The Cluster Editing Problem: Implementations and Experiments *

Frank Dehne[1], Michael A. Langston[2], Xuemei Luo[1],
Sylvain Pitre[1], Peter Shaw[3] and Yun Zhang[2]

[1] School of Computer Science, Carleton University, Ottawa, Canada
[2] Department of Computer Science, University of Tennessee, Knoxville, TN, USA
[3] Department of Computer Science, University of Newcastle, Newcastle, Australia

Abstract

In this paper, we study the cluster editing problem which is fixed parameter tractable. We present the first practical implementation of a FPT based method for cluster editing, using the approach in [6, 7], and compare our implementation with the straightforward greedy method and a solution based on linear programming [3]. Our experiments show that the best results are obtained by using the refined branching method in [7] together with interleaving (re-kernelization). We also observe an interesting lack of monotonicity in the running times for "yes" instances with increasing values of k.

1 Introduction

The *CLUSTER EDITING* problem is defined as follows. *Input*: An undirected graph $G = (V, E)$, and a non-negative integer k. *Question*: Can we transform G, by inserting and deleting at most k edges, into a graph that consists of a disjoint union of cliques? The *CLUSTER EDIT DISTANCE* for a graph G is the smallest k for which cluster editing is possible.

Our main target applications are centered around computational biology. We are in particular interested in the analysis of putative gene co-regulation (transcriptomics via microarray analysis), putative gene product co-occurrence (proteomics via mass spec or MALDI), and pathway/network elucidation in data from synthetic genetic arrays (double knockout arrays). In all these applications, the underlying biological data is very expensive and, in some cases, requires years to produce. For example, RI strains take more than 8 years to make isogenetically pure, pathway/network elucidation for yeast requires 4,5000 synthetic genetic arrays, and Affy U133 arrays contain more than 30k probesets. Because of the high value of the underlying biological data, saving computation time for the analysis by using approximation is often not acceptable. Hence, we turn to FPT based approaches for solving such problems. The cluster editing problem is an important such problem in the context outlined above.

In this paper, we present the first practical implementation of a FPT based method for cluster editing, using the approach in [6, 7]. In order to evaluate the effectiveness of the FPT approach, we also implemented a well known previous method based on linear programming [3] as well as a greedy approach for cluster editing. The total programming effort was approx. 120 person hours, producing approx. 2500 lines of code.

* This research has been supported in part by the Natural Sciences and Engineering Research Council of Canada and by the U.S. National Institutes of Health under grants 1-R01-MH-074460-01, 1-P01-DA-015027-01 and U01-AA13512.

Our experiments show that the best results for cluster editing are obtained by using the refined branching method in [7] together with interleaving (re-kernelization). Our experiments show that the refined branching method in [7] is vastly superior to the basic branching method, which is not obvious because the refined branching method is considerably more complicated and incurs larger constant factors. We also demonstrate that, in practice, branching with interleaving is indeed decidedly faster than branching without interleaving.

A surprising observation from our experiments comes with respect to the problem of determining the optimum edit value, k. In practice, we do of course not know k and the general approach would be to determine k via binary search. Things turns out to be quite different with cluster editing. If we happen to be advancing from below (that is, solving a "no" instance), then as e.g. with vertex cover run times predictably increase with rising parameter values. On the other hand, if we are advancing from above (facing a "yes" instance), then run times may decrease, increase or even stay the same with sinking parameter values. This is very different from the standard behavior e.g. for vertex cover. For cluster editing it is not the case that run times are highest around the optimum value of k. We conclude that a binary search may not be the best way to implement an FPT based approach for clustering editing, and that one may be better off steering the parameter as much as possible from below.

The remainder of this paper is organized as follows. Section 2 outlines our experimental setup used throughout the paper. Section 3 outlines our implementation of LP based and greedy based methods for cluster editing which serve as a baseline for evaluating the FPT based approach. In Section 4 we present a first practical implementation of an FPT based approach for cluster editing. Section 5 presents our experimental results for the FPT based approach and Section 6 concludes the paper.

2 Experimental Setup

In the remainder of this paper, we compare the performance of various algorithms under various criteria. Experiments were performed on a Dell OptiPlex GX280 using a 3.2GHz Pentium 4 dual processor, with 1.0 gigabytes (GB) of SDRAM, and running a Linux 2.6.8-2-686-smp kernel. Algorithms were implemented in C and compiled using gcc version 3.4.4. The various costs of implementation are listed in Table 1.

Table 1. Implementation costs for the three cluster editing algorithms studied in this paper.

Algorithm	Development Time	Lines of Code	Library Used
LP-based method	45 hours	250	lp_solve version 5.5
Greedy method	15 hours	250	None
FPT approach	60 hours	2000	None

Unless otherwise stated, we used as input synthetic graphs for which we know the optimum edit distances. For this, we built a graph generator which operates as follows. Our graph generator takes as input parameters the desired number of vertices (n), clusters (c) and the required edit distance (d). First, a random size (number of vertices) in the range $[0.5(n/c), 1.5(n/c)]$ is assigned to each cluster. We produce a clique graph G' containing c fully connected cliques (clusters) of the sizes determined above, with no edges between those cliques. Then, we execute d random edits on G' resulting in an output graph G. An edit consists of randomly inserting or deleting an edge. More precisely, for each edit we randomly decide whether to insert or delete an edge. For an insert operation, we randomly chose two vertices i and j that are not connected by an edge and then add an edge (i, j). For a delete operation, we randomly select an edge in

the graph and remove that edge. Once an edge is inserted in G it cannot be deleted by a future edit. Similarly once an edge has been deleted from G it cannot be re-inserted by another edit. For random number generation we used the Mitchell-Moore algorithm as described in [8]. Note that, the above method creates in most cases, but not always, a graph G with edit distance d. In some cases, as observed in our test runs, the edit distance of G is smaller than d because a different set of clusters than those used by our generator can be created with fewer edits.

3 A Baseline for Comparisons

In order to evaluate the effectiveness of a fixed parameter tractability approach for cluster editing, we need to establish a baseline to which we can compare our implementation. One well known previous method [3] is based on linear programming. Another alternative is to use a greedy approach for cluster editing. Needless to say, both methods only provide an approximation of the edit distance.

We implemented the LP based cluster editing method described in [3]. Given a graph, we first build a LP model by setting the objective function and constraints for every pair of vertices and every triple of vertices. For each pair of vertices i and j, a partitioning into clusters can be represented with a binary variable x_{ij}, where $x_{ij} = 0$ if i and j are in the same cluster, and $x_{ij} = 1$ if they are in different clusters. The integer constraints can be relaxed to allow rational values for x_{ij}, that is, $0 \leq x_{ij} \leq 1$. For each triple of vertices i, j and k, the triangle inequality $x_{ik} \leq x_{ij} + x_{jk}$ holds because if $x_{ij} = 0$ and $x_{jk} = 0$ then $x_{ik} = 0$. The objective is to minimize the number of edge edits: the number of edges $(i, j) \in E$ for which $x_{ij} = 1$ and the number of pairs of vertices that are not adjacent $(i, j) \notin E$ for which $x_{ij} = 0$. After the LP model is built, a linear programming C library lp_solve version 5.5 is used to solve the LP model and get the values for every variable x_{ij}. Finally, the graph is partitioned into clusters based on the variables x_{ij} in the following way: two vertices i and j are put into the same cluster if $x_{ij} \leq 0.5$. The edit distance is calculated as the summation of the number of edges that are needed to be added, which do not exist but the two endpoints are in same cluster, and the number of edges that are needed to be deleted, which exist but the two endpoints are in different clusters.

We also implemented the following greedy method for cluster editing. Consider a graph G. For an edge insertion consider all possible edges, for an edge deletion consider only the edges in G. Calculate a cost for each insertion/deletion as follows. For an edge $e = (i, j)$ define the common neighborhood as the set of common neighbors of i and j, and define the non-common neighborhood as the set of non-common neighbors of i and j. For an insertion of an edge $e = (i, j)$, define the cost as the number of edge insertions required to transform the common and non-common neighborhood of e into a clique. For a deletion of an edge $e = (i, j)$, define the cost as the number of edge deletions required to disconnect the common neighborhood of e. Select the edit operation with smallest cost and iterate until a graph of disjoint cliques is obtained. To implement this greedy method, we initially mark every pair of vertices as unmarked. For each unmarked pair of vertices i and j, the smaller of the cost of having an edge between them and the cost of not having an edge between them is chosen as the cost of i and j. We select the pair with least cost, perform the edits associated, and mark the pair. This is repeated until all pairs are marked, which will give a set of connected components. The edit distance is calculated as the number of edge editions to get the set of connected components plus the number of edge editions to transform those connected components into cliques.

The results of our experimental evaluation of the LP and greedy methods are shown in Figures 1 and 2. Our experiments show that the LP based cluster editing method is consistently better than the greedy method with respect to both, the computation time and the value for k obtained.

Hence, for the remainder of this paper, we will compare our implementation of a fixed-parameter tractability approach only to the LP based cluster editing method.

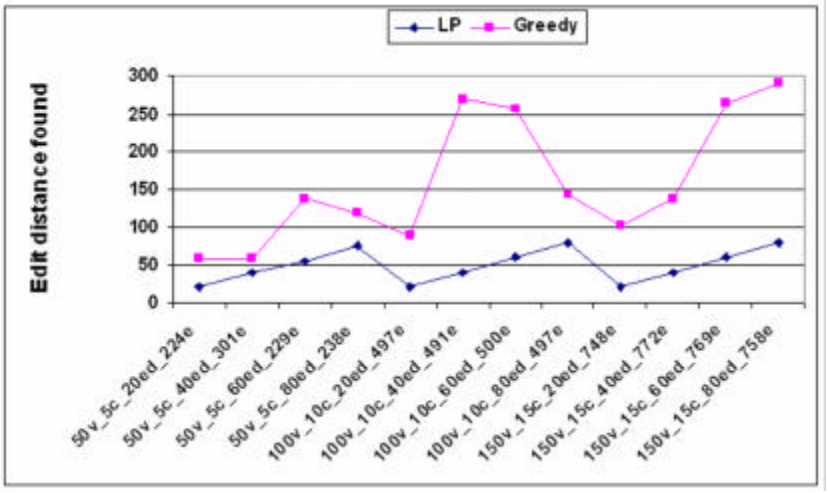

Fig. 1. A comparison of edit distances computed by LP versus the greedy method.

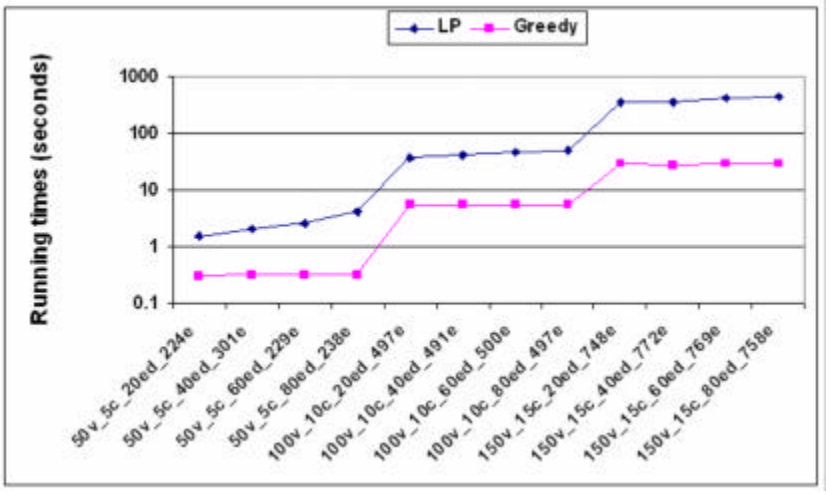

Fig. 2. A comparison of run times required by LP versus the greedy method.

4 An FPT-Based Approach

In this section we present an outline of our adaptation of the algorithm in [7, 6] that we used to obtain the first practical implementation for exact cluster edit distance computation.

Algorithm 1 Solving the Cluster Edit Problem Via a Fixed-Parameter Tractability Approach
(1) Extract highly connected (e.g., 2- and 3-connected) components. Our motivation is to eliminate sparse parts of the input. (Note: level of connectivity depends on the application.)
(2) Bound the search space for k:
  (a) Let k_{LP} be the edit distance determined by the linear programming method [3].
  (b) The search interval for the true edit distance k is $[k_{LP}/4, k_{LP}]$.
(3) For increasing k, starting with $k_{LP}/4$:
  (a) Execute the kernelization method described in [7], Section 7.2 (see also [6]).
  (b) Execute either the basic or the refined bounded tree search method described in [7], Section 10.1 or Section 10.2, respectively (see also [6]).
  (c) Use "interleaving": at each branch node in the bounded tree search, execute again the kernelization method from Step 3a.
— End of Algorithm —

Two kernelization rules for the cluster editing problem have been described. The first rule is based on the neighborhood of every pair of vertices $u, v \in V$. (1) If u and v have more than k common neighbors, then (u, v) has to belong to E; if $(u, v) \notin E$, we add it to E. (2) If u and v have more than k non-common neighbors, then (u, v) cannot belong to E; if $(u, v) \in E$, we delete it. (3) If u and v have both more than k common and more than k non-common neighbors, then the given instance has no size k solution. The other kernelization rule is to delete the connected components that are cliques from the graph.

The bounded search tree method for cluster editing is based on the observation that an induced path P_3, a path with three vertices and two edges, is forbidden for a graph consisting of disjoint cliques. Given a graph, considering any induced $P_3 = \{u, v, w\}$ with edges (u, v) and (u, w), we can branch with three cases: delete edge (u, v), delete edge (u, w), or add edge (v, w). For either case, the parameter k is decreased by one. For basic branching, this leads the search tree size of $O(3^k)$ where a resulting graph with disjoint cliques is found for k edits. At each branch node, if the parameter k goes down to be non-positive, no solution of size $\leq k$ exists on that branch. If no solution can be found on all branches, then we conclude that no solution of size k exists for the given graph.

The bounded tree search method can be improved by making a case distinction of P_3 with three cases and giving each case a branching rule. Consider a $P_3 = \{u, v, w\}$ with edges (u, v) and (u, w). There are three cases based on the neighborhood of u, v and w: (1) v and w do not share a common neighbor other than u; (2) v and w have a common neighbor x other than u, and x is adjacent to u; (3) v and w have a common neighbor x other than u, but x is not adjacent to u. For each pair of vertices, an annotation mapping is employed to facilitate the branching rules. Each vertex pair u and v is assigned one of the following annotations: "permanent" meaning $(u, v) \in E$ and (u, v) cannot be deleted, "forbidden" meaning $(u, v) \notin E$ and (u, v) cannot be inserted, or "none" meaning no information available and it can be edited. For every three vertices $u, v, w \in V$, if (u, v) and (u, w) are permanent, (v, w) has to be permanent, and if (u, v) is permanent and (u, w) is forbidden, (v, w) has to be forbidden.

Algorithm 2 Given a graph $G = (V, E)$ and parameter k, consider a $P_3 = \{u, v, w\}$ with edges (u, v) and (u, w). The refined branching strategy using the above annotation mapping works as follows.

(1) If v and w do not share a common neighbor other than u, then branch with
  (a) $(G \setminus \{(u, v)\}, k - 1)$, and
  (b) $(G \setminus \{(u, w)\}, k - 1)$.
(2) If v and w have a common neighbor $x \neq u$ and $(u, x) \in E$, then branch with five subcases:
  (c) $(G \cup \{(v, w)\}, k - 1)$;
  (d) Set (v, w) to forbidden, and branch with $(G \setminus \{(u, v), (v, x)\}, k - 2)$;
  (e) Set (v, w) to forbidden, (v, x) to permanent, and branch with $(G \setminus \{(u, v), (u, x), (w, x)\}, k - 3)$;
  (f) Set (v, w) to forbidden, and branch with $(G \setminus \{(u, w), (w, x)\}, k - 2)$;
  (g) Set (v, w) to forbidden, (w, x) to permanent, and branch with $(G \setminus \{(u, w), (u, x), (v, x)\}, k - 3)$.
(3) If v and w have a common neighbor $x \neq u$ and $(u, x) \notin E$, then branch with five subcases:
  (h) $(G \setminus \{(u, v)\}, k - 1)$;
  (i) Set (u, v) to permanent, (v, w) to forbidden, and branch with $(G \setminus \{(u, w), (v, x)\}, k - 2)$;
  (j) Set (u, v) to permanent, (v, w) to forbidden, (v,x) to permanent, and branch with $(G \cup \{(u, x)\} \setminus \{(u, w), (w, x)\}, k - 3)$;
  (k) Set (u, v) and (u, w) to permanent, and branch with $(G \cup \{(v, w)\} \setminus \{(w, x), (v, x)\}, k - 3)$;
  (l) Set (u, v) and (u, w) to permanent, and branch with $(G \cup \{(v, w), (u, x)\}, k - 2)$.

— End of Algorithm —

Initially, all vertex pairs are set to "none". When an edge is added it is set to "permanent", and when an edge is deleted it is set to "forbidden". The algorithm also stops when the parameter k reaches 0 or below or when the graph G contains no induced P_3. The search tree size for the refined branching strategy is $O(2.27^k)$.

For both basic and refined bounded tree search methods, we applied the kernelization method at each branch node. Both methods are implemented as recursive functions. For future improvement, we plan to implement them as iterative functions to achieve better performance.

5 Experimental Results

To gauge the practical merit of an approach based on fixed-parameter tractability, we tested the methods just described against each other and against our LP implementation on a variety of both synthetic and real graphs. We have already described the process by which we generate synthetic graphs. By "real" graphs, we mean those that naturally arise in application domains, in the present case, from protein domain sequence similarity. In all cases we report branching times only because the time needed for initial preprocessing and kernelization is insignificant compared to that required during branching.

It seems that refined branching is vastly superior to basic branching. The run times reported in Figure 3, where the edit distance is set to 20, are typical of those we observe. This was not obvious in advance. It is simply not always the case that asymptotically faster methods in the worst case translate into better algorithms in the average case. Unless the data is contrived, additional overhead and complexities incurred by ever more sophisticated branching strategies can ofttimes negate any real gains in efficiency.

It also seems that branching with interleaving is decidedly faster than branching without interleaving. The run times reported on larger instances in Figure 4, where the edit distance is

set to 40, are typical. Again, this makes sense, but is neither obvious nor necessarily the case in general.

Of course we do not know the optimum edit value in advance, and so generally determine it by performing a binary search. One might expect run times to be rather predictable. With vertex cover, for example, we have long observed [1, 2] that the most difficult computations are centered around the point at which a "no" instance becomes a "yes" instance, with the "no" the harder of the two (with "no," our algorithms cannot find a solution and halt early). Because we are interested in clique, the situation is reversed but still monotonic above and below the optimum value. A standard example is illustrated in Figure 5.

Things turn out to be quite different with cluster editing. If we happen to be advancing from below (that is, solving a "no" instance), then as with vertex cover run times predictably increase with rising parameter values. On the other hand, if we are advancing from above (facing a "yes" instance), then run times may decrease, increase or even stay the same with sinking parameter values. See Figures 6 and 7.

In retrospect, we find the answer to this conundrum lies in the way solutions are distributed and the way branching works with parameter values above the optimum. With vertex cover, for example, a cover of size i ensures a cover of size $i + 1$ (as long as $i < |V|$). With cluster editing, however, it is conceivable that there is a solution with edit distance i yet no solution with distance $i + 1$ (or $i + 2$ and so forth). Thus a higher parameter value may mean only that a cluster editing algorithm has to do more work. See Figure 8, which depicts search tree traversals on the graph used to report run times in Figure 7. From this we conclude that a binary search may not be the best way to implement an FPT-based approach for clustering editing, and that one may be better off steering the parameter as much as possible from below.

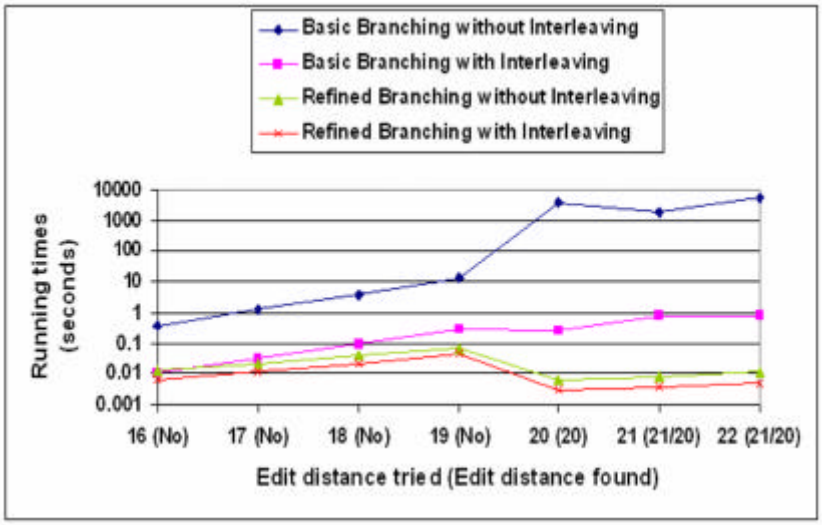

Fig. 3. FPT run times on a graph with 50 vertices, 5 clusters and edit distance 20.

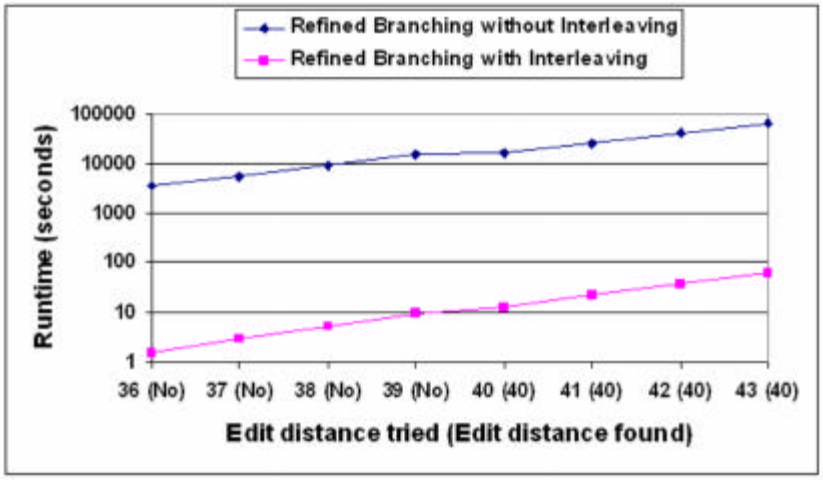

Fig. 4. FPT run times on a graph with 100 vertices, 10 clusters and edit distance 40.

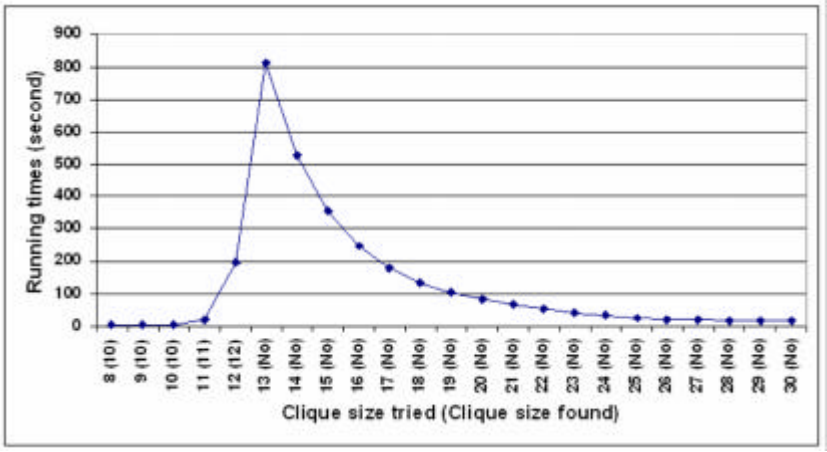

Fig. 5. The monotonicity of FPT parameter effects as seen when solving clique with vertex cover.

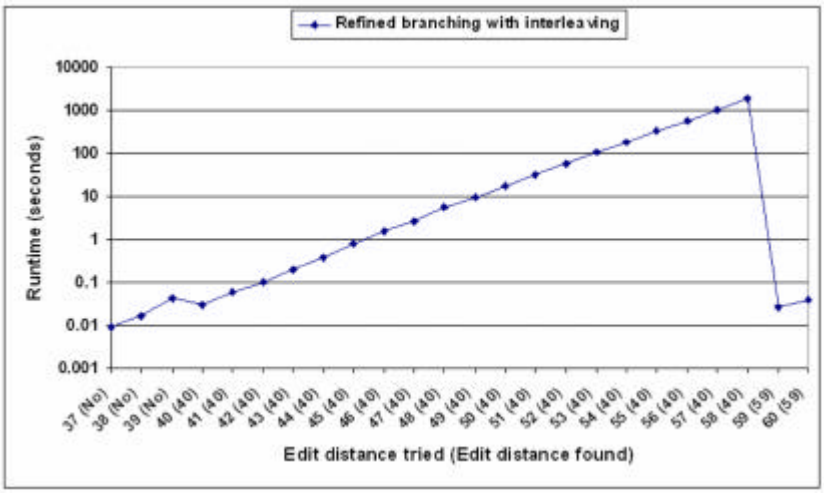

Fig. 6. FPT parameter effects on a graph with 100 vertices, 5 clusters and edit distance 40.

Fig. 7. FPT parameter effects on a graph with 100 vertices, 10 clusters and edit distance 40.

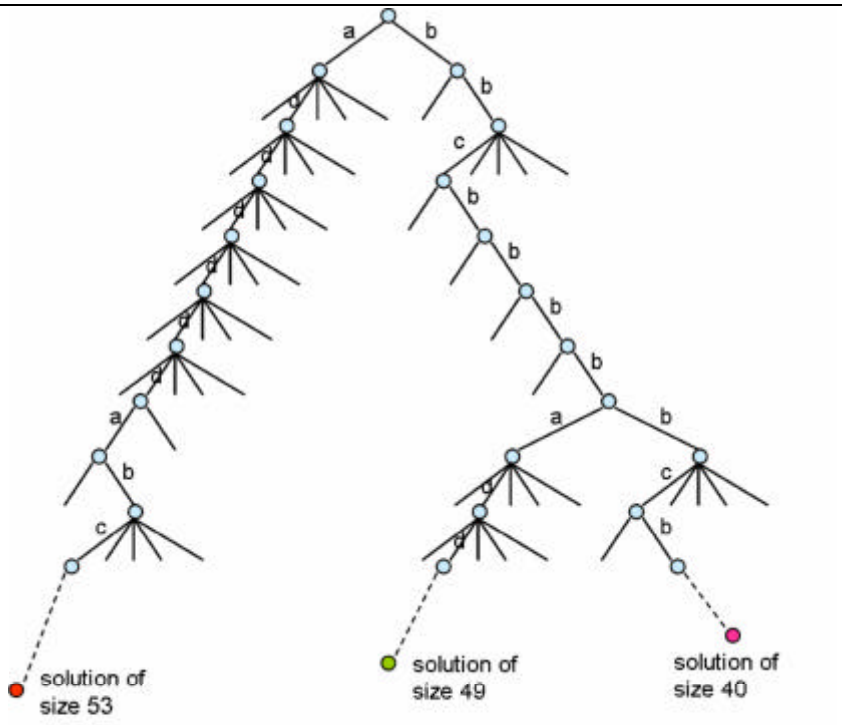

Fig. 8. Search trees depend on parameter values.

6 Concluding Remarks

The issue of scalability deserves scrutiny, especially if we are to scale to genome-sized problem instances. Even for covers and cliques, supercomputers and monolithic memory may be heavily taxed [9]. In this respect, it is noteworthy that well-known problems such as vertex cover require searching a parameter space of size $|V|$, while cluster editing possesses a search space of size $|E|$. At some problem size, of course, approximation should better optimization. The exact size probably depends on many factors, including graph density, relative efficiency of implementations, and even machine architecture. Initial experiments have produced interesting comparisons. See Figure 9.

Related questions abound. For example, we are interested in the difficulty of enumerating solutions within some fixed number of edge additions and deletions [4]. Such an enumeration may prove useful in making decisions between multiple and possibly ambiguous solutions. We are also interested in relaxing the requirement that cluster editing cliques be disjoint [5]. Such a relaxation makes particular sense in applications for which vertices represent genes or gene products, because these are often pleiotropic and thus may rightfully belong in overlapping cliques.

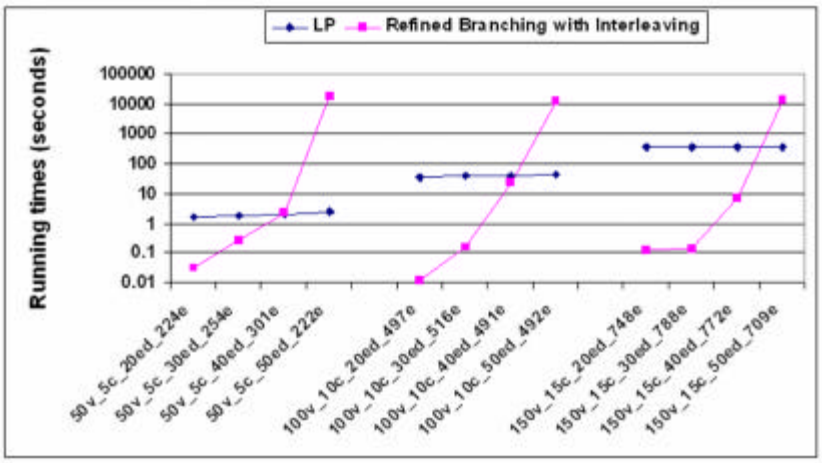

Fig. 9. The scalability of approximation via LP versus optimization via FPT.

References

1. F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Proceedings, Workshop on Algorithm Engineering and Experiments,* New Orleans, Louisiana, 2004.
2. F. N. Abu-Khzam, M. A. Langston, P. Shanbhag, and C. T. Symons. Scalable parallel algorithms for FPT problems. *Algorithmica*, 2006, accepted for publication.

3. M. Charikar, V. Guruswami, and A. Wirth. Clustering with qualitative information. *Journal of Computer and System Sciences*, 71:360–383, 2005.
4. P. Damaschke. On the fixed-parameter enumerability of cluster editing. In *Proceedings, International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 283–294, 2005.
5. P. Damaschke. Fixed-parameter tractable generalizations of cluster editing. In *Proceedings, International Conference on Algorithms and Complexity*, 2006.
6. J. Gramm, J. Guo, F. Hueffner, and R. Niedermeier. Graph-modeled data clustering: Fixed-parameter algorithms for clique generation. *Theory of Computing Systems*, 38(4):373 – 392, 2005.
7. J. Guo. *Algorithm design techniques for parameterized graph modification problems*. PhD thesis, Univ. Jena, 2005.
8. D. E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical algorithms. Addison-Wesley, 3 edition, 1997.
9. Y. Zhang, F. N. Abu-Khzam, N. E. Baldwin, E. J. Chesler, M. A. Langston, and N. F. Samatova. Genome-scale computational approaches to memory-intensive applications in systems biology. In *Proceedings, Supercomputing,* Seattle, Washington, 2005.