+(u)
        Return
    G = G union (u,v)
    K = K -1
    Set uv to permanent
    c+(u) = c+(u) -1 and c+(v) = c+(v) – 1
    if c+(u) = 0
```

```
        Add u to ZeroAdditionCapacities
    If c+(v) = 0
        Add v to ZeroAdditionCapacities
    If ZeroAdditionCapacities is not empty or ZeroAdditionCapacities is not
    empty
        UpdateZeroCapacities
```

Delete(u,v)

```
If c-(u) = 0 or c-(v) = 0
        Return
    If d(u) – c-(u) > d(v) + c+(v) or d(v) – c-(v) > d(u) + c+(u)
        Return
    G = G /(u,v)
    K = K -1
    Set uv to forbidden
    c-(u) = c-(u) -1 and c-(v) = c-(v) – 1
    if c-(u) = 0
        Insert u to ZeroDeletionCapacities
    If c-(v) = 0
        Insert v to ZeroDeletionCapacities
    If ZeroAdditionCapacities is not empty or ZeroAdditionCapacities is not
    empty
        UpdateZeroCapacities
```

UpdateZeroCapacities

```
    For each vertex x in ZeroAdditionCapacities
        For each vertex y neighbor to x and xy doesn't belong to G
            Set xy to forbidden then apply R1 and R2
    For each vertex x in ZeroDeletionCapacities
        For each vertex y neighbor to x and xy belong to G
            Set xy to permanent then apply R1 and R2
```

BacktrackAddition(u,v)

```
    G =G\(u,v)
    K = K + 1
```

BacktrackDeletion(u,v)

```
    G =G union (u,v)
    K = K + 1
```

To show the efficiency of the above reduction rules and constraints, they were implemented and tested. The results are presented in Chapter 4.

III - Guided Branching Using Dynamic Edge-Existence Probabilities

Why to branch randomly if the current state of the graph can inspire us about promising areas of the search tree? Thus, based on the current state of the graph, we introduce a heuristic that guides the branching towards a smaller search tree (i.e., faster search algorithm).

As mentioned before, for simplicity, we use the basic algorithm, which looks for a conflict and tries to resolve the conflict either by deleting one of its existing edges or by adding the missing one.

3.1 Dynamic Edge-Existence Probabilities

In order to guide the branching of this algorithm, an edge-existence probability is calculated for each edge of a conflict triple. As the name suggests, this is a value that tries to measure the likelihood of an edge to be in the solution graph.

The existence probability of an edge uv, p(*uv*), is a number between *zero* and *one* that is calculated based on the structure of the edges between vertices that have a distance *one* or *two* from *u* or *v*. So once one of these edges is modified, p(*uv*) is modified and has to be recalculated. That is why we refer to this heuristic technique as "dynamic edge existence probabilities."

In order for an edge to exist in the solution graph, its two end-points must have the same neighbors. Moreover, these neighbors must be connected. Based on this, we calculate the dynamic edge-existence probabilities, for an edge uv, as follows:

Let **n1** be the number of common neighbors of u and v.

Let **n2** be the sum of all neighbors of u and v.

Then **r1 = n1/n2** represents the ratio of common neighbors of u and v over the total number of neighbors.

Note that r1 = 1 when all pairs of common neighbors of u and v are adjacent and hence uv is more likely to be in the solution graph.

Now let **e1** be the number of edges between the common neighbors of u and v.

Let **e2** be the total number of edges that must be between the common neighbors of u and v for *uv* to exist in the solution graph.

So **e2 = n1*(n1-1)/2**.

Then **r2 = e1/e2** is close to *one* when *uv* is more likely to be in the solution graph and close to zero otherwise.

Since the edge existence of *uv* depends on both ratios, **p(uv) = αr1 + βr2** where **α + β = 1.**

Logically, **α** should be greater than **β** and experiments have shown that the best values for **α** and **β** are **0.8** and **0.2**.

3.2 Niedermeier's Algorithm after applying the above dynamic edge-existence heuristic

PseudoCode

```
Graph G
Int K
P(uv) represents the dynamic edge existence of edge uv

ClusterEdit( G , K)
If K ≥ 0
        Find a conflict (u,v,w). Assume uv is the missing edge.
        If p(uw) ≤ p(vw) and p(vw) ≤ 1 – p(uv)
                ClusterEdit( G\(u,w), K-1)
                ClusterEdit( G\(v,w), K-1)
                ClusterEdit( G U (v,w), K-1)
        If p(uw) ≤ 1 – p(uv) and 1 – p(uv) ≤ p(vw)
```

```
        ClusterEdit( G\(u,w), K-1)
        ClusterEdit( G U (v,w), K-1)
        ClusterEdit( G\(v,w), K-1)

If p(vw) ≤ p(uw) and p(uw) ≤ 1 – p(uv)
        ClusterEdit( G\(v,w), K-1)
        ClusterEdit( G\(u,w), K-1)
        ClusterEdit( G U (v,w), K-1)

If p(vw)≤ 1 – p(uv) and 1 – p(uv) ≤ p(uw)
        ClusterEdit( G\(v,w), K-1)
        ClusterEdit( G U (v,w), K-1)
        ClusterEdit( G\(u,w), K-1)

If 1 – p(uv) ≤ p(uw) and p(uw) ≤ p(vw)
        ClusterEdit( G\(u,w), K-1)
        ClusterEdit( G U (v,w), K-1)
        ClusterEdit( G\(v,w), K-1)

If p(vw)≤ 1 – p(uv) and 1 – p(uv) ≤ p(uw)
        ClusterEdit( G\(v,w), K-1)
        ClusterEdit( G U (v,w), K-1)
        ClusterEdit( G\(u,w), K-1)
```

The dynamic edge existence probabilities were implemented on the algorithm of O (3^k) and experiments show the huge difference in the running time. Details about the implementation and the successful results are shown in Chapter 4.

IV - Experimental Results

4.1 Implementation Strategy

We implemented the basic recursive backtracking algorithm that branches on conflict triples, as described in [2]. We used adjacency matrices to represent graphs. This choice was made due to the heavy usage of the adjacency query in our code. We implemented three algorithms. The first implements our annotated version only, the second implements the edge-existence heuristic, while the third is a hybrid of the two.

Evaluation platform: all algorithms were implemented using C language. Running times were measured on an AMD Opteron-2.327 GHz with 2 GB of memory.

4.2 Data Sets

In the absence of publicly available un-weighted graph datasets that meet our requirements, we used synthetic data generated using the following algorithm.

Random un-weighted graphs: Given the number of vertices N, the maximum number of edge modification k, and the number of clusters C that should be present in the graph, we randomly group the N vertices into C groups where the size of each group is randomly selected such that the sum of all sizes is N. Then we fully connect all vertices of each group each forming a cluster. After that we choose k distinct vertex pairs *(u,v)* and delete edge uv if u and v are in the same cluster or insert the edge *uv* if u and v are in two different clusters.

4.3 Experimental Running Time for the Annotated Version

In order to test the efficiency of the add-delete capacities, several instances were generated using the above random graph generator and were ran on two different codes.

The first is an implementation of the generic branching algorithm of [2]. The second is an implementation of the same algorithm, equipped with the add-delete capacities as shown in the pseudo-code described in Chapter 2.

# of Edge Modification = 50								
# of Vertices = 100								
# of clusters	3	4	5	6	7	8	9	10
Generic Branching Runtime	0.05	0.04	13.7	33.4	2033	> 1d	> 1d	> 1d
Capacitated Algorithm Runtime	0.02	0.02	0.41	1.3	1.6	1.6	2.05	2.1

Table 1-Generic vs. Capacitated Set1

# of Edge Modification = 50										
# of Vertices = 200										
# of clusters	5	6	7	8	9	10	11	12	13	14
Generic Branching Runtime	0.62	0.52	0.6	0.6	0.81	19.43	25.71	241.04	2192.79	16393.76
Capacitated Algorithm Runtime	0.46	0.45	0.45	0.36	0.77	9.06	12.64	5.61	38.41	13.54

Table 2-Generic vs. Capacitated Set2

Tables 1 and 2 show the power of the capacitated version, which is several hundred times faster than the original one. Some instances of the original algorithm did not finish in one day, so we had to interrupt the experiments, while solutions were found by the capacitated one in a few seconds.

The results shown in the above tables are obtained on a number of instances. So we report the average running times. In table 2, all codes completed the search and we were able to measure the running times. The ratio of the running time shows that the capacitated version is 290 times faster than the original one.

4.4 Experimental Running Time for the Edge-Existence Heuristic

We use the same input instances as in the case of the algorithm based on add-delete capacities. The following tables show the efficiency and power of the edge existence heuristic. Again, the basic algorithm of Niedermeier et al. was compared to a tuned version that guides the branching via the edge-existence heuristic as described in the pseudo-code of chapter 3.

# of Edge Modification = 50								
# of Vertices = 100								
# of clusters	3	4	5	6	7	8	9	10
Generic Branching Runtime	0.05	0.04	13.7	33.4	2033	> 1d	> 1d	> 1d
Edge-Existence Heuristic Algorithm Runtime	0.04	0.04	1.3	1.4	12.7	18.6	55.43	124.98

Table 3-Generic vs. Edge-Existence Heuristic Set1

# of Edge Modification = 50										
# of Vertices = 200										
# of clusters	5	6	7	8	9	10	11	12	13	14
Generic Branching Runtime	0.62	0.52	0.6	0.6	0.81	19.43	25.71	241.04	2192.79	16393.76
Edge-Existence Heuristic Algorithm Runtime	0.35	0.35	0.36	0.37	0.77	8.16	14.22	36.61	88.32	176.12

Table 4-Generic vs. Edge-Existence Heuristic Set2

As the above tables show, the edge-existence heuristic algorithm outperforms the generic one in several hundred times. Many tested instances took more than one day without finishing when using Niedermier's algorithm. Our tuned version managed to solve such instances in minutes.

4.5 The Hybrid Algorithm

Our hybrid algorithm includes the add-delete capacities to each vertex and branching is guided using the edge-existence probabilities. Several instances were tested. Instances and results are shown in table 5, table 6, and table 7.

# of Edge Modification = 75										
# of Vertices = 200										
# of clusters	5	6	7	8	9	10	11	12	13	14
Capacitated Algorithm Runtime	0.48	0.66	108.46	1210.33	>5h	>5 h	>5h	>5h	>5h	>5h
Hybrid Algorithm Runtime	0.37	0.57	9.51	6.59	48.52	54.92	65.15	63.05	87.3	117

Table 5-Generic vs. Hybrid Set1

# of Edge Modification = 100										
# of Vertices = 200										
# of clusters	5	6	7	8	9	10	11	12	13	14
Capacitated Algorithm Runtime	44	74	83	35	303	227	357	264	351	481
Hybrid Algorithm Runtime	1.2	2.1	2.71	2.9	3.96	4.3	4.3	4.8	5.1	5.6

Table 6-Generic vs. Hybrid Set2

# of Edge Modification = 100										
# of Vertices = 500										
# of clusters	5	6	7	8	9	10	11	12	13	14
Capacitated Algorithm Runtime	4.5	6.1	4.4	6.3	5.7	5.4	8.4	10	117.3	158
Hybrid Algorithm Runtime	2.6	4.2	2.54	4.48	3.9	3.8	3.7	5.1	12.1	24.6

Table 7-Generic vs. Hybrid Set3

The above results prove the significant improvement incurred by introducing capacities and using the edge-existence heuristic together. Even though the capacitated version is fast and can solve hard instances, the hybrid is much faster and hard instances are solved in few seconds. Table 5 shows some hard instances that took more than 5 hours without finding a solution by the capacitated version, while it took an average of 73 seconds to find a solution by the hybrid version.

Using tables 6 and 7, the average runtime for both algorithms was calculated and results show that the hybrid version is about 24 times faster than the capacitated one.

Experiments show that the two techniques “Pareto-Annotation” and “Edge – Existence Probabilities” perform faster on more dense graphs than on sparse graphs. As we can see in table 6 and table 7, once the number of clusters increases i.e. the graph gets more sparse or less dense, the hybrid and the capacitated versions take more time while the edge modification variable and the number of vertices is still fixed. This is

because the probabilities are more accurate on dense graphs than sparse ones. Moreover, because capacities can serve better in pruning the search tree, by reducing the edge modification variable faster when a vertex gets zero capacity, on dense graphs.

V – Conclusion

We have presented two new search techniques that lead to a faster experimental performance for the un-weighted Cluster Edit problem. The add-delete capacities play the central role of pruning the search tree by enhancing a forward-checking method that allows for early detection of failure (or dead end), and by reducing the edge modification variable faster. Besides enhancing the experimental performance, the introduced add-delete capacities can model applications more realistically, especially in social networks applications. The edge-existence probability technique helps in guiding the search during branching, hence targeting better (or promising) areas in the search tree.

We implemented our approaches based on the simple algorithm of Cluster Edit that runs in O (3^k) time. We tested the performance on randomly generated instances. Results showed that our techniques are much faster than the original algorithm on all tested instances.

References:

1. S. Bocker, S. Briesemeister, Q. B. A. Bui, and A. Trub, ”A fixed-parameter approach for weighted cluster editing”, In *Proceedings of Asia-Pacific Bioinformatics Conference,* London, 2008, pp. 211-220.

2. R. Niedermeier and J. Guo. (2006). Fixed-parameter algorithms. [Online]. Available: http://theinf1.informatik.uni-jena.de/teaching/niedermeier-guo-fpt.pdf.

3. J. Frisvad, ”Cluster Analysis“, in *Cluster analysis for researchers,* Belmont: H.C. Romesburg, 1984, pp. 49-60.

4. N. Entwistle and T. Brennan, " Types of successful students ", *British Journal of Educational Psychology,* vol. 41, no. 3, pp. 268-276, 1971.

5. R. Shamir, R. Sharan, and D. Tsur, “Cluster graph modification problems“, *Discrete Applied Mathematics*, vol. 144, pp. 173–182, 2004.

6. J. Gramm, J. Guo, F. Hufner, and R. Niedermeier, “Automated generation of search tree algorithms for hard graph modification problems“, *Algorithmica*, vol. 39, no. 4, pp. 321-347, 2004.

7. S. Bocker, S. Briesemeister, and A. Truss, “Going weighted: Parameterized algorithms for cluster editing”, *Lecture Notes in Computer Science*, vol. 5165, pp. 1–12, 2008.

8. A. Ben-Dor, R. Shamir, and Z. Yakhini,“Clustering gene expression patterns“, *Journal of Computational Biology*, vol. 6, no. 4, pp. 281-297, 1999.

9. E. Hartuv, A. Schmitt, J. Lange, S. Meier-Ewert, H. Lehrach, and R. Shamir, “An algorithm for clustering CDNA fingerprints“, *Genomics*, vol. 66, no. 3, pp. 249-256, 2000.

10. R. Sharan, A. Maron-Katz, and R. Shamir, “CLICK and EXPANDER: a system for clustering and visualizing gene expression data”, *Bioinformatics*, vol. 19 no. 14, pp. 1787-1799, 2003.

11. T. Wittkop, J. Baumbach, F. Lobo, and S. Rahmann,"Large scale clustering of protein sequences with Force - a layout based heuristic for weighted cluster editing", *Bioinformatics*, vol. 8, no. 1, pp. 396, 2007.

12. A. Zuylen and D. Williamson, "Deterministic algorithms for rank aggregation and other ranking and clustering problems", In *Proceedings of Workshop on Approximation and Online Algorithms*, 2007, pp. 77-84.