Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.K. Lenstra

Algorithmics and heuristics in combinatorial optimization

Department of Operations Research and System Theory Note OS-N8801 February

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Algorithmics and Heuristics in Combinatorial Optimization

Jan Karel Lenstra
Centre for Mathematics and Computer Science (CWI), Amsterdam
Erasmus University, Rotterdam

This is the text of a plenary address at the DGOR/NSOR meeting, delivered in Veldhoven, The Netherlands, on September 23, 1987.

1980 Mathematics Subject Classification (1985 Revision): 90C27, 90Bxx.
Key Words & Phrases: operations research, combinatorial optimization, history, complexity, intractability, geometry, randomization, parallelism, interaction.
Note: This text will be published in the proceedings of the meeting.

I would like to start by saying that I am pleased that this annual DGOR meeting is being held in the Netherlands and that I am honored that the program committee has invited me to give a plenary lecture.

There exist long-standing relations between the operations research communities in Germany and the Netherlands, and as far as DGOR is concerned, these relations have been especially close in the area of stochastic operations research. I am gratified that it has been decided to choose a topic in deterministic operations research for this presentation, and I have hopes that this signifies a broadening of the contacts between DGOR and the Dutch OR Society.

The subject of today is combinatorial optimization. In view of my affiliation with the CWI, you will not be surprised to hear that I will talk in particular about some tools from mathematics and computer science that are available to help us to solve problems in combinatorial optimization. I will thus concentrate more on techniques and less on problems.

Mr. Kuilman has spoken about flexible manufacturing, where the use of techniques from operations research in general and combinatorial optimization in particular could be beneficial. Logistics in a broad sense, covering production planning and distribution management, is just one application area of combinatorial optimization. Others are timetabling, marketing, investment planning, health care, network design, and circuit layout. It is fair to say that combinatorial optimization as a research area would not exist without the demand for effective and efficient planning and design methods in a variety of practical situations.

Let me give you an example of optimization in practice. For me, it is a historical example.

History

The first time that I encountered the subject of today was in the mid 1950's, in a novel called *Professor Sealingwax and his cuckoo*. Professor Sealingwax was a remarkable operations researcher. Many of the actions he took were based on calculations he carried out in pencil on the back of a sheet of wallpaper.

Once upon a time he found himself on a ship, together with his wife Sweetie and their cuckoo,

Note OS-N8801
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

but without a crew, without coal, and without wind. They wanted to get to a country by the name of Foreveronia. This is how it goes.

He multiplied Foreveronia with Southwind, added Sealingwax, Sweetie, ten stokers, three engineers and forty sailors, extracted the square root, divided by Nip Tune, and then by starboard and the Great Bear, subtracted steam and smoke and added lull. Then he multiplied the result by lifeboat, removed the first and the last digit, changed the middle digit nine into a four, and cried with pleasure: 'Sweetie, I've got it! One bucket of mustard, a pot of pepper, a bag of onions and a bag of garlic in the middle funnel of the ship, and that should do the trick.' 'Sealingwax,' Sweetie said, 'I have always trusted you, and I also trust you this time.'

There is little doubt that this is an algorithm, with components of a real-life situation as input and a decision as output. It is also clear that there is mathematics and computing involved, and the sheet of wallpaper suggests the infinite tape of a Turing machine, so he also used some sort of computer. We could call this operations research.

But Sealingwax calculates with real objects. There is no transformation of a practical problem situation into a mathematical problem type, which is called *modelling*, and there is no translation of a mathematical solution into a practical decision, which is called *implementation*. There is no gap between theory and practice. Abstraction and reality are the same. I will return to this relation later.

When I became seriously interested in combinatorial optimization in the 1960's, it went without saying that we needed mathematics and computers. One question continued to intrigue us in those days: how should we use our computers? More precisely, how do we implement our algorithms efficiently? It came as something of a surprise that a new discipline was emerging that addressed exactly this issue: the art of computer programming.

But we needed more. Some problems could be solved efficiently; other problems escaped us and no truly efficient method for their solution could be found. And the question was: how far can we go? That is, where are the limits to efficiency? It came as an even bigger surprise, in the early 1970's, to find that the difference between easy and hard problems could be explained. This is what computational complexity theory is about.

Complexity

You all know what a graph is: a collection of nodes and a collection of edges, each of which links two nodes together. The graph in Figure 1 is connected, because you can get from each node to any other. The graph in Figure 2 is disconnected. The graph in Figure 3 is Hamiltonian, since it contains a cycle that visits each node exactly once. The graph in Figure 4 is not Hamiltonian; you might want to prove this.

Suppose you are attending a conference and you want to buy a present for the people at home. You go to a graph store and ask for a connected graph. The shopkeeper puts a box on the counter (Figure 5(a)). You want to check this, open the box, and take out the graph. When it sticks together, it is connected (Figure 5(b)); when it falls apart, it is not (Figure 5(c)). The point I want to make here is that you can easily test a graph for connectivity by yourself; it takes an amount of time proportional to the number of edges.

Now you want to buy something special: a Hamiltonian graph. Again, there is a box (Figure 6(a)). You open it - but now you may find yourself in trouble (Figure 6(b)): there is no fast method available that can test any given graph for Hamiltonicity. Trial and error may work, but not necessarily so. However, the shopkeeper can easily convince you, namely by pointing out a Hamiltonian cycle as in Figure 3; this takes an amount of time proportional to the number of nodes.

This is exactly the difference between the problem classes P and NP. Both classes contain only decision problems, which require a yes/no answer; I will return to optimization problems shortly. P contains all those problems for which one can easily come up with the correct answer. NP contains all those problems for which one can easily be convinced of the correctness of the yes answer by checking a given structure: a Hamiltonian cycle in the example, a *certificate* in terms of complexity theory.

These definitions only make sense if the notion of *easiness* is formalized. A computation is easy if its running time is bounded by a polynomial function of the size of the problem under con-

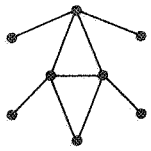


FIGURE 1. A connected graph.

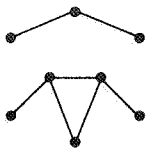


FIGURE 2. A disconnected graph.

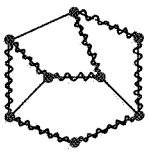


FIGURE 3. A Hamiltonian graph.

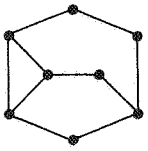
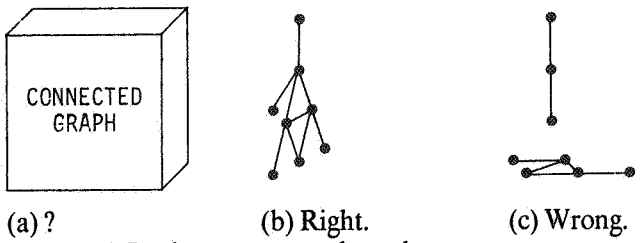
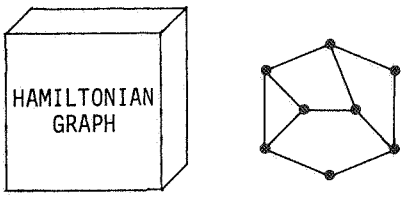


FIGURE 4. A non-Hamiltonian graph.



(a) ? (b) Right. (c) Wrong.

FIGURE 5. Buying a connected graph.



(a) ? (b) ??

FIGURE 6. Buying a Hamiltonian graph.

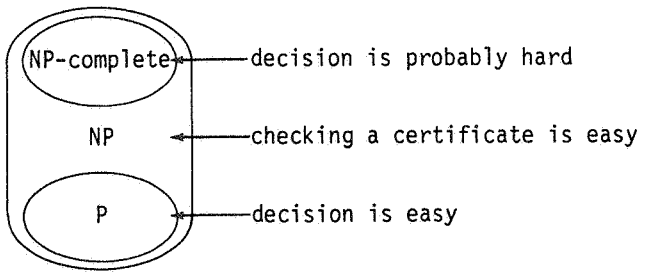


FIGURE 7. A likely map of *NP*.

sideration. For a graph on n nodes, checking all nodes or all edges takes time polynomial in n, but generating all permutations of the node set in the hope of finding a Hamiltonian cycle is superexponential.

What are the virtues of an algorithm when it runs in polynomial time? First of all, its robustness. An algorithm that is polynomial on one machine is polynomial on any other reasonable type of machine, including theoretical models and commercial computers (but excluding parallel machines). Secondly, its asymptotic behavior. Any polynomial function in n is ultimately, when n is large enough, smaller than any exponential function. In the third place, its practical efficiency. Polynomial algorithms tend to work well in practice. Some polynomial algorithms are pretty bad, but it seems to be the case that once a problem has been shown to belong to P, a truly efficient method is found sooner or later. Finally, polynomiality allows us to come to grips with computational complexity in a theoretical sense. It serves to explain why some problems appear to be harder than others. More generally, it has proved to be a fundamental concept in the broad area of computational mathematics.

Any problem in P also belongs to NP, so P is a subclass of NP. I have indicated that the connectivity problem is a member of P and that Hamiltonicity is in NP. If it could be shown that Hamiltonicity is outside P, then the problem would have no solution in polynomial time and one would justifiably call it *hard*. Such a proof seems to be beyond the reach of present-day mathematics. However, we can do slightly less. It can be shown that the Hamiltonicity problem is a generalization of any other problem in NP. Hamiltonicity is *NP-complete*, i.e., it is representative of the entire class NP. If Hamiltonicity would belong to P, then all other problems in NP would be easy as well and P would be equal to NP. No one believes this to be true, for the simple reason that NP seems to be so much richer than P. It follows that the Hamiltonicity problem is unlikely to be easy and therefore *probably hard*. See Figure 7.

Next to Hamiltonicity, many other combinatorial decision problems have been shown to be NP-complete. I have not told you how results of this type are obtained. That is of secondary importance here; suffice it to say that it is conceptually a simple affair, although it can be technically very intricate.

What is the use of all this for combinatorial optimization? Complexity theory deals with yes/no problems, we consider optimization problems. If a problem has a polynomial optimization algorithm, then it is said to be *easy* (or well solved, or tractable). If a problem is at least as hard as some NP-complete problem, then it is said to be *NP-hard*. This should not be the last word on the problem, but the first. It tells you that you cannot expect to find a guaranteed optimum in worst case polynomial time. You have to give in on either speed or solution quality.

Intractability

The next question is, of course: how to solve these hard problems? How to cope with intractability? At this point, we enter the area of the design and analysis of algorithms in combinatorial optimization. There is no way that I could review all that has happened here since the mid 1970's, even if you would give me the time.

The main criteria to be taken into account are *efficiency* and *effectivity*, or speed and quality. How do we make optimization methods more efficient, that is, run faster? How do we make approximation algorithms more effective, that is, achieve better solutions?

The performance of an algorithm can be analyzed in terms of its worst case or average case behavior. Worst case analysis is a pessimistic approach, since it has to take the isolated difficult problem instance into account, but it provides solid performance guarantees. Average case analysis is a complementary approach, which presupposes some probability distribution over the problem instances. I do not want to go into any detail here. Let me just mention one brief example: the simplex method for linear programming. This algorithm requires exponential time in the worst case but performs very satisfactorily in practice. In order to explain this, you have to resort to an analysis of the average case, and significant progress has been made in this direction.

I would like to indicate four recent tools in the design and implementation of combinatorial algorithms. *Geometric* and *randomized* methods are algorithmic approaches of a mathematical nature. *Parallelism* and *interaction* employ new

architectures, which have become available due to achievements in computer engineering.

Geometry

Geometric algorithms, and in particular those based on polyhedral combinatorics, represent an extremely important topic, which is worthy of a full plenary address at one of your future meetings. That is all I will say about it.

Randomization

Before talking about randomization, I should clarify the role of stochastics in combinatorial optimization. Stochasticity occurs at three levels.

First, there are the stochastic *problem types*, which occur in areas like stochastic programming and queueing theory, but also in routing and scheduling. The model is stochastic in the sense that one has to determine a solution before the data is realized, so as to optimize a global parameter like the expected criterion value. Secondly, the problem type is deterministic but the *problem instances* are random. This refers to the case of probabilistic analysis. For each realized instance, an optimal or approximate solution is defined, and one is looking for a probabilistic characterization of its value. Finally, the problem type and its instances are deterministic, but the *algorithm* is randomized in the sense that it is able to toss a coin at certain points in order to decide how to proceed. In the context of machine scheduling, for example, one may want to minimize the expected makespan at the first level, to find the expected minimum makespan at the second, and to minimize the makespan in a randomized fashion at the third.

Let me give you an example of the use of randomization. Consider the problem of determining whether a given number is prime. It is an open question whether primality can be tested in polynomial time. However, there exist randomized algorithms that run in polynomial time and that behave as follows: if the output is 'no', then the number is definitely composite; if the output is 'yes', then the number is prime with probability at least one half, and repeated trials can reduce the error probability. Such a test gives moral rather than mathematical certainty, but it does so very fast.

Randomized methods in combinatorial optimization are, of course, approximation algorithms. A few randomization schemes are indicated here, with the randomized action in italics:

Monte Carlo:
  [*generate*]$^{\text{many}}$;
Monte Carlo & local search:
  [*generate* $\rightarrow$ improve]$^{\text{many}}$;
simulated annealing:
  generate $\rightarrow$ improve or [*deteriorate*]$^{\text{prob}\downarrow 0}$;
sampling & clustering:
  [*generate*]$^{\text{many}}$ $\rightarrow$ select $\rightarrow$ [improve]$^{\text{some}}$;
extension & rotation:
  extend $\rightleftarrows$ *rotate*.

Monte Carlo is an old approach: a number of feasible solutions is generated and the best one is selected. The combination with local search is slightly more sophisticated: each generated solution is subjected to iterative improvement on the basis of neighborhood search. Several variations on this approach have recently been proposed.

Simulated annealing is a technique for iterative improvement, again based on neighborhood search, which accepts deteriorations with a small and decreasing probability in the hope of avoiding bad local optima and getting settled in the global optimum. Sampling & clustering is another variation, which selects only one starting point out of each cluster of sampled solutions that are likely to lead to the same local optimum. Extension & rotation is an altogether different idea.

The investigation of the randomization principle in OR is at a relatively early stage. Some notable successes have been obtained, and more can be expected.

Parallelism

What has been achieved in parallel computing?

Architectures. - In a parallel computer, several processors operate in parallel and communicate with each other. We distinguish three classes. In vector machines, the operations are pipelined rather than parallelized. In SIMD (single instruction multiple data) machines, the processors perform at each point in time the same operation on local data. Usually, there is a large number of small processors and a fast interconnection network. Both classes are suitable for regular computations, where many operations of the same type have to be performed in a synchronized fashion. In MIMD (multiple instruction multiple

data) machines, the processors can perform different instructions at a time. In practice, there is a moderate number of processors that operate in an asynchronous mode and communicate through a slow network or a shared memory.

Computations. - Parallel computing has provided a new playground for computational OR. Most experience has been obtained with numerical algorithms and nonlinear optimization on vector and SIMD machines. In combinatorial optimization, these types of machines perform well as long as the computational process is regular, as in dynamic programming. MIMD seems to be more suitable if the structure of the computation is not known in advance. An example is branch and bound, where the processors should explore different parts of the search tree and communicate only if the need occurs.

Computational models. - In comparison to sequential computing, there are two problems. First, existing machines are by no means equivalent; implementations tend to be highly machine dependent. Secondly, realistic models are lacking; theoretical analyses have to take account of physical features of the computational environment. Much attention has been paid to the PRAM (parallel random access machine), which allows for unbounded parallelism and unit-time interprocessor communication, but such a model is hardly realistic.

And what are the perspectives?

Computational models. - We need an investigation of severe restrictions on parallelism and communication. In particular, a robust theory for models with at most a linear number of processors that communicate over a bounded degree network would be very useful.

Architectures. - The main obstacle for a breakthrough of parallel computing is not the lack of realistic models but the chaos in the real world of architectures. We need a consensus on a single concept of a flexible MIMD computer in which the user can define the sort of parallelism he desires. Before attempting a hardware realization of this machine, we should build it in software and analyze its performance. This requires a flexible set of tools, including a versatile programming language, which does not bother the user with the internal machine structure.

Computations. - We need a theoretical approach towards the design and analysis of parallel algorithms for hard problems. The fundamental question is how to distribute the computational effort over the processors and how to arrange the communication so as to maximize the speedup. Operations researchers are well positioned to model and solve this complicated design problem.

Interaction

Man-machine interaction is a less formal and more fashionable topic than geometry, randomization, or parallelism. I will concentrate on *interactive planning systems*, also known as *decision support systems* (DSS). By this, I mean systems that are designed to support decision making in practical planning situations by the integration of human perception and mechanical algorithmics in an interactive environment.

Where do we place DSS if we look at it from an OR point of view? OR has various sides. The mathematics of OR is a *normative* occupation that intends to develop a theory of models and algorithms. Practical OR is an *empirical* activity in which quantitative tools are put to use in actual problem situations in a heuristic fashion. DSS is then nothing but a novel approach towards practical OR, made possible by advances in information technology. DSS merges the areas of OR and information systems. From this point of view, one should not expect a formal theory of DSS. One should, however, expect the influence of other disciplines, including database theory and software engineering, computer graphics and computational geometry, and even pattern recognition and psychology.

Figure 8 displays the structure of a DSS. The top level represents practice. The system receives data and tentative decisions from the outside world and returns decision support. At the bottom, we have models and algorithms, a collection of black boxes that contain the quantitative tools on which the DSS relies. It is illuminating to distinguish two types of models. *Evaluative* models are designed to answer the question: given a decision, what is its quality? *Generative* models do, in some sense, the reverse: given a desired quality, what is an appropriate decision? This distinction reflects two functions that the system should be able to perform: either it *assists* in representing and evaluating decisions proposed by the user;

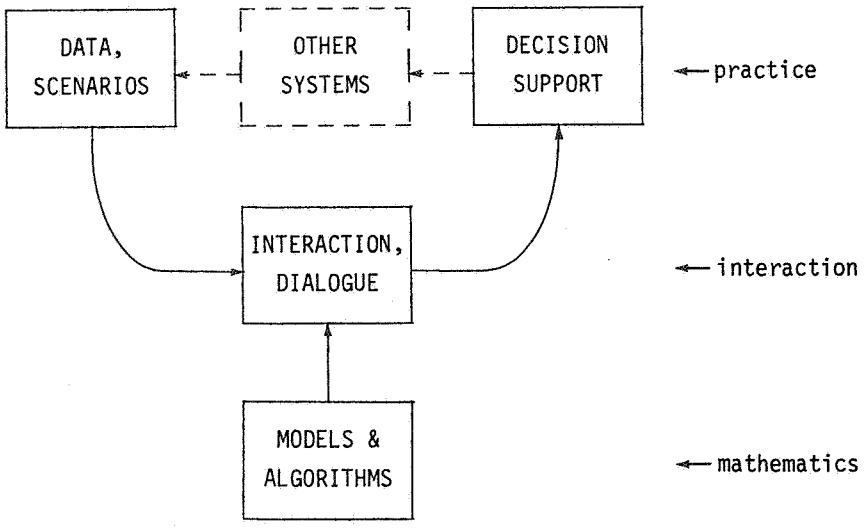


FIGURE 8. Structure of a DSS.

or it *advises* and generates complete plans by itself. These two roles are the extremes of the spectrum, and there is much in between. An important feature of a DSS is that it is always the user who is in charge, irrespective of the mode of operation.

The middle level is the interface where the interaction between human insight and experience and algorithmic power and precision takes place. It is the core of the system. A few remarks are in order here.

Interaction is possible in a technical sense, but why is it desirable? The brief answer is that practical planning problems tend to be both *hard* and *soft*. Hard on grounds of complexity considerations; in order to obtain solutions of an acceptable quality within an acceptable amount of time, one has to resort to approximation algorithms, and even to heuristic procedures in which man and machine divide the tasks in accordance with their respective capabilities. Soft because notions like feasibility and optimality are not as precise as in mathematics but are carried implicitly in the value judgement of the decision maker; interaction is the obvious way to cope with this. As a result, interaction adds to effectivity, efficiency, and acceptability. Better solutions are obtained faster, and an interactive planning system is more readily adopted than a black box approach.

While the user interface is the most visible part of a DSS, its only purpose is to create the opportunity to manipulate information in a convenient way. Whether information and manipulation make sense depends on the context, which consists of the planning situation on the one hand and the models and methods on the other. One might say that the role of information technology pertains to the form, while practice and mathematics provide the substance.

Much more could be said about the desirable functional properties of a DSS, about the design of a user interface, and about the need to turn the art of representation into a science. But due to limitations of time, I will leave it at this.

PROPOSITIONS

I would like to finish with a number of propositions. Some of you may disagree with some of them. Please view them as a contribution towards the discussions during this meeting.

PROPOSITION 1. *A heuristic is not necessarily an approximation algorithm.*

An algorithm is an unambiguous recipe and belongs to the normative science of mathematics, whether it is deterministic or randomized, and whether it optimizes, approximates, or performs some other function. A heuristic, in the original meaning of the word, is an empirical approach to practical problem solving by trial and error.

A DSS is a typical form of a heuristic. It may

rely on all kinds of things, including approximation algorithms. It would clean up OR terminology if we would distinguish these concepts more carefully.

PROPOSITION 2. *There is no gap between theory and practice.*
I know that some of my colleagues are exclusively interested in theory, while others are completely occupied by practice. For many of them, the gap does exist. What I really want to say is that *the gap is in the mind of the beholder.*

Many of the truly outstanding OR projects over the past fifteen years were carried out by people who knew how to create innovative modelling concepts, develop original solution techniques, and achieve successful implementations. They are generally too busy to realize the distinction between theory and practice, let alone to worry about a gap. What they do is true OR.

PROPOSITION 3. *There is no crisis in OR.*
I am not sure if I agree with this myself. After all, as far as the *systems* is concerned, we see that OR is in danger of becoming dominated by the areas of computer engineering and information systems. And as to the *ideas*, one could defend the point that the easy work has been done and that the hard work is safer in the hands of mathematicians and computer scientists.

However, it is the task of OR to bring ideas and systems together, and there is a bright future ahead. The use of OR models and methods requires computing facilities, and it is only in the last few years that these are becoming widely available at affordable prices. The current revolution in information technology will have a dramatic impact on the demand for OR techniques.

PROPOSITION 4. *Away with applications.*
Both theory and practice are vital to OR, so what is wrong with applications? They are of no concern to either side.

Theoreticians are interested in theorems, not in applications. If they talk about applications, they usually refer to examples. Although these can be very helpful, their purpose is to illustrate theory, without any practical motive.

Practitioners are not interested in applications but in solutions. They are not concerned with problem types or models, but they are faced with a problem situation. They do not ask for the application of some form of abstraction, but for a practical solution - whether you arrive at it by mathematics or by black magic.

Thank you for your attention. I wish you a successful meeting.

BIBLIOGRAPHICAL NOTES
I have used material from [3], [2] and [1] in the sections on complexity, parallelism and interaction, respectively. The quotation from [4] was translated by Joke Sterringa.

REFERENCES
1. J.M. ANTHONISSE, J.K. LENSTRA, M.W.P. SAVELSBERGH (1988). Behind the screen: DSS from an OR point of view. *Decision Support Systems*, to appear.
2. G.A.P. KINDERVATER, J.K. LENSTRA, A.H.G. RINNOOY KAN (1988). Perspectives on parallel computing. *Oper. Res.*, to appear.
3. J.K. LENSTRA (1986). Interfaces between operations research and computer science. Γ.Π. Πραστακος (ed.). *Επιχειρησιακη Ερευνα και Ηλεκτρονικοι Υπολογιστες*, EEEE, Athens, 35-48.
4. D. ZONDERLAND (1969). *De reisavonturen van professor Zegellak*, Het Spectrum, Utrecht.