A program generator for simulated annealing problems

Citation for published version (APA):

Document status and date:
Published: 01/01/1989

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.
Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
A Program Generator for Simulated Annealing Problems

by

M.W.H. Hesen  A.T.M.Aerts  K.M. van Hee

89/13

December, 1989
This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
All rights reserved
Editors: prof.dr.M.Rem
         prof.dr.K.M. van Hee
Abstract
This article is concerned with the description of a specification language for Simulated Annealing problems. It should facilitate the specification and investigation of a large class of combinatorial optimization problems.

Keywords: general application, language processor
Introduction

This article is concerned with the description of a specification language for Simulated Annealing (SA) problems. SA is a very general technique for combinatorial optimization problems which has been shown to give good results for many problems. The main reason for the general applicability is the ease with which problems can be cast in a form appropriate for SA. From this form it's not very difficult to design a SA program.

If we compare this situation to the use of Linear Programming (LP) then we note that these days almost anyone who wants to use LP uses standard library routines instead of programming his own application. This makes use of LP very easy. The goal of our investigation is to make a start to creating the same situation for SA. This should facilitate the specification and investigation of certain types of combinatorial optimization problems.

During the last few decades there has been a great deal of interest for problems in this field. A well known example is the travelling salesman problem (TSP) for which approximation and exact algorithms have been found. More problems of this kind are described in [2].

How can we describe a combinatorial optimization problem? This is usually done by means of a pair \((S, C)\) where \(S\) is a finite, or countably infinite, set of configurations and \(C\) a cost function: \(C : S \rightarrow R\). The function \(C\) assigns a real number to every configuration. The problem can be stated now as finding a configuration for which the corresponding value of \(C\) is minimal. Formally stated:

\[
\text{find } i_0 \text{ such that } C(i_0) = \min_{i \in S} C(i).
\]

For the above mentioned TSP \(S\) corresponds to the set of tours and \(C\) to the length of a tour. It was found that many combinatorial problems are "hard" to solve. For a class of these problems no algorithms are known with an amount of computational effort bounded by a polynomial function of the size of the problem. This class is usually referred to as NP. For large problems in this class it is impossible to find an exact solution within a reasonable amount of time. Therefore we are forced to use an approximation algorithm of some sort.

These algorithms can be divided into two categories: algorithms tailored to a specific problem or generally applicable algorithms. Algorithms of the first category have the disadvantage of their limited applicability and therefore limited use although their performance is usually good. This paper is concerned with an example of the second category: SA.

The SA algorithm is a robust general optimization technique which has been shown to give good results for many problems. SA can be seen as a generalization of Iterative Improvement, a general technique based on local search.

We define \(R_i\) (the neighbourhood of \(i\)) to be the set of configurations that can be reached in one transition from configuration \(i\). Iterative Improvement is based on a start configuration
and a transition mechanism which describes the neighbourhood structure of the set of configurations. The algorithm can be described as follows: starting from the start configuration new configurations are generated. A transition to a generated configuration is made if it's cost value ($C$) is lower than the cost value of the old configuration. This algorithm has several disadvantages. The first is that always a local optimum is obtained for which it is impossible to determine whether it is also a global minimum. Secondly the solution obtained depends on the start configuration that is chosen. The last disadvantage is the impossibility of giving an upper bound for the computation time. One solution to the problems mentioned would be to accept deteriorations of the cost value in a limited way. This is the basis for the SA algorithm.

This algorithm was introduced independently by Kirkpatrick e.a. [5.] and Černy [1.]. It avoids the first and second disadvantage mentioned above and for some problems also the third. Unfortunately it is generally slower than the iterative improvement algorithm. Below we will give a short introduction to the algorithm. For more information we refer to [6.]. The algorithm is based on the analogy between annealing of solids and the solving of combinatorial problems. Annealing of solids is a process in which a solid is heated until it is in the liquid phase. Then the liquid is cooled slowly. In this way the material is allowed to reach a state of equilibrium at each temperature. Eventually it will reach a state of minimal energy. This analogy should be kept in mind when trying to understand how the algorithm works.

The description of the algorithm is basically the same as for iterative improvement except for the acceptance of transitions. In SA a transition is not only accepted when the new cost value is lower but also with a certain probability when the cost value is higher. This probability is determined by the ratio of the difference in cost values and a parameter $c$, the cooling parameter, which has the same function as the temperature in the annealing process. A chain of transitions is generated (under a fixed $c$) until the system is close enough to equilibrium. Then the value for $c$ is lowered and again a chain of transitions is generated until equilibrium is close enough. This is repeated until a stop criterion is fulfilled. It can be mathematically proved that the system approaches an equilibrium configuration [6.]. We can now describe the algorithm in pseudo-Pascal:

\begin{verbatim}
Initialize;
REPEAT
  REPEAT
    Generate(configuration);
    IF $\Delta$Cost $\leq$ 0
      THEN Accept(configuration)
    ELSE IF $\exp(-\Delta$Cost)$>$Random[0,1]
  END IF
END REPEAT
\end{verbatim}
THEN Accept(configuration);
IF Accept(configuration) THEN Update(configuration);
UNTIL Close to Equilibrium
    Lower(c);
UNTIL Stop Criterion

The function $\Delta\text{Cost}$ in this description is defined as the difference in cost value between the current configuration and the proposed new configuration. The function Lower decreases the value of a parameter.

As can be seen from this description a number of parameters have to be specified before the algorithm can be implemented. These parameters are:

1. Initial value of $c$.
2. Stop criterion.
3. (Close to) Equilibrium criterion.
4. Function $f$ for determining the new value of $c$.

A choice for these parameters is called a cooling schedule. There are many different approaches to this problem. The particular choice of a cooling schedule is not of much interest to the subject of this article. The interested reader is referred to [6].

The Cooling Side of SA: Library Routines

In the introduction we mention that SA is a very general technique for combinatorial problems. The goal of our investigation is to simplify the use of SA.

We divide this problem into two parts. First we design library routines which can be used in application programs for SA. The next step is to automate the process of turning a problem specification into a program for SA. This is achieved by designing a program generator. The program generator is described later.

To simplify usage of the library routines we try to make as few assumptions as possible about the program in which the routines are incorporated. In the library routines we use the cooling schedule that was proposed by Aarts en Van Laarhoven. For a discussion of this and other schedules we refer to [6].

We choose this schedule because it is claimed to be quite robust for a wide variety of problems.
The schedule is incorporated in the routines in a modular way so that changing to another
schedule should prove easy.
In addition to the choice of a cooling schedule a choice for the parameters has to be made.
It's very difficult to give a set of parameters which will give good results for all problems. So
this choice is left to the user of the routines. To ease this problem it was decided to make
two "versions" of the routines. In one version the routine keeps administration of selected
information about the execution of the routine. The information recorded is chosen to help
the user to decide whether the current choice of parameters is satisfactory. To simplify ex­
perimenting with the algorithm several stop criteria have been incorporated in the procedure.
The algorithm can be ended in the "usual" way: based on the cost value. Other ways to
stop the execution are: based on the CPU-time used or the number of chains of transitions
generated. The second version is identical to the first except that all the administration has
been left out.
The proposed way to use these procedures is: use version one to iteratively obtain a satisfac­
tory set of parameters for a certain type of problem. From then on use version two to obtain
results for this type of problem. We shall now describe what information will be recorded
and why this information was chosen.
In version one information is given about the behaviour of the temperature (cooling param­
ter), the transitions and the cost function. Among other things, maxima, minima and current
values are given to help the user to monitor the program and tune the parameters.
How do we use this procedure?
The user has to make some declarations in the surrounding program: a textfile in which the
results will be recorded, datatypes for "transitions" and "configuration". Variables of these
types have to be declared and initialized. Also some functions and procedures with prescribed
headings are to be declared:

• a function to calculate the cost value of a certain configuration.

• a function to calculate the difference in cost value resulting from a transition.

• a procedure to select a transition in a random way.

• a procedure to determine the resulting configuration after a transition.

After these demands have been met the user can start the iteration process with an initial
set of parameter values for the length of the chain of transitions, the stop and equilibrium
criteria, starting temperature, maximum CPU-time to be used, the maximum number of
chains of transitions to be generated. Some rules of thumb can be given for the initial choice
and for the adjustments that are made during the iterative process [4]. Once a satisfactory
set of values for the parameters mentioned above has been obtained the second version of the
procedure can be used to solve other problems of the same type.

The Problem Side of SA: A Specification Language

The main disadvantage of the approach sketched in the previous paragraph is that it still requires programming skills of the user. To eliminate this disadvantage it was decided to design a specification language for Simulated Annealing problems. This language is the basis for a program generator which automates the process of turning a problem specification into a SA program. In this paragraph we describe the specification language. The program generator is described in the next paragraph.

In order to be able to use recursive descent parsing we design the language as an attribute grammar with disjoint look ahead sets. For a discussion of attribute grammars and recursive descent parsing the reader is referred to [3].

For the implementation we use Pascal because we have most experience with this language. Implementing in another language should give no additional problems. In order to simplify the conversion to a Pascal program we use some Pascal conventions in our language: the way expressions are formed, the naming of variables etc., standard mathematical functions and the way comments are incorporated. For a complete list the reader is referred to [4].

Starting-point of the design process is the requirement that a specification should contain all the information necessary to create a SA program. We can distinguish six information elements to be specified:

- all the parameters that are relevant to the type of problem investigated.
- the way in which the configuration is recorded.
- the form in which a transition is recorded.
- three functions:
  - determination of the configuration after transition.
  - calculation of the cost value associated with a configuration.
  - calculation of the difference in cost ($\Delta$Cost) resulting from a transition.

We now turn our attention to how these elements are to be specified.

Parameters are specified by giving their name, type and value. Two types of variables are distinguished: simple and complex. Simple variables are similar to real and integer variables. Their value can be given by means of an expression or by a numerical value. Complex variables can be compared to Pascal-arrays. Their indexbounds have to be specified besides their
values. The values can be specified either by an expression or by enumerating the values. Complex variables can have more than one dimension.

The specification language should be functional because we want the specification language to be as versatile as possible and corresponding as close as possible to a mathematical problem formulation. We base the function specifications on LISP-like structures. From [7] and [8] a selection was made of suitable LISP-functions. These are implemented in Pascal. For a complete list the reader is referred to [4]. These functions can be called recursively. A function specification consists of a Pascal-like heading (name, formal parameter list, resulttype) and a body in which the function is described in terms of standard LISP-functions and previously user-specified functions. Three functions have to be specified: transitionfunction, costfunction and $\Delta$costfunction as described above.

Since the three functions that have to be specified all operate on lists the configurations are represented by means of linear lists. In many problems this is a very natural way of representing information about configurations and transitions. Another reason for this choice was the ease with which linear lists can be implemented (in the limited time available for this investigation). In addition to the length of the list also initial values have to be specified for it's elements.

Transitions are also represented by linear lists for the same reasons as configurations. For each field in the list a range of possible values has to be specified. The user has to specify possible constraints by means of a boolean expression.

A grammar for this language contains 59 rules. In order to include context-sensitive requirements this grammar is augmented with attributes (17 types) and boolean expressions in terms of attribute-variables describing the requirements. The requirements can be divided in three groups:

- type-requirements: the types to be used, priorities, types of operands and results of various operators.
- scope- and uniqueness-requirements: identifiers have to be different from Pascal reserved words, formal parameters of functions have to be unique.
- other requirements: the heading of the three functions to be specified has to have a prescribed form.

A complete list of requirements, a complete grammar and attributegrammar can be found in [4]. We construct a recursive descent parser based on this information. For a discussion of this method we refer to [3].

This parser takes a specification as its input. The specification is analysed. All the information necessary for constructing the SA application is taken from it. The output is a file in which all this information is listed in a suitable way. This file serves as input to the program generator.
that will be described in the next section.

The Program Generator

The first step in the design of the program generator is deciding what tasks the resulting programs had to carry out. The basic task is helping to use the library routines that are described above. Consequently these routines have to be included in the program. From the requirements these routines impose we can deduce the structure of the resulting program. The program can be divided in the following parts:

- a menu-driven main module from which the library routines are called.
- the library routines.
- the functions required to make the library routines work.
- a procedure to randomly choose a transition from the ones possible.
- an initialization-procedure.
- an appropriate heading with type and variable declarations.

Initially the parameter values are input. Subsequently calls to the library routines are made. The results are output to a file which can be inspected from within the program. It is also possible to get a short introduction. These options are available in the menu.

The library routines are basically the ones described before with a few minor adjustments. More attention is necessary for the required functions: transition-, cost- and Δcostfunction. These Pascal-functions are constructed by manipulating the information that is obtained from the specification. In order to simplify this manipulation the grammar for the functions is chosen such that this transformation is basically a syntactic one instead of semantic.

The generation of a random transition is done in the simplest way possible. For every field of the transition-variable values are chosen randomly from the range specified. Afterwards these values are checked with the constraints specified by the user. If these constraints are not fulfilled the random choice is repeated.

In the initialization-procedure user-defined parameters are initialized as well as program variables. Program-variables are distinguished from user-defined variables by prefixing them with a string that is determined in such a way that it is different from every user-defined parameter.

How to construct the Simulated Annealing program from the information obtained from the
specification? This is done in a similar way to recursive descent parsing. A grammar can be constructed describing which information was used in what part of the resulting program. This grammar is turned into a program the same way as an attribute grammar is turned into a program. A major advantage of using this method is the ease with which changes can be implemented. This can be done in a very systematic way. The complete grammar can be found in [4].

Conclusions

From tests it can been concluded that the specification language permits compact and understandable problem specifications. A limiting property of the resulting program is that transitions are much slower. Experiments show a factor 10 in difference. This needs attention in future investigations.

The approach is not suitable for all types of problems. Especially problems which require large datastructures which are hard to convert to linear lists may pose problems. The size is troublesome because of the functional nature of the specification language: the frequent copying takes a lot of time. The difficult conversion is problematic since it takes place very frequently during the execution of the resulting program.

Use of the program generator has the advantage that the resulting program has a very simple, modular structure. It should be easy to comprehend and change if necessary. This means that the program is not necessarily the terminus of the solution of the problem at hand. The user can optimize the program by incorporating problem-specific changes. The main advantage though is the easy way in which insight can be obtained in the formulation and characteristics of small problems.

Of course many improvements can be made to the specification language and the program generator. New investigations could look into:

- using pointer-structures.
- better transition-choice.
- the possibility of starting batch jobs from within the program: because of the time consuming execution of SA programs.
- changes that reduce the number of requirements that have to be met by the specification.
Literature


<table>
<thead>
<tr>
<th>No.</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>85/01</td>
<td>R.H. Mak</td>
<td>The formal specification and derivation of CMOS-circuits.</td>
</tr>
<tr>
<td>85/02</td>
<td>W.M.C.J. van Overveld</td>
<td>On arithmetic operations with M-out-of-N-codes.</td>
</tr>
<tr>
<td>85/03</td>
<td>W.J.M. Lemmens</td>
<td>Use of a computer for evaluation of flow films.</td>
</tr>
<tr>
<td>85/04</td>
<td>T. Verhoeff</td>
<td>Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate.</td>
</tr>
<tr>
<td>86/01</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems.</td>
</tr>
<tr>
<td>86/02</td>
<td>G.A. Bussing</td>
<td>ELISA, A language for formal specification of information systems.</td>
</tr>
<tr>
<td>86/03</td>
<td>Rob Hoogerwoord</td>
<td>Some reflections on the implementation of trace structures.</td>
</tr>
<tr>
<td>86/04</td>
<td>G.J. Houben</td>
<td>The partition of an information system in several systems.</td>
</tr>
<tr>
<td>86/05</td>
<td>J.L.G. Dietz</td>
<td>A framework for the conceptual modeling of discrete dynamic systems.</td>
</tr>
<tr>
<td>86/06</td>
<td>Tom Verhoeff</td>
<td>Nondeterminism and divergence created by concealment in CSP.</td>
</tr>
<tr>
<td>86/07</td>
<td>R. Gerth</td>
<td>On proving communication closedness of distributed layers.</td>
</tr>
<tr>
<td>86/08</td>
<td>R. Koymans</td>
<td>Compositional semantics for real-time distributed computing (Inf.&amp;Control 1987).</td>
</tr>
<tr>
<td>86/09</td>
<td>C. Huizing</td>
<td>Full abstraction of a real-time denotational semantics for an OCCAM-like language.</td>
</tr>
<tr>
<td>86/10</td>
<td>J. Hooman</td>
<td>A compositional proof theory for real-time distributed message passing.</td>
</tr>
<tr>
<td>86/11</td>
<td>W.P. de Roever</td>
<td>Questions to Robin Milner - A responder's commentary (IFIP86).</td>
</tr>
<tr>
<td>86/12</td>
<td>A. Boucher</td>
<td>A timed failures model for extended communicating processes.</td>
</tr>
</tbody>
</table>
Specifying passing systems requires extending temporal logic.

On the existence of sound and complete axiomatizations of the monitor concept.

Federatieve Databases.

A formal approach to distributed information systems.

Delay-insensitive codes - An overview.

Enforcing non-determinism via linear time temporal logic specification.

Temporele logica specificatie van message passing en real-time systemen (in Dutch).

Specifying message passing and real-time systems with real-time temporal logic.

The maximum number of states after projection.

Language extensions to study structures for raster graphics.

Three families of maximally nondeterministic automata.

Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.

OR and AI approaches to decision support systems.

Playing with patterns - searching for strings.

A compositional proof system for an occam-like real-time language.

A compositional semantics for statecharts.

Normal forms for a class of formulas.

Modelling of discrete dynamic systems framework and examples.
<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>87/18</td>
<td>C.W.A.M. van Overveld</td>
<td>An integer algorithm for rendering curved surfaces.</td>
</tr>
<tr>
<td>87/19</td>
<td>A.J. Seebregts</td>
<td>Optimalisering van file allocatie in gedistribueerde database systemen.</td>
</tr>
<tr>
<td>87/20</td>
<td>G.J. Houben, J. Paredaens</td>
<td>The $R^2$-Algebra: An extension of an algebra for nested relations.</td>
</tr>
<tr>
<td>87/21</td>
<td>R. Gerth, M. Codish, Y. Lichtenstein, E. Shapiro</td>
<td>Fully abstract denotational semantics for concurrent PROLOG.</td>
</tr>
<tr>
<td>88/01</td>
<td>T. Verhoeff</td>
<td>A Parallel Program That Generates the Möbius Sequence.</td>
</tr>
<tr>
<td>88/03</td>
<td>T. Verhoeff</td>
<td>Settling a Question about Pythagorean Triples.</td>
</tr>
<tr>
<td>88/04</td>
<td>G.J. Houben, J. Paredaens, D. Tahon</td>
<td>The Nested Relational Algebra: A Tool to Handle Structured Information.</td>
</tr>
<tr>
<td>88/06</td>
<td>H.M.J.L. Schols</td>
<td>Notes on Delay-Insensitive Communication.</td>
</tr>
<tr>
<td>88/09</td>
<td>A.T.M. Aerts, K.M. van Hee</td>
<td>A Tutorial for Data Modelling.</td>
</tr>
<tr>
<td>88/10</td>
<td>J.C. Ebergen</td>
<td>A Formal Approach to Designing Delay Insensitive Circuits.</td>
</tr>
<tr>
<td>88/12</td>
<td>A.E. Eiben</td>
<td>Abstract theory of planning.</td>
</tr>
<tr>
<td>88/13</td>
<td>A. Bijlsma</td>
<td>A unified approach to sequences, bags, and trees.</td>
</tr>
<tr>
<td>Page</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>88/14</td>
<td>H.M.M. ten Eikelder, R.H. Mak</td>
<td>Language theory of a lambda-calculus with recursive types.</td>
</tr>
<tr>
<td>88/15</td>
<td>R. Bos, C. Hemerik</td>
<td>An introduction to the category theoretic solution of recursive domain equations.</td>
</tr>
<tr>
<td>88/16</td>
<td>C. Hemerik, J.P. Katoen</td>
<td>Bottom-up tree acceptors.</td>
</tr>
<tr>
<td>88/17</td>
<td>K.M. van Hee, G.J. Houben, L.J. Somers, M. Voorhoeve</td>
<td>Executable specifications for discrete event systems.</td>
</tr>
<tr>
<td>88/18</td>
<td>K.M. van Hee, P.M.P. Rambags</td>
<td>Discrete event systems: concepts and basic results.</td>
</tr>
<tr>
<td>88/19</td>
<td>D.K. Hammer, K.M. van Hee</td>
<td>Fasering en documentatie in software engineering.</td>
</tr>
<tr>
<td>89/20</td>
<td>K.M. van Hee, L. Somers, M. Voorhoeve</td>
<td>EXSPECT, the functional part.</td>
</tr>
<tr>
<td>89/1</td>
<td>E.Zs.Lepoeter-Molnar</td>
<td>Reconstruction of a 3-D surface from its normal vectors.</td>
</tr>
<tr>
<td>89/2</td>
<td>R.H. Mak, P. Struik</td>
<td>A systolic design for dynamic programming.</td>
</tr>
<tr>
<td>89/3</td>
<td>H.M.M. Ten Eikelder, C. Hemerik</td>
<td>Some category theoretical properties related to a model for a polymorphic lambda-calculus.</td>
</tr>
<tr>
<td>89/4</td>
<td>J. Zwiers, W.P. de Roever</td>
<td>Compositionality and modularity in process specification and design: A trace-state based approach.</td>
</tr>
<tr>
<td>89/5</td>
<td>Wei Chen, T. Verhoeff, J.T. Udding</td>
<td>Networks of Communicating Processes and their (De-)Composition.</td>
</tr>
<tr>
<td>89/6</td>
<td>T. Verhoeff</td>
<td>Characterizations of Delay-Insensitive Communication Protocols.</td>
</tr>
<tr>
<td>89/7</td>
<td>P. Struik</td>
<td>A systematic design of a parallel program for Dirichlet convolution.</td>
</tr>
<tr>
<td>89/9</td>
<td>K.M. van Hee, P.M.P. Rambags</td>
<td>Discrete event systems: Dynamic versus static topology.</td>
</tr>
<tr>
<td>89/10</td>
<td>S. Ramesh</td>
<td>A new efficient implementation of CSP with output guards.</td>
</tr>
<tr>
<td>89/11</td>
<td>S. Ramesh</td>
<td>Algebraic specification and implementation of infinite processes.</td>
</tr>
</tbody>
</table>
A.T.M. Aerts  A concise formal framework for data modeling.
K.M. van Hee

A.T.M. Aerts  A program generator for simulated annealing
K.M. van Hee  problems.
M.W.H. Hesen