A Fascinating Science

Jan van Leeuwen

Information and Computing Sciences, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

1 Algorithmic Systems

... decades of stunningly rapid advances in processing speed, storage and networking, along with the development of increasingly clever software, have brought computing into science, business and culture in ways that were barely imagined years ago. The quantitative changes delivered through [science and] smart engineering opened the door to qualitative changes. Computing changes what can be seen, simulated and done. (Stephen Lohr, 2006)

The term *Informatica* was introduced in the Dutch language in 1964. Few people foresaw at the time that Informatics would develop into the major scientific discipline that it is now. Every day we marvel at the discoveries in Informatics and their wide-ranging impact. At one end of the spectrum, there are the applications of computers as we see them around us: palmtops, laptops, desktops, network computers, and all the software that is running on them. Businesses and administrations depend on the systems that run on top of them, systems which take informaticians considerable ingenuity to design and get right for us as users. If you are not an 'informaticus', I guess that this is the part of the spectrum that you associate most with Informatics.

There is another end of the spectrum. This is the creative world in which we explore new application domains and problems that have not yet been mastered, or not sufficiently well, by effective techniques of informatisation. Here we find the fascinating *science* of informatics. It is the domain of many new problems that we have yet to understand and capture in a model or a processing method that works nicely, and in a jiffy, on a computer in your institute, your company, at home or, if you like, in your car. Subsequent challenges in information structuring, programming and interaction design arise in order to render and/or program the discovered solutions and embed, release and maintain the solutions with novel software-, information- and/or media technologies.

How are information-based processes modelled and how are the models designed for efficient processing? This is the field of *algorithmic systems research* and of the study of the *inherent complexity of problems and information processing*. It is the frontline of Informatics and an exciting world of computational ideas that should go into future software and information systems. An example is: how would one compute the fastest route from your home to your office (assuming it needs a car-ride), given predications about traffic density on all roads and updating it real-time with factual data from traffic sensors along the roads, with a 99% accurate estimate of the travel time, all online and updating it while you're driving. The example leaves room even for new notions of distributed intelligence and computational learning, conceivably leading to algorithms that evolve depending on their interaction with an unpredictable environment.

Here are some examples of recent developments in algorithmic systems research:

- Bio-informatics: algorithms for sequence alignment, gene finding, protein structure alignment, and simulations of cellular systems.
- Business intelligence: data analysis, algorithms for datamining.
- Communication technology: data compression algorithms for multimedia.
- Corporate structure analysis ('understanding organisations'): optimizing work-flow, resource management along supply chains.
- E-commerce: trustworthy and secure e-commerce using algorithmic agents, algorithmic mechanisms and auctions, computing Nash equilibria.
- E-science: algorithms for massive data, multi/many-core programming, grid computing, high performance computing.
- Game design: 3D modelling, indoors/outdoors rendering, character animation, cameras, texturing, lighting and shading algorithms, efficient algorithms for computer graphics.
- Green IT: algorithms for power saving, energy-efficient computing, virtualisation, teleworking protocols.

- Logistics: bus- and railway schedules, airport planning, multi-modal transport planning, vehicle routing, navigation systems.
- Production: work planning and scheduling, algorithms for resource management, delivery planning.
- Security: enhancements in public key encryption standards, improvements on standing cryptographic protocols.
- Sensor networks: network structure analysis, distributed communication protocols for ad hoc networks, amorphous computing.
- Web science: modeling the World Wide Web, understanding power laws, principles of social networks.

Ingredients for algorithmic systems research are a solid understanding of the problem domain, a deep analysis of the structure of the problems at hand, an extensive range of algorithmic modeling techniques, programming and experimentation skills, and a superb understanding of the (desirable) powers of future computing systems.

An important question is of course: *what will future computing systems look like* and *what will they be capable of*? Most people agree that future computers will likely consist of many, autonomous components that interact, self-organize, learn and adapt to their environment. They will be always on and operate in an infrastructure ('cloud') that connects them to many other systems and services. It is also speculated that new and faster forms of computing may be gleaned from suitable phenomena in nature, e.g. from *quantum mechanics* or *biomolecular science*. The possible impact of quantum computing or molecular machines on the practice of algorithmic systems is an intriguing question.

2 Complexity Matters

My favorite way to describe computer science is to say that it is the study of algorithms. (D.E. Knuth, 1974)

Programs are the single most powerful modeling tool for science at large, hence algorithms and algorithm design are in the center of Informatics. The Department of Information and Computing Sciences in Utrecht has been very active and prominent in algorithmic systems research from its beginning in

1975 onward. In several chairs, research eventually aims at the design or the effective implementation of algorithms.

Algorithm design may seem easy, and sometimes it is, but in general it isn't. Problems often seem to resist an adequate modeling or even a complete algorithmic rendering of the essential processes that lie at their heart. This leads to the notion of *complexity* of problems and, likewise, of algorithms and computations, of programming, and of any system or organization that informaticians design. It has implied that the following questions are leading in our work:

- **understanding complexity**, and
- **developing the concepts for dealing with it**.

The extreme case in understanding complexity is that a problem can be *algorithmically unsolvable.* We give an example from the familiar field of data compression.

The main idea of data compression is that one has an original object x (e.g. a text string, a photo or a video-stream), and wants to compute a transformed image of x, $A(x)$ such that $A(x)$ has fewer bits than x. Clearly one also wants a feasible algorithm D such that $D(A(x)) = x$, i.e. such that $A(x)$ can be decompressed back to the original x. The question now rises whether there could be a single algorithm which compresses all inputs optimally. Let's argue why such an algorithm cannot exist. Suppose we had an algorithm K such that $K(x)$ is the smallest input-less program which outputs x. Think of $K(x)$ as the smallest program that holds the compression of x and that decompresses it upon call.

Proposition 1 *$K(x)$ is not a computable function, i.e. there does not exist a program that computes $K(x)$ given x for all x.*

Proof: Assume $K(x)$ is computable by program π of size s. Let $1, \ldots$ be an enumeration of the infinitely many possible inputs x by increasing size, $|x|$. Now make a program π' that looks like this (we ignore the issue of a required standard programming framework here):

```
for $i = 1$ to $\infty$ do
if $|K(i)| > 10 \cdot s + 1000$ then output $i$ and stop fi
od.
```

This program halts, because there can be only finitely many different programs of any given size. It will output an i such that $|K(i)| > 10 \cdot s + 1000$.

Note that π' is a program that outputs i and is of size $s + log(s)$ + some small constants, which is less than $10 \cdot s + 1000$. Thus π' is a program for i that is smaller than $K(i)$. This is a contradiction. Thus $K(x)$ is not computable. ■

Thus a universal algorithm that compresses all inputs optimally does not exist. It implies that compression methods necessarily must be 'lossy' or not always optimal, especially for large inputs.

2.1 Challenges

The computational problems we study in Informatics are basically all finite but they can be so large that they can hardly be distinguished from infinite one's. The biggest challenge that we face in algorithm design is therefore: *how to cope with the (very) large.* Or to say it differently: *how to solve a problem and how to stretch the limited capabilities of memory and processing power of a given machine in order to be able to solve problem instances of largest possible size within acceptable time bounds.* We study this for many problems: in organizational systems, information networks, supply chains, transportation systems, security, and so on. In all these domains we try to figure out what the intrinsic problem complexities are and what the boundaries are of what is practically solvable by discrete algorithms. And in the process we develop algorithms that stand the test.

Computational problems are studied with the powers of contemporary computers and networks in mind. However, the ongoing evolution of computational systems is explicitly of interest as well. In Informatics, the future is often only a few years ahead. It thus pays to anticipate the new computational features in visionary *models of computation* and to think of the new algorithm designs needed. But, how to create, program and exploit these visionary systems? It often challenges the intuitions of computability theory known since the 1930's. Clearly we can't expect algorithmically unsolvable problems to suddenly become solvable by new hardware inventions, but computational limits may shift dramatically in practice.

Already for a number of years we study models of computation. The question is: what would be a good generic theory that enables one to demarcate the computational powers of new, future computing systems? Among the concrete systems we study at the moment are *sensor-based systems*: to understand their computational abilities requires a mix of geometry, graph theory and distributed computing. Also, as sensors typically are very small and have very small batteries, we run into a relatively new phenomenon: *energy-aware computations*, in which we must arrange for the sensors to compute as much as possible and as long as possible – as long as their batteries last. Easy problems can become hard if computations must be spread so the energy usage by the sensors gets balanced.

3 Algorithmic Modeling

> *At the heart of cell phones, airlines and airplanes, financial transactions, company management, publishing of any kind, the Internet and World Wide Web, industrial plant control and all other devices and processes that make today's world run, lie algorithms and data structures devised by computer scientists.* (In: 'Why study computer science', ETH Zürich, 2007.)

Algorithms are solutions to problems. But how do we get from a problem to an acceptable solution or processing method? The answer is through algorithmic modeling and design, an important part of the *development cycle* of any system.

Algorithmic modeling is applied to a multitude of problems. Applications range from programming and game design to optimizing *business processes.* Algorithmic modeling applied to business processes is widely applied in domains like *management science* and *operations research.*

Algorithmic modeling uses a more or less standard approach consisting of the following phases:

1. *model the problem,*
2. *determine the quality of the feasible solutions,*
3. *assess the quality of the algorithms to solve the model,*
4. *implement the solution of choice,* and
5. *integrate and/or scale the solution.*

The result of this approach should be an effective, implemented (component of an) algorithmic system. Clearly there are feedback loops in the methodology: if no good algorithms are found in step 3, one may need to change the model (step 1) or relax the quality of the solutions considered acceptable (step 2).

Algorithmic systems is a prominent area of research world-wide. the results are applied in many disciplines worldwide. Many journals, conferences, workshops and international research projects are devoted to it, and Utrecht is a well-known center of it. Some examples of problem domains in algorithmic systems that are pursued in in Utrecht are the following:

- discovery of patterns in large data sets ('data mining'),
- complexity analysis of graphical models for probabilistic inference,
- distributed coordination of processes in ad-hoc mobile networks,
- combinatorial optimization e.g. in railway and airport planning,
- planning and scheduling of activities along product lines and supply chains,
- structural analysis of networks and design of network algorithms,
- optimal design of (large-scale) business processes, and
- computational techniques for the support of management decisions.

3.1 Techniques

Among the typical modeling tools of algorithmic design are: decomposition, dynamic programming, graphs and networks, greedy and heuristic algorithms, integer linear and nonlinear programming, local search, probabilistic algorithms, p-reduction, randomisation, transition systems, Petri nets, and transformations into applied logic frameworks. It is only a sample of the tools and techniques that are applied by the modern algorithmician.

Qualities of algorithms can be investigated by theoretical analysis, by experimentation, or by a combination of the two. In particular, *experimental algorithm design* is concerned both with the experimental evaluation and engineering of algorithms and with the *invention of new algorithms* that cannot be thought of just from existing theory. Clearly, the methodology of algorithmic modelling requires that, ideally, the qualities of algorithms are assessed in some way, by mathematical proof or otherwise.

We give a classical example of algorithmic modelling from the domain of *scheduling*. Scheduling always deals with tasks ('jobs') on the one hand and agents ('machines') on the other. Can one compute schedules with minimum *makespan*, i.e. in which the latest completion time of any job is smallest? When modeling a scheduling problem one must take many aspects into account:

- *the properties of the jobs.* This involves e.g. their processing time, release time, deadline, and priority or weight.
- *the properties of the machines.* One may assume e.g. identical machines or unrelated machines, in which case jobs may take a different processing time on different machines. Typically, all machines are active in *in parallel.*
- *the properties of the processing.* One may consider several constraints, e.g. precedence constraints, allowed waiting times, penalties for delays (lateness), whether or not pre-emption is allowed, or whether we consider on-line or off-line scheduling.
- *the objective of the scheduling.* Usually one looks for a schedule of minimum cost, whatever the cost criterion is.

Consider m machines and n jobs with respective processing times $p_1, \ldots, p_n \in \mathbb{N}_+$. We assume that the jobs are given in a linear list, which is equivalent to assuming that the n jobs arrive in an on-line manner with no special ordering of their processing times. All machines are equivalent and pre-emptions are not allowed. The question is whether there exists a good algorithm for minimum makespan scheduling. We look at a simple algorithm called *list scheduling.*

```
List scheduling
while the list is not empty do
begin
    • take the next job from the head of the list and delete it from the list
    • assign the job to the machine with the least amount of work assigned to it so far
end
return the schedule so obtained.
```

The list scheduling algorithm is easy to implement and efficient, by keeping track of the workload of the machines in a heap. Let the *performance ratio* of an algorithm be the multiplicative factor that indicates how far the output of the algorithm is guaranteed to be close to the optimal solution. Performance ratios are a simple measure for comparing algorithms that do not always produce optimal solutions.

Proposition 2 *List scheduling achieves a performance ratio* $\leq 2 - \frac{1}{m}$.

Proof: Let $W = \sum_{i=1}^{n} p_i$ be the sum of all processing times to be accommodated. We know that the total processing time available in an optimal schedule on the machines is $m \cdot OPT$. So, $OPT \geq \frac{W}{m}$. Moreover, $OPT \geq p_k$ for every k. Let A be the makespan of the schedule produced by the algorithm. By definition there must be a job k, with processing time p_k, that ends at the makespan time. No machine can end its task before $A - p_k$, because then job k would have been scheduled on that machine, thus reducing the makespan. So all machines are busy from time 0 through $A - p_k$. Consequently, $W \geq m \cdot (A - p_k) + p_k$ and thus

$$A \leq \frac{W}{m} + \frac{m-1}{m} p_k \leq OPT + (1 - \frac{1}{m})OPT = (2 - \frac{1}{m})OPT.$$

■

List scheduling can do badly if long jobs at the end of the list spoil an even division of processing times. The performance ratio is the best in worst case. Interestingly, if all jobs are known in advance and *sorted* in order of decreasing processing times: $p_1 \geq p_2 \geq \ldots \geq p_n$ ('largest processing time first'), then the provable performance ratio of list scheduling improves to $\frac{4}{3} - \frac{1}{3m}$. These results were already proved by R.L. Graham in the 1960's.

The always returning question in algorithmic modelling is: can one do better? It is known that for minimum makespan scheduling every smaller performance ratio $1 + \epsilon$ with $\epsilon > 0$ can be achieved, provided one is willing to allow an algorithm of greater complexity, with a running time of $\mathcal{O}(n^k)$ for some $k = k(\frac{1}{\epsilon})$. Of course, one could enumerate all possible schedules and pick the best one: this guarantees an *optimal* solution but enumerative algorithms tend to become exceedingly time-consuming as n grows large. The fascinating aspect of algorithmic modelling is that almost all problems that one encounters are computationally complex when one wants to solve them exactly.

Take the jobs to be 'flights' and the machines to be 'gates' at an airport, and suddenly the abstract scheduling problem turns into a very realistic problem in airport planning. Many details must be added to model the gate assignment problem and various related transportation problems. Computing a good schedule for the gate assignments of the hundreds of flights that arrive and depart daily at a large international airport may take in the order of hours, and a large effort also if flights get delayed and a precomputed schedule must be computed again, maybe even from scratch. In recent research by one of our PhD students a new and integrated approach to gate and bus planning was found, in which a robust scheduling algorithm was developed that cuts the needed computing time to less than 15 minutes.

Many problems in algorithmic modelling derive from *the need to make better decisions* in complex situations in which many information streams must be managed and resources are limited.

4 Concepts and Theories

Solving a problem is a triumph over the unknown. (In: P.G. Csicery, 'N is a Number: Portrait of Paul Erdös', documentary, 1993.)

What are the right concepts to understand computational problems, and *why do certain algorithmic techniques work well for some problems but not for others*? These questions are of great importance in computer science, but in general we have hardly scratched the surface in solving them. We illustrate this with a well-known example, the *P-versus-NP* question, which appears to be at the heart of many computationally complex problems.

It is nowadays accepted that feasible computations on problem instances x ideally are polynomially bounded, i.e. have a time complexity bounded by $p(|x|)$ where p is some polynomial and $|x|$ is the problem size (in bits). Consider yes/no-problems. Let P be the class of yes/no-problems that can be solved (answered) by a polynomial time-bounded algorithm, the ideal class. Instead of 'computing' answers, we may prefer to 'verify' them.

Definition 3 *An algorithm $A(x,\cdot)$ is a polynomial-time verifier for a yes/no-problem D if there exist polynomials p, q such that for all problem instances x of D:*

- *$A(x,y)$ runs in time bounded by $p(|x)$ for all inputs y with $|y| \leq q(|X|)$.*
- *if the answer to x is 'yes', then there must be a proof input y with $|y| \leq q(|X|)$ such that $A(x,y)$ outputs 'yes'.*
- *if the answer to x is 'no', then for* all *inputs y with $|y| \leq q(|X|)$ $A(x,y)$ outputs 'no'.*

For many problems (e.g. 'is there a schedule with makespan $\leq C$') it seems to be much easier to 'verify' a proposed solution than to compute one. This is precisely captured by the notion of a polynomial-time verifier: it can verify 'proofs' quickly (read: in polynomial time), but no statement is made about how to find proofs that can certify a 'yes'-answer quickly. Can it? Let NP be the class of yes/no-problems that have polynomial-time verifiers. Clearly we have $P \subseteq NP$. A main open question in computer science is:

$$\textbf{is } P = NP\textbf{?}$$

Or to phrase it differently: do all problems that have a polynomial-time verifier, also have a polynomial-time solver? The problem is open for at least 37 years and still is, although there are continuing attempts to solve it.

It should be realized that many yes/no-problems are in NP. For example, consider networks $G =< V, E >$ and call a subset $S \subseteq V$ *independent* in G if no two nodes of S are connected by an edge in E. The problem $D =$ '*does G have an independent set of size $\geq k$*' can easily be seen to have a polynomial-time verifier and thus belongs to NP. But no one until now has been able to prove that $D \in P$.

Problem D is only one of many problems in practice with this same property, although harder problems, i.e. problems that are not even in NP, occur as well. Theoretically, concepts of problem reduction have been devised that can be used to show that many problems in NP are *NP-hard*: if one succeeds in proving that one of them is in P, then all of them are and $P = NP$. D is NP-hard. Another well-known example is the *traveling salesman problem*. Although we have stated the concepts for yes/no-problems, very similar concepts and conclusions hold for NP-optimization problems.

4.1 Approaches

How to solve hard problems? As there is no fixed recipe, theories are developing that aim at solving problems of all sorts. NP-hard problems are well-known and notorious because until now they have resisted any attempt to solve them by means of simple and efficiently scalable (read: polynomial-time) algorithms. Algorithm designers in the 1950's and 1960's already tackled these problems by so-called *exact algorithms*, necessarily exponential ones at that, which was realized already in these early days of complexity theory. Currently, in our group in Utrecht and at a few other places, the design of exact algorithms is fully back on the the agenda. This has already led to a considerable number of publications on *moderately exponential-time algorithms.*

Another popular approach to attack NP-hard problems is not to solve them *exactly* but to compute approximate answers for them using so-called *approximation algorithms* with 'small performance ratio'. We already saw an example, namely the list scheduling algorithm for minimum makespan scheduling. Algorithms of this kind may not be easy to find but can be very useful, especially when combined with other techniques e.g. randomisation. The right concepts to understand approximation complexity are far from settled. In recent research we have uncovered various new types of approximation schemes, and we are now in the process of figuring out what their algorithmic possibilities are.

How do 'hard problems' arise in algorithmic modeling? We give a classical example from the research on sensor networks. A sensor can be modelled by a point and a circular range, thus a disk in the plane. Assuming we have a set C of identical sensors planted in the plane, what is the largest possible set of non-interfering sensors in C? To solve it, we model C by a *unit disk graph*, with the disks in the plane as nodes and an edge between two nodes if their disks intersect.

The problem we posed now translates to a familiar form: given a unit disk graph G, compute a maximum independent set in G! We discussed the independent set problem earlier. In a precise sense, the optimization version is NP-hard in its class, even for the case of unit disk graphs. Until we know better, we're faced with the difficulty of finding an efficient and scalable algorithm for it. By exhaustive enumeration one can solve the problem exactly, but this quickly becomes infeasible as the number of sensors becomes large, which it is in practice. However, one can show again that the problem has an efficient approximation algorithm for any desired performance ratio, no matter how close to 1 we want it to be.

Proposition 4 *Given any fixed $\epsilon > 0$, there is a polynomial time approximation algorithm for the maximum independent set problem on unit disk graphs ('sensor networks') that achieves a performance ratio of at least $1 - \epsilon$.*

Proof: We begin by putting a suitable grid over the plane. For some integer $p > 0$ and $0 \leq i, j \leq p-1$, let $R_{i,j}$ be the grid with horizontal lines at $y \equiv j \pmod{p}$ and vertical lines at $x \equiv i \pmod{p}$. Let G be a unit disk graph of n nodes and let OPT be a maximum independent set of disks in G. Given a $R_{i,j}$, a disk can intersect at most one horizontal and at most one vertical grid line because lines are at p units distance. In fact, for any fixed j, a disk will intersect a vertical line of precisely one the $R_{i,j}$ when we shift from

$i = 0$ to $i = p - 1$. Thus, for some j there must be a grid $R_{i',j}$ of which the vertical lines intersect at most $|OPT|/p$ disks. By shifting vertically, one can find a j' such that the horizontal lines of $R_{i',j}$ intersect at most $|OPT|/p$ disks as well. Suppose we knew i' and j'. Consider $R_{i',j'}$ and eliminate all disks that intersect a horizontal or vertical grid line of $R_{i',j'}$. (The case in which disks precisely touch horizontal and/or vertical grid lines can be taken into account as well.)

The first observation is that this process leaves us with a set of disks of which at least $(1 - \frac{2}{p})|OPT|$ disks are independent. The second observation is that the remaining disks are all located *inside* the $p \times p$ cells of the grid. Thus any independent set in the remaining set must consist of pieces that are each fully contained within a cell. The size of an independent set in a single cell will be bounded by the packing number of unit disks in a $p \times p$ square, which is $\leq p^2$. Thus, viewing p as a constant and by exhaustively enumerating all subsets of up to p^2 disks, one can find a maximum independent set within each occupied cell in constant time. As there are at most n occupied cells, we obtain an independent set of the original unit disk graph of size $\geq (1 - \frac{2}{p})|OPT|$ efficiently. Choosing $p = \lceil \frac{2}{\epsilon} \rceil$ we seem to obtain the result we want.

The difficulty of course is that we do *not* know OPT and thus neither i' nor j' beforehand. But note that there are only p^2, i.e. constantly many different grids. Thus, by carrying out the computation we outlined above for *every* $R_{i,j}$ and taking the *largest* of the independent sets we obtain for them, we definitely include the hideous grid $R_{i',j'}$ and obtain a result with at least $(1 - \frac{2}{p})|OPT|$ disks, by an extra factor of p^2 in computation time. Note that the approximation algorithm remains polynomial time-bounded for fixed ϵ, but the polynomial will quickly become worse as a time-bound when ϵ becomes small. ■

Optimization problems on models of sensor networks are actively studied in the Center for Algorithmic Systems.

5 Conclusion

Computer science is having a revolutionary impact on scientific research and discovery. ...it is nearly impossible to do scholarly research in any scientific or engineering discipline without an ability to think computationally. (J.M. Wing, 2006)

Informatics is a way of exploring the natural and man-made world with the help of computers. It is only a small step to say that it is a way of exploring the world with the help of algorithms. We increasingly see algorithm design spreading out in all science domains, as a most powerful method for modeling and simulation. Chazelle even called algorithms *the most disruptive scientific development since quantum mechanics.* Is this so? Fact is that the strong societal need for ICT causes the science of Informatics to branch out to any field of scientific, industrial, business and societal relevance. It has even spurred the development of a philosophy of 'computational thinking'.

In this essay we have described the field of interest of the Center for Algorithmic Systems. Our research aims at understanding complexity in all information-driven systems, wherever they occur, and the discovery of the right concepts for dealing with it. The ultimate goal is to discover the algorithms that make them work or can make them work, develop the algorithmic principles and programs for them, and let them work for us and for the benefit of others. We hope you join us on this fascinating road of discovery in the world of algorithmic modelling and complexity!

The **Center for Algorithm Systems** (http://www.cs.uu.nl/groups/AD/) in 2009: Dr.ir. Marjan van den Akker, dr. Hans Bodlaender, Thomas van Dijk MSc, dr. Han Hoogeveen, ir.drs. J Kwisthout, prof.dr. Jan van Leeuwen, Eelco Penninkx MSc, drs. Johan van Rooij, prof.dr. Wim Scheper, dr. Gerard Tel, dr. Marinus Veldhorst.

References

[1] J. van Leeuwen and J. Wiedermann. How We Think of Computing Today, in: *Logic and Theory of Algorithms*, 4th Conference on Computability in Europe (CiE 2008), Proceedings, Lecture Notes in Computer Science, Vol. 5028, Springer-Verlag, Berlin, 2008, pp. 579-593.