many existing algorithms. One of these recurring building blocks is the Fast Fourier Transform (FFT) algorithm. The SFT algorithm offers a great efficiency improvement over the FFT algorithm for applications where it suffices to deal only with the significant Fourier coefficients. In such applications, replacing the FFT building block with the SFT algorithm accelerates the $\Theta(N \log N)$ complexity in each application of the FFT algorithm to $poly(\log N)$ complexity [1]. Lossy compression is an example of such an application [1,5,8]. To elaborate, central component in several transform compression methods (e. g., JPEG) is to first apply Fourier (or Cosine) transform to the signal, and then discard many of its coefficients. To accelerate such algorithms —instead of computing the entire Fourier (or Cosine) transform—the SFT algorithm can be used to directly approximate only the significant Fourier coefficients. Such an accelerated algorithm achieves compression guarantee as good as the original algorithm (and possibly better), but with running time improved to $poly(\log N)$ in place of the former $\Theta(N \log N)$.

Cross References

- ▶ Abelian Hidden Subgroup Problem
- ▶ Learning Constant-Depth Circuits
- ▶ Learning DNF Formulas
- ▶ Learning Heavy Fourier Coefficients of Boolean Functions
- ▶ Learning with Malicious Noise
- ▶ List Decoding near Capacity: Folded RS Codes
- ▶ PAC Learning
- ▶ Statistical Query Learning

Recommended Reading

1. Akavia, A., Goldwasser, S.: Manuscript submitted as an NSF grant, awarded (2005) CCF-0514167
2. Akavia, A., Goldwasser, S., Safra, S.: Proving hard-core predicates using list decoding. In: Proceedings of the 44th Symposium on Foundations of Computer Science (FOCS'03), pp. 146–157. IEEE Computer Society (2003)
3. Atici, A., Servedio, R.A.: Learning unions of $\omega(1)$-dimensional rectangles. In: ALT, pp. 32–47 (2006)
4. Blum, M., Micali, S.: How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. SIAM J. Comput. **4**(13), 850–864 (1984)
5. Cormode, G., Muthukrishnan, S.: Combinatorial algorithms for compressed sensing. In: Structural Information and Communication Complexity, 13th International Colloquium, SIROCCO (2006), Chester, UK, July 2–5, 2006 pp. 280–294
6. Gilbert, A.C., Guha, S., Indyk, P., Muthukrishnan, S., Strauss, M.: Near-optimal sparse fourier representations via sampling. In: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, pp. 152–161. ACM Press (2002)
7. Gilbert, A.C., Muthukrishnan, S., Strauss, M.J.: Improved time bounds for near-optimal sparse fourier representation via sampling. In: Proceedings of SPIE Wavelets XI, San Diego, CA 2005 (2005)
8. Gilbert, A.C., Strauss, M.J., Tropp, J.A., Vershynin, R.: One sketch for all: Fast algorithms for compressed sensing. In: 39th ACM Symposium on Theory of Computing (STOC'07)
9. Goldreich, O., Levin, L.: A hard-core predicate for all one-way functions. In: 27th ACM Symposium on Theory of Computing (STOC'89) (1989)
10. Mansour, Y.: Randomized interpolation and approximation of sparse polynomials. SIAM J. Comput. **24**, 357–368 (1995)
11. Sudan, M.: List decoding: algorithms and applications. SIGACT News **31**, 16–27 (2000)

LEDA: a Library of Efficient Algorithms

1995; Mehlhorn, Näher

Christos Zaroliagis
Department of Computer Engineering and Informatics, University of Patras, Patras, Greece

Keywords and Synonyms

LEDA platform for combinatorial and geometric computing

Problem Definition

In the last forty years, there has been a tremendous progress in the field of computer algorithms, especially within the core area known as *combinatorial algorithms*. Combinatorial algorithms deal with objects such as lists, stacks, queues, sequences, dictionaries, trees, graphs, paths, points, segments, lines, convex hulls, etc, and constitute the basis for several application areas including network optimization, scheduling, transport optimization, CAD, VLSI design, and graphics. For over thirty years, asymptotic analysis has been the main model for designing and assessing the efficiency of combinatorial algorithms, leading to major algorithmic advances.

Despite so many breakthroughs, however, very little had been done (at least until 15 years ago) about the practical utility and assessment of this wealth of theoretical work. The main reason for this lack was the absence of a standard *algorithm library*, that is, of a software library that contains a systematic collection of robust and efficient implementations of algorithms and data structures, upon which other algorithms and data structures can be easily built.

The lack of an algorithm library limits severely the great impact which combinatorial algorithms can have.

The continuous re-implementation of basic algorithms and data structures slows down progress and typically discourages people to make the (additional) effort to use an efficient solution, especially if such a solution cannot be re-used. This makes the migration of scientific discoveries into practice a very slow process.

The major difficulty in building a library of combinatorial algorithms stems from the fact that such algorithms are based on complex data types, which are typically not encountered in programming languages (i. e., they are not built-in types). This is in sharp contrast with other computing areas such as statistics, numerical analysis, and linear programming.

Key Results

The currently most successful algorithm library is LEDA (Library for Efficient Data types and Algorithms) [4,5]. It contains a very large collection of advanced data structures and algorithms for combinatorial and geometric computing. The development of LEDA started in the early 1990s, it reached a very mature state in the late 1990s, and it continues to grow. LEDA has been written in C++ and has benefited considerably from the object-oriented paradigm.

Four major goals have been set in the design of LEDA.

1. *Ease of use*: LEDA provides a sizable collection of data types and algorithms in a form that they can be readily used by non-experts. It gives a precise and readable specification for each data type and algorithm, which is short, general and abstract (to hide the details of implementation). Most data types in LEDA are parameterized (e. g., the dictionary data type works for arbitrary key and information type). To access the objects of a data structure by position, LEDA has invented the *item concept* that casts positions into an abstract form.
2. *Extensibility*: LEDA is easily extensible by means of parametric polymorphism and can be used as a platform for further software development. Advanced data types are built on top of basic ones, which in turn rest on a uniform conceptual framework and solid implementation principles. The main mechanism to extend LEDA is through the so-called LEDA extension packages (LEPs). A LEP extends LEDA into a particular application domain and/or area of algorithms that is not covered by the core system. Currently, there are 15 such LEPs; for details see [1].
3. *Correctness*: In LEDA, programs should give sufficient justification (proof) for their answers to allow the user of a program to easily assess its correctness. Many algorithms in LEDA are accompanied by *program checkers*. A program checker C for a program P is a (typically very simple) program that takes as input the input of P, the output of P, and perhaps additional information provided by P, and verifies that the answer of P in indeed the correct one.
4. *Efficiency*: The implementations in LEDA are usually based on the asymptotically most efficient algorithms and data structures that are known for a problem. Quite often, these implementations have been fine-tuned and enhanced with heuristics that considerably improve running times. This makes LEDA not only the most comprehensive platform for combinatorial and geometric computing, but also a library that contains the currently fastest implementations.

Since 1995, LEDA is maintained by the Algorithmic Solutions Software GmbH [1] which is responsible for its distribution in academia and industry.

Other efforts for algorithm libraries include the Standard Template Library (STL) [7], the Boost Graph Library [2,6], and the Computational Geometry Algorithms Library (CGAL) [3].

STL [7] (introduced in 1994) is a library of interchangeable components for solving many fundamental problems on sequences of elements, which has been adopted into the C++ standard. It contributed the *iterator concept* which provides an interface between *containers* (an object that stores other objects) and algorithms. Each algorithm in STL is a function template parameterized by the types of iterators upon which it operates. Any iterator that satisfies a minimum set of requirements can be used regardless of the data structure accessed by the iterator. The systematic approach used in STL to build abstractions and interchangeable components is called *generic programming*.

The Boost Graph Library [2,6] is a C++ graph library that applies the notions of generic programming to the construction of graph algorithms. Each graph algorithm is written not in terms of a specific data structure, but instead in terms of a graph abstraction that can be easily implemented by many different data structures. This offers the programmer the flexibility to use graph algorithms in a wide variety of applications. The first release of the library became available in September 2000.

The Computational Geometry Algorithms Library [3] is another C++ library that focuses on geometric computing only. Its main goal is to provide easy access to efficient and reliable geometric algorithms to users in industry and academia. The CGAL library started in 1996 and the first release was in April 1998.

Among all libraries mentioned above LEDA is by far the best (both in quality and efficiency of implementations) regarding combinatorial computing. It is worth

mentioning that the late versions of LEDA have also incorporated the iterator concept of STL.

Finally, a notable effort concerns the Stony Brook Algorithm Repository [8]. This is not an algorithm library, but a comprehensive collection of algorithm implementations for over seventy problems in combinatorial computing, started in 2001. The repository features implementations coded in different programming languages, including C, C++, Java, Fortran, ADA, Lisp, Mathematic, and Pascal.

Applications

An algorithm library for combinatorial and geometric computing has a wealth of applications in a wide variety of areas, including: network optimization, scheduling, transport optimization and control, VLSI design, computer graphics, scientific visualization, computer aided design and modeling, geographic information systems, text and string processing, text compression, cryptography, molecular biology, medical imaging, robotics and motion planning, and mesh partition and generation.

Open Problems

Algorithm libraries usually do not provide an interactive environment for developing and experimenting with algorithms. An important research direction is to add an interactive environment into algorithm libraries that would facilitate the development, debugging, visualization, and testing of algorithms.

Experimental Results

The are numerous experimental studies based on LEDA, STL, Boost, and CGAL, most of which can be found in the world-wide web. Also, the web sites of some of the libraries contain pointers to experimental work.

URL to Code

The afore mentioned algorithm libraries can be downloaded from their corresponding web sites, the details of which are given in the bibliography (Recommended Reading).

Cross References

- ► Engineering Algorithms for Large Network Applications
- ► Experimental Methods for Algorithm Analysis
- ► Implementation Challenge for Shortest Paths
- ► Shortest Paths Approaches for Timetable Information
- ► TSP-Based Curve Reconstruction

Recommended Reading

1. Algorithmic Solutions Software GmbH, http://www.algorithmic-solutions.com/. Accessed February 2008
2. Boost C++ Libraries, http://www.boost.org/. Accessed February 2008
3. CGAL: Computational Geometry Algorithms Library, http://www.cgal.org/. Accessed February 2008
4. Mehlhorn, K., Näher, S.: LEDA: A Platform for Combinatorial and Geometric Computing. Commun. ACM. **38**(1), 96–102 (1995)
5. Mehlhorn, K., Näher, S.. LEDA: A Platform for Combinatorial and Geometric Computing. Cambridge University Press, Boston (1999)
6. Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library. Addison-Wesley, Cambridge (2002)
7. Stepanov, A., Lee, M.: The Standard Template Library. In: Technical Report X3J16/94–0095, WG21/N0482, ISO Programming Language C++ Project. Hewlett-Packard, Palo Alto CA (1994)
8. The Stony Brook Algorithm Repository, http://www.cs.sunysb.edu/~algorith/. Accessed February 2008

Leontief Economy Equilibrium

2005; Codenotti, Saberi, Varadarajan, Ye 2005; Ye

Yin-Yu Ye
Department of Management Science and Engineering, Stanford University, Stanford, CA, USA

Keywords and Synonyms

Exchange market equilibrium with the leontief utility

Problem Definition

The Arrow–Debreu exchange market equilibrium problem was first formulated by Léon Walras in 1874 [7]. In this problem everyone in a population of m traders has an initial endowment of a divisible goods and a utility function for consuming all goods – their own and others'. Every trader sells the entire initial endowment and then uses the revenue to buy a bundle of goods such that his or her utility function is maximized. Walras asked whether prices could be set for everyone's goods such that this is possible. An answer was given by Arrow and Debreu in 1954 [1] who showed that, under mild conditions, such equilibrium would exist if the utility functions were concave. In general, it is unknown whether or not an equilibrium can be computed efficiently, see, e. g., ► General Equilibrium.

Consider a special class of Arrow–Debreu's problems, where each of the n traders has exactly one unit of a divisible and distinctive good for trade, and let trader i, $i = 1, \ldots, n$, bring good i, which class of problems is called the *pairing class*. For given prices p_j on good j, consumer