A New Implementation and Detailed Study of Breakpoint Analysis

Bernard M.E. Moret
Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131

Stacia Wyman
Dept. of Computer Science
University of Texas
Austin, TX 78712

David A. Bader
Dept. of Electrical & Computer Engineering
University of New Mexico
Albuquerque, NM 87131

Tandy Warnow
Dept. of Computer Science
University of Texas
Austin, TX 78712

Mi Yan
Dept. of Electrical & Computer Engineering
University of New Mexico
Albuquerque, NM 87131

Phylogenies derived from gene order data may prove crucial in answering some fundamental open questions in biomolecular evolution. Yet very few techniques are available for such phylogenetic reconstructions. One method is *breakpoint analysis*, developed by Blanchette and Sankoff [2] for solving the "breakpoint phylogeny." Our earlier studies [5,6] confirmed the usefulness of this approach, but also found that `BPAnalysis`, the implementation developed by Sankoff and Blanchette, was too slow to use on all but very small datasets. We report here on a reimplementation of `BPAnalysis` using the principles of algorithmic engineering. Our faster (by 2 to 3 orders of magnitude) and flexible implementation allowed us to conduct studies on the characteristics of breakpoint analysis, in terms of running time, quality, and robustness, as well as to analyze datasets that had so far been considered out of reach. We report on these findings and also discuss future directions for our new implementation.

1 Introduction

Some organisms have a single chromosome or contain single-chromosome organelles (mitochondria or chloroplasts), the evolution of which is mostly independent of the evolution of the nuclear genome. Given a particular strand from a single chromosome (whether linear or circular), we can infer the ordering of the genes along with the directionality of the genes, thus representing each chromosome by an ordering of oriented genes. The evolutionary process that operates on the chromosome may include inversions and transpositions, which change the order in which genes occur in the genome as well as their orientation. Other events, such as insertions, deletions, or duplications, change the number of times and the positions in which a gene occurs. Gene order, orientation, and number represent a new source of data for phylogeny reconstruction. Appropriate tools for analyzing such data may help resolve some difficult phylogenetic reconstruction problems; indeed, this new source of data has been embraced by many biologists in their phylogenetic work.[14,15,17]

A natural optimization problem for phylogeny reconstruction from this type of data explicitly attempts to reconstruct an evolutionary scenario with a minimum number of permitted evolutionary events (e.g., duplications, insertions, deletions, inversions, and transpositions) on the tree. Such approaches are computationally very intensive (all are known or conjectured to be NP-hard); worse, to date, no automated tools exist for solving such problems. Another approach is first to estimate leaf-to-leaf distances (based upon some metric) between all genomes, and then to use a standard distance-based method such as *neighbor-joining* [19] to construct the tree. Such approaches are quite fast and may prove valuable in reconstructing the underlying tree, but cannot recover the ancestral gene orders.

Blanchette and Sankoff developed a technique, breakpoint phylogeny, for the special case in which the genomes all have the same set of genes, and each gene appears once. Our earlier simulation study suggests that the approach works well for certain datasets (i.e., it obtains trees that are close to the model tree), but that the implementation, the `BPAnalysis` software, is too slow to be used on anything other than small datasets with a few genes[5]. In this paper we describe our reimplementation of `BPAnalysis` and how we have obtained speedups of 2 to 3 orders of magnitude.

2 Definitions

When each genome has the same set of genes and each gene appears exactly once, a genome can be described by an ordering (circular or linear) of these genes, each gene given with an orientation that is either positive (g_i) or negative ($-g_i$). Given two genomes G and G' on the same set of genes, a *breakpoint* in G is defined as an ordered pair of genes, (g_i, g_j), such that g_i and g_j appear consecutively in that order in G, but neither (g_i, g_j) nor $(-g_j, -g_i)$ appears consecutively in that order in G'. The breakpoint distance between two genomes is the number of breakpoints between that pair of genomes. The breakpoint score of a tree in which each node is labelled by a signed ordering of genes is then the sum of the breakpoint distances along the edges of the tree.

Given three genomes, we define their *median* to be a fourth genome that minimizes the sum of the breakpoint distances between it and the other three. The *Median Problem for Breakpoints* (MPB) is to construct such a median and is NP-hard [16]. Sankoff and Blanchette developed a reduction from MPB to the Travelling Salesman Problem (TSP), perhaps the most studied of all optimization problems[7]. Their reduction produces an undirected instance of the TSP from the directed instance of MPB by the standard technique of representing each gene by a pair of cities connected by an edge that must be included in any solution.

3 BPAnalysis

`BPAnalysis` (see Figure 1) is the method developed by Blanchette and Sankoff to solve the breakpoint phylogeny. Within a framework that enumerates all trees, it uses an iterative heuristic to label the internal nodes with signed gene orders. This procedure is computationally very intensive. The outer loop enumerates all $(2n-5)!!$

```
Initially label all internal nodes with gene orders
Repeat
    For each internal node v, with neighbors A, B, and C, do
        Solve the MPB on A, B, C to yield label m
        If relabelling v with m improves the score of T, then do it
until no internal node can be relabelled
```

Figure 1: `BPAnalysis`

leaf-labelled trees on n leaves, an exponentially large value.[a] The inner loop runs an unknown number of iterations (until convergence), with each iteration solving an instance of the TSP (with a number of cities equal to twice the number of genes) at each internal node. The computational complexity of the entire algorithm is thus exponential in *each* of the number of genomes and the number of genes, with significant coefficients. The procedure nevertheless remains a heuristic: even though all trees are examined and each MPB problem solved exactly, the tree-labeling phase does not ensure optimality unless the tree has only three leaves.

4 Study Objectives

Our earlier experiments with various techniques for reconstructing phylogenies from gene order data suggested that Sankoff and Blanchette's implementation of `BPAnalysis` is much too slow. On a collection of Campanulaceae[5] with 13 genomes of 105 gene segments, we estimated that Sankoff and Blanchette's `BPAnalysis` would take well over 200 years to complete—an estimate based on the average number of trees processed by the code per unit time and extended to the 13,749,310,575 tree topologies on 13 leaves. Although an exhaustive search of tree topologies is clearly impossible for more than 15 genomes (there are $0.2 \cdot 10^{15}$ trees on 16 genomes), even selective exploration of tree space requires very fast labeling of the internal nodes of a tree.

Our objective was therefore to develop a much faster implementation of `BPAnalysis`, prior to modifying the method used for searching tree space. Our three major goals were flexibility (e.g., the ability to change TSP solvers or to change distance measures between genomes), the introduction of approximate TSP solvers (which are required for large number of genes), and overall speed. To achieve the last goal, we used a process that we helped pioneer and termed *algorithmic engineering*[11,12]—a combination of low-level algorithmic changes, data structures changes, and coding strategies that combine to eliminate bottlenecks in the code, balance its computational tasks, and make it cache-sensitive.

[a] The double factorial is a factorial with a step of 2, so we have $(2n-5)!! = (2n-5) \cdot (2n-7) \cdot \ldots \cdot 3$

5 Algorithmic Aspects of our Implementation

Tree generation Exploring tree space, whether exhaustively or selectively, requires the efficient generation of tree topologies. We need a generating mechanism that is interruptible and restartable at any point. We chose to generate a preorder encoding of the tree, then to produce the topology from the encoding. Generating the next tree in the ordering takes amortized constant time. This enables us to produce only trees that are refinements of a given constraint tree, as well as to generate only every kth tree for a given stepping value k. The stepping value is a crucial feature for sampling-based exploration. By generating every kth tree, we investigate a wide range of tree topologies for large datasets. Without the step, we are limited to a fraction of very similar tree topologies. Detailed profiling shows that the time taken by tree generation does not rise above the noise level in our time measurements.

Tree labeling Labeling the internal nodes of a tree is the most challenging part of the problem—indeed, no algorithm is known that would produce an optimal solution for more than three leaves. Although the problem is NP-hard even for a three-leaf tree, it is possible to produce an optimal solution for many realistic problems on three leaves by using the TSP reduction. The approach used by Sankoff and Blanchette to label an entire tree is to do a postorder traversal of the tree; at each internal node, use the labels of its three neighbors to define an instance of the MPB, solve that problem, and assign the new label to the node if the number of breakpoints is thereby lowered; and repeat the entire process until no change occurs. This process is rather wasteful—an NP-hard problem must be solved at each internal tree node, over and over, with most solutions discarded because they do not bring about any improvement.

Our implementation only generates an MPB problem for nodes that saw at least one of their three neighbors relabeled over the last pass. We also score the tree incrementally in constant time after each relabeling (whereas `BPAnalysis` calls a tree-scoring routine that requires linear time to run) and do so only if the label has changed—a relatively rare occurrence. These changes brought about a speed-up on the Campanulaceae on the order of 1.5.

Condensation Sets of closely related genomes often share a number of adjacencies; even when not closely related, three genomes will often share some as well. In those cases when all genomes in a set contain shared adjacencies, we *condense* the shared adjacencies: we redefine gene fragments to consist of the longest shared subsequences and replace the original instance by one given in terms of the new gene fragments. Such condensation does not affect labeling or any of the rearrangement-based distance measures (breakpoint, inversion, transposition). Condensation can be implemented (and reversed) efficiently and may save large amounts of time by producing significantly smaller numbers of genes in the genomes—and hence smaller TSP instances. We use both an initial condensing of the entire dataset and a dynamic condensing of each triple of genomes when computing the median. In our Campanulaceae dataset, the 13 genomes have sufficient runs in common that they can be initially condensed from 105 down to just 36 gene segments. When only three genomes are considered at a time (as in the TSP instances), condensing can have an even greater impact. Combining initial and dynamic condensation on the Campanulaceae dataset results in a speed-up by a factor of 6.

Approximate TSP solvers Each MPB problem is solved through reduction to a TSP instance. The number of such instances solved in the analysis of a dataset is very large—and, of course, the TSP problem is itself NP-hard. We used the *Concorde* library[4] for two of our approximate solvers—the chained and the simple versions of the famous Lin-Kernighan heuristic[9]. These heuristics typically come within a few percent of optimal for the simple version and even closer for the slower chained version, at least for the geometric TSP instances used in most testbeds[7]. Unfortunately, the LK solvers are quite slow—even the simple LK solver takes cubic time and suffers from significant coefficients.

We also implemented the standard greedy algorithm for TSP (also known as "coalesced simple paths"[13]): this algorithm successively adds the next available edge of least cost, subject to not creating a short cycle nor a vertex of degree three. For our instances of the TSP, this method can be implemented to run in very fast linear time, but tends to yield significantly poorer solutions than the LK solvers.

Our exact TSP solver We implemented a standard include-exclude backtracking search with pruning—the most basic technique for exhaustive search of a state space—along the lines of the `BPAnalysis` code. This approach orders the edges by cost, then recursively attempts first to include, then to exclude each edge in turn, the inclusions subject again to not creating a short cycle nor a vertex of degree three. (In effect, the greedy method described earlier is simply the first probe sequence of this search method.) The recursion stops when a solution is obtained, when it runs out of edges (non-trivial edges only, for which see below), or when a lower bound computation indicates that no tree can be found to improve upon the current upper bound (the value of the current best solution).

In comparison with `BPAnalysis`, we used more streamlined data structures, better bounding, and some features tailored to the special nature of the instances generated in the reduction. Of the $\Theta(n^2)$ edges of an instance produced by the reduction to TSP, at most $3n$ are nontrivial—those that correspond to adjacent genes segments in the three genomes. Our exact TSP solver considers only nontrivial edges, treating the others as an undifferentiated pool—a refinement that allows each step in the search to run in linear rather than quadratic time. Our lower bound is computed with as much information as can easily be maintained in linear time—excluded from the computation are not just edges that have already been considered (as in `BPAnalysis`), but also any edges that would create a short cycle or a vertex of degree three. Finally, we provide the solver with what often proves a very tight upper bound by determining which of the current label and its three neighbors would provide the best median, and initializing the solver with this solution. Our lower and upper bounds prove tight enough that over two thirds of the calls to our TSP solver are pruned immediately, without a single attempt to include or exclude an edge. This combination of algorithmic changes accounted for a speed-up factor of 10 on the Campanulaceae dataset.

Initial labeling Since the labeling procedure is iterative, assigning initial labels can make a big difference in performance. Sankoff and Blanchette proposed several initializations. We implemented all but one of them (their last heuristic, based on an *ad hoc* solution method for a set of linear equations to define the parameters of each TSP instance, is extremely slow), along with several of our own devising. The choice of an initialization procedure is crucial, because little to no relabeling is done after a good initialization. All but one of the methods run in linear time (one of the two methods described by Sankoff and Blanchette as "onerous" takes quadratic time), with the exception of the cost of the calls to the exact TSP solver. Some of the methods make no such call, some make one, while the more demanding methods make one such call at each internal node. After much experimentation with these methods, we settled on one of those proposed by Sankoff and Blanchette as the best compromise of accuracy and speed—but our code provides another 6 methods. The chosen method sets up a TSP instance for each internal node by using the closest already labeled neighbor along each of the three directions out of that internal node; the computed median is assigned to the internal node, which is then considered to be labeled.

6 Coding Aspects of our Implementation

Algorithmic engineering suggests a refinement cycle in which the behavior of the current implementation is studied in order to identify problem areas which can include excessive resource consumption or poor results. We used extensive profiling and testing throughout our development cycle, which allowed us to identify and eliminate a number of such problems. For instance, converting the MPB into a TSP instance dominates the running time whenever the TSP instances are not too hard to solve. Thus we lavished much attention on that routine, down to the level of hand-unrolling loops to avoid modulo computations and allowing reuse rather than recomputation of intermediate expressions; we cut the running time of that routine down by a factor of at least six—and thereby nearly tripled the speed of the overall code. We lavished equal attention on distance computations and on the computation of the lower bound, with similar results. Constant profiling is the key to such an approach, because the identity of the principal "culprits" in time consumption changes after each improvement, so that attention must shift to different parts of the code during the process—including revisiting already improved code for further improvements. These steps provided a speed-up by a factor of 6–8 on the Campanulaceae dataset.

The original BPAnalysis is written in C++ and uses a space-intensive full distance matrix, as well as many other data structures. It has a significant memory footprint (over 60MB when running on the Campanulaceae dataset) and poor locality (a working set size of about 12MB). Our implementation has a tiny memory footprint (1.8MB on the Campanulaceae dataset) and mostly good locality (nearly all of our storage is in arrays preallocated in the main routine), which enables it to run almost completely in cache (the working set size is 600KB). Cache locality can be improved by returning to a FORTRAN-style of programming, in which records (structures/classes) are avoided in favor of separate arrays, in which simple iterative loops that traverse an array linearly are preferred over pointer dereferencing, in which code is replicated to process each array separately, etc. While we cannot measure exactly how much we gain from this approach, studies of cache-aware algorithms[8,20] indicate that the gain is likely to be substantial—factors of anywhere from 2 to 40 have been reported. Many new memory hierarchies show differences in speed between cache and main memory that exceed two orders of magnitude.

Low-level coding details also affect the quality of the solution. For instance, the exact method used in creating the MPB subproblems makes a significant, if not consistent, difference: in which order are the internal nodes in need of relabeling handled? are the previous or current values of their neighbors used in creating the TSP instance? in the initialization phase, are newly labeled nodes assimilated to leaves or skipped over? We experimented with many such combinations in our code.

We checked correctness throughout the development. Direct comparison with BPAnalysis is not feasible: the order in which it enumerates trees differs from the order our code uses and the way in which it breaks ties is dictated by details of its data structures, which differ completely from ours. We can and did compare the value of solutions obtained (tree scores) after completion on small datasets and verified that differences were unbiased—i.e., that the expected difference was zero. We wrote several independent procedures specifically for verifying parts of the computation—e.g., through independent scoring of a tree. We also tested our code throughout the process to ensure that the functional behavior of each successive version remained consistent with that of earlier versions. All of our tests confirm that our reimplementation produces the same quality of solution as BPAnalysis when run with the same parameters.

7 Experimental Procedure

We had two objectives for our experiments. We wanted to compare the raw running times of our versions and of BPAnalysis and investigate their dependency on the number of genomes, the number of genes, and the rate of evolution in the model. We also wanted to study the impact on the quality of solutions of using an approximation algorithm for TSP.

We used a simulator we developed to create datasets for our experiments. The datasets are created as follows. First, a random tree topology is generated. Then evolution is simulated on the tree using inversions and transpositions as the evolutionary events. We examined different parameter settings by varying the expected number of inversions and transpositions per edge, as well as varying the number of genes and the number of genomes. We generated datasets with each of $2, 4$, or 8 events per edge to simulate different rates of evolution. The numbers of genes in each dataset were 10, 20, 40, 80, 160, and 320.

In order to obtain statistically significant data, we followed the recommendations of McGeoch[10,11] and Moret[12] and used *runs* of *trials*. Each trial is one dataset, while a run is composed of a number of independent trials. One then retains only the median value for the entire run and repeats the process with additional runs, with each run yielding a single value. These values are then examined for consistency. This method produces robust results even when, as in our case, the enormous size of the sample space precludes any fair sampling; using the median also gives a more honest picture of the situation when large deviations from the mean are expected.

8 Experiments

In order to test our TSP solvers, we ran a number of tests with just three genomes. These problems have only one tree topology and only one internal node to label, so that only one call is made to the TSP solver. We used six different numbers of genes per genome, and up to three rates of evolution. For large numbers of genes, $r = 8$ was too high a rate of evolution to allow the two exact solvers to finish within reasonable time. We ran the 3-genome problem under 5 different methods: `BPAnalysis` itself (except on 320-gene problems, which it could not handle), our exact solver, the greedy algorithm, the basic Lin-Kernighan, and the chained Lin-Kernighan.

In our first experiment, we tested the different solvers under the different rates of evolution. In our second experiment, we used the same data to compare the relative running times of the 5 solvers. In our third experiment, we computed the percent over optimal that the approximate solvers obtained. Our fourth experiment explored tree-processing rates for the various solvers. We used both real and synthetic datasets for this experiment, including our Campanulaceae dataset. Most datasets did not run to completion, but we ran them long enough to obtain an accurate count of the number of trees processed per unit time and thus to be able to predict the running time to completion (in the case of our code, we also used the sampling option to ensure that the estimates were not biased towards the small fraction of trees explored in a large problem).

Our experiments were run on a 233MHz Pentium II laptop with 128MB of memory and 512KB of off-chip L2 cache running Linux; we compiled all code (including `BPAnalysis`) with the GNU `gcc` compiler with options `-O3 -mpentiumpro`.

9 Results and Discussion

Experiments on 3-genome problems Figure 2 shows the running times of our four versions and of `BPAnalysis` on 3-genome problems of increasing sizes and at different rates of evolution. The rates $r = 2, 4, 8$ reflect the expected number of inver-

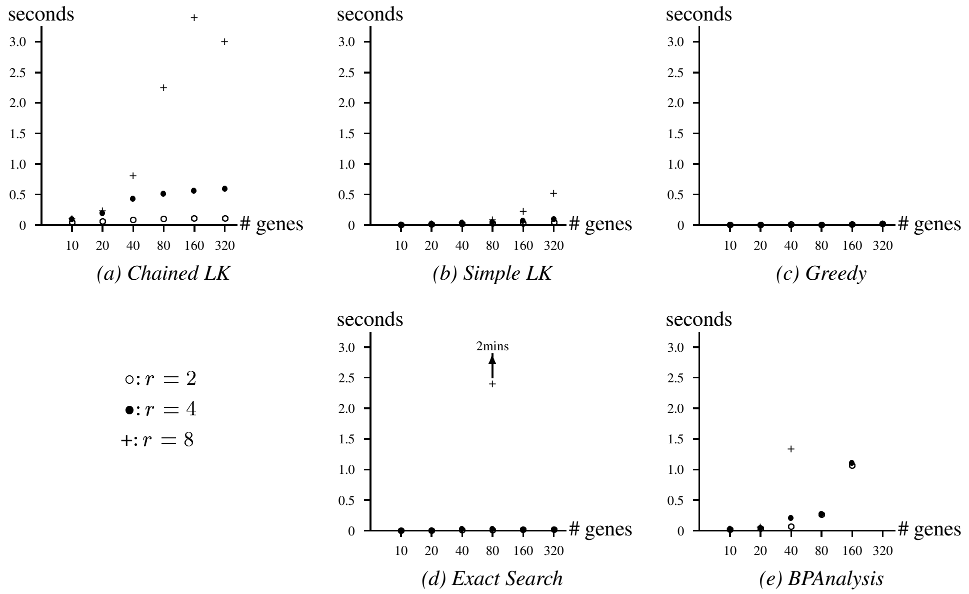

Figure 2: Speed of the 5 solvers on various 3-genome problems under 3 different rates of evolution

sions on each tree edge. The greedy solver is so fast that all of its times fall on the horizontal axis, as do the times of the exact solver for $r = 2$ and $r = 4$. Higher rates of evolution clearly induce much harder instances of the TSP, so that the two exact solvers suffer; when they require more than 20 minutes of computation, we do not show their times.

Figure 3 presents the same data, this time per evolutionary rate so as to facilitate comparisons of running times. We know from algorithm analysis that the greedy solver runs (in our special case) in linear time, the two LK solvers in roughly cubic time, and the exact solvers in exponential time. Our figures clearly demonstrate the exponential behavior of `BPAnalysis`, but our exact solver stays in a flat part of its exponential curve all the way to 320 genes for $r = 2$ and $r = 4$. The two LK solvers show at least quadratic behavior, while the greedy solver give us nearly flat values (below noise level) for the entire range. Because high rates of evolution induce numerous breakpoints, the resulting instances of the TSP have relatively undifferenti-

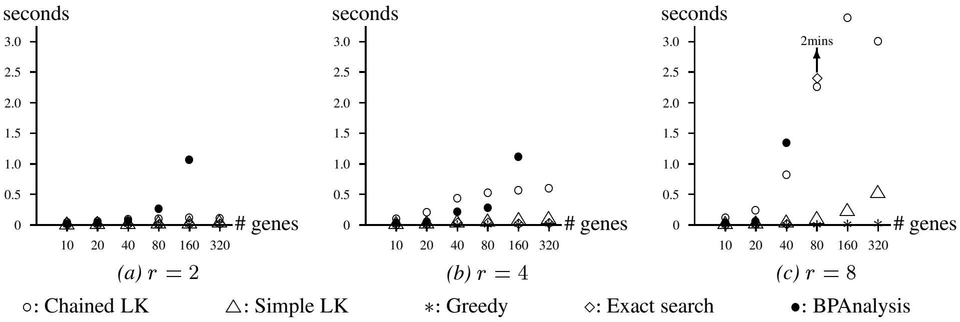

Figure 3: Relative running times of the five methods on 3-genome problems of various sizes

ated edges costs: thus most of the edges have maximal or near maximal cost, making it difficult to find an optimal solution quickly, so that the exact solvers suffer—indeed, the problems rapidly become intractable. In contrast, the Chained LK solver slows down a bit (more phases) and the simple LK and greedy solvers not at all.

Quality of approximation of heuristics Figure 4 shows the percentage by which the solutions returned by the simple LK and fast greedy solvers exceed the optimal. In contrast, the Chained LK solver returned optimal solutions for all of these test in-

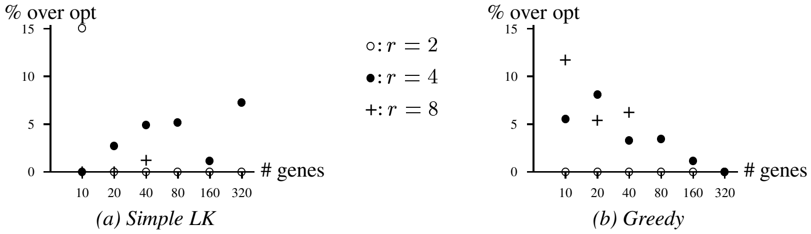

Figure 4: Percentage excess over optimal for LK and greedy solvers

stances. The error percentage can be artificially larger for smaller evolutionary rates because, for these lower rates, the number of breakpoints in the optimal solution is quite small, thus magnifying percentages. Rather surprisingly, the fast and unsophisticated greedy algorithm holds its own against the simple LK algorithm—not a situation encountered on typical TSP benchmarks[7].

Tree-processing rates In order to estimate the raw speed of each method, we ran all given methods on real and synthetic datasets, letting them process thousands of trees until a fixed time bound was attained. We then normalized the results and, since the greedy algorithm was always the fastest, computed ratios of processing rates for the other four methods against the rate of the greedy method. Table 1 shows the results;

Table 1: Ratios of tree-processing rates of 5 methods to the rate of the greedy method on various datasets.

method	13/105/-	8/105/-	10/20/2	10/20/4	10/20/8	10/160/2	60/20/2	80/100/2
Greedy baseline in trees/s	23,100	73,400	15,300	15,200	13,550	5,250	2,400	975
Simple LK	68	66	31	33	31	40	45	100
Chained LK	280	220	225	310	300	210	250	330
Exact solver	3.5	1.1	3.4	4.3	3.6	1.7	2.6	2.6
`BPAnalysis`	2,000	3,820	220	250	225	840	350	500

in the table, $n/N/r$ denotes a problem with n genomes, N genes, and r inversions per model tree edge; the first two data sets are the full Campanulaceae dataset and its first 8 members, respectively. The figure shown for the greedy method is the actual processing rate of that method, in trees processed per second. The high processing rate of our exact solver (we have observed rates from 70 to 5,000 times faster than `BPAnalysis`) makes it possible to solve problems with 10–12 genomes on a single processor. Chained LK is much too slow to be of use, and even simple LK, while significantly faster than `BPAnalysis`, is far slower than our exact solver. On the other hand, the greedy algorithm, while faster than the exact solver, tends to yield worse solutions and thus should be reserved for difficult instances (large numbers of genes and high rates of evolution).

10 Conclusions and Future Work

We have presented a new implementation of breakpoint analysis that improves on the original `BPAnalysis` by 2 to 3 orders of magnitude—an improvement reached through algorithmic engineering techniques. Our implementation makes it possible to analyze much larger datasets; it can be combined with massive parallelism, reducing the running time for the Campanulaceae dataset from two centuries down to a day when run on a 512-processor supercluster. Our code can be obtained from URL `www.cs.unm.edu/~moret/GRAPPA-09.tar.gz`; it has been tested under Linux, FreeBSD, Solaris, and Windows NT, as well as on parallel clusters. Breakpoint scores may not be the measure of choice; our latest implementation includes a fast linear-time computation of inversion distances [1], allowing us to minimize either measure and compare their relative use. These and other improvements pale against the main drawback of the approach: enumerating all tree topologies is impossible for 16 or more genomes; thus an implicit exploration of tree space is the next step.

Acknowledgments

This work was supported in part by NSF ITR 00-81404 (Moret and Bader), NSF DEB 99-10123 (Bader), DOE SUNAPP AX-3006 (Bader), NSF 94-57800 (Warnow), and the David and Lucile Packard Foundation (Warnow).

References

1. Bader, D.A., Moret, B.M.E., and Yan, M., "A fast linear-time algorithm for inversion distance with an experimental comparison," submitted to RECOMB 01.
2. Blanchette, M., Bourque, G., & Sankoff, D., "Breakpoint phylogenies," in *Genome Informatics 1997*, Miyano, S., and Takagi, T., eds., Univ. Academy Press, Tokyo, 25–34.
3. Blanchette, M., Kunisawa, T., & Sankoff, D., "Gene order breakpoint evidence in animal mitochondrial phylogeny," *J. Mol. Evol.* **49** (1999), 193–203.
4. Applegate, D., Bixby, R., Chvátal, V., & Cook, W., "CONCORDE: Combinatorial Optimization and Networked Combinatorial Optimization Research and Development Environment," available at `www.keck.caam.rice.edu/concorde.html`
5. Cosner, M.E., Jansen, R.K., Moret, B.M.E., Raubeson, L.A., Wang, L.-S., Warnow, T., & Wyman, S., "A new fast heuristic for computing the breakpoint phylogeny and experimental phylogenetic analyses of real and synthetic data," *Proc. 8th Int'l Conf. on Intelligent Systems for Molecular Biology ISMB-2000*, San Diego (2000).
6. Cosner, M.E., Jansen, R.K., Moret, B.M.E., Raubeson, L.A., Wang, L.S., Warnow, T., & Wyman, S., "An empirical comparison of phylogenetic methods on chloroplast gene order data in Campanulaceae," *Proc. Gene Order Dynamics, Comparative Maps, and Multigene Families DCAF-2000*, Montreal (2000).
7. Johnson, D.S., & McGeoch, L.A., "The traveling salesman problem: a case study," in *Local Search in Combinatorial Optimization*, E. Aarts & J.K. Lenstra, eds., John Wiley, New York (1997), 215–310.
8. LaMarca, A., and Ladner, R.E., "The influence of caches on the performance of heaps," *ACM J. Exp. Algorithmics* **1**, 4 (1996), `www.jea.acm.org/1996/LaMarcaInfluence/`
9. Lin, S., & Kernighan, B.W., "An effective heuristic algorithm for the traveling salesman problem," *Operations Res.* **21** (1973), 498–516.
10. McGeoch, C.C., "Analyzing algorithms by simulation: variance reduction techniques and simulation speedups," *ACM Comput. Surveys* **24**, 2 (1992), 195–212.
11. McGeoch, C.C., "Toward an experimental method for algorithm simulation," *INFORMS J. Comput.* **8** (1996), 1–15.
12. Moret, B.M.E., "Towards a discipline of experimental algorithmics," *Proc. 5th DIMACS Challenge*, available at `www.cs.unm.edu/~moret/dimacs.ps`.
13. Moret, B.M.E., and Shapiro, H.D. *Algorithms from P to NP, Vol. I: Design and Efficiency*. Benjamin-Cummings, Menlo Park, CA, 1991.
14. Olmstead, R.G., & Palmer, J.D., "Chloroplast DNA systematics: a review of methods and data analysis," *Amer. J. Bot.* **81** (1994), 1205–1224.
15. Palmer, J.D., "Chloroplast and mitochondrial genome evolution in land plants," in *Cell Organelles*, Herrmann, R., ed., Springer Verlag (1992), 99–133.
16. Pe'er, I., & Shamir, R., "The median problems for breakpoints are NP-complete," *Elec. Colloq. on Comput. Complexity*, Report 71, 1998.
17. Raubeson, L.A., & Jansen, R.K., "Chloroplast DNA evidence on the ancient evolutionary split in vascular land plants," *Science* **255** (1992), 1697–1699.
18. Sankoff, D., & Blanchette, M., "Multiple genome rearrangement and breakpoint phylogeny," *J. Computational Biology* **5** (1998), 555–570.
19. Saitou, N., and Nei, M., "The neighbor-joining method: A new method for reconstructing phylogenetic trees," *Mol. Biol. & Evol.* **4** (1987), 406–425.
20. Xiao, L., Zhang, X., and Kubricht, S.A., "Improving memory performance of sorting algorithms," to appear in *ACM J. Exp. Algorithmics*.