Operations Research 4.0

SoftAS GmbH

Operations Research is a product of SoftAS GmbH.

Mathematica® is a registered trademark of Wolfram Research, Inc. All other product names mentioned are trademarks of their producers.

November 2007
Fourth edition
Requires *Mathematica* Version 6.x

Software and manual written by: Jochen Kripfganz and Holger Perlt

Content

About Operations Research 4.0

Operations Research is a *Mathematica* application package used from within *Mathematica* to provide the user with a number of tools for handling a broad range of topics in business optimization.

Operations Research consists of the *Mathematica* package *OperationsResearch.m*, a this manual and data files. The complete package is loaded with the following command

```
<< "OperationsResearch.m";
```

If this command fails, please refer to your installation instructions.

This documentation is a complete tutorial for using the *Operations Research* package. It is organized into four main chapters. The user may wish to read these chapters in turn, although each chapter can be used independently as well. References and appendices with additional material are also included.
This manual cannot serve as a comprehensive introduction into the field of operations research. The reader is referred to the vast literature. We give the following short list of introductionary books because we found them particularly useful, and some of our notations and conventions have been taken from them

1. **Ronald L. Rardin**,
Optimization In Operations Rersearch, Prentice Hall, 1998
2. **H. P. Williams**,
Model Building in Mathematical Programming, John Wiley & Sons, 1993
3. **M. Asghar Bhatti**,
Practical Optimization Methods, Springer Telos, 2000
4. **James P. Ignizio and Tom C. Cavalier**,
Linear Programming, Prentice Hall, 1994

Linear Programming

Simplex Algorithm

Introduction

Mathematica 6.x itself provides efficient tools for Linear Programming, im particular through the built-in functions Maximize or Minimize, resp., and LinearProgramming. In particular for teaching, however, it may be desirable showing intermediate steps of the Simplex calculation. This is provided by the package function `Simplex`, which is called as `Simplex[Z, Constraints, Var, Options].`

It calculates the optimal objective function Z of variables Var subject to the list of constraints Constraints.

Sometimes, it may be useful to study the model dual to the primary model characterized by Z, Constraints, and Var. The dual model is provided by the package function `GenerateDualModel`. The result of `GenerateDualModel` is given in a form which can be used as input for `Simplex`

The sensitivity analysis is supported by the package functions `SensitivitySecVar` and `SensitivityTargetCoef`, resp. They start from the optimal tableau Zopt and ReplBasicVarOpt, expressing the target function and the basic varables (VarBasicOpt) in terms of the secondary (non-basic) ones (VarSecundaryOpt). These quantities can directly be copied from the output of the package function `Simplex`.

`SensitivitySecVar` describes the response of the optimal tableau if a secondary variable secvar is moved away from zero. The output also shows the range of deviations for which the optimal tableau remains valid.

Similarily, `SensitivityTargetCoef` describes the response of the optimal tableau if the target function coefficient of the variable Var is varied. The call is

`SensitivityTargetCoef[`*`Zopt, ReplBasicVarOpt, VarBasicOpt, VarSecondary-Opt, Var`*`]`

Functions - call and output syntax

Simplex

Call:

`Simplex[Z, Constraints, Var, Options]`

with

Z: objective function

Constraints: list of constraints, it expects the decision variables on the lhs of the relational operator, and constant additive terms on the rhs.

Var: list of variables, it has the structure {VarPos,VarNeg,VarNs}, with VarPos the list of variables with $x \geq 0$, VarNeg for variables with $x \leq 0$ and VarNs for variables without sign restrictions.

Options:

Option	Value	Description
PrintOpt	True	Prints intermediate results to the screen
Maximize	True	Maximizes the objective function
	False	Minimizes the objective function
NewName	u (default)	Prefix string given to the additional slack variables

Options for Simplex.

Output syntax:

```
{ZRes, VarRes, {ZResWithNonBasicVars,
  BasicVarValues, ListOfBasicVars, ListOfNonbasicVars}}
```

with

ZRes: optimal value of objective function

VarRes: values for the variables

ZResWithNonBasicVars: optimal value of objective function as function of the nonbasic<variables

BasicVarValues: values of the basic variables as functions of the nonbasic variables

ListOfBasicVars: list of basic variables

ListOfNonbasicVars: list of non basic variables

GenerateDualModel

Call:

```
GenerateDualModel[Z, Constraints, Var,  Options]
```

with

Z: objective function

Constraints: list of constraints, it expects the decision variables on the lhs of the relational operator, and constant additive terms on the rhs.

Var: list of variables, it has the structure {VarPos,VarNeg,VarNs}, with VarPos the list of variables with $x \geq 0$, VarNeg for variables with $x < 0$ and VarNs for variables without sign restrictions.

Options:

Option	Value	Description
NewName	y (default)	Prefix string given to the new dual variables

Option for GenerateDualModel.

Output syntax:

```
{Z, Constraints, Var}
```

with

Z: objective function

Constraints: list of constraints

Var: list of variables

SensitivitySecVar

Call:

```
SensitivitySecVar[Zopt,  ReplBasicVarOpt,  VarBasicOpt,  VarSecundaryOpt,
SecundaryVar]
```

with

Zopt: optimal tableau

ReplBasicVarOpt: replacement list for the optimal values of the basic variables

VarBasicOpt: list of optimal basic variables

VarSecundaryOpt: list of secondary variables

SecundaryVar: secundary variable which should be varied

Output syntax:

```
{Z, Solution, {SecundaryVar, sv0, sv1}}
```

with

Z: objective function as function of the chosen secondary variable SecundaryVar

Solution: basic variables as function of SecundaryVar

{SecundaryVar,sv0,sv1}: range of SecundaryVar (sv0 < SecundaryVar < sv1) for which the solution remains optimal

SensitivityTargetCoef

Call:

```
SensitivityTargetCoef[Zopt, ReplBasicVarOpt, VarBasicOpt, VarSecundary-
Opt, Var]
```

with

Zopt: optimal tableau

ReplBasicVarOpt: replacement list for the optimal values of the basic variables

VarBasicOpt: list of optimal basic variables

VarSecundaryOpt: list of secondary variables

Var: variable the coefficient of that in the objective function should be varied

Output syntax:

```
{Z, {△Coeff, cf0, cf1}}
```

with

Z: objective function as function of the change of the corresponding coefficient

{ΔCoeff,cf0,cf1}: range of coefficient (cf0 < ΔCoeff < cf1) for which the solution remains optimal

Example: production program - maximal profit under limited ressources

We consider a problem of production planning. Three different products are being produced, using a certain amount of time on two different machines. The coefficients of the target function represent the profit margins per unit of the correspondung product.

```
Z = 400 x1 + 100 x2 + 100 x3;
(* → Max *)
```

For the task, machine 1 may be available 10 hours per day , and machine 2 12 hours. Product 1 demands 1 hour of machine 1 and 1.5 hours of machine 2. Similar data apply to the other products. The data are summarized by the constraints

$$\text{lconstr} = \left\{ x1 + 2\,x2 + x3 \le 10,\ \frac{3}{2}\,x1 + 2\,x2 + \frac{1}{2}\,x3 \le 12 \right\};$$

The lot sizes are denoted by x1,x2 and x3 and should obviously be greater or equal zero

```
lvar = {x1, x2, x3};
llvar = {lvar, {}, {}};
```

Because of the <= constraints a feasible starting tableau is immediately generated by the introduction of slack variables u1 and u2 for the two machines. They represent the amount of machine time not used by the task. If a slack variable appears to be nonbasic (i.e. has value zero) the corresponding machine represents a bottleneck.

```
res = Simplex[Z, lconstr, llvar, newname -> "u"];
```

list of variables {x1, x2, x3, u1, u2}

system of equations $\begin{pmatrix} u1 + x1 + 2\,x2 + x3 == 10 \\ u2 + \frac{3\,x1}{2} + 2\,x2 + \frac{x3}{2} == 12 \end{pmatrix}$

first feasible tableau

$$\begin{pmatrix} 1 & 2 & 1 & 1 & 0 & 10 \\ \frac{3}{2} & 2 & \frac{1}{2} & 0 & 1 & 12 \\ -400 & -100 & -100 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & \frac{2}{3} & \frac{2}{3} & 1 & -\frac{2}{3} & 2 \\ 1 & \frac{4}{3} & \frac{1}{3} & 0 & \frac{2}{3} & 8 \\ 0 & \frac{1300}{3} & \frac{100}{3} & 0 & \frac{800}{3} & 3200 \end{pmatrix}$$

optimal tableau found

```
res
```

$$\left\{3200,\ \{x1 \to 8,\ x2 \to 0,\ x3 \to 0\},\ \left\{3200 - \frac{800\,u2}{3} - \frac{1300\,x2}{3} - \frac{100\,x3}{3},\right.\right.$$
$$\left.\left.\left\{u1 \to 2 + \frac{2\,u2}{3} - \frac{2\,x2}{3} - \frac{2\,x3}{3},\ x1 \to 8 - \frac{2\,u2}{3} - \frac{4\,x2}{3} - \frac{x3}{3}\right\},\ \{u1,\ x1\},\ \{x2,\ x3,\ u2\}\right\}\right\}$$

In this particular case, 8 units of product 1 are produced. u1 belongs to the basic variables. Only 8 hours of machine 8 are required, out of the 10 availble. u2 is nonbasic, i.e. machine 2 represents a bottleneck.

The optimal tableau provides more information than just the optimal production program. Shadow prices of bottleneck ressources are given by the coefficient of the target function with respect to the corresponding slack variable. For machine 2, the corresponding shadow price is read off as 800/3. Machine 1 represents no bottleneck. Accordingly, he corresponding shadow price is zero.

Shadow prices may also be obtained from the solution of the corresponding dual problem.

```
dualmodel = GenerateDualModel[Z, lconstr, llvar, newname → "y"]
```

$$\left\{10\,y1 + 12\,y2,\ \left\{y1 + \frac{3\,y2}{2} \geq 400,\ 2\,y1 + 2\,y2 \geq 100,\ y1 + \frac{y2}{2} \geq 100\right\},\ \{\{y1,\ y2\},\ \{\},\ \{\}\}\right\}$$

The dual target function should be minimzed. Because of the >= constraints a first feasible tableau is generated by temporarily introducing additional auxiliary variables in addition to the slack variable v (simplex step 1).

```
resdual = Simplex[dualmodel[[1]],
   dualmodel[[2]], dualmodel[[3]], newname → "v", Maximize → False];
```

list of variables $\{y1, y2, u1, u2, u3\}$

system of equations $\begin{pmatrix} -u1 + y1 + \frac{3\,y2}{2} == 400 \\ -u2 + 2\,y1 + 2\,y2 == 100 \\ -u3 + y1 + \frac{y2}{2} == 100 \end{pmatrix}$

Identity Matrix incomplete

Simplex Step 1 initialized

starting tableau Step 1

$$\begin{pmatrix} 1 & \frac{3}{2} & -1 & 0 & 0 & 1 & 0 & 0 & 400 \\ 2 & 2 & 0 & -1 & 0 & 0 & 1 & 0 & 100 \\ 1 & \frac{1}{2} & 0 & 0 & -1 & 0 & 0 & 1 & 100 \\ -4 & -4 & 1 & 1 & 1 & 0 & 0 & 0 & -600 \end{pmatrix}$$

$$\begin{pmatrix} 0 & \frac{1}{2} & -1 & \frac{1}{2} & 0 & 1 & -\frac{1}{2} & 0 & 350 \\ 1 & 1 & 0 & -\frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 & 50 \\ 0 & -\frac{1}{2} & 0 & \frac{1}{2} & -1 & 0 & -\frac{1}{2} & 1 & 50 \\ 0 & 0 & 1 & -1 & 1 & 0 & 2 & 0 & -400 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & -1 & 0 & 1 & 1 & 0 & -1 & 300 \\ 1 & \frac{1}{2} & 0 & 0 & -1 & 0 & 0 & 1 & 100 \\ 0 & -1 & 0 & 1 & -2 & 0 & -1 & 2 & 100 \\ 0 & -1 & 1 & 0 & -1 & 0 & 1 & 2 & -300 \end{pmatrix}$$

$$\begin{pmatrix} -2 & 0 & -1 & 0 & 3 & 1 & 0 & -3 & 100 \\ 2 & 1 & 0 & 0 & -2 & 0 & 0 & 2 & 200 \\ 2 & 0 & 0 & 1 & -4 & 0 & -1 & 4 & 300 \\ 2 & 0 & 1 & 0 & -3 & 0 & 1 & 4 & -100 \end{pmatrix}$$

$$\begin{pmatrix} -\frac{2}{3} & 0 & -\frac{1}{3} & 0 & 1 & \frac{1}{3} & 0 & -1 & \frac{100}{3} \\ \frac{2}{3} & 1 & -\frac{2}{3} & 0 & 0 & \frac{2}{3} & 0 & 0 & \frac{800}{3} \\ -\frac{2}{3} & 0 & -\frac{4}{3} & 1 & 0 & \frac{4}{3} & -1 & 0 & \frac{1300}{3} \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Step 1 : optimal plateau found

Simplex Step 2 initialized

first feasible plateau

$$\begin{pmatrix} -\frac{2}{3} & 0 & -\frac{1}{3} & 0 & 1 & \frac{100}{3} \\ \frac{2}{3} & 1 & -\frac{2}{3} & 0 & 0 & \frac{800}{3} \\ -\frac{2}{3} & 0 & -\frac{4}{3} & 1 & 0 & \frac{1300}{3} \\ 2 & 0 & 8 & 0 & 0 & -3200 \end{pmatrix}$$

optimal tableau found

resdual

$$\left\{3200, \left\{y1 \to 0, y2 \to \frac{800}{3}\right\}, \left\{3200 + 8\,u1 + 2\,y1, \left\{u3 \to \frac{100}{3} + \frac{u1}{3} + \frac{2\,y1}{3}, y2 \to \frac{800}{3} + \frac{2\,u1}{3} - \frac{2\,y1}{3}, u2 \to \frac{1300}{3} + \frac{4\,u1}{3} + \frac{2\,y1}{3}\right\}, \{u3, y2, u2\}, \{y1, u1\}\right\}\right\}$$

The shadow price for machine 2 now occurs as the optimal value of y2, the dual variable related to the constraint representing machine two.

The dual solution shows that constraint 2 of the primal problem is satiesfied as equation (u2=0), whereas u1 is basic. This information would be sufficient for reconstructing the primal solution.

Sensitivity analysis

The optimal response to small variations of external parameters or requirements may by derived from the optimal tableau without restarting the simplex calculation.

```
Zopt = res[[3, 1]]
replbasvaropt = res[[3, 2]]
lvarbasopt = res[[3, 3]]
lvarsecopt = res[[3, 4]]
```

$$3200 - \frac{800\,u2}{3} - \frac{1300\,x2}{3} - \frac{100\,x3}{3}$$

$$\left\{u1 \to 2 + \frac{2\,u2}{3} - \frac{2\,x2}{3} - \frac{2\,x3}{3},\ x1 \to 8 - \frac{2\,u2}{3} - \frac{4\,x2}{3} - \frac{x3}{3}\right\}$$

$$\{u1,\ x1\}$$

$$\{x2,\ x3,\ u2\}$$

Consider e.g. the possibility of making more (or less) time available on machine two. This can be modelled by a negative (or positive) shift of the slack variable u2.

```
SensitivitySecVar[Zopt, replbasvaropt, lvarbasopt, lvarsecopt, u2]
```

$$\left\{3200 - \frac{800\,u2}{3},\ \left\{u1 \to 2 + \frac{2\,u2}{3},\ x1 \to 8 - \frac{2\,u2}{3}\right\},\ \{u2,\ -3,\ 12\}\right\}$$

Up to three more hours can be made available for machine 2 without leaving the optimality range of the current basic solution. After this point, machine one becomes the bottleneck.
If less time is provided for machine 2, the tableau remains optimal all the way down to no machine time at all (u2 = 12). At this point, machine one is of course also not being used (u1→ 10).

Similarily, one could consider the case that for marketing reasons some units of product 3 must be produced. This could be handled by redoing the calculation with an additional constraint. The consequences of the enforced production of x3 units of product three can also be seen by shifting x3 in the optimal tableau, however.

```
SensitivitySecVar[Zopt, replbasvaropt, lvarbasopt, lvarsecopt, x3]
```

$$\left\{3200 - \frac{100\,x3}{3},\ \left\{u1 \to 2 - \frac{2\,x3}{3},\ x1 \to 8 - \frac{x3}{3}\right\},\ \{x3,\ -\infty,\ 3\}\right\}$$

The tableau remains optimal as long as no more than three units of product three will be produced. After this point, machine one would become the bottleneck.
Technically, the tableau would also remain optimal if x3 is reduced below zero. This is economically meaningless, of course.

Up to now, we have only considered external variations affecting the constraints. The target function could also be modified, however. E.g., the profit margins could vary because of market price variations. As long as these external variations do not become too large, they can be describes using the optimal tableau.

Let us first consider consider an increase or decrease of the profit margin of product one, by the amount del.

```
SensitivityTargetCoef[Zopt, replbasvaropt, lvarbasopt, lvarsecopt, x1]
```

$$\{3200 + 8\,del,\ \{delcofx1,\ -100,\ \infty\}\}$$

An increase of this profit margin (del >0) will boost the profit by 8 del (because 8 units of product one are being produced). A small decrease will not affect the optimal production program either. As soon as del=-100 is passed, however, it becomes more profitable producing also some units of product three. This can be seen directly by expanding Z + del x1 in the optimal base solution.

```
Expand[Zopt + del * x1 /. replbasvaropt]
```

$$3200 + 8\,\text{del} - \frac{800\,u2}{3} - \frac{2\,\text{del}\,u2}{3} - \frac{1300\,x2}{3} - \frac{4\,\text{del}\,x2}{3} - \frac{100\,x3}{3} - \frac{\text{del}\,x3}{3}$$

Similar efects occur if the profit margin changes for products currently not being produced (non-basic variables). Consider e.g. product 3.

```
SensitivityTargetCoef[Zopt, replbasvaropt, lvarbasopt, lvarsecopt, x3]
```

$$\left\{3200, \left\{\text{delcofx3}, -\infty, \frac{100}{3}\right\}\right\}$$

The optimal programm remains unaffected as long as the profit margin for product three does nit increase by more than $\frac{100}{3}$.

Example: blends at minimal costs

The previous example covered less-equal constraints only. In this case, a first feasible tableau can be written down immediately. In general, this is not the case. As a second example we consider the problem of composing a diet from certain raw materials (e.g. fruits, or meat) whereby restrictions are set for the content of specified ingrediants like vitamins, or fat. Greater-equal constraints (vitamins), as well as less-equal ones (fat) may now be appropriate.
The total cost should be minimal.

A particular example with 3 ingrediants could be

```
Z = 9/10 x1 + 14/10 x2 + 12/10 x3;
(* → Min  *)
```

```
lconstr =
  {3 x1 + 11 x2 + 10 x3 ≥ 9, 6 x1 + 2 x2 + 2 x3 <= 5, 150 x1 + 220 x2 + 120 x3 ≥ 194};
```

```
lvar = {x1, x2, x3};
llvar = {lvar, {}, {}};
```

Consideruing the following output please note that internally - Z is maximized.

```
res = Simplex[Z, lconstr, llvar, Maximize → False];
```

list of variables $\{x1, x2, x3, u1, u2, u3\}$

system of equations $\begin{pmatrix} -u1 + 3\,x1 + 11\,x2 + 10\,x3 == 9 \\ u2 + 6\,x1 + 2\,x2 + 2\,x3 == 5 \\ -u3 + 150\,x1 + 220\,x2 + 120\,x3 == 194 \end{pmatrix}$

Identity Matrix incomplete

Simplex Step 1 initialized

starting tableau Step 1

$$\begin{pmatrix} 3 & 11 & 10 & -1 & 0 & 0 & 1 & 0 & 9 \\ 6 & 2 & 2 & 0 & 1 & 0 & 0 & 0 & 5 \\ 150 & 220 & 120 & 0 & 0 & -1 & 0 & 1 & 194 \\ -153 & -231 & -130 & 1 & 0 & 1 & 0 & 0 & -203 \end{pmatrix}$$

$$\begin{pmatrix} \frac{3}{11} & 1 & \frac{10}{11} & -\frac{1}{11} & 0 & 0 & \frac{1}{11} & 0 & \frac{9}{11} \\ \frac{60}{11} & 0 & \frac{2}{11} & \frac{2}{11} & 1 & 0 & -\frac{2}{11} & 0 & \frac{37}{11} \\ 90 & 0 & -80 & 20 & 0 & -1 & -20 & 1 & 14 \\ -90 & 0 & 80 & -20 & 0 & 1 & 21 & 0 & -14 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & \frac{38}{33} & -\frac{5}{33} & 0 & \frac{1}{330} & \frac{5}{33} & -\frac{1}{330} & \frac{128}{165} \\ 0 & 0 & \frac{166}{33} & -\frac{34}{33} & 1 & \frac{2}{33} & \frac{34}{33} & -\frac{2}{33} & \frac{83}{33} \\ 1 & 0 & -\frac{8}{9} & \frac{2}{9} & 0 & -\frac{1}{90} & -\frac{2}{9} & \frac{1}{90} & \frac{7}{45} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Step 1 : optimal plateau found

Simplex Step 2 initialized

first feasible plateau

$$\begin{pmatrix} 0 & 1 & \frac{38}{33} & -\frac{5}{33} & 0 & \frac{1}{330} & \frac{128}{165} \\ 0 & 0 & \frac{166}{33} & -\frac{34}{33} & 1 & \frac{2}{33} & \frac{83}{33} \\ 1 & 0 & -\frac{8}{9} & \frac{2}{9} & 0 & -\frac{1}{90} & \frac{7}{45} \\ 0 & 0 & \frac{64}{165} & \frac{2}{165} & 0 & \frac{19}{3300} & -\frac{2023}{1650} \end{pmatrix}$$

optimal tableau found

res

$$\left\{\frac{2023}{1650}, \left\{x1 \to \frac{7}{45}, x2 \to \frac{128}{165}, x3 \to 0\right\},\right.$$
$$\left\{\frac{2023}{1650} + \frac{2\,u1}{165} + \frac{19\,u3}{3300} + \frac{64\,x3}{165}, \left\{x2 \to \frac{128}{165} + \frac{5\,u1}{33} - \frac{u3}{330} - \frac{38\,x3}{33},\right.\right.$$
$$\left.\left.\left.u2 \to \frac{83}{33} + \frac{34\,u1}{33} - \frac{2\,u3}{33} - \frac{166\,x3}{33}, x1 \to \frac{7}{45} - \frac{2\,u1}{9} + \frac{u3}{90} + \frac{8\,x3}{9}\right\}, \{x2, u2, x1\}, \{x3, u1, u3\}\right\}\right\}$$

For the cost - minimal solution, raw material 3 is not used, and the upper limit for ingrediant 2 (fat) is not fully utilized.

Sensitivity analysis

Varying constraints may now arise because of changing guidelines and recommendations. The optimal response to small variations may again by derived from the optimal tableau, without restarting the simplex calculation.

Consider e.g. a loosening or tightening of the lower bound for ingrediant 1. This can be modelled by varying u1. A positive u1 would correspond to a tightening of the bound, and vica versa.

```
Zopt = res[[3, 1]]
replbasvaropt = res[[3, 2]]
lvarbasopt = res[[3, 3]]
lvarsecopt = res[[3, 4]]
```

$$\frac{2023}{1650} + \frac{2\,u1}{165} + \frac{19\,u3}{3300} + \frac{64\,x3}{165}$$

$$\left\{x2 \to \frac{128}{165} + \frac{5\,u1}{33} - \frac{u3}{330} - \frac{38\,x3}{33},\ u2 \to \frac{83}{33} + \frac{34\,u1}{33} - \frac{2\,u3}{33} - \frac{166\,x3}{33},\ x1 \to \frac{7}{45} - \frac{2\,u1}{9} + \frac{u3}{90} + \frac{8\,x3}{9}\right\}$$

{x2, u2, x1}

{x3, u1, u3}

```
SensitivitySecVar[Zopt, replbasvaropt, lvarbasopt, lvarsecopt, u1]
```

$$\left\{\frac{2023}{1650} + \frac{2\,u1}{165},\ \left\{x2 \to \frac{128}{165} + \frac{5\,u1}{33},\ u2 \to \frac{83}{33} + \frac{34\,u1}{33},\ x1 \to \frac{7}{45} - \frac{2\,u1}{9}\right\},\ \left\{u1,\ -\frac{83}{34},\ \frac{7}{10}\right\}\right\}$$

Tightening of the first bound obviosly increases the cost. The optimal tableau remains valid up to u1= $\frac{7}{10}$ where x1 becomes zero, and raw material one would be replaced by raw material three.

If on the other hand the bound on ingrediant one is loosened, the total cost is decreasing. At u1=$-\frac{83}{34}$ u2 becomes zero, i.e. the constraint for ingrediant 2 would start to be violated. Further loosening of the constraint for ingrediant one would be ineffective.

One may also ask the question on consequences of price changes of raw materials. Consider e.g. raw material one.

```
SensitivityTargetCoef[Zopt, replbasvaropt, lvarbasopt, lvarsecopt, x1]
```

$$\left\{\frac{2023}{1650} + \frac{7\,del}{45},\ \left\{delcofx1,\ -\frac{24}{55},\ \frac{3}{55}\right\}\right\}$$

A price increase by del > $\frac{3}{55}$ has the consequence that raw material one is no longer being used but only raw material two is being adopted.

A price decrease going beyond del = - $\frac{24}{55}$ leads to a different basic solution where all three raw materials are being used, and all restrictions are satisfied as equalities.

In order to draw these conclusions one has to perform one more simplex step. The sensitivity interval $(-\frac{24}{55}, \frac{5}{55})$ only tells us the range of cost changes where the original basic solution remains the optimal one.

Goal Programming

Introduction

Real-world problems typically involve multiple objectives. Some of them may be formulated in terms of linear objective functions. Different decisions, i.e. choices of the decision variables will lead to better values for some objectives, but are worse with respective to other ones. Obviously, this causes some ambiguity for the decisicion process. In any case, it will be strongly influenced by preferences of the decision maker.

A number of ways of aggregating various objectives into a single objective function have been proposed. They will not be discussed in detail here. We just provide one particular implementation of a method called goal programming. The idea behind is as follows. Decision makers should have a more or less clear idea which results should be achieved with respect to various objectives. The correspnding target values could be implemented as constraints. In this case, a feasable solution will only be found if these objective levels can actually all be reached. On the other hand, one could also consider these target values as goals. This opens the possibility of setting target values some of which may perhaps be not completely in reach. In this case, the deviation z_i of the corresponding objective function Z_i from the goal value $goal_i$ may be considered as a positive slack variable. For each goal a new constraint is added where $sign_i$ is +1 if Z_i is to be maximized, and -1 otherwise.

$$Z_i + sign_i\, z_i = goal_i,\ z_i \geq 0$$

If $goal_i$ can be reached, z_i will turn out to be zero. Otherwise the deviation z_i will be positive.

The question remains what could be understood as optimal solution. A useful idea is the minimax principle which requires the largest of the deviations from the goals to be minimal

$$zz = \max[\{z_1, z_2, ...\}] \rightarrow \min$$

This leads to a linear approximation problem if an additional constraint z_i <= zz is introduced for each goal, and zz ist used as aggregated objective

function to be minimized.

Functions - call and output syntax

GoalProgramming

Call:

```
GoalProgramming[Zgoal, Constraints, Var, Options]
```

with

Zgoal: list of {{Z_1,$goal_1$, $sign_1$},...}

Constraints: list of constraints, it expects the decision variables on the lhs of the relational operator, and constant additive terms on the rhs.

Var: list of variables, it has the structure {VarPos,VarNeg,VarNs}, with VarPos the list of variables with $x \geq 0$, VarNeg for variables with $x < 0$ and VarNs for variables without sign restrictions.

Options:

Option	Value	Description
NewName	z (default)	Prefix string given to the goal deviations

Option for GoalProgramming.

Output:

{ZRes, VarRes}

with

ZRes: optimal value of objective function

VarRes: values for the variables

Example

The following example can be considered as food production planning, with limited ressources (labour, and area of cultivable land). Two products (e.g. vegetables and corn) could be grown, with different content of carbohydrate and protein. In some sense, both the content of carbohydrate and corn should be maximal. This leads to the objective functions

```
Z1 = 12 x1 + 6 x2;
Z2 = 4 x1 + 6 x2;
```

with x1 and x2 the number of units of grown products. The constraints due to the limited ressources are asumed to be

```
lconstr = {x1 + x2 ≤ 300, x1 + 4 x2 ≤ 800};
llvar = {{x1, x2}, {}, {}};
```

The procedure now depends on the preferences of the decision makers. If content one is considered to be far more important one would reduce the problem to maximizing Z1

```
res1 = Simplex[Z1, lconstr, llvar, newname → "u", PrintOpt → False];
```

```
res1[[1]]
res1[[2]]
```

```
3600
```

```
{x1 → 300, x2 → 0}
```

Only pruduct one would be produced. For Z2, this implies

```
Z2 /. res1[[2]]
```

```
1200
```

If on the other hand content 2 is considered as more important one obtains

```
res2 = Simplex[Z2, lconstr, llvar, newname → "u", PrintOpt → False];
```

```
res2[[1]]
res2[[2]]
```

$$\frac{4600}{3}$$

$$\left\{x1 \to \frac{400}{3},\ x2 \to \frac{500}{3}\right\}$$

Goal Programming now requires a decision on "desirable" goals for both carbohydrate and protein. This could e.g. mean setting goals at 90 per cent of the respective objective values obtained disregarding the other substance of content.

```
lZgoal = {{Z1, res1[[1]] * 9/10, 1}, {Z2, res2[[1]] * 9/10, 1}};
```

```
resgoal = GoalProgramming[lZgoal, lconstr, llvar, NewName -> "z"]
```

$$\left\{45,\ \left\{x1 \to \frac{465}{2},\ x2 \to \frac{135}{2},\ z1 \to 45,\ z2 \to 45,\ zz \to 45\right\}\right\}$$

These goals cannot be achieved completely. The deviation is 45 in both cases. The reachable objective values are

```
Z1 /. resgoal[[2]]
Z2 /. resgoal[[2]]
```

```
3195
```

```
1335
```

Graph Algorithms

Dijkstra

Introduction

A large class of problems requires the knowledge of the shortest path on a two dimensional network. The Operations Research package provides a solution based on dynamic programming and on software agent methods. The user can choose between two algorithms

SPDijkstra : giving the network and one starting node, the program calculates the shortest path from this node to all other nodes
RevSPDijkstra: giving the network one starting node, the programm calculates the shortest path from all nodes to the starting node

There are two supporting routines which have to be called in order to prepare the main computation or to bring the results into a simple form:

Links2Actions : calculates the list of actions based on the list of nodes and the corresponding list of links
Links2RevActions: calculates the list of reversed actions based on the list of nodes and the corresponding list of links

For a better output there are four help routines

SPDijkstraOptPath : provides the ordered list of nodes of the optimal path obtained from SPDijkstra
RevSPDijkstraOptPath: provides the ordered list of nodes of the optimal path obtained from RevSPDijkstra

VD : provides the length of the shortest path obtained from SPDijkstra
VRD: provides the length of the shortest path obtained from RevSPDijkstra

At this point we should define the meaning of action:

An action is always assigned to a node. It is a list of tuples {node_i, linklength_i} from the node under consideration to all directly accessable nodes with the corresponding link length. A reversed action is a list of tuples {node_i, linklength_i} where node_i is a node that can reach the node under consideration

The network is defined by two files: a configuration file and a topology file. The configuration file must have a definite syntax. Each row has three numbers: the number of the outgoing node, the number of the incoming node and the length of the link between them. This means that this syntax requires directed links.

```
1     2     12
5     1     20
6     5     18
5     6     18
2     6     32
... ... ... ... ... ... ... ... ... ... ... .. etc
```

Configuration file example (part)

For n nodes the topology file has n rows with three numbers: the number of the node, the x-coordinate and the y-coordinate of the node in the x-y-plane:

```
1     0.222     1.222
2     3.344     20.234
3     5.222     18.001
4     11.2      19.34
5     6.44      32.334
... ... ... ... ... ... ... ... ... ... ... .. etc
```

Topology file example (part)

The network defined by the configuration file must not exhibit any negative dicycles.

All nodes should be given as positive numbers in a consecutive sequence.

Functions - call and output syntax

SPDijkstra

Call:

`SPDijkstra[actionlist, start]`

with

actionlist:	list of actions as obtained from Links2Actions
start:	starting node

Output syntax:

`{v, d}`

with

v_{ij}	length of shortest path from node $_i$ to node $_j$ ($v_{ii} = 0$), node $_i$is the fixed start node
d_{ij}	node preceding j in the optimal path from node$_i$ to node$_j$ ($d_{ii} = 0$), node $_i$is the fixed start node

RevSPDijkstra

Call:

`RevSPDijkstra[actionlist, start]`

with

actionlist:	list of actions as obtained from Links2RevActions
start:	starting node

Output syntax:

`{v, d}`

with

v_{ij}	length of shortest path from node $_i$ to node $_j$ ($v_{ii} = 0$), node $_i$is the fixed start node

d_{ij} node preceding j in the optimal path from $node_i$ to $node_j$ ($d_{ii} = 0$), $node_i$ is the fixed start node

Links2Actions

Call:

```
Links2Actions[nodelist, linklist]
```

with

nodelist: list of nodes in the x-y-plane of the form { ..., {$node_i$, x_i, y_i}, ...}

linklist: list of links of the form { ..., {$node_i$, $node_j$, action from nod_i to $node_j$}, ...}

Output syntax:

```
ListOfActions
```

with ListOfAction being a nested list, where at the i_{th}position all nodes with their corresonding actions are listed which can be reached from $node_i$.

Links2RevActions

Call:

```
Links2RevActions[nodelist, linklist]
```

with

nodelist: list of nodes in the x-y-plane of the form { ..., {$node_i$, x_i, y_i}, ...}

linklist: list of links of the form { ..., {$node_i$, $node_j$, action from nod_i to $node_j$}, ...}

Output syntax:

```
ListOfActions
```

with ListOfAction being a nested list, where at the i_{th}position all nodes with their corresonding actions are listed which end on $node_i$.

SPDijkstraOptPath

Call:

```
SPDijkstraOptPath[d, startnode, finalnode]
```

with

d matrix of connected nodes in the shortest path as obtained from SPDijkstra

startnode: starting node

finalnode: final node

Output syntax:

```
ListOfNodes
```

with ListOfNodes being the ordered list nodes connecting startnode with finalnode.

RevSPDijkstraOptPath

Call:

```
RevSPDijkstraOptPath[d, startnode, finalnode]
```

with

d matrix of connected nodes in the shortest path as obtained from RevSPDijkstra

startnode: starting node

finalnode: final node

Output syntax:

```
ListOfNodes
```

with ListOfNodes being the ordered list nodes connecting startnode with finalnode.

VD

Call:

VD[v]

with

v	matrix of shortest paths obtained from SPDijkstra

Output syntax:

ShortesPath

with ShortestPath being the length of the shortest path obtained from SPDijkstra.

VRD

Call:

VRD[v]

with

v	matrix of shortest paths obtained from RevSPDijkstra

Output syntax:

ShortesPath

with ShortestPath being the length of the shortest path obtained from RevSPDijkstra.

Example

In this section we introduce the two methods based on a small example *littelville*:

Starting configuration

First, we show the contents of the configuration file *littelville.net*. It stores the directed links together with the corresponding link lengths.

```
FilePrint["littleville.net"]
```

```
1 2 12
5 1 20
6 5 18
5 6 18
2 6 32
6 2 32
8 5 18
6 9 25
9 6 25
8 9 36
9 8 36
10 8 28
9 10 40
10 9 40
7 9 21
9 7 21
7 10 49
10 7 49
2 3 18
3 7 30
7 3 30
3 4 13
4 10 38
6 7 28
7 6 28
```

Column one means that there is a link from node 1 to node 2 with link length 12.
The topology file *littleville.top* contains the ten nodes with their locations in the x-y-plane.

```
FilePrint["littleville.top"];
```

```
1 1 1
2 3 1
3 5 1
4 1 3
5 3 3
6 5 3
7 1 5
8 3 5
9 5 5
10 1 7
```

Column two means that node 2 has x-y-coordinates (3,1).
We extract from the file *littleville.top* the list of nodes and the number of nodes:

```
ListOfNodes = Sort[ReadList["littleville.top", Number, RecordLists → True]];
NumberOfNodes = Length[ListOfNodes]
```

```
10
```

From the file *littleville.net* we read in the configuration of the network:

```
ListOfLinks = ReadList["littleville.net", Number, RecordLists → True]
```

```
{{1, 2, 12}, {5, 1, 20}, {6, 5, 18}, {5, 6, 18}, {2, 6, 32}, {6, 2, 32},
 {8, 5, 18}, {6, 9, 25}, {9, 6, 25}, {8, 9, 36}, {9, 8, 36}, {10, 8, 28},
 {9, 10, 40}, {10, 9, 40}, {7, 9, 21}, {9, 7, 21}, {7, 10, 49}, {10, 7, 49},
 {2, 3, 18}, {3, 7, 30}, {7, 3, 30}, {3, 4, 13}, {4, 10, 38}, {6, 7, 28}, {7, 6, 28}}
```

From *NumberOfNodes* and *ListOfLinks* we calculate the list of actions:

```
ListOfActions = Links2Actions[NumberOfNodes, ListOfLinks]
```

```
{{{2, 12}}, {{6, 32}, {3, 18}}, {{7, 30}, {4, 13}}, {{10, 38}}, {{1, 20}, {6, 18}},
 {{5, 18}, {2, 32}, {9, 25}, {7, 28}}, {{9, 21}, {10, 49}, {3, 30}, {6, 28}},
 {{5, 18}, {9, 36}}, {{6, 25}, {8, 36}, {10, 40}, {7, 21}}, {{8, 28}, {9, 40}, {7, 49}}}
```

Let us discuss this result in some detail. The action assigend to node 2 is given as

```
ListOfActions[[2]]
```

```
{{6, 32}, {3, 18}}
```

This means, that from node 2 we can directly reach node 6 (with link length 32) and node 3 (with link length 18). This can be verified by inspection of configuration file *littleville.net*.

SPDijkstra

Let us determine the optimal paths from node 1 to all nodes in the network with the Dijkstra method.

```
StartNode = 1;
Result = SPDijkstra[ListOfActions, StartNode];
VD = Result[[1]]
d = Result[[2]]
```

```
{0., 12, 30, 43, 62, 44, 60, 105, 69, 81}
```

```
{0, 1, 2, 3, 6, 2, 3, 9, 6, 4}
```

The list *VD* gives the shortest distance from node 1 to all nodes, the list d contains the optimal sequence of nodes. Let us look for the shortest path to

node 5:

```
FinalNode = 5;
ShortestPathTo5 = SPDijkstraOptPath[d, StartNode, FinalNode]
LengthOfTheOptimalPath = VD[[FinalNode]]
```

```
{1, 2, 6, 5}
```

```
62
```

A more advanced example

Starting configuration

We shall now test the performance of different methods considering a larger graph with 2401 nodes and 6907 (one-way) links. The starting configuration is given by the files *net2401.net* and *net2401.top*, respectively.

We start by reading in the graph and transforming it into the appropriate structure

```
ListOfLinks = ReadList["net2401.net", Number, RecordLists → True];
```

```
ListOfNodes = Sort[ReadList["net2401.top", Number, RecordLists → True]];
NumberOfNodes = Length[ListOfNodes]
```

```
2401
```

We internally use an enumeration of the nodes from one to number of nodes. This needs not to be the case in the original data. If we talk about e.g. node number three we refer to the third node in the ordered list *ListOfNodes*.

The link matrix is now transformed into the action format

```
ListOfActions = Links2Actions[NumberOfNodes, ListOfLinks];
```

The action assigned to node 223 is

```
ListOfActions[[223]]
```

```
{{222, 1.20738}, {271, 1.39164}, {224, 1.10095}, {272, 0.981541}}
```

This means that one can reach from node 223 the nodes 222 (with link length 1.20738), 271 (1.39164), 224 (1.10095) and 272 (0.981541).

SPDijkstra

We are looking for the optimal path between the following points (all computational times are obtained on a 450 GHz Pentium II computer with 256 MB RAM)

```
StartNode = 1;
FinalNode = NumberOfNodes;
```

We first test the method SPDijkstra (and compare the time behavior)

```
Result = SPDijkstra[ListOfActions, StartNode] // Timing;
ComputationTimeDisjkstra = Result[[1]]
VD = Result[[2, 1]];
d = Result[[2, 2]];
```

```
0.985
```

```
ShortestPath1 = SPDijkstraOptPath[d, StartNode, FinalNode]
```

```
{1, 2, 52, 101, 151, 103, 152, 202, 252, 301, 351, 352, 353, 402, 452, 501, 551, 552,
 553, 554, 603, 653, 701, 751, 801, 802, 851, 901, 853, 903, 952, 1001, 1051, 1100,
 1101, 1151, 1201, 1251, 1301, 1253, 1254, 1255, 1207, 1158, 1208, 1258, 1259, 1309,
 1358, 1408, 1458, 1507, 1508, 1558, 1559, 1608, 1657, 1707, 1756, 1806, 1855, 1905,
 1906, 1907, 1957, 2005, 2054, 2104, 2153, 2202, 2252, 2301, 2351, 2350, 2400, 2401}
```

The length of the shortest path is given by *VD[[FinalNode]]*

```
LengthOfShortestPath1 = VD[[FinalNode]]
```

```
90.5777
```

In problems like car navigation the position of the car varies while the final node remains fixed. In this case it is more appropriate to use the reverse SPDijkstra algorithm providing the optimal path from any point of the graph to the final node. This is achieved as follows. We first introduce a modified data structure where links are grouped by the second node

```
ListOfReverseActions = Links2RevActions[NumberOfNodes, ListOfLinks];
```

For example, ListOfReverseActions[[2]] covers all links ending at node 2:

```
ListOfReverseActions[[2]]
```

```
{{3, 1.14078}, {52, 1.5747}, {1, 0.756256}}
```

The result is obtained as

```
Result = RevSPDijkstra[ListOfReverseActions, FinalNode] // Timing;
ComputationTimeReverseDisjkstra = Result[[1]]
VRD = Result[[2, 1]];
d = Result[[2, 2]];
```

```
1.078
```

```
ShortestPath1 = RevSPDijkstraOptPath[d, StartNode, FinalNode]
```

```
{1, 2, 52, 101, 151, 103, 152, 202, 252, 301, 351, 352, 353, 402, 452, 501, 551, 552,
 553, 554, 603, 653, 701, 751, 801, 802, 851, 901, 853, 903, 952, 1001, 1051, 1100,
 1101, 1151, 1201, 1251, 1301, 1253, 1254, 1255, 1207, 1158, 1208, 1258, 1259, 1309,
 1358, 1408, 1458, 1507, 1508, 1558, 1559, 1608, 1657, 1707, 1756, 1806, 1855, 1905,
 1906, 1907, 1957, 2005, 2054, 2104, 2153, 2202, 2252, 2301, 2351, 2350, 2400, 2401}
```

```
VRD[[StartNode]]
```

```
90.5777
```

If someone deviates from the optimal path for one reason or another one can still evaluate the new optimal path for the current position, without any lengthy new calculation.

The situation is less favorable if the environment changes. Traffic jams may appear or disappear, or floating car data may provide changing speed informations on the whole network. The SPDijkstra method does not take into account prior knowledge like a previously calculated path under modified conditions. The calculation would have to be redone completely. In the next subsection we shall describe an approach more adapted to navigation in changing environments.

Network Flows

Introduction

Network flows are special cases of linear programming. Network flow models arise on structures called directed graphs. A directed graph is defined by

nodes - these are the intersections or transfer points of the network. They are charaterized by so called net demands: negative numbers for sink (or demand) nodes, and positive numbers for source (or supply) nodes

arcs - these quantities join the nodes of the network. They are characterized by the flows , their direction and the possible capacities (upper bounds on the flows). Moreover, to each arc a cost is assigned.

At every node conservation of flows is enforced:

(total flow in) - (total flow out) = net demand

Moreover, the sum of all demands and supplies of the network should add to zero. If the original model violates this constraint an artificial node (sink or source) is added to the network joined by arcs with zero cost.

Network flow problems are solved with the routine NetworkFlow[arg, Options]. It calculates the optimal network flow as determined by the input arg.

The routine NetworkFlow can get as input two different types of arguments arg

List_of_elements - list of all variables needed to define the network model

File - string denoting the corresponding input files

Let us discuss both cases in some detail:

List_of_elements:

The input has to be given with the following structure:

{list_of_nodes, list_of_arcs, list_of_coordinates}

They have the form:

list_of_nodes = {{ $node_1$, $demand_1$}, { $node_2$, $demand_2$}, ... }

If $demand_i < 0$ then $node_i$is a sink otherwise it is a source.

list_of_arcs = {{ $node_i$, $node_j$, $cost_{ij}$, $bound_{ij}$}, ... }

The $bound_{ij}$- quantities (> 0) are optional. If no bound is specified the corresponding link is assumed to be unbounded corresponding to their capacity.

list_of_coordinates = {{ $node_i$, x_i, y_i}, ... }

This list is optional. It must be given, if the network flow should be represented graphically.

Functions - call and output syntax

Example

Let us demonstrate the use of this input structure by a small example. The input variables are given by

```
arcs = {{1, 2, 2}, {1, 3, 3}, {2, 3, 4}, {3, 4, 1}, {2, 4, 5}};
nodes = {{1, 16}, {2, 5}, {3, -7}, {4, -8}};
coordinates = {{1, 0, 1}, {2, 1, 2}, {3, 1, 0}, {4, 2, 1}};
```

The solution is obtained with the call

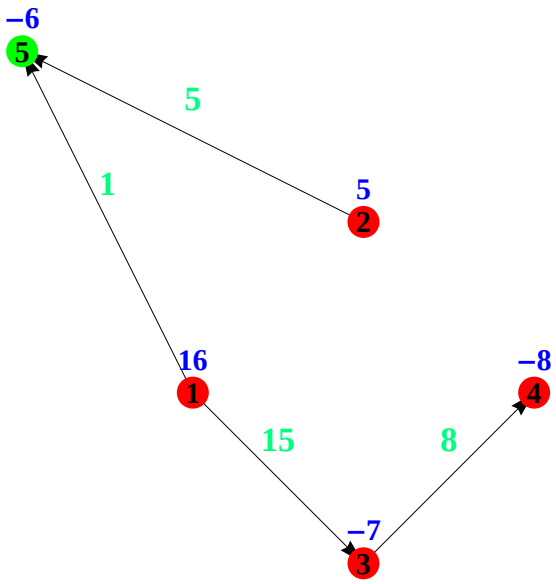
```
res = NetworkFlow[{nodes, arcs, coordinates}]
```

```
{53, {{1, 2, 0}, {1, 3, 15}, {2, 3, 0}, {3, 4, 8}, {2, 4, 0}, {1, 5, 1}, {2, 5, 5}}}
```

```
res = NetworkFlow[{nodes, arcs, coordinates}, ShowNetwork → True];
res[[1]]
res[[2]]
Show[Graphics[res[[3]]], AspectRatio → Automatic]
```

```
53
```

```
{{1, 2, 0}, {1, 3, 15}, {2, 3, 0}, {3, 4, 8}, {2, 4, 0}, {1, 5, 1}, {2, 5, 5}}
```



Mixed Integer Problems

Branchbound

Mixed integer problems are solved with the routine `BranchBound[Z, Constraints, Var, Options]`

with

Z:	objective function
Constraints:	list of constraints
Var:	list of decision variables

The list of decision variables must be of the form Var = {ListOfReals, ListOfIntegers, ListOfBinaries}!

BranchBound is controlled by the following options:

Option	Value	Description
Maximize	True (default)	Maximizes the objective function Z
	False	Minimizes the objective function Z
SaveDegenerate	False (default)	Returns always the last integer solution found
	True	Returns all optimal integer solutions found
Method	BestValue (default)	Uses the best value strategy for traversing the tree
	DepthFirst	Uses the depth first strategy for traversing the tree
	Mixed	Uses first DepthFirst until it finds a first solution, then it continues with BestValue
ModPrint	100 (default)	Defines the number of steps after which intermediate results are printed to screen
DeltaZ0	0 (default)	Determines the minimal required improvement from a previously known integer solution in order to be accepted as new integer solution
Z0start	-10^9 (default)	Starting value for bound on objective function either from feasible solutions already known, or from other insight into the problem

Options for BranchBound.

The default method *"BestValue"* selects, out of all active nodes of the branching tree, the node with the best actual value for the objective function Z, and performes the branching on that node. *"DepthFirst"* runs along one branch of the tree up to the end determined either by an infeasible or an integer solution.

The decision variables *var* in the argument list of *BranchBound* must have the form
var = {ListOfReals, ListOfIntegers, ListOfBinaries}

The default value for *Z0Start* is given the correct sign for minimizing or maximizing the objective function Z internally.

Let us consider a simple example taken from personnel planning. Office counters have to be occupied round the clock, with a minimum number of staff for each two-hour period (r.h.s. of the constraints). A shift may start every second hour and lasts 8 hours. xi are the number of staff per shift. Payment (coefficients of the cost function Z) varies depending on the amount of night work involved.
First of all, decision variables are defined:

```
IntegerVar = {x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12};
RealVar = {};
BinaryVar = {};
DecisionVariables = {RealVar, IntegerVar, BinaryVar};
```

The objective function Z is defined through the coefficient list c

```
c = {9.2, 8.8, 8.8, 8.4, 8., 8., 8.4, 8.8, 8.8, 9.2, 9.6, 9.6};
Z = c.IntegerVar
```

```
9.2 x1 + 9.2 x10 + 9.6 x11 + 9.6 x12 + 8.8 x2 + 8.8 x3 + 8.4 x4 + 8. x5 + 8. x6 + 8.4 x7 + 8.8 x8 + 8.8 x9
```

The list of constraints is:

```
Constraints = {x1 + x9 + x10 + x12 >= 3,
x1 + x2 + x10 + x11 >= 3,
x2 + x3 + x11 + x12 >= 8,
x1 + x3 + x4 + x12 >= 8,
x1 + x2 + x4 + x5 >= 10,
 x2 + x3 + x5 + x6 >= 10,
 x3 + x4 + x6 + x7 >= 8,
x4 + x5 + x7 + x8 >= 8,
x5 + x6 + x8 + x9 >= 14,
x6 + x7 + x9 + x10 >= 14,
x7 + x8 + x10 + x11 >= 5,
x8 + x9 + x11 + x12 >= 5};
```

There are no special bounds - all integer variables are in the range from 0 to ∞

```
Bounds1 = {};
```

In the first case the objective function Z is minimized using the *"BestValue"* method:

```
res = BranchBound[Z, Constraints, DecisionVariables, Bounds1, Maximize -> False]
```

```
Integer solution: 253.2

Integer solution: 253.2

Integer solution: 253.2

Integer solution: 253.2

Integer solution: 253.2

Integer solution: 253.2

Integer solution: 253.2

⌗ iterations:   100  ⌗ active nodes:  36  Z0:  253.2  Z1:  250.6

⌗ iterations:   200  ⌗ active nodes:  34  Z0:  253.2  Z1:  251.4
```

```
{253.2,
 {x1 → 0, x2 → 4, x3 → 1, x4 → 4, x5 → 2, x6 → 11, x7 → 3, x8 → 2, x9 → 0, x10 → 0, x11 → 0, x12 → 3}}
```

Every integer solution found is printed immediately. Additionally, every 100 steps (default value) the program prints the number of actual active nodes, the objective function *Z0* for the best integer solution found so far , the current bound for the objective function *Z1*, and the computational time for *ModPrint* steps in seconds. The range between *Z0* and *Z1* represents the region where better integer solutions could still be found.
The test for feasibility gives:

```
Map[ReplaceAll[#, res[[2]]] &, Constraints]
```

```
{True, True, True, True, True, True, True, True, True, True, True, True}
```

The control print indicates that this problem has degenerate integer solutions. We can store all of them with the option *SaveDegenerate:*

```
res = BranchBound[Z, Constraints, DecisionVariables,
  Bounds1, 1 Maximize -> False, SaveDegenerate → True]
```

```
Integer solution: 253.2

Integer solution: 253.2

Integer solution: 253.2

Integer solution: 253.2

Integer solution: 253.2

Integer solution: 253.2

Integer solution: 253.2

⌗ iterations:   100  ⌗ active nodes:  36  Z0:  253.2  Z1:  250.6

⌗ iterations:   200  ⌗ active nodes:  34  Z0:  253.2  Z1:  251.4
```

```
{{253.2, {x1 → 0, x2 → 2, x3 → 1, x4 → 4,
   x5 → 4, x6 → 11, x7 → 3, x8 → 0, x9 → 0, x10 → 0, x11 → 2, x12 → 3}},
 {253.2, {x1 → 0, x2 → 3, x3 → 1, x4 → 4, x5 → 3, x6 → 11, x7 → 3, x8 → 1, x9 → 0,
   x10 → 0, x11 → 1, x12 → 3}}, {253.2, {x1 → 0, x2 → 3, x3 → 2, x4 → 3,
   x5 → 4, x6 → 9, x7 → 4, x8 → 1, x9 → 1, x10 → 0, x11 → 0, x12 → 3}},
 {253.2, {x1 → 0, x2 → 2, x3 → 2, x4 → 3, x5 → 5, x6 → 9, x7 → 4, x8 → 0, x9 → 1,
   x10 → 0, x11 → 1, x12 → 3}}, {253.2, {x1 → 0, x2 → 2, x3 → 1, x4 → 3,
   x5 → 5, x6 → 10, x7 → 4, x8 → 0, x9 → 0, x10 → 0, x11 → 1, x12 → 4}},
 {253.2, {x1 → 0, x2 → 3, x3 → 1, x4 → 3, x5 → 4, x6 → 10, x7 → 4, x8 → 1, x9 → 0,
   x10 → 0, x11 → 0, x12 → 4}}, {253.2, {x1 → 0, x2 → 4, x3 → 1, x4 → 4,
   x5 → 2, x6 → 11, x7 → 3, x8 → 2, x9 → 0, x10 → 0, x11 → 0, x12 → 3}}}
```

The same calculation may be performed with the *"DepthFirst"* method:

```
res = BranchBound[Z, Constraints, DecisionVariables, Bounds1,
  Method → "DepthFirst", Maximize -> False, SaveDegenerate → True]
```

```
Integer solution: 254.

Integer solution: 253.6

Integer solution: 253.2

Integer solution: 253.2

Integer solution: 253.2

Integer solution: 253.2

Integer solution: 253.2

⌗ iterations: 100  ⌗ active nodes: 19  Z0: 253.2  Z1: 249.8
```

```
Integer solution: 253.2

Integer solution: 253.2

⌗ iterations: 200  ⌗ active nodes: 22  Z0: 253.2  Z1: 250.6

      {{253.2, {x1 → 0, x2 → 2, x3 → 1, x4 → 4,
          x5 → 4, x6 → 11, x7 → 3, x8 → 0, x9 → 0, x10 → 0, x11 → 2, x12 → 3}},
       {253.2, {x1 → 0, x2 → 3, x3 → 1, x4 → 4, x5 → 3, x6 → 11, x7 → 3, x8 → 1, x9 → 0,
          x10 → 0, x11 → 1, x12 → 3}}, {253.2, {x1 → 0, x2 → 3, x3 → 2, x4 → 3,
          x5 → 4, x6 → 9, x7 → 4, x8 → 1, x9 → 1, x10 → 0, x11 → 0, x12 → 3}},
       {253.2, {x1 → 0, x2 → 2, x3 → 1, x4 → 3, x5 → 5, x6 → 10, x7 → 4, x8 → 0, x9 → 0,
          x10 → 0, x11 → 1, x12 → 4}}, {253.2, {x1 → 0, x2 → 2, x3 → 2, x4 → 3,
          x5 → 5, x6 → 9, x7 → 4, x8 → 0, x9 → 1, x10 → 0, x11 → 1, x12 → 3}},
       {253.2, {x1 → 0, x2 → 3, x3 → 1, x4 → 3, x5 → 4, x6 → 10, x7 → 4, x8 → 1, x9 → 0,
          x10 → 0, x11 → 0, x12 → 4}}, {253.2, {x1 → 0, x2 → 4, x3 → 1, x4 → 4,
          x5 → 2, x6 → 11, x7 → 3, x8 → 2, x9 → 0, x10 → 0, x11 → 0, x12 → 3}}}
```

One can see that in this case the number of active nodes is somewhat smaller. The computation time is comparable. For other problems, the two branching methods may dramatically differ in their performance.

```
res = BranchBound[Z, Constraints,
  DecisionVariables, Bounds1, Method → "Mixed", Maximize -> False]
```

```
solution   {253.2, {0, 2, 1, 4, 4, 11, 3, 0, 0, 0, 2, 3}}

solution   {253.2, {0, 3, 3, 4, 3, 9, 3, 1, 2, 0, 1, 1}}

⌗ iterations  100  ⌗ active nodes  6  lower bound  251.4  upper bound  253.2

      {{253.2, {x1 → 0, x2 → 2, x3 → 1, x4 → 4,
          x5 → 4, x6 → 11, x7 → 3, x8 → 0, x9 → 0, x10 → 0, x11 → 2, x12 → 3}},
       {253.2, {x1 → 0, x2 → 4, x3 → 4, x4 → 4, x5 → 2, x6 → 8, x7 → 3, x8 → 2, x9 → 3,
          x10 → 0, x11 → 0, x12 → 0}}, {253.2, {x1 → 0, x2 → 3, x3 → 4, x4 → 4,
          x5 → 3, x6 → 8, x7 → 3, x8 → 1, x9 → 3, x10 → 0, x11 → 1, x12 → 0}},
       {253.2, {x1 → 0, x2 → 3, x3 → 1, x4 → 3, x5 → 4, x6 → 10, x7 → 4,
          x8 → 1, x9 → 0, x10 → 0, x11 → 0, x12 → 4}},
       {253.2, {x1 → 0, x2 → 4, x3 → 3, x4 → 4, x5 → 2, x6 → 9, x7 → 3, x8 → 2, x9 → 2,
          x10 → 0, x11 → 0, x12 → 1}}, {253.2, {x1 → 0, x2 → 3, x3 → 3, x4 → 4,
          x5 → 3, x6 → 9, x7 → 3, x8 → 1, x9 → 2, x10 → 0, x11 → 1, x12 → 1}}}
```

Binary Branchbound

The *BinaryBranchBound* routine solves linear problems with binary decision variables only. It is directed by the following options:

Option	Value	Description
Maximize	True (default) False	Maximizes the objective function Z Minimizes the objective function Z
NumberOfIterations	∞ (default)	Number of steps the binary branch tree can be scanned though
ModPrint	Null 100 (default)	Defines the number of steps after that actual results are printed out

Options for BinaryBranchBound.

Let us consider a simple example. First, the objective function Z is defined

```
c = {4, 5, 6, 2, 3};
Var = {x1, x2, x3, x4, x5};
Z = c.Var
```

```
4 x1 + 5 x2 + 6 x3 + 2 x4 + 3 x5
```

There are only two constraints:

```
Constraints = {-4 x1 - 2 x2 + 3 x3 - 2 x4 + x5 ≤ -1,
        -x1 - 5 x2 - 2 x3 + 2 x4 - 2 x5 ≤ -5};
```

One looks for a solution to this binary problem that minimizes Z

```
res = BinaryBranchBound[Z, Constraints, Var, Maximize → False]
```

```
{5, {x1 → 0, x2 → 1, x3 → 0, x4 → 0, x5 → 0}}
```

Again, the test of the solution for its feasibility gives

```
Map[ReplaceAll[#, res[[2]]] &, Constraints]
```

```
{True, True}
```

Examples

MPS format files

There is the possibility to read MPS format files with the *Mathematica* routine `Import` (see the Documentation Center). Its output is not quite compatible to the input required by our routine `BranchBound`. Therefore, we have written a wrapping routine `ReadMPS` which is based on `Import`. One problem of `Import` is that it cannot handle comments in MPS format files. Also, it cannot extract binary variables explicitely - they are given via their constraints. There are some MPS format file examples which cannot be read in by `Import` (and therefore by `ReadMPS`) corrrectly. For these (rare) cases the user can take the routine `ReadMPS1` which is the old `ReadMPS` from our Operations Research package version 3.x.

The ReadMPS1 routine reads a file in MPS format and transformes its contents into a form which can be used as input for the optimization routines of this package. MPS has been developed by IBM and serves as a quasi standard for commercial optimization software. Nevertheless, there are some "dialect" elements which differ from system to system. ReadMPS can handle the most commonly used macros and elements. However, there is a list of exceptions:

-ReadMPS1 cannot handle the macro RANGES

-ReadMPS1 cannot handle unrestricted constraints marked by N
(N is reserved for the objective function)

-ReadMPS1 cannot treat the macros FR and MI inside the
BOUNDS section

-ReadMPS1 transforms dots in variable names into "p",
e.g. d34.001 → d34p001

-ReadMPS1 transforms variable names represented by pure
numbers as follows: e.g. 2235 → XX2335

MPS file examples from miplib3

In this section we discuss some example problems from the miplib3 library for mixed integer problems. This library is available e.g. from http://www.caam.rice.edu/~bixby/miplib/miplib3.html and contains real world benchmark problems for pure integer and mixed integer programming. Most of the problems are customized for commercial solvers and exceed the possibility of *Mathematica* based routines to solve them in a reasonable time.

flupl.mps

This problem is rather small. Therefore, we will give all input variables explicitely.

```
res = ReadMPS1["flugpl.mps"];
```

The result is a list with seven elements. The first element contains the objective function

```
Z = res[[1]]
```

```
1500 ANM10000 + 1500 ANM20000 + 1500 ANM30000 + 1500 ANM40000 + 1500 ANM50000 + 1500 ANM60000 +
 2700 STM10000 + 2700 STM20000 + 2700 STM30000 + 2700 STM40000 + 2700 STM50000 + 2700 STM60000 +
 30 UE100000 + 30 UE200000 + 30 UE300000 + 30 UE400000 + 30 UE500000 + 30 UE600000
```

The second element contains the list of constraints:

```
Constraints = res[[2]]
```

```
{STM10000 == 60, -100 ANM10000 + 150 STM10000 + UE100000 ≥ 8000,
 -20 STM10000 + UE100000 ≤ 0., ANM10000 + 0.9 STM10000 - STM20000 == 0.,
 -100 ANM20000 + 150 STM20000 + UE200000 ≥ 9000, -20 STM20000 + UE200000 ≤ 0.,
 ANM20000 + 0.9 STM20000 - STM30000 == 0., -100 ANM30000 + 150 STM30000 + UE300000 ≥ 8000,
 -20 STM30000 + UE300000 ≤ 0., ANM30000 + 0.9 STM30000 - STM40000 == 0.,
 -100 ANM40000 + 150 STM40000 + UE400000 ≥ 10 000, -20 STM40000 + UE400000 ≤ 0.,
 ANM40000 + 0.9 STM40000 - STM50000 == 0., -100 ANM50000 + 150 STM50000 + UE500000 ≥ 9000,
 -20 STM50000 + UE500000 ≤ 0., ANM50000 + 0.9 STM50000 - STM60000 == 0.,
 -100 ANM60000 + 150 STM60000 + UE600000 ≥ 12 000, -20 STM60000 + UE600000 ≤ 0.}
```

The third element is the list of real variables:

```
RealVar = res[[3]]
```

```
{STM10000, UE100000, UE200000, UE300000, UE400000, UE500000, UE600000}
```

The fourth element is the list of integer variables:

```
IntegerVar = res[[4]]
```

```
{ANM10000, ANM20000, ANM30000, ANM40000, ANM50000,
 ANM60000, STM20000, STM30000, STM40000, STM50000, STM60000}
```

The fifth element is the list of binary variables:

```
BinaryVar = res[[5]]
```

```
{}
```

The sixth element contains the list of lower bounds:

```
LowerBounds = res[[6]]
```

```
{STM20000 ≥ 57, STM30000 ≥ 57, STM40000 ≥ 57, STM50000 ≥ 57, STM60000 ≥ 57}
```

The last element contains the list of upper bounds:

```
UpperBounds = res[[7]]
```

```
{ANM10000 ≤ 18, STM20000 ≤ 75, ANM20000 ≤ 18, STM30000 ≤ 75, ANM30000 ≤ 18, STM40000 ≤ 75,
 ANM40000 ≤ 18, STM50000 ≤ 75, ANM50000 ≤ 18, STM60000 ≤ 75, ANM60000 ≤ 18}
```

One can use these elements as the input for our *BranchBound* routine, reorganizing them in the required way

```
Var = {RealVar, IntegerVar, BinaryVar};
Bounds1 = Join[LowerBounds, UpperBounds];
```

```
result = BranchBound[Z, Constraints, Var, Bounds1, Maximize → False]
```

```
⌗ iterations:   100  ⌗ active nodes:  33  Z0:  1. × 10^9  Z1:  1.18628 × 10^6
⌗ iterations:   200  ⌗ active nodes:  67  Z0:  1. × 10^9  Z1:  1.19078 × 10^6
⌗ iterations:   300  ⌗ active nodes:  100  Z0:  1. × 10^9  Z1:  1.19285 × 10^6
⌗ iterations:   400  ⌗ active nodes:  124  Z0:  1. × 10^9  Z1:  1.19489 × 10^6
Integer solution: 1.2015 × 10^6
⌗ iterations:   500  ⌗ active nodes:  126  Z0:  1.2015 × 10^6  Z1:  1.19675 × 10^6
⌗ iterations:   600  ⌗ active nodes:  97  Z0:  1.2015 × 10^6  Z1:  1.19869 × 10^6
⌗ iterations:   700  ⌗ active nodes:  39  Z0:  1.2015 × 10^6  Z1:  1.20053 × 10^6
```

```
{1.2015 × 10^6, {ANM10000 → 6, ANM20000 → 6, ANM30000 → 16, ANM40000 → 7,
  ANM50000 → 12, ANM60000 → 0, STM20000 → 60, STM30000 → 60, STM40000 → 70,
  STM50000 → 70, STM60000 → 75, STM10000 → 60., UE100000 → 0., UE200000 → 600.,
  UE300000 → 600., UE400000 → 200., UE500000 → 0., UE600000 → 750.}}
```

egout.mps

p0033.mps

enigma.mps

Personnel Scheduling

Introduction

This class of problem arises whenever a minimum number of personnel has to be garanteed for each work shift, and the total wage should be minimized.

Model

In this example, there are 12 shifts of two hours each, and the costs vary over the day due to various supplements.

Mathematical formulation & solution

The variables x_i represent the number of workers starting their work at shift i. They all work 8 hours, i.e. 4 subsequent shifts. We define the integer decision variables:

```
IntegerVar = {x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12};
```

The objective cost function Z to be minimized is determined by costs varying over the shifts

```
Costs = {9.2, 8.8, 8.8, 8.4, 8., 8., 8.4, 8.8, 8.8, 9.2, 9.6, 9.6};
Z = Costs.IntegerVar
```

```
9.2 x1 + 9.2 x10 + 9.6 x11 + 9.6 x12 + 8.8 x2 + 8.8 x3 + 8.4 x4 + 8. x5 + 8. x6 + 8.4 x7 + 8.8 x8 + 8.8 x9
```

The constraints are determined by the minimal number of workers to be present at each shift:

```
Constraints = {x1 + x10 + x12 + x9 >= 3, x1 + x10 + x11 + x2 >= 3,
 x11 + x12 + x2 + x3 >= 8, x1 + x12 + x3 + x4 >= 8,
 x1 + x2 + x4 + x5 >= 10, x2 + x3 + x5 + x6 >= 10,
 x3 + x4 + x6 + x7 >= 8, x4 + x5 + x7 + x8 >= 8,
 x5 + x6 + x8 + x9 >= 14, x10 + x6 + x7 + x9 >= 14,
 x10 + x11 + x7 + x8 >= 5, x11 + x12 + x8 + x9 >= 5};
```

```
Bounds1 = {};
```

First, one could try to minimize disregarding the fact that the x_i should be integer.

```
Hv = Join[{Z, Constraints}]
```

```
{9.2 x1 + 9.2 x10 + 9.6 x11 + 9.6 x12 + 8.8 x2 + 8.8 x3 + 8.4 x4 + 8. x5 + 8. x6 + 8.4 x7 + 8.8 x8 + 8.8 x9,
 {x1 + x10 + x12 + x9 ≥ 3, x1 + x10 + x11 + x2 ≥ 3, x11 + x12 + x2 + x3 ≥ 8, x1 + x12 + x3 + x4 ≥ 8,
  x1 + x2 + x4 + x5 ≥ 10, x2 + x3 + x5 + x6 ≥ 10, x3 + x4 + x6 + x7 ≥ 8, x4 + x5 + x7 + x8 ≥ 8,
  x5 + x6 + x8 + x9 ≥ 14, x10 + x6 + x7 + x9 ≥ 14, x10 + x11 + x7 + x8 ≥ 5, x11 + x12 + x8 + x9 ≥ 5}}
```

```
RelaxationResult = NMinimize[Hv, IntegerVar]
```

```
{248.8, {x1 → 0., x10 → -0.75, x11 → 0., x12 → 3.75, x2 → 3.75,
  x3 → 0.5, x4 → 3.75, x5 → 2.5, x6 → 10.25, x7 → 4.5, x8 → 1.25, x9 → 0.}}
```

For this type of problem, a feasible integer solution can be generated from the LP relaxation RelaxationResult by choosing the nearest larger integers.

```
repl0 = {};
Do[
    AppendTo[repl0,
  Rule[RelaxationResult[[2, i, 1]], Ceiling[RelaxationResult[[2, i, 2]]]]],
    {i, Length[IntegerVar]}]
```

```
repl0
```

```
{x1 → 0, x10 → 0, x11 → 0, x12 → 4, x2 → 4, x3 → 1, x4 → 4, x5 → 3, x6 → 11, x7 → 5, x8 → 2, x9 → 0}
```

This leads to a cost value

```
Zupperbound = Z /. repl0
```

```
287.6
```

This will turn out to be far away from the optimal integer solution. Zupperbound could be used as upper bound for the branch-and-bound algorithm, however.

```
RealVar = {};
BinaryVar = {};
DecisionVar = {RealVar, IntegerVar, BinaryVar};
```

This time we chose a deltaZ0 ≠ 0, because Z varies in steps of 0.4 for integer solutions

```
delZ0 = 0.3999;
```

```
BestValueResult =
 BranchBound[Z, Constraints, DecisionVar, Bounds1, Maximize -> False, DeltaZ0 -> delZ0]
```

```
Integer solution: 253.2

⌗ iterations:   100  ⌗ active nodes:  36  Z0:  253.2  Z1:  250.6

⌗ iterations:   200  ⌗ active nodes:  34  Z0:  253.2  Z1:  251.4

      {253.2,
       {x1 → 0, x2 → 2, x3 → 1, x4 → 4, x5 → 4, x6 → 11, x7 → 3, x8 → 0, x9 → 0, x10 → 0, x11 → 2, x12 → 3}}
```

```
DepthFirstResult = BranchBound[Z, Constraints, DecisionVar, Bounds1,
  Method → "DepthFirst", Maximize -> False, DeltaZ0 -> delZ0, Z0Start → Zupperbound]
```

```
Integer solution: 254.

Integer solution: 253.6

Integer solution: 253.2

⌗ iterations: 100  ⌗ active nodes: 20  Z0: 253.2  Z1: 249.8

⌗ iterations: 200  ⌗ active nodes: 21  Z0: 253.2  Z1: 250.6

      {253.2,
       {x1 → 0, x2 → 2, x3 → 1, x4 → 4, x5 → 4, x6 → 11, x7 → 3, x8 → 0, x9 → 0, x10 → 0, x11 → 2, x12 → 3}}
```

There are degenerate optimal integer solutions (each method only finds one because delZ0 != 0). The optimal value of the cost function is about 10% lower than that of the feasable integer solution guessed from the LP relaxation.

Factory Planning

(Example taken from Williams)

Introduction

Planning models support decisions what to do, and when and where to do it. They are of great importance for users in government institutions as well as enterprises.

Model

A factory produces different products on various machines. Under certain constraints determined by the productivity of the machines and the market demands one has to find the best months for shutting down the various machines for maintenance.

Mathematical formulation & solution

The factory has five different machines on which seven types of products can be made. Moreover, he factory works with two shifts of 8 hours each day. A month is assumed to consist of 24 working days. Therefore, the following variables can be set

```
NumberOfProducts = 7;
NumberOfMachines = 5;
Months = 6;
HoursPerMonth = 24 * 8 * 2;
```

First, we define the number of machines (four grinders, two vertical drills, three horizontal drills, one borer, and one planer):

```
ProductionMachines = {4, 2, 3, 1, 1};
```

The factory produces seven different products. The contribution to profit from each of the products is

```
ProfitContribution = {10, 6, 8, 4, 11, 9, 3};
```

Up to 100 pieces of each product can be stored at any time, at a cost of 0,5 per unit per month. There are no stocks present at the beginning. A stock of 50 units of each type of product is required at the end of June (i.e. after 6 periods).

The required processing time for aech product (column) on each of the machine types (row) is given by

```
ProductionTime = {{0.5, 0.7, 0., 0., 0.3, 0.2, 0.5},
   {0.1, 0.2, 0., 0.3, 0., 0.6, 0.}, {0.2, 0., 0.8, 0., 0., 0., 0.6},
   {0.05, 0.03, 0., 0.07, 0.1, 0., 0.08}, {0., 0., 0.01, 0., 0.05, 0., 0.05}};
ProductionTime // MatrixForm
```

$$\begin{pmatrix} 0.5 & 0.7 & 0. & 0. & 0.3 & 0.2 & 0.5 \\ 0.1 & 0.2 & 0. & 0.3 & 0. & 0.6 & 0. \\ 0.2 & 0. & 0.8 & 0. & 0. & 0. & 0.6 \\ 0.05 & 0.03 & 0. & 0.07 & 0.1 & 0. & 0.08 \\ 0. & 0. & 0.01 & 0. & 0.05 & 0. & 0.05 \end{pmatrix}$$

Market allows to sell the following quantities of the seven products during January through June

```
Market = N[{{500,    600,    300,    200,    0,    500},
{1000,    500,    600,    300,    100,    500},
{300,    200,    0,    400,    500,    100},
{300,    0,    0,    500,    100,    300},
{800,    400,    500,    200,    1000,    1100},
{200,    300,    400,    0,    300,    500},
{100,    150,    100,    100,    0,    60}}];
Market // MatrixForm
```

$$\begin{pmatrix} 500. & 600. & 300. & 200. & 0. & 500. \\ 1000. & 500. & 600. & 300. & 100. & 500. \\ 300. & 200. & 0. & 400. & 500. & 100. \\ 300. & 0. & 0. & 500. & 100. & 300. \\ 800. & 400. & 500. & 200. & 1000. & 1100. \\ 200. & 300. & 400. & 0. & 300. & 500. \\ 100. & 150. & 100. & 100. & 0. & 60. \end{pmatrix}$$

Now, the decision variables are defined:
$Aprod_{ij}$: quantities of product i produced during month j

```
Clear[prod];
Aprod = Table[prod[i, j], {i, 1, NumberOfProducts}, {j, 1, Months}]
```

```
{{prod[1, 1], prod[1, 2], prod[1, 3], prod[1, 4], prod[1, 5], prod[1, 6]},
 {prod[2, 1], prod[2, 2], prod[2, 3], prod[2, 4], prod[2, 5], prod[2, 6]},
 {prod[3, 1], prod[3, 2], prod[3, 3], prod[3, 4], prod[3, 5], prod[3, 6]},
 {prod[4, 1], prod[4, 2], prod[4, 3], prod[4, 4], prod[4, 5], prod[4, 6]},
 {prod[5, 1], prod[5, 2], prod[5, 3], prod[5, 4], prod[5, 5], prod[5, 6]},
 {prod[6, 1], prod[6, 2], prod[6, 3], prod[6, 4], prod[6, 5], prod[6, 6]},
 {prod[7, 1], prod[7, 2], prod[7, 3], prod[7, 4], prod[7, 5], prod[7, 6]}}
```

$Astore_{ij}$: quantities of product i stored during month j

```
Astore = Table[store[i, j], {i, 1, NumberOfProducts}, {j, 1, Months}]
```

```
{{store[1, 1], store[1, 2], store[1, 3], store[1, 4], store[1, 5], store[1, 6]},
 {store[2, 1], store[2, 2], store[2, 3], store[2, 4], store[2, 5], store[2, 6]},
 {store[3, 1], store[3, 2], store[3, 3], store[3, 4], store[3, 5], store[3, 6]},
 {store[4, 1], store[4, 2], store[4, 3], store[4, 4], store[4, 5], store[4, 6]},
 {store[5, 1], store[5, 2], store[5, 3], store[5, 4], store[5, 5], store[5, 6]},
 {store[6, 1], store[6, 2], store[6, 3], store[6, 4], store[6, 5], store[6, 6]},
 {store[7, 1], store[7, 2], store[7, 3], store[7, 4], store[7, 5], store[7, 6]}}
```

$Asell_{ij}$: quantities of product i sold during month j

```
Asell = Table[sell[i, j], {i, 1, NumberOfProducts}, {j, 1, Months}]
```

```
{{sell[1, 1], sell[1, 2], sell[1, 3], sell[1, 4], sell[1, 5], sell[1, 6]},
 {sell[2, 1], sell[2, 2], sell[2, 3], sell[2, 4], sell[2, 5], sell[2, 6]},
 {sell[3, 1], sell[3, 2], sell[3, 3], sell[3, 4], sell[3, 5], sell[3, 6]},
 {sell[4, 1], sell[4, 2], sell[4, 3], sell[4, 4], sell[4, 5], sell[4, 6]},
 {sell[5, 1], sell[5, 2], sell[5, 3], sell[5, 4], sell[5, 5], sell[5, 6]},
 {sell[6, 1], sell[6, 2], sell[6, 3], sell[6, 4], sell[6, 5], sell[6, 6]},
 {sell[7, 1], sell[7, 2], sell[7, 3], sell[7, 4], sell[7, 5], sell[7, 6]}}
```

$Adown_{ij}$: number of machine i down for maintenance during month j

```
Adown = Table[down[i, j], {i, 1, NumberOfMachines}, {j, 1, Months}]
```

```
{{down[1, 1], down[1, 2], down[1, 3], down[1, 4], down[1, 5], down[1, 6]},
 {down[2, 1], down[2, 2], down[2, 3], down[2, 4], down[2, 5], down[2, 6]},
 {down[3, 1], down[3, 2], down[3, 3], down[3, 4], down[3, 5], down[3, 6]},
 {down[4, 1], down[4, 2], down[4, 3], down[4, 4], down[4, 5], down[4, 6]},
 {down[5, 1], down[5, 2], down[5, 3], down[5, 4], down[5, 5], down[5, 6]}}
```

The profit to be maximized is given by

```
Profit = Sum[Sum[ProfitContribution[[i]] * Asell[[i, t]] - 0.5 * Astore[[i, t]],
   {i, NumberOfProducts}], {t, Months}]
```

```
10 sell[1, 1] + 10 sell[1, 2] + 10 sell[1, 3] + 10 sell[1, 4] + 10 sell[1, 5] +
 10 sell[1, 6] + 6 sell[2, 1] + 6 sell[2, 2] + 6 sell[2, 3] + 6 sell[2, 4] + 6 sell[2, 5] +
 6 sell[2, 6] + 8 sell[3, 1] + 8 sell[3, 2] + 8 sell[3, 3] + 8 sell[3, 4] + 8 sell[3, 5] +
 8 sell[3, 6] + 4 sell[4, 1] + 4 sell[4, 2] + 4 sell[4, 3] + 4 sell[4, 4] + 4 sell[4, 5] +
 4 sell[4, 6] + 11 sell[5, 1] + 11 sell[5, 2] + 11 sell[5, 3] + 11 sell[5, 4] +
 11 sell[5, 5] + 11 sell[5, 6] + 9 sell[6, 1] + 9 sell[6, 2] + 9 sell[6, 3] +
 9 sell[6, 4] + 9 sell[6, 5] + 9 sell[6, 6] + 3 sell[7, 1] + 3 sell[7, 2] + 3 sell[7, 3] +
 3 sell[7, 4] + 3 sell[7, 5] + 3 sell[7, 6] - 0.5 store[1, 1] - 0.5 store[1, 2] -
 0.5 store[1, 3] - 0.5 store[1, 4] - 0.5 store[1, 5] - 0.5 store[1, 6] - 0.5 store[2, 1] -
 0.5 store[2, 2] - 0.5 store[2, 3] - 0.5 store[2, 4] - 0.5 store[2, 5] - 0.5 store[2, 6] -
 0.5 store[3, 1] - 0.5 store[3, 2] - 0.5 store[3, 3] - 0.5 store[3, 4] - 0.5 store[3, 5] -
 0.5 store[3, 6] - 0.5 store[4, 1] - 0.5 store[4, 2] - 0.5 store[4, 3] - 0.5 store[4, 4] -
 0.5 store[4, 5] - 0.5 store[4, 6] - 0.5 store[5, 1] - 0.5 store[5, 2] - 0.5 store[5, 3] -
 0.5 store[5, 4] - 0.5 store[5, 5] - 0.5 store[5, 6] - 0.5 store[6, 1] - 0.5 store[6, 2] -
 0.5 store[6, 3] - 0.5 store[6, 4] - 0.5 store[6, 5] - 0.5 store[6, 6] - 0.5 store[7, 1] -
 0.5 store[7, 2] - 0.5 store[7, 3] - 0.5 store[7, 4] - 0.5 store[7, 5] - 0.5 store[7, 6]
```

The constraints are given as below:

```
Constraints = {};
Do[
    Do[
    AppendTo[Constraints,
    Sum[ProductionTime[[j, i]] * Aprod[[i, t]], {i, NumberOfProducts}] +
      HoursPerMonth * Adown[[j, t]] <= ProductionMachines[[j]] * HoursPerMonth],
   {j, NumberOfMachines}], {t, Months}];
Do[AppendTo[Constraints, Aprod[[i, 1]] - Asell[[i, 1]] - Astore[[i, 1]] == 0],
  {i, NumberOfProducts}];
Do[
    Do[
    AppendTo[Constraints,
    Astore[[i, t - 1]] +     Aprod[[i, t]] - Asell[[i, t]] - Astore[[i, t]] == 0],
   {i, NumberOfProducts}], {t, 2, Months}];
AppendTo[Constraints, Sum[Adown[[1, t]], {t, Months}] == 2];
Do[AppendTo[Constraints, Sum[Adown[[j, t]], {t, Months}] == ProductionMachines[[j]]],
  {j, 2, NumberOfMachines}];
```

```
Constraints // Length
```

```
77
```

There are special bounds on our decision variables.
First the upper bounds.

```
UpperBounds = {};
Do[
    Do[
        AppendTo[UpperBounds, Asell[[i, t]] <= Market[[i, t]]],
   {i, NumberOfProducts}], {t, Months}];

Do[
    Do[
        AppendTo[UpperBounds, Astore[[i, t]] <= 100], {i, NumberOfProducts}],
  {t, Months - 1}];

Do[AppendTo[UpperBounds, Adown[[1, t]] <= 2], {t, Months}];

Do[
    Do[
        AppendTo[UpperBounds, Adown[[j, t]] <= ProductionMachines[[j]]],
   {t, Months}], {j, 2, NumberOfMachines}];

Do[AppendTo[UpperBounds, Astore[[i, Months]] <= 50], {i, NumberOfProducts}];
```

Each of the machines must be down in one of the six month, with the exception of grinding machines. In this case, only two of the four machines need be down during the six month.

Then the lower bounds:

```
LowerBounds = {};
Do[AppendTo[LowerBounds, Astore[[i, Months]] >= 50], {i, NumberOfProducts}];
```

```
Bounds1 = Flatten[{UpperBounds, LowerBounds}];
```

Before calling the branch and bound routine the variable types have to be fixed:

```
Var = Flatten[{Adown, Aprod, Astore, Asell}];
BinaryVar = {down[4, 1], down[5, 1], down[4, 2], down[5, 2], down[4, 3], down[5, 3],
   down[4, 4], down[5, 4], down[4, 5], down[5, 5], down[4, 6], down[5, 6]};
IntegerVar = {down[1, 1], down[2, 1], down[3, 1], down[1, 2], down[2, 2], down[3, 2],
   down[1, 3], down[2, 3], down[3, 3], down[1, 4], down[2, 4], down[3, 4],
   down[1, 5], down[2, 5], down[3, 5], down[1, 6], down[2, 6], down[3, 6]};
RealVar = Complement[Var, Join[IntegerVar, BinaryVar]];
DecisionVar = {RealVar, IntegerVar, BinaryVar};
```

We use the BestValue-method first:

```
BestValueResult = BranchBound[Profit, Constraints, DecisionVar, Bounds1, ModPrint → 5];
```

⌗ iterations: 5 ⌗ active nodes: 5 Z0: $-1.\times 10^{9}$ Z1: 116 455.

⌗ iterations: 10 ⌗ active nodes: 10 Z0: $-1.\times 10^{9}$ Z1: 111 305.

Integer solution: 108 855.

For comparison, we repeat the optimization with the DepthFirst-method:

```
DepthFirstResult = BranchBound[Profit, Constraints,
   DecisionVar, Bounds1, Method → "DepthFirst", ModPrint → 5];
```

⌗ iterations: 5 ⌗ active nodes: 5 Z0: $-1.\times 10^{9}$ Z1: 116 455.

⌗ iterations: 10 ⌗ active nodes: 10 Z0: $-1.\times 10^{9}$ Z1: 111 305.

Integer solution: 108 855.

We decode the list results into a mor readable form:

```
MonthNames = {"January", "February", "March", "April",
   "May", "June"};
MachineNames = {"grinder", "hor. drill", "vert. drill",
   "borer", "planer"};
Do[
 Print[Adown[[i, j]] /. DepthFirstResult[[2]],
  " ", MachineNames[[i]], " are down at ", MonthNames[[j]]],
 {i, 1, NumberOfMachines}, {j, 1, 6}]
```

```
0 grinder are down at January
0 grinder are down at February
0 grinder are down at March
0 grinder are down at April
2 grinder are down at May
0 grinder are down at June
0 hor. drill are down at January
0 hor. drill are down at February
0 hor. drill are down at March
1 hor. drill are down at April
1 hor. drill are down at May
0 hor. drill are down at June
0 vert. drill are down at January
0 vert. drill are down at February
0 vert. drill are down at March
0 vert. drill are down at April
1 vert. drill are down at May
2 vert. drill are down at June
0 borer are down at January
0 borer are down at February
0 borer are down at March
1 borer are down at April
0 borer are down at May
0 borer are down at June
0 planer are down at January
0 planer are down at February
0 planer are down at March
1 planer are down at April
0 planer are down at May
0 planer are down at June
```

The costs are:

```
DepthFirstResult[[1]]
```

```
108 855.
```

Combinatorial Optimization

Introduction

This chapter deals with combinatorial problems, especially with a class of problems which can be mapped onto tour problems. The travelling salesman problem is the most famous one of this type. Additionally, we provide a special routine for the well-known knapsack problem.

The heuristic algorithms are:

Simulated annealing

This method dates back to early investigations of Metropolis and coworkers. Valid solutions to the problem are reordered according to certain prescriptions. The results are judged by their amount of improvement compared to the best result obtained so far. The acceptance of new results is governed by a variable called temperature which allows for fluctuations around the optimum found so far. During an iteration process this temperature is cooled down and the system is frozen into the current optimum.

Ant Colony System

Ant Colony System uses mobile agents to explore the system, e.g. a network. While traversing the network the agents change the routing tables on the nodes. The changes depend on the immediate reward the agent receives - in this case the time the agent needs from node to node.

The exact algorithm which is able to proof optimality is

Branch and bound - special implementation

Usually, the common branch and bound algorithm is very slow and inefficient in finding optimal solutions for combinatorial problems. In this chapter we present a special implementation suited for handling the traveling salesman problem. It uses an assignment model formulation for estimating appropriate bounds for the nodes of the branching tree.

Compared with heuristics, the Branch-and-Bound approach is able to proof optimality. However, it is usually slower in coming up with useful sub-optimal solutions.

Genetic algorithms also fall into the class of methods appropriate for handling combinatorial optimization. We do not cover genetic algorithms in this package because this would lead us too far away.

Knapsack

Introduction

Suppose , a hiker wants to take along a number of items with weights (or volumes) a_i, i = 1, ...,n. Each item i has a certain utility val_i. A knapsack obviously has a limited capacity alim. Thus, the problem arises chosing a selection of items with maximum utility, fitting the knapsack.

We chose binary decision variables x_i , with x_i equal to one if the i^{th} item is selected, and zero otherwise. Then the knapsack problem can be formulated as follows

$$Z = \sum_{1=1}^{n} \text{val}_i \, x_i \to \max$$

$$\sum_{1=1}^{n} a_i \, x_i \leq \text{alim}$$

x binary

From the point of view of Operations Research, the knapsack problem may be considered as one of production planning with a single bottle neck, as job scheduling, or as freight optimization indeed.

The package provides two functions, a greedy heuristics KnapsackBinaryGreedy, and a specialized branch bound KnapsackBinaryBB.

The greedy heuristics starts with reordering the items such that $\frac{\text{val}_i}{a_i}$ is falling. KnapsackBinaryGreedy starts with filling the knacksack in this order up to the limiting capacity alim. The resulting solution is in general not feasible if only a fraction x < 1 of the last accepted item fits in. However, this procedure provides an upper bound on the possible utility (or gain) Z and is used as relaxation for the branch bound approach KnapsackBinaryBB.

The greedy heuristics continues in dropping the last, partially accepted item, and searches the ordered list of the remaining non-accepted items for those still fitting the knapsack.

Functions - call and output syntax

KnapsackBinaryGreedy as well as KnapsackBinaryBB

Call:

```
KnapsackBinaryGreedy[lval, la, alim]
KnapsackBinaryBB[lval, la, alim, Options]
```

with

lval: list of utilities

lval list of weights

alim limiting capacity of the backpack

Options (for KnapsackBinaryBB only) :

Option	Value	Description
ModPrint	5 (default)	Prints after given number of iterations
MaxIter	100(default)	Maximal number of iterations

Options for KnapsackBinaryBB only.

Output syntax:

{ZRes, lsol}

with

ZRes: optimal value of objective function

lsol: solution vector

Example

We consider the following randomly generated example

```
lval = {23, 14, 28, 21, 27, 5, 36, 39, 16, 31, 15, 15, 32, 0, 3, 18, 16, 13, 19, 13};
la = {15, 13, 19, 12, 14, 18, 20, 15, 11, 8, 16, 13, 2, 20, 4, 9, 10, 11, 11, 17};
alim = 100;
```

The greedy approximation yields a solution with utility value 226.

```
res0 = KnapsackBinaryGreedy[lval, la, alim]
```

```
{226, {0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0}}
```

The knapsack binary branch bound starts with the heuristics and finds two improved intermediate solutions until the optimal Solution with utility value 234 is obtained.

```
res = KnapsackBinaryBB[lval, la, alim, ModPrint → 10]
```

```
gain   226  solution  {0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0}
#iter  1  # active nodes  1  lower limit  226  upper limit  237.4
#iter  10  # active nodes  6  lower limit  226  upper limit  236.8
#iter  20  # active nodes  5  lower limit  226  upper limit  235.333
#iter  30  # active nodes  6  lower limit  226  upper limit  234.867
gain   230  solution  {1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0}
#iter  40  # active nodes  6  lower limit  230  upper limit  234.867
#iter  50  # active nodes  6  lower limit  230  upper limit  234.867
gain   232  solution  {0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0}
#iter  60  # active nodes  6  lower limit  232  upper limit  234.
gain   234  solution  {0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0}
{234, {0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0}}
```

Travelling Salesman

Introduction

The travelling salesman problem (TSP) might be the most studied problem in combinatorial optimization. It has obvious applications in vehicle routing and job scheduling. The basic variant of the TSP is described as follows. A travelling salesman starts from a given city, visits all cities of a given problem exactly ones, and returns to the origin. A distance matrix has to be given, showing the distance between any two cities. Any permutation of the list of cities leads to a feasible solution. The objective is finding a cycle minimizing the total distance.

In some cases, the distance matrix may be given in terms of a metric (e.g. euclidean) regarding a list of points. More often, however, it will be derived from an underlying graph (road map). In those cases, the distance matrix will satisfy the triangle equation. The locations considered in the TSP may be a subset of the nodes of the underlying graph which can be visited more than once.

In general, the distance matrix will be asymmetric. For the symmetric TSP there are some specialized algorithms which we do not consider.

Basis of the formulation of the asymmetric TSP can be the assignment problem, which is of interest in its own right. It may be desceribed as the problem of assigning tasks to machines where each task is assigned to only one machine, and each machine to only one task. A binary programming formulation may be given as

assignment problem: $Z = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \rightarrow \min$

$\sum_{j=1}^{n} x_{ij} = 1,\ i = 1,2,..,n$

$\sum_{i=1}^{n} x_{ij} = 1,\ j = 1,2,..,n$

x binary

c_{ij} refers to a cost matrix. x_{ij} equals one if task i is assigned to machine j, and zero otherwise. In the context of the TSP, x_{ij} would be one if location i would be directly followed by location j. The assignment problem can in fact be considered as a linear programming problem and can be solved in polynomial time $O(n^{3})$. Asolution is provided by the function AssignmentGKA.

The constraints of the assignment problem are not sufficient for the TSP, however. Several short cycles may appear, i.e. the starting location may be reached again before all locations are visited. Those would be assigned to other short tours starting at other locations. Additional constraints are required in order to exclude short cycles. Several formulations exist. They will not be discussed here because a general integer programming approach to the TSP is not very promissing.

With the function TSBranchBound we present a specialized branch bound algorithm for the TSP. The corresponding assignment problem is used as relaxation approximation. Removing short cycles from the assignment solution obviously increases costs. Therefore, assignment provides lower bounds for the nodes of the branching tree.

We first provide the implementation of a number of heuristics. They usually lead to some solution fast. The quality of the solution is hard to estimate, however.

The branch bound approach provides a lower bound on the optimal solution at each step of the calculation. The quality of the best solution found thus far can therefore be estimated. Optimality is proved eventually. In practice, the optimality proof can be achieved for medium size problems only, because of the non-polynomial time complexity. We demonstrate the method with an example of 36 cities. TSBranchBound is significantly improved compared to release 3.1.

Utility Functions

There are two routines which serve as utility functions for our pacakge routines related to the travelling salesman problem. Therefore, they are presented at the beginning.

Functions - call and output syntax

ShowTour

Call:

```
ShowTour[tour, cities]
```

with

tour: list representing the optimal tour (result of the corresponding routine)

cities: list of cities corresponding to the two-dimensional coordinates of nodes (cites)

Output:

Graphic showing the two-dimensional map of the nodes (cities) with the corresponding optimal connections

TourLength

Call:

```
TourLength[tour, dist]
```

with

tour: list representing the optimal tour (result of the corresponding routine)

dist: distance matrix of the nodes (cities)

Output:

Real number representing the length of the optimal tour

Assignment Problem

In the context of the travelling salesman problem AssignmentGKA has only a support function. We release this function nevertheless because the assignment problem is interesting in itself. The algorithm is due to Glover and Klingman.

Functions - call and output syntax

Call:

```
AssignmentGKA[distmat]
```

with

distmat: cost matrix for the assignment

Output syntax:

```
{lsol, Zres}
```

with

lsol: list of assignments

Zres: minimal assignment cost

TSP Heuristics

FindShortesTour

`FindShortestTour` is a *Mathematica* routine new in 6.0. Several heuristics are implemented which can be addressed by the option *Method*. Methods are e.g. "TwoOpt" and "OrOpt" (originally designed for the symmetric TSP), but also "Greedy" and "SimulatedAnnealing"

TSPatching

TSPatching is based on a general improvement algorithm starting from the solution of the corresponding assignment problem. This solution usually contains short cycles. A solution of the TSP is generated by "patching" short cycles together. The method is less efficient for the symmetric TSP because in this case the assignment solution usually consists of a large number of very short cycles.

Functions - call and output syntax

```
TSPatching[distmat]
```

with

distmat: distance matrix; diagonal elements ∞

Output syntax:

```
{ZRes, lsol}
```

with

ZRes: length of the round trip

lsol: solution vector

Simulated Annealing

About simulated annealing

Simulated annealing consists of a certain sequence of acception and rejection steps. As in the branch and bound case an objective function has to be defined, usually it has to be minimized (which is assumed here). Allowed manipulations on the model are tested with respect to increasing or decreasing the value of the objective function. If it decreases the move is always accepted. In case of enlarging the objective function the underlying manipulation is accepted with a certain probability determined by the so-called process temperature. This concept is closely related to energy state distribution of systems in thermal equilibrium - the energy E of this system is distributed according to the probability

$$p(E) \sim e^{-\frac{E}{kT}},$$

where T is the temperature and k denotes the Boltzmann constant.

The advantage of this algorithm consists in the chance for the system to overcome local minima, and finding the global minimum. Especially for a large number of variables simulated annealing is of great importance. The disadvantage arises from the somewhat experimental determination of the temperature: at the beginning it should allow for numerous fluctuations, towards the end the temperature has to be lowered until the system is frozen in its global minimum.

In the case of traveling salesman the above mentioned allowed manipulations are randomly chosen cross overs and reversions of sections of the total path. After each change of the path the total length is computed and subjected to the probability demon.

Functions - call and output syntax

Call:

```
Annealing[distmat, temp, Options]
```

with

distmat: distance matrix of nodes (cities)

temp: starting temperature

Options:

Option	Value	Description
RedTempFactor	0.9 (default)	Factor by which the temperature is reduced in every iteration step
TempStepMax	100 (default)	Maximal number of iteration steps
SuccessLimit	10 (default)	Maximal number of successfull reordering moves – it is multiplied with number of cities
MaxNumberMoves	100 (default)	Maximal number of reordering moves – it is multiplied with number of cities

Options for Annealing.

Output syntax:

$\{rnode_{i1}, rnode_{i2}, \ldots, rnode_{ik}\}$

with

{...}: list of optimal sequence of the nodes in the tour

Ant Colony System

About Ant Colony System

Ant Colony System (ACS) is a new distributed algorithm proposed by Marco Dorigo and Luca Maria Gambardella (IEEE Transaction On Evolutionary Computation Vol 1, No 1, 1997). The name represents the method: ACS really mimics the behaviour of ants to find the shortest path between two points, even if there is an obstacle on the way. In computer language these ants are mobile cooperative agents changing the natural "routing tables" during their travel on the possible paths. Especially for the traveling salesman problem example calculations have shown the superiority of ACS over traditional methods like simulated annealing or genetic algorithms.
In the following we give a short introduction into this algorithm and explain the main quantities needed for the calculation. To each edge (r,s) of the graph three matrix elements are attached:

δ(r,s) - the cost measure (e.g. the lentgh of the edge)
η(r,s) - $1/\delta$(r,s) - the inverse of the corresponding cost matrix element
τ(r,s) - the *pheromone*, i.e. a measure of the visiting frequency of an ant on this edge; every time an ant visits this edge this quantity is updated.

We investigate the symmetric problem, therefore we have δ(r,s)=δ(s,r)and τ(r,s)=τ(s,r).
The algorithm performs a number of iterations. In each iteration a certain number of ants starts at randomly chosen nodes of the graph to their travel. One iteration is finished if each ant has visited all nodes once. During the iteration the mobile agents have to do two calculational steps:

(1) ant *k* being at node *r* has to chose next node *s*

$s = \arg\max_{u \in J(k,r)}\{\tau(r,u)\,\eta(r,u)^{\beta}\}$ if $q \le q_0$ (exploitation)

$= S$ otherwise (biased exploration).

q is a random number $\in [0,1]$, $q_0$is a parameter $\in[0,1]$. J(k,r) is the set of nodes that remain to be visited by ant k from node r. S is a random variable selected according to the rule

$$p_k(r,s) = \frac{\tau(r,s)^{\alpha}\,\eta(r,s)^{\beta}}{\sum \tau(r,u)^{\alpha}\,\eta(r,u)^{\beta}} \quad u \in J(k,r);\ \text{if } s \in J(k,r)$$

$= 0$ otherwise

$p_k(r, s)$ is the probability for ant k to go from node r to node s. α (>0) and β (>0) determine the relative importance of pheromone vs. distance.

(2) ant *k* changes the pheromone on the visited edge *(r,s) (local update)*

$\tau(r,s) \Longleftarrow (1-\rho)\,\tau(r,s) + \rho\,\Delta\tau(r,s)$

There are different possible choices for the increment $\Delta\tau(r,s)$. We set

$\Delta\tau(r,s) = \gamma \max_{z \in J(k,s)} \tau(s,z)$.

This setting has its origin in the field of reinforcement learning. γ is a learning parameter ($0 \le \gamma \le 1$), ρ describes the relative importance between old values of the pheromone and changes.

After all ants have completed their tours - the iteration is finished - a global update of the pheromone trail is performed. This should resemble the cooperative opinion of all agents obtained during the iteration. The global update is defined by

$\tau(r,s) \Longleftarrow (1-\alpha)\,\tau(r,s) + \alpha\,\Delta\tau(r,s)$ for all edges

with

$\Delta\tau(r,s) = \frac{w}{L_{gb}}$ if (r,s) $\in$ global best tour

$= 0$ otherwise.

L_{gb} is the length of the globally best tour from the beginning of the trial, $\alpha \in (0,1)$, w is an appropriate weight.

The iterations should be finished if no shorter path is expected to be found. In the ideal case the final τ matrix has such entries that one path is overwhelmingly dominant - the resulting probabilities are essentially one for the shortest path found. This is, of course, no rigorous proof of optimality - a situation very similar for simulated annealing. Chosing "wrong" parameters $(\alpha, \beta, \rho, \gamma)$ the iteration can end up in a local minimum far away from the global one.

Function - call and output syntax

Call:

`AntColonySystem[dismat, τ, noa, noi, α, β, γ, ρ, q0, w, Options]`

with

distmat:	distance matrix of the nodes (cities)
noa:	number of ants traversing the network
noi:	number of iterations
$(\alpha, \beta, \gamma, \rho$, q0, w,$\tau)$:	defined above

Options:

Option	Value	Description
ModPrint	5 (default)	Number of steps after which results are printed to the screen
ModAdd	1 (default)	Number of steps after which results are stored into arrays

Options for AntColonySystem.

Output syntax:

```
{BestTour,BestLength,AverageLengthVec,BestLengthVec}
```

with

BestTour:	best round trip obtained by AntColonySystem (list of ordered nodes (cities))
BestLength:	the length of the best round trip
AverageLengthVec:	the vector of average lengths obtained in every ModAdd - iteration step
BestLengthVec:	the vector of best lengths achieved at every ModAdd - iteration step so far

One Short Example

In the following we define a short trivial example data set which we shall use to demonstrate the use of the routines described above. We choose 7 randomly distributed cities.

```
NumberOfCities = 7;
Cities = 5 Table[RandomReal[{0, 1}, 2], {i, NumberOfCities}];
```

For the symmetric traveling salesman problem the corresponding distance matrix is obtained as

```
DistanceMatrix =
  Table[√((Cities[[i, 1]] - Cities[[j, 1]])^2 + (Cities[[i, 2]] - Cities[[j, 2]])^2),
        {i, 1, NumberOfCities}, {j, 1, NumberOfCities}];
DistanceMatrix // MatrixForm
```

```
⎛0.        4.92721  3.5309    3.56426   0.437736  4.44776  4.10208⎞
⎜4.92721   0.       1.60942   1.37414   4.83591   1.22012  1.96316⎟
⎜3.5309    1.60942  0.        0.534164  3.36535   1.93178  0.97097⎟
⎜3.56426   1.37414  0.534164  0.        3.46218   1.42583  1.46084⎟
⎜0.437736  4.83591  3.36535   3.46218   0.        4.46335  3.85197⎟
⎜4.44776   1.22012  1.93178   1.42583   4.46335   0.       2.70586⎟
⎝4.10208   1.96316  0.97097   1.46084   3.85197   2.70586  0.      ⎠
```

Mathematica-routine FindShortestTour:

```
Clear[DistFun];
DistFun[pt1_, pt2_] := DistanceMatrix[[pt1, pt2]];
```

```
lnames = Table[i, {i, NumberOfCities}];
```

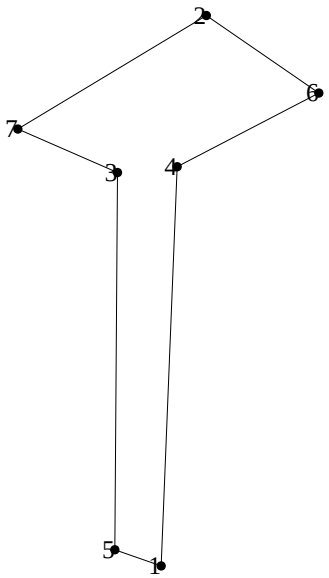
```
FSTResult = FindShortestTour[lnames, DistanceFunction → DistFun]
```

```
{12.9474, {1, 4, 6, 2, 7, 3, 5}}
```

```
FSPLength = FSTResult[[1]]
```

```
12.9474
```

```
ShowTour[FSTResult[[2]], Cities]
```



Simulated Annealing :

```
SAResult = Annealing[DistanceMatrix, 0.5, RedTempFactor → 0.5]
```

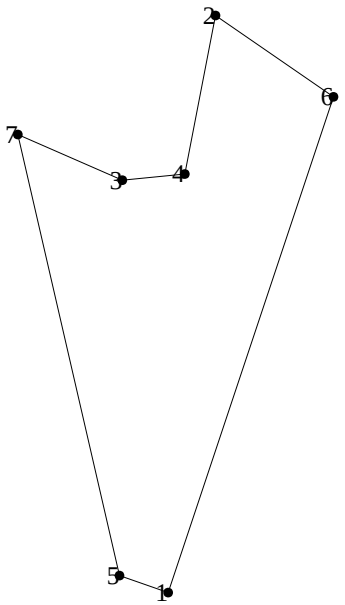
```
starting path length: 21.8043
T = 0.5 Path length = 13.0803
Temperature step = 1 Successful moves = 70
T = 0.25 Path length = 13.1026
Temperature step = 2 Successful moves = 70
T = 0.125 Path length = 12.9474
Temperature step = 3 Successful moves = 55
T = 0.0625 Path length = 12.8369
Temperature step = 4 Successful moves = 14
T = 0.03125 Path length = 12.8369
Temperature step = 5 Successful moves = 0
```

```
{7, 3, 4, 2, 6, 1, 5}
```

```
SALength = TourLength[SAResult, DistanceMatrix]
```

```
12.8369
```

```
ShowTour[SAResult, Cities]
```



Ant colony:

```
NumberOfAnts = 20;
β = 2; (* relative weight of distance vs. pheromone *)
α = 1; (* relative weight of pheromone vs. distance *)
q0 = 0.9; (* exploitation parameter  *)
γ = 0.8; (* learning parameter *)
ρ = 0.5; (* pheromone update parameter  *)
w = 1; (* weight for global - best - length update *)
τ = Table[If[i ≠ j, 1, 0], {i, 36}, {j, 36}];
(* starting pheromone matrix *)
NumberOfIterations = 20;
```

```
ACSResult = AntColonySystem[DistanceMatrix,
   τ, NumberOfAnts, NumberOfIterations, α, β, γ, ρ, q0,

   w];
```

```
iteration number: 5

tour lengths vector of all ants:
 {13.1538, 13.6549, 12.8369, 12.8369, 12.8369, 13.6549, 15.4134, 13.6549, 13.1026, 13.7395,
  13.6549, 13.7395, 13.6549, 13.1538, 15.4134, 13.1538, 13.1026, 15.4134, 13.1026, 13.7395}

average length: 13.6506 best length: 12.8369

iteration number: 10

tour lengths vector of all ants:
 {13.8081, 12.8369, 13.8081, 12.8369, 12.8369, 12.8369, 13.8081, 12.8369, 12.8369, 13.5268,
  13.8081, 13.8823, 13.0484, 12.8369, 12.8369, 13.0484, 13.5268, 12.8369, 13.5268, 12.9639}

average length: 13.2144 best length: 12.8369

iteration number: 15

tour lengths vector of all ants:
 {13.8081, 12.8369, 13.8081, 12.8369, 13.8081, 12.8369, 12.8369, 13.8081, 13.0484, 12.8369,
  12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 13.8823, 12.8369}

average length: 13.094 best length: 12.8369

iteration number: 20

tour lengths vector of all ants:
 {12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 12.8369,
  12.8369, 12.8369, 12.8369, 13.8081, 12.8369, 12.8369, 12.8369, 12.8369, 13.8081, 13.8081}

average length: 12.9826 best length: 12.8369
```

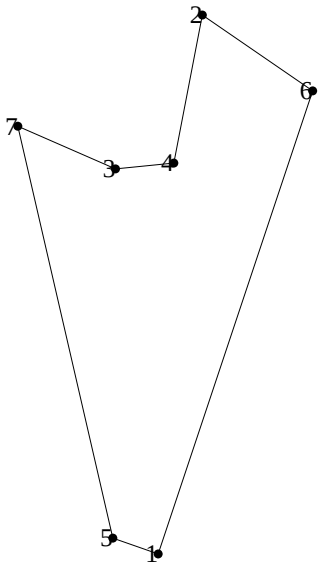
```
ACSResult
```

```
{{6, 2, 4, 3, 7, 5, 1}, 12.8369,
 {13.4148, 13.5111, 13.6719, 13.1744, 13.6506, 13.4638, 13.3297, 13.325, 13.2728, 13.2144,
  13.0794, 13.017, 13.0454, 13.3617, 13.094, 12.9377, 12.8854, 12.934, 12.934, 12.9826},
 {12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 12.8369,
  12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 12.8369,
  12.8369, 12.8369, 12.8369, 12.8369, 12.8369, 12.8369}}
```

```
ACSLength = TourLength[ACSResult[[1]], DistanceMatrix]
```

```
12.8369
```

```
ShowTour[ACSResult[[1]], Cities]
```



All three heuristics give the same result which is not a surprise for this small and symmetric example.

TSP Branchbound

TSBranchBound first calls TSPatching in order to provide a first solution, and to set up an upper bound on the optimal solution. Better solutions are printed as soon as they are found. Otherwise, the actual lower and upper bound and the number of active nodes are printed after certain intervalls. The evaluation stops if the optimal solution is found, or the maximal number of iterations is reached. In the last case, the remaining active nodes are returned besides of the solutions found so far.

Functions - call and output syntax

Call:

```
TSBranchBound[distmat, Options]
```

with

distmat: distance matrix; diagonal elements ∞

Options :

Option	Value	Description
ModPrint	100 (default)	Printing after given number of iterations
MaxIter	10000(default)	Maximal number of iterations

Options for TSBranchBound

Output syntax:

```
{{{ZRes, lsol}}, lnodes}
```

with

{ZRes,lsol}: length of round trip, solution vector

lnodes: list of remaining active nodes

Summarizing Example

With the m-file Data_Xville.m we provide one set of example data (road map), 100 nodes in 2 d - plane, and 1017 directed edges.

```
<< Data_Xville.m
```

```
res = FInput[1];
lnodes0 = res[[1]];
llinks0 = res[[2]];
distmat0 = res[[3]];
Length[lnodes0]
Length[llinks0]
```

```
100
```

```
1017
```

The corresponding distance matrix has already been calculated with the Deijkstra method. The call shows how the data are organized.

From this basic graph, smaller input samples for travelling salesman problems can be derived by chosing subsets of nodes. In the following example, a subset of 36 nodes has been randomly chosen.

```
lnodes = {3, 9, 15, 16, 17, 18, 19, 20, 22, 25, 38, 40, 42, 43, 44, 45, 47,
   52, 55, 60, 61, 67, 75, 77, 78, 79, 80, 82, 84, 85, 87, 90, 91, 93, 95, 100};
dim = Length[lnodes]
```

```
36
```

.Distances do not have to be recalculated as long as we stay on the same underlying road map. The new distance matrix simply results as submatrix of distmat0.

```
distmat = distmat0[[lnodes, lnodes]];
```

Make sure that the diagonal elements of distmat are ∞ This will not result from the distance calculation on the graph but is required in order to prevent short cycles where the node refers to itself.

```
Do[
  distmat[[i, i]] = ∞, {i, Length[distmat]}];
```

We first demonstrate the result for the corresponding assignment problem.

```
res = AssignmentGKA[distmat];
```

```
assignment = res[[1]]
cost = res[[2]]
```

```
{{1, 4}, {2, 14}, {3, 7}, {4, 1}, {5, 23}, {6, 25}, {7, 3}, {8, 10}, {9, 19}, {10, 27},
 {11, 26}, {12, 30}, {13, 2}, {14, 13}, {15, 17}, {16, 5}, {17, 15}, {18, 34}, {19, 12},
 {20, 35}, {21, 29}, {22, 24}, {23, 16}, {24, 22}, {25, 6}, {26, 11}, {27, 8}, {28, 36},
 {29, 21}, {30, 9}, {31, 32}, {32, 33}, {33, 31}, {34, 18}, {35, 20}, {36, 28}}
```

```
83.1376
```

This provides a lower bound of 83.14 on the optimal travelling salesman solution. The assignment solution is not feasiible as solution for the corresponding TSP. It actually contains 15 short cycles.

A feasible solution may be obtained with TSPatching, leading to a value of 109.49 for the length of a valid round trip.

```
res0 = TSPatching[distmat]
```

```
{109.485, {9, 17, 15, 19, 8, 10, 27, 7, 3, 4, 1, 31, 32, 33, 29, 21, 5,
  23, 16, 6, 25, 11, 26, 36, 28, 24, 22, 2, 14, 13, 18, 34, 20, 35, 12, 30, 9}}
```

The *Mathematica* routine FindShortestTour finds a longer tour in this case. This varies from example to example, however.

```
Clear[DistFun];
DistFun[pt1_, pt2_] := distmat[[pt1, pt2]];
```

```
lnames = Table[i, {i, dim}];
```

```
FindShortestTour[lnames, DistanceFunction → DistFun]
```

```
{118.176, {1, 3, 7, 8, 27, 10, 13, 14, 2, 34, 18, 9, 30, 19, 15, 17,
  12, 35, 20, 22, 24, 36, 28, 26, 11, 25, 6, 16, 5, 23, 21, 29, 31, 33, 32, 4}}
```

We now generate the optimal solution with TSBranchBound. The length of the optimal tour turns out to be 102.83.

```
res1 = TSBranchBound[distmat, ModPrint → 100, MaxIter → 10 000];
```

```
lower bound  83.1376

cost  109.485   solution cycle   {9, 17, 15, 19, 8, 10, 27, 7, 3, 4, 1, 31, 32, 33,
  29, 21, 5, 23, 16, 6, 25, 11, 26, 36, 28, 24, 22, 2, 14, 13, 18, 34, 20, 35, 12, 30, 9}

iter  0  # of active nodes  1  lower bound  83.1376  upper bound  109.485

iter  100  # of active nodes  101  lower bound  92.3251  upper bound  109.485

$Aborted
```

```
lres = res1[[1]]
lactivenodes = res1[[2]]
```

```
{{102.834, {1, 31, 32, 33, 29, 21, 5, 23, 16, 6, 25, 11, 26, 28, 36, 24,
   22, 20, 35, 12, 17, 15, 19, 30, 9, 18, 34, 2, 14, 13, 8, 10, 27, 7, 3, 4, 1}},
 {103.055, {1, 31, 32, 33, 29, 21, 5, 23, 16, 6, 25, 11, 26, 28, 36, 24, 22,
   20, 35, 12, 15, 17, 19, 30, 9, 18, 34, 2, 14, 13, 8, 10, 27, 7, 3, 4, 1}},
 {109.485, {9, 17, 15, 19, 8, 10, 27, 7, 3, 4, 1, 31, 32, 33, 29, 21, 5, 23, 16,
   6, 25, 11, 26, 36, 28, 24, 22, 2, 14, 13, 18, 34, 20, 35, 12, 30, 9}}}
```

```
{}
```

Let us test our other two heuristic routines with this non-trivial example. First the simulated annealing routine:

```
SAResult = Annealing[distmat, 0.5, RedTempFactor → 0.5]
```

```
starting path length: 399.316

T = 0.5 Path length = 140.185

Temperature step = 1 Successful moves = 207

T = 0.25 Path length = 124.154

Temperature step = 2 Successful moves = 65

T = 0.125 Path length = 124.493

Temperature step = 3 Successful moves = 57

T = 0.0625 Path length = 115.19

Temperature step = 4 Successful moves = 39

T = 0.03125 Path length = 120.88

Temperature step = 5 Successful moves = 33

T = 0.015625 Path length = 115.408
```

```
Temperature step = 6 Successful moves = 26
T = 0.0078125 Path length = 112.315
Temperature step = 7 Successful moves = 40
T = 0.00390625 Path length = 117.048
Temperature step = 8 Successful moves = 44
T = 0.00195313 Path length = 103.878
Temperature step = 9 Successful moves = 18
T = 0.000976563 Path length = 105.049
Temperature step = 10 Successful moves = 16
T = 0.000488281 Path length = 117.634
Temperature step = 11 Successful moves = 22
T = 0.000244141 Path length = 106.445
Temperature step = 12 Successful moves = 28
T = 0.00012207 Path length = 116.336
Temperature step = 13 Successful moves = 25
T = 0.0000610352 Path length = 115.349
Temperature step = 14 Successful moves = 28
T = 0.0000305176 Path length = 114.654
Temperature step = 15 Successful moves = 23
T = 0.0000152588 Path length = 114.618
Temperature step = 16 Successful moves = 30
T = 7.62939×10^-6 Path length = 120.395
Temperature step = 17 Successful moves = 27
T = 3.8147×10^-6 Path length = 117.633
Temperature step = 18 Successful moves = 21
T = 1.90735×10^-6 Path length = 107.028
Temperature step = 19 Successful moves = 34
T = 9.53674×10^-7 Path length = 117.08
Temperature step = 20 Successful moves = 27
```

Annealing::error2 : Exponent in Metropolis selection function is too small because of very low temperature. Take result obtained so far.

```
T = 4.76837×10^-7 Path length = 115.107
Temperature step = 21 Successful moves = 1
```

Annealing::error2 : Exponent in Metropolis selection function is too small because of very low temperature. Take result obtained so far.

```
T = 2.38419×10^-7 Path length = 115.107
Temperature step = 22 Successful moves = 0
```

```
{27, 7, 3, 1, 4, 33, 31, 32, 29, 21, 23, 16, 5, 6, 25, 11, 26,
 28, 36, 24, 22, 18, 9, 30, 19, 15, 17, 12, 35, 20, 34, 14, 2, 13, 8, 10}
```

The result is better than the result obtained from FindShortestTour but - as expected - not optimal.

Now we try the ant colony algorithm. First, we define the parameter needed by the routine

```
NumberOfAnts = 20;
β = 2; (* relative weight of distance vs. pheromone *)
α = 1; (* relative weight of pheromone vs. distance *)
q0 = 0.9; (* exploitation parameter  *)
γ = 0.8; (* learning parameter *)
ρ = 0.5; (* pheromone update parameter  *)
w = 1; (* weight for global - best - length update *)
τ = Table[If[i ≠ j, 1, 0], {i, 36}, {j, 36}];
(* starting pheromone matrix *)
NumberOfIterations = 20;
```

Now we try the ant colony algorithm itself:

```
ACSResult =
  AntColonySystem[distmat, τ, NumberOfAnts, NumberOfIterations, α, β, γ, ρ, q0,
       w];
```

```
iteration number: 5

tour lengths vector of all ants:
 {132.35, 133.77, 139.145, 136.544, 147.162, 145.966, 149.03, 133.017, 148.019, 150.28,
  138.645, 120.056, 129.136, 138.461, 143.29, 135.816, 142.104, 135.881, 132.414, 140.139}

average length: 138.561 best length: 120.056

iteration number: 10

tour lengths vector of all ants:
 {170.482, 117.859, 126.42, 126.413, 125.987, 128.351, 132.896, 117.95, 146.383, 145.525,
  127.097, 116.338, 165.726, 137.613, 149.089, 118.158, 146.38, 157.976, 147.106, 151.093}

average length: 137.742 best length: 116.338

iteration number: 15

tour lengths vector of all ants:
 {113.974, 113.229, 113.229, 113.974, 113.229, 115.612, 133.373, 122.814, 113.229, 136.035,
  113.974, 113.974, 136.462, 113.229, 142.75, 127.991, 113.229, 115.782, 113.974, 113.974}

average length: 119.702 best length: 113.229

iteration number: 20

tour lengths vector of all ants:
 {139.795, 142.826, 113.787, 117.94, 144.315, 116.183, 113.043, 113.043, 128.922, 113.043,
  140.262, 113.043, 113.043, 138.833, 129.369, 142.826, 128.634, 115.193, 113.787, 131.283}

average length: 125.458 best length: 113.043
```

The is result is better than FindShortestPath and Annealing - but not optimal.

Installation

Windows: the user should execute the installation program ORInstall.exe which is contained in the Windows directory on the installation CD. During the installation procedure he is requested to fix the installation directory

Linux: the user should execute the installation script ORInstall which is contained in the Linux directory on the installation CD. During the installation procedure he is requested to fix the installation directory

It is recommended to copy the files into the directory

$TopDirectory/AddOns/Applications

or into a directory which is included in the search path of *Mathematica*.

The installation procedure generates one directory.