FFI NOTAT

HOTDOG : A HEURISTIC ON THE DETERMINATION OF OPTIMUM GLOBALLY USING THE MATHEMATICAL PROGRAMMING LANGUAGE AMPL

GROTMOL Øyvind, SUKKESTAD Jens Arne, BRAATHEN Sverre

FFI/NOTAT-2001/03011

FFISYS/807/161

Kjeller 9 August 2001

HOTDOG : A HEURISTIC ON THE DETERMINATION OF OPTIMUM GLOBALLY USING THE MATHEMATICAL PROGRAMMING LANGUAGE AMPL

GROTMOL Øyvind, SUKKESTAD Jens Arne,
BRAATHEN Sverre

FFI/NOTAT-2001/03011

FORSVARETS FORSKNINGSINSTITUTT
Norwegian Defence Research Establishment
P O Box 25, NO-2027 Kjeller, Norway

FORSVARETS FORSKNINGSINSTITUTT (FFI)
Norwegian Defence Research Establishment

P O BOX 25
N0-2027 KJELLER, NORWAY

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE
(when data entered)

REPORT DOCUMENTATION PAGE

1) PUBL/REPORT NUMBER	2) SECURITY CLASSIFICATION	3) NUMBER OF PAGES
FFI/NOTAT-2001/03011	UNCLASSIFIED	29
1a) PROJECT REFERENCE	**2a) DECLASSIFICATION/DOWNGRADING SCHEDULE**	
FFISYS/807/161	-	

4) TITLE

HOTDOG : A HEURISTIC ON THE DETERMINATION OF OPTIMUM GLOBALLY USING THE MATHEMATICAL PROGRAMMING LANGUAGE AMPL

5) NAMES OF AUTHOR(S) IN FULL (surname first)

GROTMOL Øyvind, SUKKESTAD Jens Arne, BRAATHEN Sverre

6) DISTRIBUTION STATEMENT

Approved for public release. Distribution unlimited. (Offentlig tilgjengelig)

7) INDEXING TERMS

IN ENGLISH:	IN NORWEGIAN:
a) Global Optimization	a) Global Optimering
b) Mathematical Programming	b) Matematisk programmering
c) Nonlinear Programming	c) Ikke-lineær programmering
d) Large-scale problems	d) Stor-skala problem
e) Gradient-based search	e) Gradient-basert søk

THESAURUS REFERENCE:

8) ABSTRACT

An approach for robust, large-scale global optimization is developed, where robust refers to obtaining feasible local optimal solutions within constraints tolerances, and large-scale may imply nonconvex problems with hundreds to thousands of decision variables.

A heuristic algorithm HOTDOG ("Heuristic On The Determination of Optimum Globally") is described for problems with a restricted number of local optima. A simpler version of the algorithm called MICIO ("MInimum Computation by Iterative Optimization") that uses the same local search procedure as HOTDOG, is also developed for use when the number of local optima is very large. Both algorithms are developed using the features and generic functions in the mathematical programming language AMPL, and both the HOTDOG and MICIO heuristics may therefore be invoked by a standard *include*-statement command in AMPL (corresponding to *m*-files in Matlab).

To apply these heuristics certain AMPL modeling conventions have to be used. A modeling template describes these conventions with an example. Test results comparing HOTDOG/MICIO with other algorithms are also given.

9) DATE	AUTHORIZED BY	POSITION
	This page only	
9 August 2001	Bent Erik Bakken	Director of Research

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE
(when data entered)

CONTENTS

HOTDOG : A HEURISTIC ON THE DETERMINATION OF OPTIMUM GLOBALLY USING THE MATHEMATICAL PROGRAMMING LANGUAGE AMPL

1 INTRODUCTION

Global optimization of nonconvex, constrained problems is a challenging and difficult task, but nevertheless often required for solving practical, realistic problems. Many suggestions and algorithms have been reported with different properties and results (1)(2)(4)(6)(7)(8)(9). There is (as yet) no altogether "best" method found, even though tunneling type algorithms show promise (6)(7)(9). An approach for robust, large-scale global optimization is therefore of interest, where robust refers to obtaining feasible local optimal solutions within constraints tolerances, and large-scale may imply nonconvex problems with hundreds to thousands of decision variables.

Here, a heuristic algorithm HOTDOG ("Heuristic On The Determination of Optimum Globally") is described for problems with a restricted number of local optima. A simpler version of the algorithm called MICIO ("MInimum Computation by Iterative Optimization") that uses the same local search procedure as HOTDOG, is also developed for use when the number of local optima is very large, much in the spirit of multi-level single-linkage type algorithms (1). Both algorithms are developed using the features and generic functions in AMPL itself (5), and both the HOTDOG and MICIO heuristics may therefore be invoked by a standard *include*-statement command in AMPL (5) (corresponding to *m*-files in Matlab). This may seem ineffective compared to ordinary solvers, but experience shows that this is not very serious for small problems (where response times are short anyway), and may also be preferred for large problems where other methods may fail to find feasible solutions. The pace of technology also works to reduce the response times for large problems.

First, a general problem formulation is presented with a transformation to a total penalty function form used by both heuristics. Then, in chapter 2 the general features of the heuristics for HOTDOG and MICIO are presented, together with a detailed description of the common local optimum search procedure used. A flowchart guide to both heuristics is also included with a description of user input parameters. To apply these heuristic algorithms certain AMPL modeling conventions have to be used. A general modeling template in chapter 3 describes these conventions with an example. Finally, some test results comparing HOTDOG/MICIO with other algorithms are given in chapter 4.

1.1 Problem formulation

A general nonconvex, constrained mathematical programming problem may be formulated as shown in Problem 1, where $f(\mathbf{x})$ is the basic objective function of the bounded decision variables $\mathbf{x} \in \Re^n$ to be optimized, and where both inequality $g(\mathbf{x})$ and equality $h(\mathbf{x})$ constraint

functions are included. When one or more of the functions are nonconvex this is a global optimization problem. *Note that the symbol* $\boldsymbol{f}$ *is reserved for the basic objective function.*

Problem 1:

$$\begin{aligned} &\min_{\mathbf{x}} f(\mathbf{x}) \\ &g_i(\mathbf{x}) \le 0, \quad i = 1,\ldots,I \\ &h_j(\mathbf{x}) = 0, \quad j = 1,\ldots,J \\ &\mathbf{x} \in [\mathbf{l}, \mathbf{u}] \end{aligned} \tag{1.1}$$

This original problem is reformulated as a total penalty function to return a real number for all real input values, even if the constraints are violated (i.e., the input value is an infeasible point). The constraints are reformulated as expressions with nonpositive values when the constraints are satisfied. This means that an equality constraint like $h_j(x) = 0$ will be included as $| h_j(x) |$ in the penalty function, while inequality constraints as represented above are unchanged. For each constraint a tolerance limit *Tol* and a penalty weight *Pen* also has to be given to determine the corresponding penalty function term. The penalty function term for each constraint is reformulated as a combination of a quadratic term within the tolerance limit and a linear term outside. If the constraint expression is $v_k(\mathbf{x})$, then the corresponding penalty function term combination will be:

$$L_k(\mathbf{x}) = \frac{Pen_k}{2 \cdot Tol_k} \cdot \min\left(Tol_k, \max\left(0, v_k(\mathbf{x})\right)\right)^2 + Pen_k \cdot \max\left(0, v_k(\mathbf{x}) - Tol_k\right) \tag{1.2}$$

The derivatives of this penalty function term with respect to **x** are continuous wherever $v_k(\mathbf{x})$ is continuous. This makes the penalty function term easier to handle for the search algorithm. The purpose of expressing the penalty function term in this way is to try to get a progression towards the feasible area as a linear function, and at the same time trying to achieve convergence as a quadratic function within the feasible area. The value of Pen_k should be set large enough to make the value of the total penalty function deteriorate when outside the feasible area.

The reformulated version of Problem 1 with a total penalty function $F(\mathbf{x})$ may then be expressed as follows:

Problem 2:

$$\begin{aligned} &\min_{\mathbf{x}} F(\mathbf{x}) = f(\mathbf{x}) + \sum_{k=1}^{I+J} L_k(\mathbf{x}) \\ &\mathbf{x} \in [\mathbf{l}, \mathbf{u}] \end{aligned} \tag{1.3}$$

This problem reformulation will have the same minimizing solution as Problem 1 when the constraints are satisfied within their tolerances and the $L_k(\mathbf{x})$ terms are negligible.
(Note that a maximizing formulation may be expressed by instead minimizing $-F(\mathbf{x})$.)

The heuristic algorithms HOTDOG and MICIO are developed to solve the reformulated Problem 2 version of the original Problem 1. An AMPL general modeling template is described in chapter 3 to guide the Problem 1 reformulation process to a Problem 2 representation.

Equalities are in general more difficult to handle than inequalities, since the equalities are always active. The equalities should therefore mainly be used to eliminate variables by using the defined variable features of AMPL (5). Thereby, both the number of decision variables and the number of constraints are reduced. The equalities kept should be given the highest possible acceptable tolerance limit.

2 ALGORITHM

To solve Problem 2, two heuristic algorithms are developed for effective and robust search for a global minimum in large nonconvex, constrained problems with a restricted number of local minima. First, a general description of the main ideas of the algorithm HOTDOG is given. Then, the simpler version MICIO that uses the same local search procedure as HOTDOG is described. The local search procedure steps are also described in more detail together with flow-chart descriptions for both HOTDOG and MICIO. Finally, the set of input parameters of the algorithms to control the optimization is described.

2.1 General

To find the global optimum of a function, a usual procedure (1) is to start a number of local searches from different random initial start points and choose the smallest local optimum as an estimate of the global optimum. However, to search for local optima in the same search space more than once, will probably reach one of the local optima already found. Various methods of choosing a subset of starting points from the set of generated points have been developed (1)(4)(7). The main idea in these methods is to avoid searches to be started from points close to another. Some of the methods also have criteria for instance related to the function values or the gradient directions in the start points (1).

HOTDOG is a heuristic algorithm that tries to achieve improvements in performance by terminating some of the local searches after they have started. If the number of local optima is small compared to the number of searches started, we can achieve a considerable reduction in local iterations. The main idea is to terminate a search as fast as possible if it is close to an already found local optimum. In HOTDOG, this is included in the algorithm by storing the path of local search legs as a point sequence forming continuous line segments. In this way, we can, during a local search, test the distance to these previous path legs. If we are close, the search will probably reach an earlier local optimum, and the search will be terminated.

If we visualize the problem as a multi-dimensional topography with “hills” and “valleys”, local minimum searches will probably find the “valleys” quite fast. When testing for the distance to previous path legs, a certain (user input) tolerance radius will determine a possible local search cut-off, and the regions where the searches are terminated can thus be seen as a network of "hotdogs" with thickness proportional to the tolerance radius (giving an association to the

heuristic algorithm name). In a minimization problem the concentration of these will be in the lower parts of the "valleys", and most of the searches will terminate there. As new starting points are generated the local searches should therefore tend to be terminated when hitting a "hotdog" before a full search path is finished, thus saving some local iterations as shown in Figure 2.1. This termination process should be most useful when a problem has a restricted number of local optima.

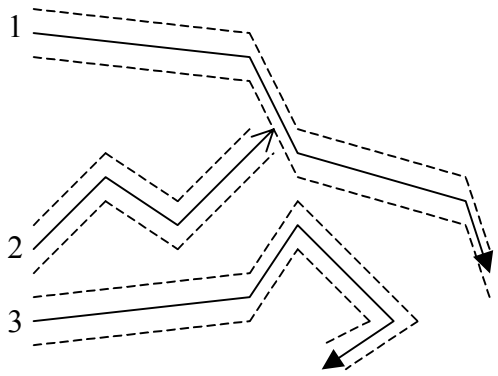


Figure 2.1 *The dotted lines represent the outer edges of the "hotdogs", while the solid lines represent the path legs in the local search. The filled arrows are the end point of a path. The simple arrow in path 2 represents the end point of a path that lies within the hotdog of another path.*

If, on the other hand, the number of local optima is large, it is considered better to finish all local searches. HOTDOG would in these cases only terminate a small part of the local searches with a probably marginal efficiency improvement in the number of iterations at an increased storage requirement. In these cases, methods that test only the initial start points are probably even less helpful (1). We can then only hope that the best local optimum found is a good estimate of the global optimum. Therefore, for these kind of problems a simpler algorithm MICIO is also developed, which uses the same local search procedure, but does not store and test previous local search path legs for cut-offs, thus giving a random multi-start algorithm (1). Every local search is then run to the end, and the best local optimum found is used as an estimate of the global optimum. This simpler logic is the only difference between HOTDOG and MICIO, but since MICIO does not store any path search legs, this algorithm may be used for slightly larger problems. Both HOTDOG and MICIO are designed for robust optimization of large problems (> 100 variables) and implemented using features and generic functions in AMPL itself only (5). Thus, there is no need for a special solver tool, since both the HOTDOG and MICIO heuristics may be invoked by a standard *include*-statement command in AMPL (5).

As a further possible improvement, a number of user specified phases can be continued for further local optimization searches. If some of the local searches in the initial phase exited with too many iterations, and the best of these are almost as good as the best end point so far, the best of these are continued with further local searches until the number of phases are equal to *max_phase*, the total number of searches equals *max_totalrep*, or the improvement in the last

phase of the particular search was too small. Thus, both algorithms may be tailored to the particular needs or characteristics of each problem.

2.2 Local search procedure

Both HOTDOG and MICIO use the same basic local search procedure for optimization problems with a gradient-based search. A vector related to the second-order derivative is also calculated, giving some of the advantages using the complete Hessian matrix, but without the extra calculations and storage requirements necessary for the Hessian (2). Thus, the algorithm is especially suitable for large, nonlinear problems where gradient methods based upon the Hessian are too costly (2)(3).

The local search procedure uses the total penalty function $F(\mathbf{x})$ of Problem 2, where each of the decision variables x_i can take all values within lower and upper bounds l_i , u_i. As seen, the constraints are included as penalty terms in the function $F(\mathbf{x})$ to be optimized. User defined penalty coefficients and constraint tolerances are supposed given relative to the basic objective function $f(\mathbf{x})$, such that this function and $F(\mathbf{x})$ are both optimized (with negligible penalty terms). A description of certain modeling template requirements for using HOTDOG or MICIO with an AMPL problem is given in chapter 3.

The local search procedure uses AMPL's internal support for gradient calculations of a function $\nabla F(\mathbf{x})$ *(* $= \mathbf{x}.rc$*)* based on the effective and elegant method of automatic differentiation (2)(5). The algorithm draws a (user specified maximum) number of random start points within the lower and upper bounds (*.lb* and *.ub*), and starts local searches from these initial start points. HOTDOG also stores the path legs that is generated, and terminates the current search if it gets too close to a previous path leg (hitting one of the "hotdogs").

An assumption for the local search procedure heuristic is that the function $F(\mathbf{x})$ may be approximated locally by quadratic functions of the variables. Therefore, major iteration steps dx_i are calculated individually for each variable x_i based upon two gradients, which are also applied in the minor iterations step updates as a quadratic interpolation local line search.

The main characteristics of the local search procedure heuristic in HOTDOG/MICIO can be summarized as follows expressed in AMPL 'pseudo-code' (**dx** is a vector notation with individual components dx_i).

0. Set local termination criteria OptTol, MaxIters, MaxRed and precision ε
1. Let **dx**:=0, **rc**:=0, make a random draw for initial start point:= **x** ;
2. **Major iterations**:
   Let f1:=F(**x**), let **x1**:=**x**, let **rc2**:=**rc**, let $rc_i := \nabla_i F(\mathbf{x}) \cdot |u_i - l_i|$;(**rc** scaled)
3. If |**rc**|≤OptTol or MaxIters=true then *stop* -> **x**,F(**x**) is local optimum.
4. Let dx_i:= if $|dx_i| \le \varepsilon$ then (if rc_i<0 then 0.01 else if rc_i>0 then -0.01)
   else $\frac{rc_i \times dx_i}{rc2_i - rc_i}$; (step:local quadratic approximation)
   (The test also keeps the variables within bounds, and also prevents too large changes or changes in the wrong direction)

5. **Minor iterations:**
   Let $x_i := x1_i + |u_i\text{-}l_i| \cdot dx_i$; (new point = rescaled update for all *i*)
6. If F(**x**)<f1 - $\varepsilon \cdot (1+|F(\mathbf{x})|)$ then go to step 2 ; (improvement found)
7. If F(**x**)<f1 + $\varepsilon \cdot (1+|F(\mathbf{x})|)$ then go to step 2 ; (no further improvement?)
8. If MaxRed=true then go to step 2 ; (too many minor iterations)
9. Let dx_i:= $dx_i \cdot$(if $dx_i \cdot \nabla_i F(\mathbf{x}) \le \varepsilon$ then 1 else $\frac{rc_i}{rc_i - \nabla_i F(\mathbf{x})}$) ;
   (step:line search with quadratic interpolation)
10. Go to step 5

If it is preferable, an ordinary gradient direction search may be made after exit to try to get an even better result.

The penalty function parameters *Pen* of the total penalty function $F(\mathbf{x})$ should initially be set to values that give a resulting penalty that outweighs the corresponding improvement of the basic objective function $f(\mathbf{x})$ if violating the constraints of the feasible region. In this way, the gradients will usually point in a direction towards the feasible area. There can, of course, be certain cases where some of the local optima are also infeasible. If this is only a small part of the total local optima found, it would be acceptable to select the best of the local feasible optima, first eliminating the infeasible ones. If we get too many infeasible solutions, the corresponding penalty function parameters for the violated constraints must be increased. However, certain kinds of constraints can make problems for the algorithm if the penalty function terms are large. Thus, they shouldn't be exaggerated. To automatically assign penalty function parameter values may be a potential area for future research in further algorithmic improvement work.

If there are less than 300 variables in the problem, the student version of AMPL is sufficient for using HOTDOG or MICIO. If either a separate analytic expression for the gradient of the penalty function, or an alternate approximate gradient calculation procedure is available, the algorithms may also be implemented as ordinary solvers for large-scale problems.

2.3 HOTDOG

A flowchart of the HOTDOG algorithm is shown in Figure 2.3 where the (coloured) dotted lines and boxes represent the special HOTDOG features of the algorithm, while the solid (black) parts are the common local search algorithm described above.

The common local search algorithm is the inner core of both the HOTDOG and MICIO algorithm and will be explained in detail also referring to the 10 step description in chapter 2.2.

The task of HOTDOG is to determine the global minimum of the basic objective function **f** of Problem 2. The modeling template for the total penalty function **F** in equation 1.3 is described in chapter 3, and the first box of the flowchart generates this total penalty function from the modeling template information using a *reserved objective function name Obj*. The first box also sets all parameters and required options, assigning preset default values to options if they aren't given as input. A header is also printed to the main output file before starting the search.

The outermost loop of the algorithm controls the phases of the optimization. This loop and all tests related to it are marked green (dotted) in the flow chart. The parameter *Qphase* counts the number of finished phases (green loops), while the parameter *Qc* counts the number of local search (partial) paths. A path that has been terminated because *max_iters* iterations have been made is called a *partial* path. For each phase (green loop), the endpoints of the partial paths in last phase are tested, and this test determines which of these paths (if any) should be continued. This test is explained in detail in *g)* below.

The following detailed description of the HOTDOG algorithm is grouped according to the letters a) to h) shown on the flowchart in Figure 2.3:

a/b)
In the first phase, we randomly draw a user determined number of starting points *_svar[i], i=1,...,QN* (the number of variables). In the other phases, end points from the partial paths in the last phase already exist, and these are used as continuation points for the next phase.

c/d)
A search starts for another point with a smaller function value. First, we test if the current point is within already stored paths, the "hotdogs". This hotdog test checks whether or not the current point is closer to any of the line segments of the paths generated than the "hotdog" radius *Qr (default* $0.05\cdot\sqrt{QN}$). To keep track of all the line segments, each point visited is stored in an array *Qpoint[Qct,l,i]* referring to the *i*-coordinate of point *l* made in repetition *Qct*. The hotdog test is done for all line segments in all paths. A hotdog test for one of these line segments is shown in Figure 2.2 for a two-dimensional case. The test determines the distance from point C to the line segment AB and checks if it is greater than *Qr*. Referring to the AMPL generic synonyms, the point we wish to test, C, is called *_svar[i]*. Point A is the first and B is the second of the two points of the line segment that is tested. These points are called *Qpoint[Qct,l-1,i]* and *Qpoint[Qct,l,i]*, and *Qdist[Qct,l-1]* in the code is equal to the squared distance c^2 between the two points in the figure.

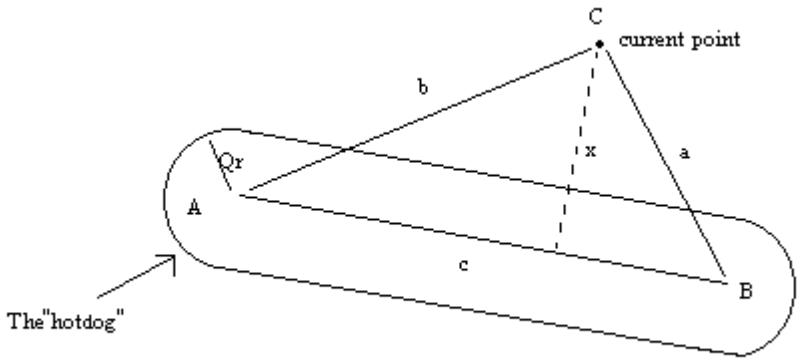


Figure 2.2 The distance from a point C to a "hotdog" line segment AB

If the angle ∠ ABC > 90° then the distance from line segment AB to C is a
If the angle ∠ BAC > 90° then the distance from line segment AB to C is b

If both angle $\angle$ BAC < 90° and $\angle$ ABC < 90° then the distance from line segment AB to C is x, found by solving the following equation:

$$x^2 = \frac{2a^2 + 2b^2 - c^2 - \frac{\left(a^2 - b^2\right)^2}{c^2}}{4}$$

If this test shows that the current point is within the "hotdog", the algorithm goes to part g) as shown in the flow chart. Otherwise, a search for a better point continues using the local search algorithm described above. If the gradient norm is less than or equal to *OptTol*, the point is considered a local minimum, and the local path is stopped. This will also happen if the total number of iterations completed exceeds *max_iters*. Otherwise, a new search direction and step length is evaluated as a vector *Qdx* by approximating the total penalty function locally by a second order function and finding a minimum for each coordinate, using 2 gradients and keeping the step inside the *x* domain. This will in general not be the best possible guess for a new point, but more memory and time-consuming methods have to be used to do better.

e)
From the current point the individual directions and step lengths of the vector *Qdx* determine a new point. If this new point is an improvement, we go directly to f). Else, the step length for each coordinate is reduced with a line search procedure using quadratic interpolation with 2 gradients. This step reduction loops no more than *max_red* times, in which case we jump to g).

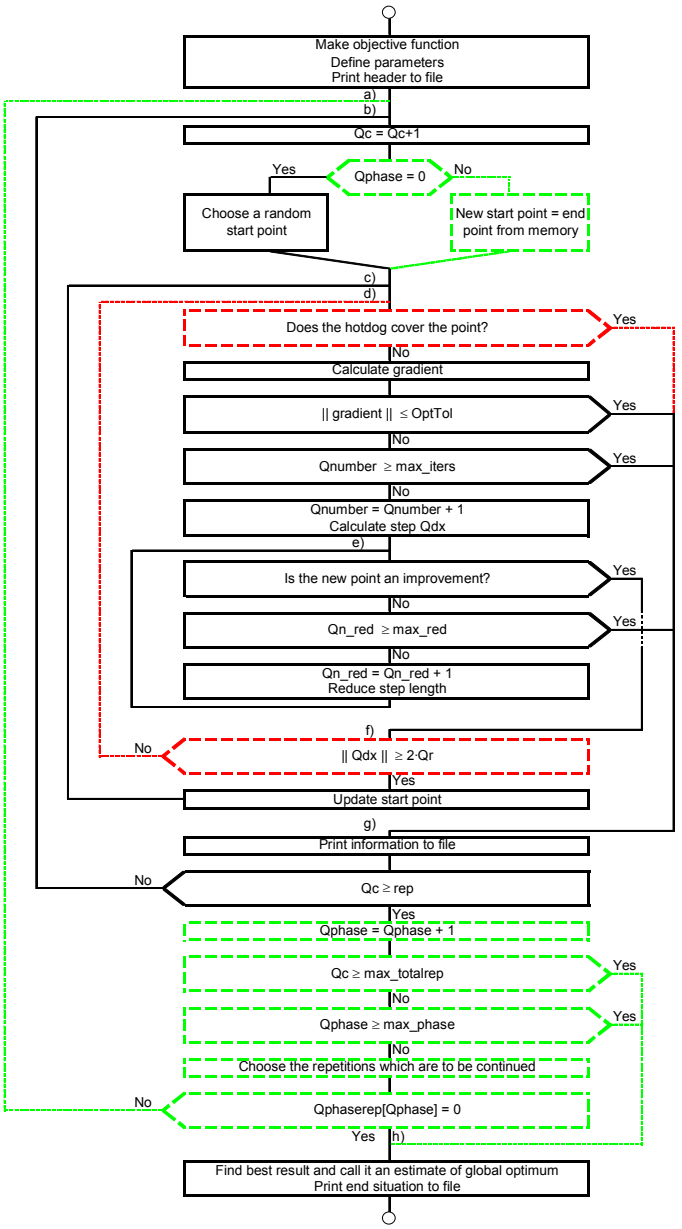
f)
Here, a new point has been found that is kept as the next point of the local search path. At this moment, there is a greater chance that the new point is inside the "hotdog" around the last line segment than ever, so it would probably be more efficient to take the hotdog-test just for this line segment now. But, we will this time just check whether the distance from the last point to this new point is less than 2·Qr. If not, this new point is accepted as the next point in the local search path, and the algorithm repeats from c).

g)
To get here, one of four conditions are TRUE:

1. The last point is within one of the "hotdogs"
2. || gradient || ≤ OptTol
3. *max_iters* iterations exceeded
4. The step length has been reduced more than *max_red* times

In cases 1, 2 and 4 a local search path is completed and a comment is printed to the output file corresponding to "Hotdog hit", "Local minimum" or "Cannot be improved" respectively. In case 3, the comment "Too many iterations" is printed to the file, but then the path will not immediately be considered as completed, but instead possibly continued as a partial path in later phases.

One local search repetition started at c) is now completed. In phase *Qphase*, this loop is run *Qphaserep[Qphase]* times. In the initial phase 0, a certain fixed number of random starting point repetitions given by the input parameter *Qphaserep[0]* = *rep* is searched. For each later phase, the number of partial paths is checked to make sure that *Qphaserep[Qphase]* number of repetitions is made. If not, the algorithm returns to step b), otherwise the repetitions limit, *max_totalrep*, or the phases limit, *max_phase* is tested. If the limits are reached, the algorithm continues at h). If not, the repetitions that are to be continued in the next phase are chosen.



Objective function:	Generated by combining the function f, which is to be optimized, and penalty function terms for the constraints
Qc:	Counter for local searches
Qphase:	Counter for phases
Hotdog:	See below
Gradient:	Scaled by domain size
OptTol:	If a point is to be regarded as a local minimum, then || gradient || has to be less than OptTol
Qnumber: max_iters:	Counter for iterations Maximum number of iterations in one local search
Qdx:	Step vector based upon local 2. order approximation within bounds
Qn_red: max_red:	Counter for step lenght reductions Maximum number of step length reductions in one iteration
Reduction of step length:	Line search with quadratic interpolation using 2 gradients
Qr:	The radius of the "hotdogs" relative to the standard value, which is 0.05*sqrt(N), where N is the number of variables
rep:	Maximum number of local searches in each phase
max_totalrep:	Maximum number of local searches in all phases
max_phase:	Maximum number of phases
Criterion for this:	See below
Qphaserep[i]:	Number of local searches we want to continue on in phase i

Hotdog: Points nearby former visited points are covered by a virtual hotdog to make sure that we don't test these points because we believe that these points will lead to the same local minimum as we already have found.

Criterion for continuing a local search:

1. The local search in phase i-1 has been stopped because Qnumber ≥ max_iters.
2. The function value of this end point is less than or equal to the best function value reached so far plus the absolute value of this function value multiplied by the parameter PhaseTol.

Figure 2.3 Flow chart for HOTDOG algorithm

To determine the partial paths that should be continued in new phases, the best of the repetitions so far is found. This repetition is stored as *best_rep*, and *Objs[best_rep]* is its objective value. Then, among the case 3 repetitions with "Too many iterations", the repetitions that are within a certain tolerance *PhaseTol* of *best_rep*, i.e. < (*Objs[best_rep]* + *PhaseTol* . | *Objs[best_rep]* |) are selected, assuming that a relatively good partial path will become better than the other partial paths. The number of repetitions to continue in the next phase *Qphaserep[Qphase]* and the indexes to the partial paths chosen *Qextra[Qc]* are then updated.

If there are no further partial paths to continue, *Qphaserep[Qphase] = 0*, and the algorithm finish at h), else a new phase starts by returning to a).

h)
At the end of the algorithm, results are printed to file. The most interesting information is of course the object function value and variable values of the global minimum estimate, which has to be evaluated first. This is done by comparing all the local minima found that have feasible solutions, choosing the best as the global minimum estimate. Other types of information of interest is the total number of gradient evaluations made, the length of all the paths, and in addition all the local minima values found.

The solution to the problem, that is, the variable values of the global minimum estimate are in general printed to another file executed from the problem run-file (see chapter 3.3), but if the number of variables is less than 10, the variable values of the local minima are also printed to the HOTDOG output file.

This ends the HOTDOG algorithm, which may be invoked by an AMPL *include*-statement. The algorithm may also be written as an ordinary solver, if gradient calculation programs/procedures are provided. The version described works as an interpreted solver in the same way as *m*-files in MATLAB, and is straightforward to use when certain modeling conventions are followed. These are described in chapter 3.

As mentioned, the HOTDOG algorithm works best for problems having a reasonably small number of local minima, since storage of local search paths and additional testing of distances to them are made. A simpler algorithm version MICIO is therefore also developed which may be faster when the number of local minima is very large.

2.4 MICIO

MICIO is an algorithm using the same random starting procedure and repetitive optimizing phases as HOTDOG, but without the hotdog test for stored path distances (see Figure 2.4). Therefore, because of these similarities no in-depth description of the contents of MICIO is made, just in general referring to chapter 2.3.

The basic parts of MICIO are the same as in HOTDOG, that is, the method of finding a local minimum is the same in both algorithms. By skipping the hotdog test, MICIO becomes less

memory consuming than HOTDOG, giving the possibility to increase the number of variables in the model or making the model more complex.

The MICIO algorithm is also grouped according to the letters a) to h) shown on the flowchart in Figure 2.4 below, with the same content except the hotdog test of d) and f) which are missing. The same phase logic of g), and final result reports of h) as in HOTDOG are also included in MICIO. The only difference should be the number of iterations and gradient evaluations made.

2.5 Parameters

The algorithms for HOTDOG and MICIO may be controlled by a set of user input parameters. The most important of these are shown in the following AMPL-code of the initializing part 'Define parameters' of the algorithms. These parameters are described together with suggested default values.

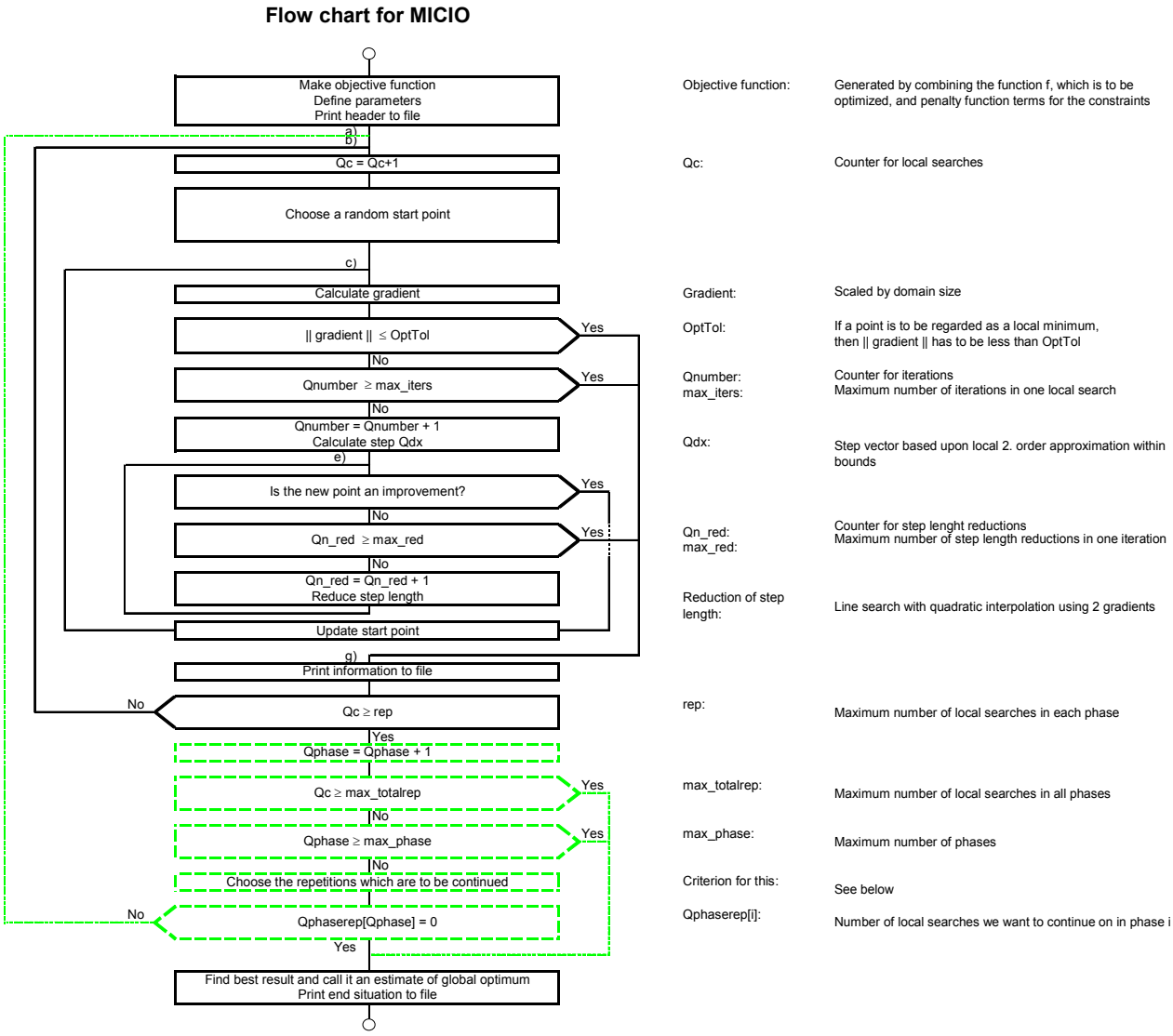


Figure 2.4 Flow chart for MICIO algorithm

AMPL-code of 'Define parameters':

```
if {'file'} not within _PARS then print "param file symbolic default 'HotDog.tmp';" >> options;
if {'output'} not within _PARS then print "param output default 2;" >> options;
if {'xl'} not within _PARS then print "param xl default 0;" >> options;
if {'StartDef'} not within _PARS then print "param StartDef default 1;" >> options;
if {'OptTol'} not within _PARS then print "param OptTol default 0.03;" >> options;
if {'rep'} not within _PARS then print "param rep default 10;" >> options;
if {'max_red'} not within _PARS then print "param max_red integer default 13;" >> options;
if {'max_iters'} not within _PARS then print "param max_iters default 250;" >> options;
if {'max_step'} not within _PARS then print "param max_step default 0.1;" >> options;
if {'max_acc'} not within _PARS then print "param max_acc default 1e-16;" >> options;
if {'max_phase'} not within _PARS then print "param max_phase default 10;" >> options;
if {'max_totalrep'} not within _PARS then print "param max_totalrep default 2*rep;" >> options;
if {'PhaseTol'} not within _PARS then print "param PhaseTol default 0.1;" >> options;
if {'Qr'} not within _PARS then print "param Qr default 1;" >> options;

include options;
```

The meanings of these parameters are as follows:

file:	The name of the file you want HOTDOG/MICIO to print the results to.
output:	The amount of information that should be written to the file: 0 gives no information at all 1 gives a short summary 2 gives a summary of each repetition 3 gives a more detailed summary of each repetition
xl:	0 gives a text file, 1 makes converting the output files to Excel-format easier
StartDef:	Set to 1 if the local search is to be started from the default values of the variables
OptTol:	If || gradient || is smaller than OptTol, the point is considered a local minimum
rep:	The number of repetitions in the first phase
max_red:	The maximal number of step reductions in each iteration
max_iters:	The maximal number of iterations in each repetition
max_step:	The maximum change in each variable relative to the difference of the upper and lower bounds of the variable
max_acc:	The maximum accuracy in the minor iterations
max_phase:	The maximal number of extra phases
max_totalrep:	The maximal number of total repetitions
PhaseTol:	If the difference of the value of **f** at the end point of a partial path and the best local point so far is greater than PhaseTol, the partial path will not be continued
Qr:	The radius of the "hotdogs" that are generated around the paths in HOTDOG, relative to the standard value 0.05·sqrt(N), where N is the number of variables. In MICIO, this parameter will have no effect.

These parameter values should be modified to suit the problem being optimized, but the default values may be used as a first trial compromise. The values are easily changed when necessary.

3 AMPL MODELING TEMPLATE

To use HOTDOG and MICIO certain AMPL modeling conventions have to be followed. A demonstration example applying an AMPL modeling template is given. Both algorithms are called from a run file, which also uses *include*–statements for the files formulating the problem according to the modeling template. The problem is conveniently divided into a model file, a constraints file, a data file and a run file.

An example will show how these files can be made. The run file has to be designed in a special way in order to make HOTDOG/MICIO work correctly using the general modeling template. The example considers the well known problem of minimizing the volume of a cube containing **n** spheres of radius 1 (10). The formulation of this problem is shown in the different file listings below for the model-, constraints- and run-file.

3.1 Model

The following code is the content of the spheres problem model file *'spheres.mod'*:

```
param n integer > 1 default 19;      # the number of spheres
var pos {i in 1..n, dim in 1..3} >=1, <=(1+2*floor((n-1)^(1/3)));
                                     # 3-dimensional sphere coordinates

# var f = <function to be minimzed>, NB! Reserved basic objective name f!
var f = (1+max {i in 1..n, dim in 1..3} pos[i,dim]);  # cube side length

# The basic object function may also be given a defined name for readability
var cube = f;
```

A separate data file may also be specified, but is not used for this example.

3.2 Constraints

Each constraint must have the form shown in Problem 2 of chapter 1.1 and be declared as a defined variable. To get a correct reference to the constraint variables two pointers *QD* and *QM* are updated using the AMPL generic feature *_nvars*. These pointers relate the constraint tolerances and penalty parameters to the corresponding defined constraint variables, and are updated in a constraint *let-block*. One such constraint *let-block* must follow each new constraint definition. To illustrate the use the spheres-example is shown; constraint definitions using the general modeling template is described in the next section.

Even though the constraint definitions may be included directly as part of the model-file, the following constraint definition code shows the application of the general modeling template written to a separate constraint file *'spheres.con'*:

```
# Constraint definitions
var constraint {i in 1..n, j in 1..n: i>j} = (4 - sum {dim in 1..3}
(pos[i,dim]-pos[j,dim])^2);   # the constraint variable as a difference

# Constraint let-block
let QM:=_nvars;                        # the current variable pointer
let {i in QD..QM} FeasTol[i] := 1e-2; # the feasibility tolerances
let {i in QD..QM} Pen[i] := 1e-2;        # the penalty parameters
let QD:=QM+1;                          # the next constraint pointer
```

One such constraint definition and constraint let-block must be given for each new constraint of the problem formulation.

3.3 Run-file and general modeling template

To apply the HOTDOG/MICIO algorithms the following general modeling template should be used. By replacing the filenames *'spheres.mod'* and *'spheres.con'* with other model- and constraint-filenames using the same modeling template, a general run-file is made. A data-file may also be included after the model-file in the usual way for AMPL models.

The following code example shows the general modeling template applied to make the content of the spheres problem run-file *'spheres.run'*:

```
# GENERAL MODELING TEMPLATE HAS THREE PARTS:
#       I) RUN PARAMETERS
#      II) MODELING PART
#     III) SOLVER PART


# I) RUN PARAMETERS

# Reset AMPL
reset;
reset options;

option omit_zero_rows 0;
option display_eps 1e-5;

option show_stats 0;
```

```
option times 0;

option linelim 0;
option substout 0;
option pl_linearize 0;
option presolve 1;


# Declarations of HOTDOG/MICIO input parameters

param QC;
param QD;
param QM;
param FeasTol {QC+1..QM};
param Pen {i in QC+1..QM};


# HOTDOG/MICIO input parameter default settings

option bane '';
param file symbolic     default 'hotdog.tmp';
param output            default 2;
param xl                default 0;
param StartDef          default 0;
param OptTol            default 0.01;
param rep               default 8;
param max_red           default 16;
param max_iters         default 2000;,
param max_step          default 0.02;
param max_acc           default 1e-16;
param max_phase         default     8;
param max_totalrep      default 2*rep;
param PhaseTol          default 0.01;
param Qr                default 1;
```

(In the spheres and the Lennard Jones$_{20}$ problems, the parameters rep, max_phase, max_totalrep *and* PhaseTol *are omitted. The search is instead stopped when a function value close to the global optimum of the problem has been found. This method is only used for testing the algorithm, and is used only on problems with known optimum values to test the performance of the algorithm.)*

```
let file := ($bane & file);
```

```
# II) MODELING PART

# Problem

# All the declarations of sets, parameters and variables of the problem are
# set here.
# Data files (if any) are to be loaded here. AMPL will generate the actual
# model for further adjustment to HOTDOG/MICIO use.
#
# The information can be put in model- and data-files respectively, and
# these can be included to give a better overview of the run, e.g. like this:
#
# include <mymodel.mod>
# data <mydatafile.dat>

include spheres.mod;  # This is the spheres-example model above, no data-file

# Mandatory let-block to define number of variables and constraint reference

let QC:=_nvars;   # Number of variables generated so far, that is, before the
                  # constraints
let QD:=QC+1;     # Used for convenient reference to the constraint variables

# All the constraints are to be set here.
# A constraint "let-block" has to be included between constraints with
# different tolerance limits or penalty parameters. All the constraints
# defined since last let-block will be associated with the values that are
# put into the let-block. It should always be a let-block at the end whether
# there are any constraints or not.
#
# Constraints are made by defining variables that are supposed to be non-
# positive when inside the feasible area.
#
# An example showing how to formulate the constraint Kari: c(x)<=g(x):
# var Kari = c(x)-g(x);
#
# An example showing how to formulate the constraint Ola:  c(x)=h(x):
# var Ola = abs(c(x)-h(x));
# As for the model variables, the constraints can be put in a constraint file
# to give a better overview:
#
# include <myconstraints.con>;

include spheres.con;       # This is the spheres-example constraints above
```

```
# Mandatory constraint let-block for each different constraint definition
# -- This is a constraint "let-block" --
# let QM:=_nvars;
# let {i in QD..QM} FeasTol[i] := 1e-1;
# let {i in QD..QM} Pen[i] := 3;    # The constraint variables are given
# tolerance and penalty values in the Objective function in HOTDOG/MICIO
# let QD:=QM+1;
# -- This is the end of the "let-block"
# Each such "let-block" is best included as part of the constraint-file


#  III) SOLVER PART

# Search for a minimizing solution with HOTDOG/MICIO

# include micio.txt     # Remove the #-mark at the line start to use MICIO
# include hotdog.txt    # Remove the #-mark at the line start to use HOTDOG

include hotdog.txt;        # This starts a spheres-example HOTDOG search

# Print the results using AMPL generic features:
# The following files are generated automatically:
# "zInf.tmp" with all the constraints where the slack exceeds FeasTol/2
# "zVar.tmp" with the value and gradient for all the variables, and also the
# slack and dual value for the constraints
# "zSol.tmp" that can be used to include the solution to restart AMPL later
# (To do this, write:
#  include zSol.tmp;
# after the model and the constraints are loaded)

print {i in 1.._snvars}: "let", _svarname[i], ":=", _svar[i], ";" > ($bane &
'zSol' & '' & '.tmp');
print {i in QC+1..QM : _var[i] > Feas_Tol[i]/2 } : _varname[i], _var[i] >
($bane & 'zInf' & '' & '.tmp');
print {i in _VARS}  ('display ' & i & ', ' & i & '.rc, ' & i & '.dual' & " >
('" & $bane & "zVar' & '' & '.tmp');") > display.tmp;
close;

include display.tmp;      # The print statements are executed

close;

#  END OF RUN_FILE
#  END OF GENERAL MODELING TEMPLATE
```

This run-file template description will work in general for all type of NLP-problems with real variables without the need of separate solvers, if the above template with the let-blocks are used and the AMPL gradient computations <*var.rc*> are available. Other gradient computations may also be used, but then the HOTDOG/MICIO gradient reference statements must be modified accordingly.

3.4 Result file

An example of the *'hotdog.tmp'* log-file for the spheres problem (19 spheres) is shown below:

```
Variables:    57
Constraints:  171

Draw          Iters  Total  Grad   Feas   Penalty      Function     End
1             351    351    3.995  1      3.7e-05      5.848        Cannot be improved
2             334    685    3.994  1      2.4e-06      5.787        Cannot be improved
3             235    920    3.845  0      0.02645      5.916        Cannot be improved
4             250    1170   3.956  1      2.4e-05      5.83         Cannot be improved
5             86     1256   3.849  0      0.00442      5.902        Cannot be improved
6             485    1741   3.961  0      0.00017      5.83         Cannot be improved
7             381    2122   3.899  0      0.00026      5.904        Cannot be improved
8             248    2370   4      0      0.03329      5.85         Cannot be improved
9             399    2769   3.985  1      8.6e-05      5.809        Cannot be improved
.
.
.
3262          1596   778100 3.996  1      6.4e-05      5.475        Cannot be improved

Gradient evaluations:      2325000

Local minimum points:
Objs[j] [*] :=
   1 5.8482
   2 5.7872
   4 5.8298
   9 5.8091
   .
   .
   .

3262 5.4746

;


Global minimum point: (?)

best_rep = 3262

f = 5.4745

Obj = 5.4746

feas = 1


sum{i in 1 .. Qc} Qline[i] = 17627

(sum{i in 1 .. Qc} Qline[i])/Qc = 5.4037
```

The output file first shows the number of variables and constraints of the *spheres* problem. Then one line for each repetition show: *Draw* (the repetition number, and also *(i-j)* if new phases *j* are tried for repetition number *i*), *Iters* (number of iterations for this repetition), *Total* (the accumulated number of iterations so far), *Grad* (the gradient norm at the end point), *Feas* (feasibility tolerance indication: 1 feasible, 0 infeasible), *Penalty* (the summed value of the penalty terms for the constraints), *Function* (the basic objective function **f** value at the end point), *End* (a search stop message for the repetition), *Last improvement* (the improvement of *(i-*

j) : a new phase *j* of repetition *i*), *Total improvement* (the accumulated improvements of *(i-j)* : all phases including *j* of repetition *i*).

Then follows the total accumulated gradient evaluations *<var.rc>* for all the iterations of all the repetitions as an indication of algorithm efficiency.

All feasible local minimum points are then listed, and the best one is suggested as a global minimum point (?) if feasible, shown with repetition number, basic objective value f, total objective value including penalty terms and feasibility indication.

Also shown are the HOTDOG accumulated line segment number for all paths, and average number of line segments of each path.

The output level of detail may be controlled by the *output* parameter. Here, *output* = 2.

4 RESULTS

4.1 Example tests

Results of benchmark tests with HOTDOG/MICIO are compared with other methods referenced (1)(4)(6)(7). The number of iterations and total gradient- and function-evaluations are used as criteria for comparison of the methods.

The test functions used are the unconstrained optimization problems Rosenbrock_{100}, Rosenbrock_{1000} (4), Shubert (4), $\text{Shekel}_{4,10}$ (4), $\text{Lennard-Jones}_{20}$ (20-atom molecule potential energy function) (1), Spheres (10), and a constrained problem labelled Test-constraints (equal to example 4.1 of (8)). These tests together exercise the global optimization properties of the HOTDOG/MICIO algorithms.

The methods referenced for comparison are Simple Linkage (SL) of (1), the standard Multi-level Single-Linkage (MLSL) (1)(6), the Enhanced Continuous Tabu Search (ECTS) of (4), the Gradient Descent Dynamic Tunneling (GRDT) of (7), the Value-Estimation Function Method of (8), and last, but not least, the Terminal Repeller Unconstrained Subenergy Tunneling (TRUST) algorithm of (6).

Table 4.1 shows the number of function evaluations required of each method for the various test problems, where the results are taken from the references where available (otherwise – is marked in the table). For HOTDOG/MICIO the tests are run with results shown as mean values based upon 10 trials runs for each example test function.

It is seen that the HOTDOG/MICIO algorithms altogether compare favourably with the other referenced methods where applicable, apart from TRUST for the Shubert function (6), which is 5-6 times more effective in the number of function evaluations. Both algorithms also solve the rather large 1000 variables Rosenbrock problem with acceptable efficiency and success rate.

Methods	Test-functions						
	$Rosenbrock_{100}$	$Rosenbrock_{1000}$	Shubert	$Shekel_{4,10}$	Lennard-$Jones_{20}$	$Spheres_{19}$	Test-Constraints
SL	_	_	_	_	358160	_	_
MLSL	_	_	_	_	291054	_	_
ECTS	162532 (.75)	_	370	898 (.75)	_	_	_
GRDT	_	_	502	469	_	_	_
VEF	_	_	_	_	_	_	149
TRUST	_	_	72	_	_	_	_
HOTDOG	10931 (.90)	18480 (.80)	364	234	318152	3817900	97
MICIO	11315 (.90)	17094 (.80)	437	207	332544	3828200	132

Table 4.1 Results of tests comparing referenced methods with HOTDOG/MICIO. The number of function evaluations required for reaching a global optimum are shown. In parenthesis are also shown success rates of the methods where applicable.

The best found minimum value of the $Spheres_{19}$ problem with HOTDOG/MICIO is 5.461. The global minimum of this problem is (currently) 5.459 (10). The number shown in the table is an average number of gradient/function evaluations required for reaching 5.475, which is 0.3% above the assumed global minimum.

The solver MINOS5.5 was used very effectively for the Rosenbrock problems (1250 function evaluations for $Rosenbrock_{100}$). However, MINOS5.5 used in a random multistart loop did not reach feasible solutions for the $Spheres_{19}$ nonconvex problem with an ordinary AMPL formulation in a comparable number of function evaluations. Of course, MINOS5.5 may be used as local search algorithm in HOTDOG/MICIO for possible improved efficiency of Problem 2 type formulations.

The test examples above with HOTDOG and MICIO were made with the following parameter configurations:

HOTDOG:

```
Rosenbrock100:      max_iters = 600   rep = 4     max_phase = 2
Rosenbrock300:      max_iters = 500   rep = 4     max_phase = 3
Rosenbrock1000:     max_iters = 800   rep = 4     max_phase = 3
Shubert:            max_iters = 10    rep = 60    max_phase = 3
Shekel:             max_iters = 30    rep = 15    max_phase = 3
LJ20:               max_iters = 1200  (rep and max_phase are not used)
Spheres19:          max_iters = 800   (rep and max_phase are not used)
Test-Constraints: max_iters = 30    rep = 50    max_phase = 3
```

MICIO:

```
Rosenbrock100:    max_iters = 600   rep = 4     max_phase = 2
Rosenbrock300:    max_iters = 500   rep = 4     max_phase = 3
Rosenbrock1000:   max_iters = 650   rep = 4     max_phase = 3
Shubert:          max_iters = 10    rep = 50    max_phase = 3
Shekel:           max_iters = 30    rep = 10    max_phase = 3
LJ20:             max_iters = 1200  (rep and max_phase are not used)
Spheres19:        max_iters = 800   (rep and max_phase are not used)
Test-Constraints: max_iters = 20    rep = 50    max_phase = 3
```

The results indicate that HOTDOG/MICIO may be used for nonconvex optimization of rather large problems, even though parameter tuning for each problem is required to get good results. Hopefully, further work may suggest more automatic computation of constraint penalty weights and parameter settings (2)(3)(9).

5 CONCLUSION

A heuristic algorithm HOTDOG is described for nonconvex problems with a restricted number of local optima, together with a simpler version of the algorithm called MICIO used when the number of local optima is very large. Both algorithms are developed using the features and generic functions in the mathematical programming language AMPL, and both the HOTDOG and MICIO heuristics may therefore be invoked by a standard *include*-statement command in AMPL (corresponding to *m*-files in Matlab).

To apply these heuristics, certain AMPL modeling conventions have to be used. A modeling template describes these conventions with an example.

Test results comparing HOTDOG/MICIO with other reference algorithms show a generally favourable performance in the number of function evaluations to reach an optimal solution.

To reach good results with HOTDOG/MICIO parameter tuning for each problem is required. Hopefully, further work may suggest more automatic computation of constraint penalty weights and parameter settings.

If there are less than 300 variables in the problem, the student version of AMPL is sufficient for using HOTDOG or MICIO. If either a separate analytic expression for the gradient of the penalty function, or an alternate approximate gradient calculation procedure is available, the algorithms may also be implemented as ordinary solvers for large-scale problems.

Both the HOTDOG and MICIO algorithms are available from the Norwegian Defence Research Establishment (FFI) upon request, subject to agreed conditions for use.

References

(1) Schoen F (1999): Global optimization methods for high-dimensional problems, *Eur Jnl of Op Res (EJOPR)* **119**, 345-352.

(2) Nash S G, Sofer A (1996): Linear and Nonlinear Programming, McGraw-Hill, N.Y., USA.

(3) Fletcher R (1987): Practical Methods of Optimization (2nd ed), J Wiley, UK.

(4) Chelouah R, Siarry P (2000): Tabu search applied to global optimization*, Eur Jnl of Op Res (EJOPR)* **123**, 256-270

(5) Fourer R, Gay D M, Kernighan B W (1993): AMPL - A Modeling Language for Mathematical Programming, Boyd & Fraser, Mass., USA.

(6) Barhen J, Protopopescu V, Reister D (1997): TRUST: A Deterministic Algorithm for Global Optimization, *Science* **276**, 1094-1097.

(7) Roychowdhury P, Singh Y P, Chansarkar R A (2000): Hybridization of Gradient Descent Algorithms with Dynamic Tunneling Methods for Global Optimization, *IEEE Trans. on Systems, Man and Cybernetics, part A* **30**, 3 (May), 384-390.

(8) Sun X L, Li D (1999): Value-Estimation Function Method for Constrained Global Optimization, *Jnl of Opt. Theory and Appl.* **102**, 2 (august), 385-409.

(9) Banze I M, Csendes T, Horst R, Pardalos P M (eds) (1997): Developments in Global Optimization, Kluwer Academic Publ..

(10) Boll D W, Donovan J, Graham R L, Lubachevsky B D (2000): Improving dense packings of equal disks in a square, *Electronic jnl of Combinatorics* (see also http://www.frii.com/~dboll/packing.html) **7**, *(#R46).*

DISTRIBUTION LIST

FFISYS **Dato:** 9 august 2001

RAPPORTTYPE (KRYSS AV)	RAPPORT NR.	REFERANSE	RAPPORTENS DATO
[ ] RAPP [X] NOTAT [ ] RR	2001/03011	FFISYS/807/161	9 august 2001
RAPPORTENS BESKYTTELSESGRAD		ANTALL EKS UTSTEDT	ANTALL SIDER
Unclassified		30	29
RAPPORTENS TITTEL		FORFATTER(E)	
HOTDOG : A HEURISTIC ON THE DETERMINATION OF OPTIMUM GLOBALLY USING THE MATHEMATICAL PROGRAMMING LANGUAGE AMPL		GROTMOL Øyvind, SUKKESTAD Jens Arne, BRAATHEN Sverre	
FORDELING GODKJENT AV FORSKNINGSSJEF:		FORDELING GODKJENT AV AVDELINGSSJEF:	

EKSTERN FORDELING

ANTALL	EKS NR	TIL
1		Øyvind Grotmol, NTNU

INTERN FORDELING

ANTALL	EKS NR	TIL
2		FFI-Bibl
1		Adm direktør/stabssjef
1		FFIE
1		FFISYS
1		FFIBM
1		Ragnvald H Solstrand, FFISYS
1		Bent Erik Bakken, FFISYS
1		Jan Erik Torp, FFISYS
1		Anne Lise Bjørnstad, FFISYS
1		Sverre Braathen, FFISYS
1		Bård Eggereide, FFISYS
1		Geir Enemo, FFISYS
1		Ole Martin Halck, FFISYS
1		Øyvind Karlsrud, FFISYS
1		Tor Langsæter, FFISYS
1		Terje Nilsen, FFISYS
1		Tor-Erik, Schjelderup, FFISYS
1		Ole-Jakob Sendstad, FFISYS
1		Jens Arne Sukkestad, FFISYS
1		Hans Olav Sundfør, FFISYS
		FFI-veven

FFI-K1 Retningslinjer for fordeling og forsendelse er gitt i Oraklet, Bind I, Bestemmelser om publikasjoner for Forsvarets forskningsinstitutt, pkt 2 og 5. Benytt ny side om nødvendig.