Alcom-FT Technical Report Series
ALCOMFT-TR-01-24

What Do We Learn from Experimental Algorithmics?*

Camil Demetrescu† and Giuseppe F. Italiano‡

March 15, 2001

Abstract

Experimental Algorithmics is concerned with the design, implementation, tuning, debugging and performance analysis of computer programs for solving algorithmic problems. It provides methodologies and tools for designing, developing and experimentally analyzing efficient algorithmic codes and aims at integrating and reinforcing traditional theoretical approaches for the design and analysis of algorithms and data structures.

In this paper we survey some relevant contributions to the field of Experimental Algorithmics and we discuss significant examples where the experimental approach helped in developing new ideas, in assessing heuristics and techniques, and in gaining a deeper insight about existing algorithms.

1 Introduction

Experiments play a crucial role in many scientific disciplines. In Natural Sciences, for instance, researchers have been extensively running experiments to learn certain aspects of nature and to discover unpredictable features of its internal organization. In Theoretical Computer Science we use mathematical tools for analyzing and predicting the behavior of algorithms. For over thirty years, asymptotic worst-case analysis has been the main model in the design of efficient algorithms, proving itself to yield enormous advantages in comparing and characterizing their behavior, and leading to major algorithmic advances.

In recent years, many areas of theoretical computer science have shown growing interest in solving problems arising in real world applications, experiencing a remarkable shift to more application-motivated research. The new demand for algorithms that are of practical utility has led researchers to take advantage from the experimental method as a tool both for refining and reinforcing the theoretical analysis of algorithms, and for developing and assessing heuristics and programming techniques for producing codes that are efficient

*This work has been partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT), by the Italian Ministry of University and Scientific Research (Project "Algorithms for Large Data Sets: Science and Engineering") and by CNR, the Italian National Research Council.

†Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy. Email: demetres@dis.uniroma1.it, URL: http://www.dis.uniroma1.it/~demetres/

‡Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Via di Tor Vergata 110, 00133 Roma, Italy. Email: italiano@disp.uniroma2.it, URL: http://www.info.uniroma2.it/~italiano/

in practice. This is clear from the surge of investigations that compare and analyze experimentally the behavior of algorithms, and from the rise of a new research area called *Experimental Algorithmics*.

Experimental Algorithmics is concerned with the design, implementation, tuning, debugging and performance analysis of computer programs for solving algorithmic problems and aims at joining the experimental method with more traditional theoretical approaches for the design and analysis of algorithms and data structures.

1.1 Goals

Major goals of Experimental Algorithmics are:

- Defining standard methodologies for algorithm engineering.
- Devising software systems to support the process of implementation, debugging and empirical evaluation of algorithms. In particular, programming environments, high-level debuggers, visualization and animation tools, and testing and simulation environments.
- Identifying and collecting problem instances from the real world and providing generators of synthetic test sets for experimental evaluation of algorithms.
- Providing standard and well documented libraries that feature efficient implementations of algorithms and data structures.
- Performing empirical studies for comparing actual relative performance of algorithms so as to identify the best ones for use in a given application. This may lead to the discovery of algorithm separators, i.e., families of problem instances for which the performances of solving algorithms are clearly different. Other important results of empirical investigations include assessing heuristics for hard problems, characterizing the asymptotic behavior of complex algorithms, discovering the speed-up achieved by parallel algorithms and studying memory hierarchy and communication effects on real machines, helping in performance prediction and finding bottlenecks in real applications, etc.
- Last, but not least, encouraging fruitful cooperation between theoreticians and practitioners.

1.2 Motivations

The main motivations for recurring to experiments for analyzing and drawing on-the-road conclusions about algorithms have already been pointed out by different authors [3, 16, 19, 26]. Among many, we cite the following:

- Many authors call "Asymptopia" the range of problem instances for which an algorithm exhibits clear asymptotic behavior. Unfortunately, for certain algorithms, "Asymptopia" may include only huge problem instances, far beyond the needs of any reasonable practical application. This means that, due to the high constants hidden in the analysis, theoretical bounds may fail to describe the behavior of algorithms on many instances of practical interest. As a typical example, the experimental study in [27] shows that the minimum spanning tree algorithm of Fredman and Tarjan improves in practice upon the classical Prim's algorithm only for huge dense graphs with more than one million nodes.

- The situation may be even worse: constants hidden in the asymptotic time bounds may be so large as to prevent any practical implementation from running to completion. The Robertson and Seymour cubic-time algorithm for testing if a graph is a minor of another graph provides an extreme example: as a matter of fact, no practical implementation can face the daunting 10^{150} constant factor embedded in the algorithm, and no substantial improvement has been proposed yet.

- Many algorithms typically behave much better than in the worst case, so considering just the worst-case bounds may lead to underestimate their practical utility. A typical example is provided by the simplex algorithm for solving linear programs, whose asymptotic worst-case time bound is exponential, while its running time seems to be bounded by a low-degree polynomial in many real-world instances.

- Many practical applications require solving NP-hard problems, for which asymptotic analyses do not provide satisfactory answers about the utility of a solving algorithm.

- There are algorithms for which no tight asymptotic bounds on the running time or on the quality of returned solutions have been theoretically proved. For instance, the experimental study proposed in [14] compares the quality of the solutions returned by different approximation algorithms for the problem of minimizing edge crossings in the drawings of bipartite graphs. It reports that an algorithm with no theoretically proved constant approximation ratio returns in practice solutions with less crossings than algorithms with small, constant approximation ratio.

- New algorithmic results often rely on previous ones, and devising them only at theoretical level may lead to a major problem: researchers who would eventually come up with a practical implementation of their results may be required to code several layers of earlier unimplemented complex algorithms and data structures, and this task may be extremely difficult.

- Adding ad-hoc heuristics and local hacks to the code may dramatically improve the practical performances of some algorithms, although they do not affect the theoretical asymptotic behavior. Many clear examples are addressed in the literature: in Section 3.1 we will discuss in detail the implementation issues of the Preflow-push Maximum Flow algorithm of Goldberg and Tarjan.

- Performing experiments on a good collection of test problems may help in establishing the correctness of a code. In particular, collecting problem instances on which a code has exhibited buggy behavior may be useful for testing further implementations for the same problem.

- Practical indicators, such as implementation constant factors, real-life bottlenecks, locality of references, cache effects and communication complexity may be extremely difficult to predict theoretically, but can be measured by means of experiments.

1.3 Common Pitfalls

Unfortunately, as in any empirical science, it may be sometimes difficult to draw general conclusions about algorithms from experiments. Common pitfalls, often experienced by researchers in their studies, seem to be:

- Dependence of empirical results upon the experimental setup:
  - Architecture of the running machine: memory hierarchy, CPU instructions pipelining, CISC vs RISC architectures, CPU and data bus speed are technical issues that may substantially affect the execution performance.
  - Operating system: CPU scheduling, communication management, I/O buffering and memory management are also important factors.
  - Encoding language: features such as built-in types, data and control flow syntactic structures and language paradigm should be taken into account when choosing the encoding language. Among others, C++, C and Fortran are most commonly used in this context. However, we point out that powerful C++ features such as method invocations, overloading of functions and operators, overriding of virtual functions, dynamic casting and templates may introduce high hidden computation costs in the generated machine code even using professional compilers.
  - Compiler's optimization level: memory alignment, register allocation, instruction scheduling, repeated common subexpression elimination are the most common optimization issues.
  - Measuring of performance indicators: time measuring may be a critical point in many situations including profiling of fast routines. Important factors are the granularity of the time measuring function (typically 1 μsec to 10 $msec$, depending upon the platform), and whether we are measuring the real elapsed time, the time used by the user's process, or the time spent by the operating system to do I/O, communication or memory management.
  - Programming skills of implementers: the same algorithm implemented by different programmers may lead to different conclusions on its practical performances. Moreover, even different successive refined implementations coded by the same programmer may greatly differ from each other.
  - Problem instances used in the experiments: the range of parameters defining the test sets used in the experiments and the structure of the problem instances themselves may lead to formulate specific conclusions on the performance of algorithms without ensuring generality. Another typical pitfall in this context consists of testing codes on data sets representing classes that are not broad enough. This may lead to inaccurate performance prediction. An extreme example is given by the Netgen problem instances for the minimum cost flow problem [20] that were used to select the best code for a multicommodity flow application [23]. That code was later proved to behave much slower than several other codes on real-life instances by the same authors of [23]. In general, it has been observed that some algorithms behave quite differently if applied on real-life instances and on randomly generated test sets. Linear programming provides a well known example.

- Difficulty in separating the behavior of algorithms: it is sometimes hard to identify problem instances on which the performance of two codes is clearly distinguishable. In general, good algorithm separators are problem families on which differences grow with the problem size [16].

- Unreproducibility of experimental results: possibly wrong, inaccurate or misleading conclusions presented in experimental studies may be extremely difficult to detect if the results are not exactly and independently reproducible by other researchers.

- Modularity and reusability of the code: modularity and reusability of the code seem to conflict with any size and speed optimization issues. Usually, special implementations are difficult to reuse and to modify because of hidden or implicit interconnections between different parts of the code, often due to sophisticated programming techniques, tricks and hacks which they are based on, but yield to the best performances in practice. In general, using the C++ language seems to be a good choice if the goal is to come up with a modular and reusable code because it allows defining clean, elegant interfaces towards (and between) algorithms and data structures, while C is especially well suited for fast, compact and highly optimized code.

- Limits of the implementations: many implementations have strict requirements on the size of the data they deal with, e.g., work only with small numbers or problem instances up to a certain maximum size. It is important to notice that ignoring size limits may lead to substantially wrong empirical conclusions, especially in the case where the used implementations, for performance reasons, do not explicitly perform accurate data size checking.

- Numerical robustness: implementations of computational geometry algorithms may typically suffer from numerical errors due to the finite-precision arithmetic of real computing machines.

Although it seems there is no sound and generally accepted solution to these issues, some researchers have proposed accurate and comprehensive guidelines on different aspects of the empirical evaluation of algorithms maturated from their own experience of in the field (see, for example [3, 16, 19, 26]). The interested reader may find in [24] an annotated bibliography of experimental algorithmics sources addressing methodology, tools and techniques.
In the remainder of this paper, we first survey some relevant contributions to the area of Experimental Algorithmics, pointing out their impact in the field. Then we discuss two case studies and the lessons that we can learn from them.

2 Tools for Experimental Analysis of Algorithms

The increasing attention of the algorithmic community to the experimental evaluation of algorithms is producing, as a side effect, several tools whose target is to offer a general-purpose workbench for the experimental validation and fine-tuning of algorithms and data structures. In particular, software libraries of efficient algorithms and data structures, collections and generators of test sets for experimental evaluation, and software systems for supporting the implementation process are relevant examples of such an effort.

2.1 Software Libraries

The need for robust and efficient implementations of algorithms and data structure is one of the main motivations for any experimental work in the field of Algorithm Engineering. Devising fast, well documented, reliable, and tested algorithmic codes is a key aspect in the transfer of theoretical results into the setting of applications, but it is fraught with many of the pitfalls described in Section 1. Without claim of completeness, we survey some examples of such an effort.

LEDA. LEDA (Library of Efficient Data Types and Algorithms) is a project that aims at building a library of efficient data structures and algorithms used in combinatorial computing [25]. It provides a sizable collection of data types and algorithms in a form which allows them to be used by non-experts. The authors started the LEDA project in 1989 as an attempt to bridge the gap between algorithm research, teaching and implementation. The library is written in C++ and features efficient implementations of most of the algorithms and data types described in classical text books such as [1, 2] and [11]. Besides, it includes classes for building graphic user interfaces and for I/O, error handling and memory management. In particular, LEDA contains:

- Data structures for arrays, stacks, queues, lists, sets, dictionaries, priority queues, ordered sequences, partitions, lines, points, planes, dynamic trees and directed, undirected and planar graphs. Efficient implementations are given for each of the data types, e.g., Fibonacci Heaps and Redistributive Heaps for priority queues, Red-black Trees and Dynamic Perfect Hashing for dictionaries, etc. Moreover, the library features generators for several different classes of graphs including complete, random, bipartite, planar, grid and triangulated graphs, and testers for many graph properties, including planarity testing.

- Basic algorithms on graphs and networks and computational geometry algorithms such as topological sort, graph visits, connected, strongly connected and biconnected components, as well as transitive closure, shortest paths, maxflow, min-cost flow, min cut, maximum cardinality matching, minimum spanning tree, st-numbering, and others.

- C++ classes for dealing with windows, widgets and menus, drawing and event handling routines, and other miscellaneous utility classes.

Additional LEDA Extension Packages (LEP) have been developed by different researchers, and include Dynamic Graph Algorithms, Abstract Voronoi Diagrams, D-Dimensional Geometry, Graph Iterators, Parametric Search, SD-Trees and PQ-Trees.

LEDA can be used with almost any C++ compiler (g++, CC, xlC, cxx, Borland, MSVC++, Watcom). It is currently being developed for commercial purposes, but can be used freely for academic research and teaching. LEDA is available over the Internet at the URL: `http://www.mpi-sb.mpg.de/LEDA/`.

The design of the LEDA library is heavily based upon features of the C++ language and the library itself is intended to be a flexible and general purpose tool: for this reason, programs based on it tend to be less efficient than special implementations. However, LEDA is often used as a practical framework for empirical studies in the field of Experimental Algorithmics.

Stony Brook Algorithm Repository. The Stony Brook Algorithm Repository is a comprehensive collection of algorithm implementations for over seventy of the most fundamental problems in combinatorial algorithms. The repository is accessible via WWW at the URL: `http://www.cs.sunysb.edu/ algorith/`. Problems are classified according to the following categories:

- Data Structures
- Numerical Problems
- Combinatorial Problems
- Graph Problems – polynomial-time problems
- Graph Problems – hard problems
- Computational Geometry
- Set and String Problems

The repository features implementations coded in in different programming languages, including C, C++, Fortran, Lisp, Mathematica and Pascal. Also available are some input data files including airplanes routes and schedules, a list of over 3000 names from several nations, and a subgraph of the Erdos-number author connectivity graph.

According to a study on WWW hits to the Stony Brook Algorithm Repository site recorded over a period of ten weeks [29], most popular problems were shortest paths, traveling-salesman, minimum spanning trees as well as triagulations and graph data structures. On the opposite, least popular problems, among others, were determinants, satisfiability and planar drawing.

2.2 Test Sets

Collecting, designing and generating good problem instances for algorithm evaluation is a fundamental task in Experimental Algorithmics. For this reason, much effort has been put in collecting and defining standard test sets and generators for both specific problems and general purpose applications. The Stanford GraphBase and a recent CATS project are two examples of such an effort.

Stanford GraphBase. The Stanford GraphBase [21] is a collection of datasets and computer programs that generate and examine a wide variety of graphs and networks. Differently from other collections of test sets, the Stanford GraphBase consists of small building blocks of code and data and is less than 1.2 megabytes altogether. Data files include the following:

- `econ.dat`: numerical data representing the input/output structure of the entire United States economy in 1985.
- `games.dat`: information about prominent football teams of U.S. colleges or universities that played each other during 1990.

- `miles.dat`: contains highway distances between 128 North American cities.
- `lisa.dat`: contains a digitized version of Leonardo da Vinci's famous painting, *Mona Lisa*.
- `anna.dat`, `david.dat`, `homer.dat`, `huck.dat` and `jean.dat`: contain "digested" versions of five classic works of literature, namely Tolstoj's *Anna Karenina*, Dickens's *David Copperfield*, Homer's *Iliad*, Twain's *Huckleberry Finn*, and Hugo's *Les Misérable*.
- `words.dat`: contains a dictionary of 5757 words, representing every five-letter word of English, compiled by Donald E. Knuth over a period of 20 years.

Several instance generators included in the package are designed to convert these data files into a large variety of interesting test sets that can be used to explore combinatorial algorithms. Other generators produce graphs with a regular structure or random instances.

CATS. CATS (Combinatorial Algorithms Test Sets) [17] is a project whose mission is to facilitate experimental research by standardizing common benchmarks, providing a mechanism for their evolution, and making them easily accessible and usable. The project aims at identifying significant open questions in the design of good test sets and the assessment of performance of existing algorithms. Other goals are to facilitate algorithm selection for applications by characterizing subproblems and the behavior of competitive algorithms on these subproblems, and to encourage the development of high-quality implementations of advanced algorithms and data structures. CATS currently features an archive of application data, synthetic data and generators of instances for problems such as Maximum Flow and Minimum Spanning Tree. More information about CATS is available at the URL: `http://www.jea.acm.org/CATS/`.

2.3 Software Systems.

Libraries are just collections of subroutines that usually provide no interactive environment for developing and experimenting with algorithms. Indeed, the need for software systems such as editors for test sets and development, debugging, visualization and analysis tools has grown continuously in the last two decades. As a matter of fact, such systems proved themselves to yield consistent and valuable support in all phases of the algorithm implementation process. In the following, we briefly address some examples of research effort in this context.

Algorithm Animation Systems. Many software systems in the algorithmic area have been designed with the goal of providing specialized environments for algorithm animation. According to a standard definition [32], algorithm animation is a form of high-level software visualization that uses interactive graphics to enhance the development, presentation, and understanding of computer programs. Systems for algorithm animation have matured significantly since the rise of modern computer graphic interfaces due to their relevance in many applications. Thanks to the capability of conveying a large amount of information in a compact form that is easily perceivable by a human observer, algorithm animation is a powerful tool for understanding the behavior of algorithms and testing their correctness on specific test sets. Actually, visual debugging techniques often help discover both errors due to a wrong implementation of an algorithm and, at a higher level of abstraction, errors due to an incorrect design of the algorithm itself. Sometimes, algorithm animation can help in designing heuristics and local improvements in the code difficult to figure out theoretically. In Section 3.1 we will show an animation example that yields sharp clues to the utility of heuristics for improving the practical performances of an algorithm for solving the maximum flow problem.

Dozens of algorithm animation systems have been developed in the last two decades. The area was pioneered in the 80's by the systems Balsa [6] and Zeus [7]. Concerning other tools, we mention Tango [30], Polka [31], UWPI [18], ZStep95 [32], TPM [15], Pavane [28], Leonardo [12], Eliot [22] and Catai [8].

Computing Environments. Among others, we cite LINK, a software system developed at the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS). LINK has been designed to be a general-purpose, extendible computing environment in which discrete mathematical objects representing real world problems can be manipulated and visualized. The system features a full Scheme interpreter with access to the Tk graphics toolkit and a graphic user interface for creation, manipulation, loading, and storing of graphs, hypergraphs, and their attributes. However, due to the interpretive approach to graphics, the system is not suited for visualizing large data sets.

The LINK project has been started to encourage experimentation with algorithms and properties of graphs, and has been designed primarily as an educational and research tool. However, its development has been discontinued in the last years. The interested reader may find more information about LINK at the URL:
`http://dimacs.rutgers.edu/~berryj/LINK.html`.

3 Case Studies

3.1 Maximum Flow

The maximum flow problem, first introduced by Berge and Ghouila-Houri in [4], is a fundamental problem in combinatorial optimization that arises in many practical applications. Examples of the maximum flow problem include determining the maximum steady-state flow of petroleum products in a pipeline network, cars in a road network, messages in a telecommunication network, and electricity in an electrical network. Given a capacited network $G = (V, E, c)$ where V is the set of nodes, E is the set of edges and c_{xy} is the *capacity* of edge $(x, y) \in E$, the maximum flow problem consists of computing the maximum amount of flow that can be sent from a given source node s to a given sink node t without exceeding the edge capacities. A flow assignment is a function f on edges such that $f_{xy} \leq c_{xy}$, i.e., edge capacities are not exceeded, and for each node v (bar the source s and the sink t), $\sum_{(u,v)\in E} f_{uv} = \sum_{(v,w)\in E} f_{vw}$, i.e., the assigned incoming flows and the outgoing flows are equal. Usually, it is required to compute not only the maximum amount of flow that can be sent from the source to the sink in a given network, but also a flow assignment that achieves that amount.

Several methods for computing a maximum flow have been proposed in the literature. In particular, we mention the network simplex method proposed by Dantzig [13], the augmenting path method of Ford and Fulkerson, the blocking flow of Dinitz, and the push-relabel technique of Goldberg and Tarjan [2].

The push-relabel method, that made it possible to design the fastest algorithms for the maximum flow problem, sends flows locally on individual edges (push operation), possibly creating flow excesses at nodes, i.e., a *preflow*. A preflow is just a relaxed flow assignment such that for some nodes, called *active nodes*, the incoming flows may overcome the outgoing flows. The push-relabel algorithms work by progressively transforming the preflow into a maximum flow by dissipating excesses of flow held by active nodes that either reach the sink or return back to the source. This is done by repeatedly selecting a current active node according to some selection strategy, pushing as much exceeding flows as possible towards adjacent nodes that have a lower estimated distance from the sink paying attention not to exceed the edge capacities, and then, if the current node is still active, updating its estimated distance from the sink (relabel operation). Whenever an active node cannot reach the sink anymore as no path to the sink remains with some residual unused capacity, its distance progressively increases due to relabel operations until it gets greater than n: when this happens, it starts sending flows back towards the source, whose estimated distance is initially forced to n. This elegant solution makes it possible to deal with both sending flows to the sink and draining undeliverable excesses back to the source through exactly the same push/relabel operations. However, as we will see later, if taken "as is" this solution is not so good in practice.

Two aspects of the push-relabel technique seem to be relevant with respect to the running time: (1) the selection strategy of the current active node, and (2) the way estimated distances from the sink are updated by the algorithm.

The selection strategy of the current active node has been proved to significantly affect the asymptotic worst-case running time of push-relabel algorithms [2]: as a matter of fact, if active nodes are stored in a queue, the algorithm, usually referred to as the FIFO preflow-push algorithm, takes $O(n^3)$ in the worst case; if active nodes are kept in a priority queue where each extracted node has the maximum estimated distance from the sink, the worst-case running time decreases to $O(\sqrt{m}n^2)$, which is much better for sparse graphs. The last algorithm is known as the highest-level preflow-push algorithm.

Unfortunately, regardless of the selection strategy, the push-relabel method yields in practice very slow codes if taken "ad litteram". Indeed, the way estimated distances from the sink are maintained has been proved to dramatically affect the practical performances of the push-relabel algorithms. For this reason, several additional heuristics for the problem have been proposed. Though these heuristics are irrelevant from an asymptotic point of view, the experimental study presented in [10] proves that two of them, i.e., the global relabeling and the gap heuristics, could be extremely useful in practice.

Global Relabeling Heuristic. Each relabel operation increases the estimated distance of the current active node from the sink to be equal to the lowest estimated distance of any adjacent node, plus one. This is done by considering only adjacent nodes joined by edges with some non-zero residual capacity, i.e., edges that can still carry some additional flows. As relabel operations are indeed local operations, the estimated distances from the sink may progressively deviate from the exact distances by losing the "big picture" of the distances: for this reason, flow excesses might not be correctly pushed right ahead towards the sink, and may follow longer paths slowing down the computation. The global relabeling heuristic consists of recomputing, say every n push/relabel operations, the exact distances from the sink, and the asymptotic cost of doing so can be amortized against the previous operations. This heuristic drastically improves the practical running time of algorithms based on the push-relabel method [10].

Gap Heuristic. Different authors [9] have observed that, at any time during the execution of the algorithm, if there are nodes with estimated distances from the sink that are strictly greater than some distance d and no other node has estimated distance d, then a gap in the distances has been formed and all active nodes above the gap will eventually send their flow excesses back to the source as they no more reach the sink. This is achieved by the algorithm through successive repeated increments of the estimated distances done via relabel operations until they get greater than n. The problem is that a huge number of such relabeling operations may be required. To avoid this, it is possible to efficiently keep track of gaps in the distances: whenever a gap occurs, the estimated distances of all nodes above the gap are immediately increased to n. This is usually referred to as the gap heuristic and, according to the study in [10], it is a very useful addition to the global relabeling heuristics if the highest-level active node selection strategy is applied. However, the gap heuristic does not seem to yield the same improvements under FIFO selection strategy.

The 5 snapshots a, b, c, d and e shown in Fig. 1 and in Fig. 2 have been produced by the algorithm animation system Leonardo [12] and depict the behavior of the highest-level preflow push algorithm implemented with no additional heuristics on a small network with 19 nodes and 39 edges. The animation aims at giving an empirical explanation about the utility of the gap heuristic under the highest-level selection. The example shows that this heuristic, if added to the code, could have saved about 80% of the total time spent by the algorithm to solve the problem on that instance. Both the network and a histogram of the estimated distances of nodes are shown in the snapshots: active nodes are highlighted both in the network and in the histogram and flow excesses are reported as node labels. Moreover, the edge currently selected for a push operation is highlighted as well. Notice that the source is initially assigned distance n and all nodes that eventually send flows back to the source get distance greater than n.

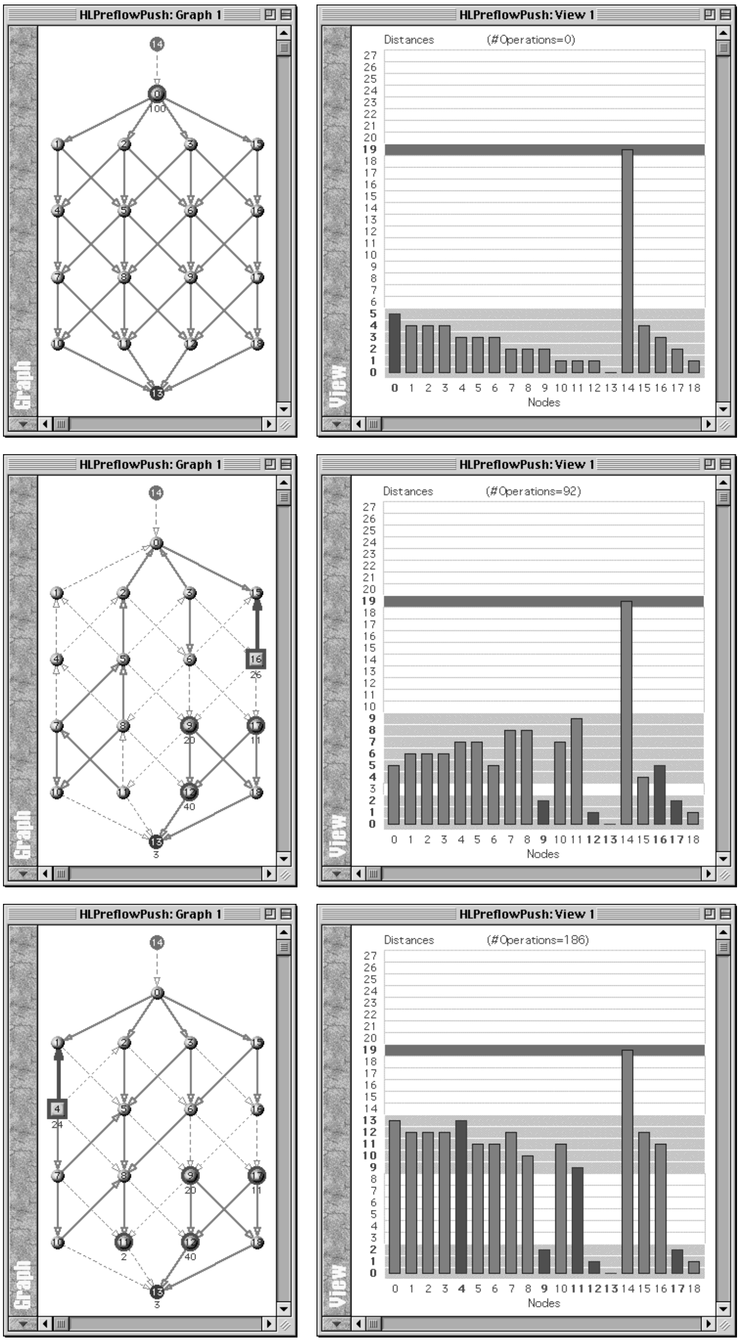
3.2 Matrix Multiplication

In this section we briefly report on an experimental study of matrix multiplication [5]. The study is an example of the fact that theoretical conclusions on locality exploitation can yield practical implementations with the desired properties. Differently from the case of Maxflow presented in Section 3.1 where experiments helped theoreticians develop suitable heuristics for improving the running time, experiments in this case provided an empirical confirmation about the precision of theoretical performance prediction related to certain matrix multiplication algorithms.

In general, as memory hierarchy is of no help to performance if the computation exhibits an insufficient amount of locality, both algorithm design and compiler optimizations should explicitly take into account locality.

For what concerns matrix multiplication, a simple approach, called *fractal* approach [5], allows it to design algorithms that expose locality at all temporal scales. The main idea consists of decomposing matrices recursively embedding any two-dimensional array X into a one-dimensional array Y as follows:

$$X = \begin{array}{|c|c|}\hline A_0 & A_1 \\ \hline A_2 & A_3 \\ \hline \end{array} \qquad\qquad Y = \begin{array}{|c|c|c|c|}\hline A_0 & A_1 & A_2 & A_3 \\ \hline \end{array}$$



(a)

Network status and distances after the initialization phase.

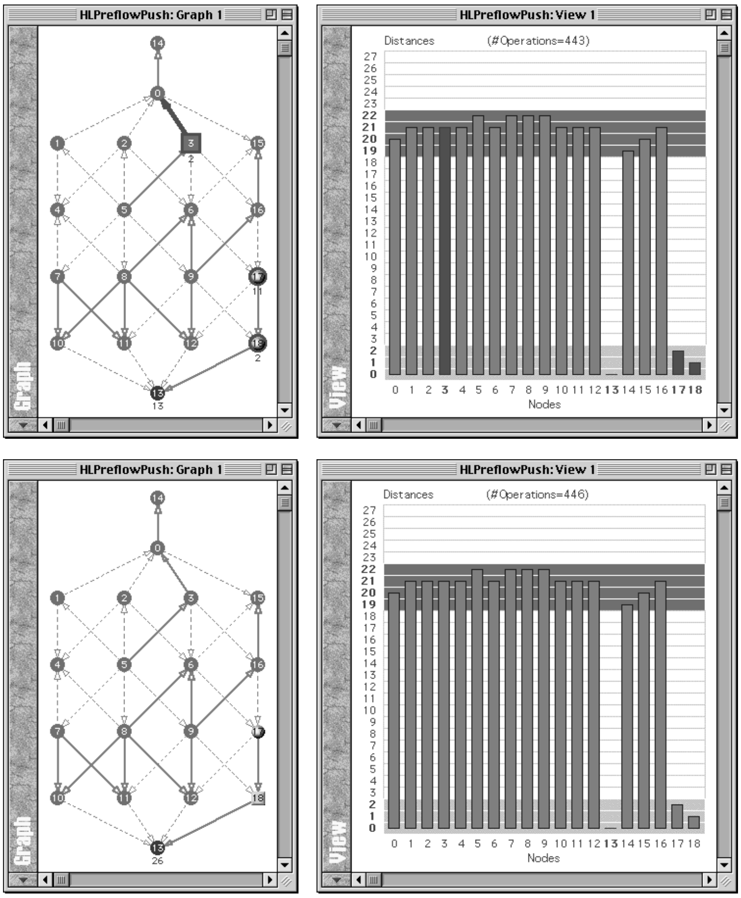
(b)

After 92 operations a gap has been formed. Nodes with distance greater than the gap no more reach the sink. Their distance should be directly increased to n through the gap heuristic.

(c)

Nodes with distance greater than the gap are being slowly relabeled step after step if the gap heuristic is not implemented.

Figure 1: Highest-level preflow push maxflow algorithm animation. Snapshots a, b, c.



(d)

After 443 operations the distances of all nodes above the gap have been increased to n and their flow excesses are being drained back to the source. The gap heuristic could have saved the last 351 operations on this instance, i.e., about 80% of the total time spent by the algorithm to solve the problem.

(e)

After 446 operations the maximum flow has been determined by the algorithm and no more active nodes remain.

Figure 2: Highest-level preflow push maxflow algorithm animation. Snapshots d, e.

Keeping in mind this decomposition, it is possible to define a class of recursive algorithms that compute a product $C \leftarrow A \cdot B$ by performing 8 recursive computations on smaller matrices. Below we show two possible orderings for computing the multiplications of sub-matrices, corresponding to two different algorithms, namely `ABC-fractal` and `CAB-fractal`:

	`ABC-fractal`
1	$C_0 \leftarrow C_0 + A_0 \cdot B_0$
2	$C_1 \leftarrow C_1 + A_0 \cdot B_1$
3	$C_3 \leftarrow C_3 + A_2 \cdot B_1$
4	$C_2 \leftarrow C_2 + A_2 \cdot B_0$
5	$C_2 \leftarrow C_2 + A_3 \cdot B_2$
6	$C_3 \leftarrow C_3 + A_3 \cdot B_3$
7	$C_1 \leftarrow C_1 + A_1 \cdot B_3$
8	$C_0 \leftarrow C_0 + A_1 \cdot B_2$

	`CAB-fractal`
1	$C_0 \leftarrow C_0 + A_0 \cdot B_0$
2	$C_0 \leftarrow C_0 + A_1 \cdot B_2$
3	$C_1 \leftarrow C_1 + A_0 \cdot B_1$
4	$C_1 \leftarrow C_1 + A_1 \cdot B_3$
5	$C_3 \leftarrow C_3 + A_2 \cdot B_1$
6	$C_3 \leftarrow C_3 + A_3 \cdot B_3$
7	$C_2 \leftarrow C_2 + A_2 \cdot B_0$
8	$C_2 \leftarrow C_2 + A_3 \cdot B_2$

From the perspective of temporal locality, there is always a sub-matrix in common between consecutive calls, which increases data reuse. In particular, it is not difficult to see that both `ABC-fractal` and `CAB-fractal` actually maximize data reuse. Moreover, the first algorithm optimizes *read* cache misses, while the second optimizes *write* cache misses. This is clear if we consider, for example, that sub-matrix sharing between consecutive calls in `CAB-fractal` is maximum for C, which is the matrix being written.

Experiments showed that these algorithms can efficiently exploit the cache hierarchy without taking cache parameters into account, thus ensuring portability of cache performance [5].

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[2] R.K. Ahuia, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[3] R. Anderson. The role of experiment in the theory of algorithms. In *Proceedings of the 5th DIMACS Challenge Workshop*, 1996. Available over the Internet at the URL: `http://www.cs.amherst.edu/ dsj/methday.html`.

[4] C. Berge and A. Ghouila-Houri. *Programming, Games and Transportation Networks.* Wiley, 1962.

[5] G. Bilardi, P. D'Alberto, and A. Nicolau. Fractal matrix multiplication: a case study on portability of cache performance. Manuscript, May 2000.

[6] M.H. Brown. *Algorithm Animation.* MIT Press, Cambridge, MA, 1988.

[7] M.H. Brown. Zeus: a System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 7-th IEEE Workshop on Visual Languages*, pages 4–9, 1991.

[8] G. Cattaneo, U. Ferraro, G.F. Italiano, and V. Scarano. Cooperative Algorithm and Data Types Animation over the Net. In *Proc. XV IFIP World Computer Congress, Invited Lecture*, pages 63–80, 1998. System Home Page: `http://isis.dia.unisa.it/catai/`.

[9] B.V. Cherkassky. *A Fast Algorithm for Computing Maximum Flow in a Network.* In A.V. Karzanov editor, *Collected Papers, Issue 3: Combinatorial Methods for Flow Problems,* pages 90-96. The Institute for Systems Studies, Moscow, 1979. In Russian. English translation appears in AMS Translations, Vol. 158, pp. 23–30. AMS, Providence, RI, 1994.

[10] B.V. Cherkassky and A.V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.

[11] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[12] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible Execution and Visualization of Programs with LEONARDO. *Journal of Visual Languages and Computing*, 11(2), 2000. Leonardo is available at the URL: `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

[13] G.B. Dantzig. *Application of the Simplex Method to a Transportation Problem.* In T.C. Hoopmans editor, *Activity Analysis and Production and Allocation*, Wiley, New York, 1951.

[14] Demetrescu, C. and Finocchi, I. Break the "Right" Cycles and Get the "Best" Drawing. In *Proc. of the 2nd International Conference on Algorithms and Experimentations (ALENEX'00), San Francisco, CA*, 2000.

[15] M. Eisenstadt and M. Brayshaw. The transparent prolog machine: An execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):1–66, 1988.

[16] A.V. Goldberg. Selecting problems for algorithm evaluation. In *Proc. 3-rd Workshop on Algorithm Engineering (WAE'99), LNCS 1668*, pages 1–11, 1999.

[17] A.V. Goldberg and B.M.E. Moret. Combinatorial algorithms test sets [CATS]: The ACM/EATCS platform for experimental research (short). In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[18] R.R. Henry, K.M. Whaley, and B. Forstall. The University of Washington Program Illustrator. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 223–233, 1990.

[19] D. Johnson. A theoretician's guide to the experimental analysis of algorithms. In *Proceedings of the 5th DIMACS Challenge Workshop*, 1996. Available over the Internet at the URL: `http://www.cs.amherst.edu/ dsj/methday.html`.

[20] D. Klingman, A. Napier, and J. Stutz. Netgen: A program for generating large scale capacitated assignment, transportation, and minimum cost network flow problems. *Management Science*, 20:814–821, 1974.

[21] Donald E. Knuth. Stanford GraphBase: A platform for combinatorial algorithms. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 41–43, New York, NY 10036, USA, 1993. ACM Press.

[22] S.P. Lahtinen, E. Sutinen, and J. Tarhio. Automated Animation of Algorithms with Eliot. *Journal of Visual Languages and Computing*, 9:337–349, 1998.

[23] T. Leong, P. Shor, and C. Stein. Implementation of a combinatorial multicommodity flow algorithm. In D.S. Johnson and C.C. McGeoch, eds., *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 387–406, 1993.

[24] C. McGeoch. A bibliography of algorithm experimentation. In *Proceedings of the 5th DIMACS Challenge Workshop*, 1996. Available over the Internet at the URL: `http://www.cs.amherst.edu/ dsj/methday.html`.

[25] K. Mehlhorn and S. Naher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.

[26] B.M.E. Moret. Towards a discipline of experimental algorithmics. In *Proceedings of the 5th DIMACS Challenge Workshop*, 1996. Available over the Internet at the URL: `http://www.cs.amherst.edu/ dsj/methday.html`.

[27] B.M.E. Moret and H.D. Shapiro. An empirical assessment of algorithms for constructing a minimal spanning tree. *Computational Support for Discrete Mathematics* N. Dean and G. Shannon eds. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 15:99–117, 1994.

[28] G.C. Roman, K.C. Cox, C.D. Wilcox, and J.Y Plun. PAVANE: a System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.

[29] S. Skiena. Who is interested in algorithms and why? lessons from the stony brook algorithms repository. In *Proc. Workshop on Algorithm Engineering (WAE'98)*, pages 204–212, 1998.

[30] J.T. Stasko. The Path-Transition Paradigm: a Practical Methodology for Adding Animation to Program Interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, 1990.

[31] J.T. Stasko. A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18:258–264, 1993.

[32] J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price. *Software Visualization: Programming as a Multimedia Experience.* MIT Press, Cambridge, MA, 1997.