1. TSP.

This program implements a branch-and-cut algorithm for the symmetric traveling salesman (TSP) problem based on the branch-and-cut framework ABACUS. It does neither provide a practically efficient algorithm for the traveling salesman problem nor it is a state-of-the-art implementation of branch-and-cut algorithm for this optimization problem. This program has nothing in common with the one presented in [Thi95] except that it is also based on ABACUS. The author of this program is aware that many functions could be implemented better. However, the ONLY purpose of this program is to show how a branch-and-cut algorithm can be implemented with ABACUS. Therefore we sacrifice efficiency for didactic reasons. Descriptions of branch-and-cut algorithms for the traveling salesman problem can be found in [PR91] and [JRT94].

On a first sight the reader of this example might get the feeling due to the length of this document that an application based on ABACUS requires a significant amount of implementation. While this can be true for sophisticated implementations for some applications this example is mainly blown up by its detailed documentation and the explanation and implementation of some additional features of ABACUS. A "minimal" branch-and-cut TSP-solver using ABACUS could be much shorter!

The problem instances that can be solved with this simple TSP-solver must be in TSPLIB format [Rei91] and the distance between two nodes must be given by the two dimensional Euclidean distance (edge weight type `EUC_2D`).

This program is written in the literate programming system CWEB [KL93]. But also readers not familiar with this system should be able to read this program.

2. Given the complete graph $K_n = (V_n, E_n)$ with edge weights c_e for every edge $e \in E_n$, the symmetric traveling salesman problem is to find a tour with minimum length, i.e., with minimum sum of its edge weights.

For a node set W of a graph we denote by $\delta(W)$ the set of edges with exactly one endnode in W (if $W = \{v\}$ we write $\delta(v)$). Edge sets induced by a node set W in this way are called cuts. For $W \subseteq V$ we denote by $E(W)$ the set of all edges in E with both endnodes in W. For an edge set F we denote by $x(F)$ the sum of the variables associated with the edges in F.

By identifying with each edge $e \in E_n$ of the graph a 0-1 variable $x_e \in \{0,1\}^{E_n}$ we obtain an integer programming formulation of the TSP:

$$
\begin{array}{lll}
\min & c^T x & \\
\text{s.t.} & x(\delta(v)) = 2 & \text{for all } v \in V \\
 & x(\delta(W)) \geq 2 & \text{for all } \emptyset \neq W \subset V \\
 & 0 \leq x \leq 1 & \\
 & x \text{ integer.} &
\end{array}
$$

The variables having value 1 in a feasible solution of this integer program are in one-to-one correspondance with the edges of a tour.

The equations require that each node is incident to exactly two edges and are called degree constraints. The inequalities forbid subtours, and are therefore called subtour elimination constraints. By subtracting the degree constraint $x(\delta(v)) = 2$ for each node $v \in W$ of a subtour elimination constraint and dividing the inequality by -2 we obtain the equivalent format $x(E(W)) \leq W - 1$.

3. The basic idea of ABACUS for the development of a new application is the derivation of problem specific classes from a small amount of base classes. For every application a problem specific master and a problem specific subproblem have to be derived from the base classes **MASTER** and **SUB**. Our master for the traveling-salesman problem is implemented in the class **TSPMASTER** and the corresponding subproblem in the class **TSPSUB**.

Problem specific structure of constraints and variables can be exploited by deriving classes from the base classes **CONSTRAINT** and **VARIABLE**. A variable in our implementation is in one-to-one correspondence with the edge of a graph. Therefore, we represent this variable by the class **EDGE**.

The degree constraints and the subtour elimination constraints are implemented in the classes **DEGREE** and **SUBTOUR**, respectively, which are derived from the abstract base class **CONSTRAINT**. The inheritance tree of these classes is presented in Figure 1. The problem specific classes are surrouned by a bold frame.

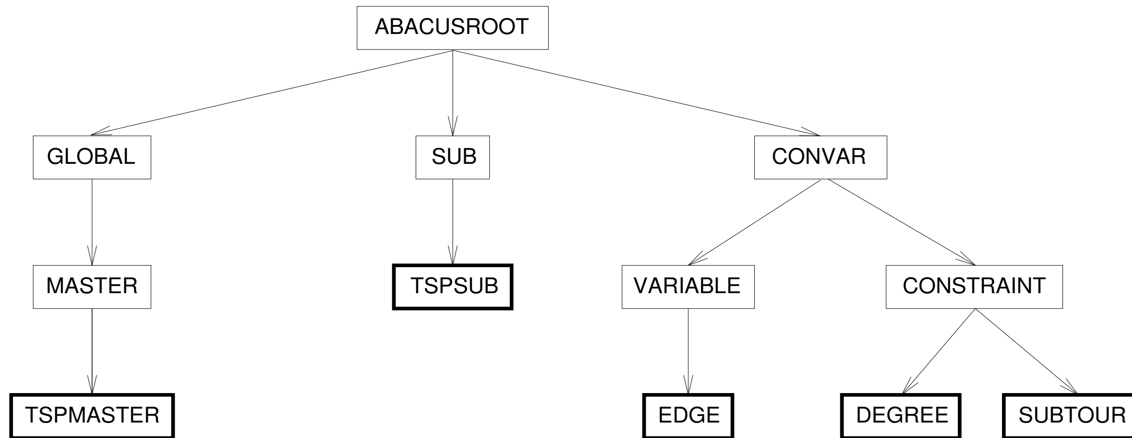

Figure 1. The TSP inheritance tree.

Since the problem specific constraint and variable classes are used in the class **TSPMASTER** and **TSPSUB**, we first describe the declaration and definition of these classes. After the implementation of the classes **TSPMASTER** and **TSPSUB** we present a short main program for starting the optimization.

4. EDGE.
As explained at the beginning each edge of the complete graph is identified with a variable in the integer programming formulation of the traveling salesman problem. Therefore, we derive the class **EDGE** from the base class **VARIABLE** in order to store the two end nodes of the edge.

```
⟨ edge.h 4 ⟩ ≡
#ifndef EDGE_H
#define EDGE_H
#include "variable.h"
  class EDGE : public VARIABLE {
  public:
    EDGE(MASTER *master, int tail, int head, double obj);
    virtual ~EDGE();
    int tail() const;
    int head() const;
  private:
    int tail_;    /* the tail node of the edge */
    int head_;    /* the head node of the edge */
  };
#endif    /* ¬EDGE_H */
```

5. All member functions are defined in the file `edge.cc`.

```
⟨ edge.cc 5 ⟩ ≡
#include "edge.h"
#include "master.h"
```

See also sections 6, 7, 8, and 9.

6. The constructor.

Arguments:
master
A pointer to the corresponding master of the optimization.
tail
The *tail* of the edge associated with the variable.
head
The *head* of the edge associated with the variable.
obj
The objective function coefficient of the variable.

In the call of the base class constructor we make sure that a variable associated with the edge of a graph is not dynamic, but globally valid. It is a binary variable, which has hence lower bound 0.0 and upper bound 1.0.

```
⟨ edge.cc 5 ⟩ +≡
EDGE::EDGE(MASTER *master, int tail, int head, double obj):
  VARIABLE(master, 0, false, false, obj, 0.0, 1.0, VARTYPE::Binary),
  tail_(tail),
  head_(head)
  { }
```

7. The destructor.

```
⟨ edge.cc 5 ⟩ +≡
  EDGE::~EDGE()
  { }
```

8. The function *tail*( ).

Return Value:
The tail nodeof the edge.

```
⟨ edge.cc 5 ⟩ +≡
int EDGE::tail() const
{
  return tail_;
}
```

9. The function *head*( ).

Return Value:
The head node of the edge.

```
⟨ edge.cc 5 ⟩ +≡
int EDGE::head() const
{
  return head_;
}
```

10. DEGREE.

The simplest constraints for the traveling salesman problem besides the trivial inequalities $0 \le x_e \le 1$, which are considered by the lower and upper bounds of the variables, are the degree constraints $x(\delta(v)) = 2$ which require that every node v is incident to exactly two edges. A degree constraint is uniquely determined by the corresponding node.

This class is derived from the abstract base class **CONSTRAINT**.

```
⟨degree.h 10⟩ ≡
#ifndef DEGREE_H
#define DEGREE_H
#include "constraint.h"
  class DEGREE : public CONSTRAINT {
  public:
    DEGREE(MASTER *master, int v);
    virtual ~DEGREE( );
    virtual double coeff (VARIABLE *v);
  private:
    int node_;    /* the node associated with the degree constraint */
  };
#endif    /* ¬DEGREE_H */
```

11. All member functions are defined in the file `degree.cc`.

```
⟨degree.cc 11⟩ ≡
#include "degree.h"
#include "edge.h"
```

See also sections 12, 13, and 14.

12. The constructor.

Arguments:

master
  A pointer to the corresponding master of the optimization.

v
  The number of the associated node.

In the call of the base class constructor we make sure that the constraint is not assiociated with a specific subproblem, is an equation, has right hand side 2.0, will not be removed from the LP-relaxation, is globally valid, and that it can be lifted.

```
⟨degree.cc 11⟩ +≡
DEGREE::DEGREE(MASTER *master, int v):
  CONSTRAINT(master, 0, CSENSE::Equal, 2.0, false, false, true),
  node_(v)
  { }
```

13. The destructor.

```
⟨degree.cc 11⟩ +≡
  DEGREE::~DEGREE( )
  { }
```

14. The function *coeff* ( ) defines the pure virtual function of the base class **CONVAR** for the computation of a coefficient of a variable in a degree constraint. The coefficient of a variable associated with an edge (t, h) is 1 if one of the two nodes t and h is the constraint defining node. Otherwise, the coefficient of the edge is 0.

Return Value:

The coefficient of edge (t, h) associated with the variable $*v$.

Arguments:

v

The pointer to the variable v must be of type **EDGE** $*$. Using RTTI the cast from **VARIABLE** $*$ to **EDGE** $*$ could be done more safely. However, this language feature is currently not supported by the GNU-compiler on all architectures.

⟨ degree.cc 11 ⟩ +≡

```
double DEGREE::coeff(VARIABLE *v)
{
  EDGE *edge = (EDGE *) v;
  if (edge→tail() ≡ node_ ∨ edge→head() ≡ node_) return 1.0;
  else return 0.0;
}
```

15. SUBTOUR.

This class implements the subtour elimination constraint for the symmetric traveling salesman problem. Given a subset W of the nodes set V with $2 \le |W| \le |V| - 2$, then the subtour elimination constraint

$$x(E(W)) \le |W| - 1$$

must hold. As explained at the beginning this inequality has the equivalent form $x(\delta(W)) \ge 2$. While we are using the first format for adding the inequality to the linear programming relaxation, the second format is used for the solution of the separation problem.

We represent the subtour elimination constraint by storing the nodes of the set W. This format (compressed format) saves memory, but the computation of the coefficient of variable requires $O(W)$ time. Therefore, we provide a second format (expanded format) that uses more memory but enables us to compute the coefficient of a variable in constant time using $O(n)$ storage space. This expanded format is given by an array of type **bool** that stores for each node if it is contained in the set W or not. The computation and the deletion of the expanded format is done by redefining the virtual functions *expand*( ) and *compress*( ) of the base class **CONVAR**.

ABACUS will automatically call the function *expand*( ) when “many” coefficients of the constraint have to be computed and remove the expanded format by calling *compress*( ) when these computations are done. In general it is not required to redefine the functions *expand*( ) and *compress*( ) but it might be useful depending on the storage format of the constraint.

```
⟨ subtour.h 15 ⟩ ≡
#ifndef SUBTOUR_H
#define SUBTOUR_H
#include "constraint.h"
  class SUBTOUR : public CONSTRAINT {
  public:
    SUBTOUR(MASTER *master, int nNodes, int *nodes);
    ~SUBTOUR( );
    virtual double coeff(VARIABLE *v);
  private:
    virtual void expand( );
    virtual void compress( );
    ARRAY⟨int⟩ nodes_;    /* the nodes of the set W */
    bool *marked_;    /* array for the expanded format */
    SUBTOUR(const SUBTOUR &rhs);    /* definition omitted */
    const SUBTOUR &operator=(const SUBTOUR &rhs);    /* definition omitted */
  };
#endif    /* ¬SUBTOUR_H */
```

16. All member functions are defined in the file `subtour.cc`.

```
⟨ subtour.cc 16 ⟩ ≡
#include "tspmaster.h"
#include "subtour.h"
#include "edge.h"
```

See also sections 17, 18, 19, 22, and 23.

17. The constructor.

Arguments:

master

A pointer to the corresponding master of the optimization.

nNodes
The number of nodes defining the constraint.
nodes
An array with the nodes defining the constraints.

When call the base class constructor we make sure that the constraint has no associated subproblem, is an $\leq$-inequality, has right hand side $nNodes - 1$, may be removed again from the LP-relaxation, is globally valid, and can be lifted.

```
⟨ subtour.cc 16 ⟩ +≡
SUBTOUR::SUBTOUR(MASTER *master, int nNodes, int *nodes):
  CONSTRAINT(master, 0, CSENSE::Less, nNodes − 1, true, false, true),
  nodes_(master, nNodes),
  marked_(0)
  {
    for (int i = 0; i < nNodes; i++) nodes_[i] = nodes[i];
  }
```

18. The destructor.

```
⟨ subtour.cc 16 ⟩ +≡
  SUBTOUR::~SUBTOUR()
  {
    if (expanded_) delete [ ]marked_;
  }
```

19. The function *coeff* ( ) computes the coefficient of the edge (t, h). It redefines the pure virtual function of the base class **CONSTRAINT**. The coefficient of an edge (t, h) of a subtour elimination constraint is 1 if both nodes t and h belong to the node set *nodes*, otherwise its coefficient is 0.

Return Value:
The coefficient of variable $*v$.

Arguments:
v
A pointer to a variable that must be of type **EDGE** $*$.

Using RTTI the cast from **VARIABLE** $*$ to **EDGE** $*$ could be done more safely. However, this language feature is currently not supported by the GNU-compiler on all architectures.

```
⟨ subtour.cc 16 ⟩ +≡
  double SUBTOUR::coeff(VARIABLE *v)
  {
    if (expanded_) {
      ⟨ compute coefficient for subtour in expanded format 20 ⟩;
    }
    else {
      ⟨ compute coefficient for subtour in compressed format 21 ⟩;
    }
  }
```

20. The coefficient is 1 if both the tail node and the head node of the edge are marked in the expanded format.

```
⟨ compute coefficient for subtour in expanded format 20 ⟩ ≡
  if (marked_[((EDGE *) v)→tail( )] ∧ marked_[((EDGE *) v)→head( )]) return 1.0;
  else return 0.0;
```

This code is used in section 19.

21. In the compressed format we have to scan the set of nodes defining the constraint. We stop the scan as soon as both the tail node and the head node of the edge are found. In this case the coefficient is one, otherwise we return 0.

```
⟨ compute coefficient for subtour in compressed format 21 ⟩ ≡
  int t = ((EDGE *) v)→tail();
  int h = ((EDGE *) v)→head();
  bool tFound = false;
  bool hFound = false;
  for (int v = 0; v < nodes_.size(); v++)
    if (nodes_[v] ≡ t) {
      if (hFound) return 1.0;
      tFound = true;
    }
    else if (nodes_[v] ≡ h) {
      if (tFound) return 1.0;
      hFound = true;
    }
  return 0.0;
```

This code is used in section 19.

22. The function *expand*( ) redefines a virtual function of the base class **CONVAR** in order to compute the expanded format. In the expanded format of a subtour elimination constraint we store in the array *marked_* of type **bool** for each node if it is contained in the set defining the constraint.

ABACUS makes sure that the function *expand*( ) is not called from any function of its kernel, if the constraint is already expanded. Since the function *expand*( ) is a private member of the class **SUBTOUR** and not called from any other function of this class we can be sure that repeated expansion cannot cause any memory leaks.

```
⟨ subtour.cc 16 ⟩ +≡
  void SUBTOUR::expand()
  {
    int n = ((TSPMASTER *) master_)→nNodes();
    marked_ = new bool [n];
    for (int v = 0; v < n; v++) marked_[v] = false;
    int nNodesSubTour = nodes_.size();
    for (int v = 0; v < nNodesSubTour; v++) marked_[nodes_[v]] = true;
  }
```

23. The function *compress*( ) deletes the array storing the expanded format. Like for the function *expand*( ) ABACUS makes sure that the constraint is not compressed again if it is already in compressed format.

```
⟨ subtour.cc 16 ⟩ +≡
  void SUBTOUR::compress()
  {
    delete []marked_;
  }
```

24. TSPMASTER.

The class **TSPMASTER** is derived from the abstract base class **MASTER**. Its main purpose is the initialization of the pools with the problem specific constraints and variable, the storage of the input data, and the memorization of the best tour.

```
⟨tspmaster.h 24⟩ ≡
#ifndef TSPMASTER_H
#define TSPMASTER_H
#include "master.h"
  class TSPMASTER : public MASTER {
  public:
    TSPMASTER(const char *problemName);
    virtual ~TSPMASTER();
    virtual SUB *firstSub();
    int dist(int t, int h);
    virtual void output();
    void newSubTours(int n);
    void updateBestTour(double *xVal);
    int nNodes() const;
    int nearestNeighbor(ARRAY⟨int⟩ &succ);
  private:
    void readTsplibFile(const char *fileName);
    virtual void initializeOptimization();
    virtual void initializeParameters();
    int nNodes_;    /* the number of nodes of the problem instance */
    double *xCoor_;    /* the x-coordinate of each node */
    double *yCoor_;    /* the y-coordinate of each node */
    int nSubTours_;    /* the number of generated subtour elimination constraints */
    int *bestSucc_;    /* the successor of each node in the best know tour */
    bool showBestTour_;    /* if true, the best tour is output finally */
    TSPMASTER(const TSPMASTER &rhs);    /* definition omitted */
    const TSPMASTER &operator=(const TSPMASTER &rhs);    /* definition omitted */
  };
#endif    /* ¬TSPMASTER_H */
```

25. First we specifiy the required include files.

```
⟨tspmaster.cc 25⟩ ≡
  extern "C"
  {
#include <stdio.h>
#include <string.h>
#include <math.h>
  }
#include "tspmaster.h"
#include "tspsub.h"
#include "edge.h"
#include "degree.h"
```

See also sections 26, 33, 34, 41, 42, 48, 55, 56, 59, 60, 65, and 66.

26. The Constructor.

Arguments:

problemName
The name of the optimization problem instance. If *problemName* starts with "./" then the input file with the same name is searched in the current working directory, if the problem name starts with "/" then the absolute path name of the problem instances is taken, otherwise the input file is searched in the directory defined by the environment variable `TSPLIB_DIR`. The input file must be in TSPLIB-format having edge weight type `EUC_2D` (two-dimensional Euclidean distance).

The constructor calls first the constructor of its base class **MASTER**. The second argument of the constructor of **MASTER** is *true* because we are using cutting plane generation for the solution of the subproblems. Since no variables are generated dynamically the third argument of the base class constructor is *false*. To indicate that the traveling salesman problem is a minimization problem we set the sense of the optimization to `OPTSENSE`::*Min*. If in another application the sense of the optimization is still unknown (e.g., if it is later read from a file), the fourth argument of **MASTER**( ) can be omitted. However, it must be later initialized by calling **MASTER**::*initializeOptSense*(*s*) where *s* is `OPTSENSE`::*Min* or `OPTSENSE`::*Max*.

In the body of the constructor the problem data is read and memory is allocated.

```
⟨tspmaster.cc 25⟩ +≡
TSPMASTER::TSPMASTER(const char *problemName):
  MASTER(problemName, true, false, OPTSENSE::Min),
  nNodes_(0),
  xCoor_(0),
  yCoor_(0),
  nSubTours_(0),
  bestSucc_(0),
  showBestTour_(false)
  {
    ⟨read the input data 27⟩;
    ⟨allocate further memory for class TSPMASTER 31⟩;
    ⟨clean up TSPMASTER::TSPMASTER() 32⟩;
  }
```

27. ⟨read the input data 27⟩ ≡

```
  ⟨check if problemName is not 0 28⟩;
  ⟨determine the complete file name 29⟩;
  readTsplibFile(fileName);
```

This code is used in section 26.

28. ⟨check if *problemName* is not 0 28⟩ ≡

```
  if (problemName ≡ 0) {
    err() ≪ "TSPMASTER::TSPMASTER():␣problem␣name␣is␣missing." ≪ endl;
    exit(Fatal);
  }
```

This code is used in section 27.

29. If *problemName* does not start with "./" or "/" we have to determine the location of the TSPLIB.

⟨determine the complete file name 29⟩ ≡

```
  char *fileName;
  if (problemName[0] ≡ '/' ∨ strlen(problemName) > 1 ∧ problemName[0] ≡ '.' ∧ problemName[1] ≡ '/')
    {
    fileName = new char [strlen(problemName) + 1];
    sprintf(fileName, "%s", problemName);
  }
  else {
```

```
    ⟨ find the location of the TSPLIB 30 ⟩;
    fileName = new char [strlen(tsplib) + strlen(problemName) + 2];
    sprintf(fileName, "%s/%s", tsplib, problemName);
  }
```

This code is used in section 27.

30. The location of the TSPLIB is determined with the help of the environment variable `TSPLIB_DIR`.

```
⟨ find the location of the TSPLIB 30 ⟩ ≡
  const char *tsplib = getenv("TSPLIB_DIR");
  if (tsplib ≡ 0) {
    err() ≪ "TSPMASTER::TSPMASTER():␣environment␣variable␣";
    err() ≪ "TSPLIB_DIR␣not␣found" ≪ endl;
    exit(Fatal);
  }
```

This code is used in section 29.

31. ⟨ allocate further memory for class **TSPMASTER** 31 ⟩ ≡

```
  bestSucc_ = new int [nNodes_];
```

This code is used in section 26.

32. ⟨ clean up **TSPMASTER**::**TSPMASTER**( ) 32 ⟩ ≡

```
  delete [ ]fileName;
```

This code is used in section 26.

33. The destructor frees the memory allocated in the constructor.

```
⟨ tspmaster.cc 25 ⟩ +≡
  TSPMASTER::~TSPMASTER()
  {
    delete [ ]xCoor_;
    delete [ ]yCoor_;
    delete [ ]bestSucc_;
  }
```

34. The function *readTsplibFile*( ) reads a problem instance in TSPLIB-format if the edge weight type is `EUC_2D`. The arrays *xCoor_* and *yCoor_* storing the coordinates of the nodes are allocated in the function. Also the number of nodes *nNodes_* is initialized.

Arguments:

fileName
  The name of the input file.

```
⟨ tspmaster.cc 25 ⟩ +≡
  void TSPMASTER::readTsplibFile(const char *fileName)
  {
    ⟨ open the input file 35 ⟩;
    ⟨ read the problem 36 ⟩;
    ⟨ close the TSPLIB file 40 ⟩;
  }
```

35. ⟨ open the input file 35 ⟩ ≡

```
  FILE *tspFile = fopen(fileName, "r");
  if (tspFile ≡ Λ) {      /* NULL is written Λ in CWEB */
```

```
    err() ≪ "TSPMASTER::TSPMASTER():␣";
    err() ≪ "TSPLIB␣file␣" ≪ fileName ≪ "␣could␣not␣be␣opened." ≪ endl;
    exit(Fatal);
  }
```

This code is used in section 34.

36. ⟨ read the problem 36 ⟩ ≡

```
  ⟨ check the problem type and read the dimension 37 ⟩;
  ⟨ have all required keywords been found in the file? 38 ⟩;
  ⟨ read the coordinates of the nodes 39 ⟩;
```

This code is used in section 34.

37. The TSPLIB provides several input formats. For simplification in this example we only can read problems having edge weight type "EUC_2D". In order to determine the number of nodes of the problem we look for a line starting with the string "DIMENSION␣:". The edge weight type of the problem instance is correct if we find a line of the form "EDGE_WEIGHT_TYPE␣:␣EUC_2D" or "EDGE_WEIGHT_TYPE:␣EUC_2D".

As soon as we reach a line starting with the string "NODE_COORD_SECTION" we can continue with stop analyzing the specification part of the TSPLIB-file.

⟨ check the problem type and read the dimension 37 ⟩ ≡

```
  const int maxCharPerLine = 1024;
  char lineBuf[maxCharPerLine + 1];
  bool dimensionFound = false;
  bool typeFound = false;
  bool coordSectionFound = false;
  while (fgets(lineBuf, maxCharPerLine, tspFile)) {
    if (strncmp(lineBuf, "DIMENSION", strlen("DIMENSION")) ≡ 0) {
      if (sscanf(lineBuf, "DIMENSION␣:␣%d", &nNodes_) ≠ 1) {
        err() ≪ "Error␣when␣reading␣dimesion␣of␣problem." ≪ endl;
        exit(Fatal);
      }
      dimensionFound = true;
    }
    else if (strncmp(lineBuf, "EDGE_WEIGHT_TYPE", strlen("EDGE_WEIGHT_TYPE")) ≡ 0) {
      if (strncmp(lineBuf, "EDGE_WEIGHT_TYPE␣:␣EUC_2D", strlen("EDGE_WEIGHT_TYPE␣:␣EUC_2D")) ∧
      strncmp(lineBuf, "EDGE_WEIGHT_TYPE:␣EUC_2D", strlen("EDGE_WEIGHT_TYPE:␣EUC_2D"))) {
        err() ≪ "Invalid␣EDGE_WEIGHT_TYPE,␣must␣be␣EUC_2D." ≪ endl;
        exit(Fatal);
      }
      typeFound = true;
    }
    else if (strncmp(lineBuf, "NODE_COORD_SECTION", strlen("NODE_COORD_SECTION")) ≡ 0) {
      coordSectionFound = true;
      break;
    }
  }
```

This code is used in section 36.

38. Before reading the coordinates of the nodes we check if all required keywords in the file have been found.

⟨ have all required keywords been found in the file? 38 ⟩ ≡

```
  if (¬typeFound) {
```

```
    err() ≪ "Keyword␣EDGE_WEIGHT_TYPE␣not␣found␣in␣file␣" ≪ fileName ≪ ".";
    err() ≪ endl;
    exit(Fatal);
  }
  if (¬dimensionFound) {
    err() ≪ "Keyword␣DIMENSION␣not␣found␣in␣file␣" ≪ fileName ≪ ".";
    err() ≪ endl;
    exit(Fatal);
  }
  if (¬coordSectionFound) {
    err() ≪ "Keyword␣␣NODE_COORD_SECTION␣not␣found␣in␣file␣" ≪ fileName;
    err() ≪ "." ≪ endl;
    exit(Fatal);
  }
```

This code is used in section 36.

39. A line of the "NODE_COORD_SECTION" consists of a number of the node and its x- and y-coordinate. We drop the node number given in the line and number all nodes consecutively from 0 to $nNodes_ - 1$.

⟨read the coordinates of the nodes 39⟩ ≡

```
  xCoor_ = new double [nNodes_];
  yCoor_ = new double [nNodes_];
  int nodeNumber;
  for (int i = 0; i < nNodes_; i++)
    if (fscanf(tspFile, "%d␣%lf␣%lf", &nodeNumber, xCoor_ + i, yCoor_ + i) ≠ 3) {
      err() ≪ "Error␣while␣reading␣coordinates␣of␣node␣" ≪ nodeNumber ≪ "." ≪ endl;
      exit(Fatal);
    }
```

This code is used in section 36.

40. ⟨close the TSPLIB file 40⟩ ≡

```
  if (fclose(tspFile)) {
    err() ≪ "TSPMASTER::TSPMASTER():␣error␣in␣closing␣file␣" ≪ fileName ≪ "." ≪ endl;
    exit(Fatal);
  }
```

This code is used in section 34.

41. The function *firstSUB*( ) redefines a pure virtual function of the base class **MASTER**.

Return Value:

A pointer to the root node of the enumeration tree.

The root of the branch-and-bound tree is initialized with an object of the type **TSPSUB**. For any other application the function *firstSub*( ) has to be implemented in the same way replacing **TSPSUB** with the name of the problem specific subproblem class derived from the class **SUB**.

⟨tspmaster.cc 25⟩ +≡

```
  SUB *TSPMASTER::firstSub()
  {
    return new TSPSUB (this);
  }
```

42. The function *initializeOptimization*( ) redefines a virtual dummy function of the base class **MASTER**. Its main purpose is the initialization of the constraint and variable pools. The initialization of these pools must also be performed in any other application using ABACUS. This initialization can also be done in the

constructor of the problem specific class derived from **MASTER**, but we recommend to follow our strategy redefining the function *initializeOptimization*( ) in a similar way.

```
⟨ tspmaster.cc 25 ⟩ +≡
  void TSPMASTER::initializeOptimization()
  {
    ⟨ output a banner 43 ⟩;
    ⟨ generate the variables 44 ⟩;
    ⟨ generate the degree constraints 45 ⟩;
    ⟨ initialize the pools 46 ⟩;
    ⟨ compute a nearest neighbor tour 47 ⟩;
  }
```

43. ⟨ output a banner 43 ⟩ ≡

```
out() << "A Simple TSP-Solver." << endl;
out() << "Copyright (c) 1996, Stefan Thienel." << endl << endl;
out() << "The intension of this program is to demonstrate various " << endl;
out() << "features of ABACUS, but NOT the fast solution of" << endl;
out() << "traveling salesman problems." << endl << endl;
```

This code is used in section 42.

44. Each variable in this traveling salesman problem solver is associated with an edge of the undirected graph. We create these variables using the class **EDGE** that is derived from the base class **VARIABLE**. The objective function coefficient of each variable is computed by the function $dist(t, h)$ giving the distance of node t and h.

```
⟨ generate the variables 44 ⟩ ≡
  int nEdges = (nNodes_ * (nNodes_ - 1))/2;
  BUFFER<VARIABLE *> variables(this, nEdges);
  for (int t = 0; t < nNodes_ - 1; t++)
    for (int h = t + 1; h < nNodes_; h++) variables.push(new EDGE (this, t, h, dist(t, h)));
```

This code is used in section 42.

45. In any solution for the traveling salesman problem each node must have exactly two incident edges. Therefore we generate for each node a degree constraint having the form $x(\delta(v)) = 2$ with an object of the class **DEGREE**.

```
⟨ generate the degree constraints 45 ⟩ ≡
  BUFFER<CONSTRAINT *> degreeConstraints(this, nNodes_);
  for (int i = 0; i < nNodes(); i++) degreeConstraints.push(new DEGREE(this, i));
```

This code is used in section 42.

46. Any constraint and variable used in the optimization has to be stored in a pool. ABACUS distinguishes three different pools: the constraint pool, the cut pool, and the variable pool.

The constraint pool stores the constraints that should be included in any LP-relaxation. Therefore, we add the degree constraints to this pool. The size of the constraint pool is adapted to the number of constraints contained in the buffer *degreeConstraints*.

We initialize the variable pool with the edges of the graph stored in the buffer *variables*. Since no variables are generated dynamically we set its size to *nEdges*. If in another application variables are generated dynamically then the size of the variable pool must not be initialized with the maximal possible number of variables because the variable pool is reallocated automatically if necessary. Therefore, only an initial guess of its size should be used in this initialization.

Finally, we specify the size of the cut pool, storing all cutting planes generated during the optimization, to $5 * nNodes_$. This value is only an estimation according to our experience with the traveling salesman

problem. Suitably estimations have to be determined for every particular optimization problem. Here, we omit the last argument of the function *initializePools*( ). Therefore its default value is taken, i.e., the constraint pool is not increased if it is full, but non-active constraints are removed instead.

⟨ initialize the pools 46 ⟩ ≡
```
initializePools(degreeConstraints, variables, nEdges, 5 * nNodes_);
```
This code is used in section 42.

47. In order to show how the primal bound can be initialized we compute a nearest-neighbor tour. The primal bound is set with the function *primalBound*( ). The computation of an initial primal bound is not required for the correctness of the optimization.

After setting the primal bound, we initialize the best tour storing in the array *bestSucc_*.

⟨ compute a nearest neighbor tour 47 ⟩ ≡
```
ARRAY<int> succ(this, nNodes_);
int length = nearestNeighbor(succ);
primalBound(length);
for (int i = 0; i < nNodes_; i++) bestSucc_[i] = succ[i];
```
This code is used in section 42.

48. The function *nearestNeighbor*( ) is a rather simple implementation for the determination of a nearest neighbor tour. We start at node 0 and insert the nearest neighbor of the previously inserted nodes until all nodes are included in the tour. Finally we have to close the tour from the last inserted node to node 0.

Return Value:

The length of the tour.

Arguments:

succ

Stores the successor of each node in the tour after the execution of the function. The size of this array must be at least the number of nodes of the traveling salesman problem.

⟨ tspmaster.cc 25 ⟩ +≡
```
int TSPMASTER::nearestNeighbor(ARRAY<int> &succ)
{
  ⟨ local variables (TSPMASTER::nearestNeighbor()) 49 ⟩;
  ⟨ initialize the partial tour with node 0 50 ⟩;
  ⟨ collect the other nodes 51 ⟩;
  ⟨ close the tour and return its length 54 ⟩;
}
```

49. ⟨ local variables (**TSPMASTER**::*nearestNeighbor*( )) 49 ⟩ ≡
```
ARRAY<bool> marked(this, nNodes_, false);   /* collected nodes become marked */
int length;   /* the length of the tour */
int front;   /* the last collected node */
int next;   /* the next collected node */
int minDist;   /* the distance between front and next */
```
This code is used in section 48.

50. ⟨ initialize the partial tour with node 0 50 ⟩ ≡
```
marked[0] = true;
front = 0;
length = 0;
```
This code is used in section 48.

51. ⟨ collect the other nodes 51 ⟩ ≡

```
for (int i = 0; i < nNodes_ − 1; i++) {
  ⟨ find the node next having minimal distance to front 52 ⟩;
  ⟨ add next to the partial tour 53 ⟩;
}
```

This code is used in section 48.

52. ⟨ find the node *next* having minimal distance to *front* 52 ⟩ ≡

```
minDist = INT_MAX;
for (int j = 0; j < nNodes_; j++)
  if (¬marked[j] ∧ (dist(front, j) < minDist)) {
    next = j;
    minDist = dist(front, j);
  }
```

This code is used in section 51.

53. ⟨ add *next* to the partial tour 53 ⟩ ≡

```
marked[next] = true;
length += minDist;
succ[front] = next;
front = next;
```

This code is used in section 51.

54. ⟨ close the tour and return its length 54 ⟩ ≡

```
succ[front] = 0;
length += dist(front, 0);
return length;
```

This code is used in section 48.

55. The function *dist*( ).

Return Value:

The two-dimensional Euclidean distance between node t and node h rounded to the nearest integer (TSPLIB specifies integer distances for all problem).

Arguments:

t
The first end node of an edge.

h
The second end node of an edge.

⟨ tspmaster.cc 25 ⟩ +≡

```
int TSPMASTER::dist(int t, int h)
{
  double xd = xCoor_[t] − xCoor_[h];
  double yd = yCoor_[t] − yCoor_[h];
  return (int) floor(sqrt(xd ∗ xd + yd ∗ yd) + 0.5);
}
```

56. The function *output*( ) redefines a virtual dummy function of the base class master to output statistics of the run and the best tour. This function is called at the end of the optimization after the output of the ABACUS statistics.

⟨ tspmaster.cc 25 ⟩ +≡

```
void TSPMASTER::output()
{
  ⟨output statistics on constraint generation 57⟩;
  ⟨output best tour 58⟩;
}
```

57. ⟨output statistics on constraint generation 57⟩ ≡

```
out() << endl;
out() << "Statistics on TSP-constraints" << endl << endl;
out() << "  Subtour Elimination Constraints: " << nSubTours_ << endl;
out() << endl;
```

This code is used in section 56.

58. The best tour is only output if *showBestTour_* is *true*. This flag can be controlled by the parameter *ShowBestTour* in the configuration file `.tsp`.

⟨output best tour 58⟩ ≡

```
if (showBestTour_) {
  out() << "Best tour: 0";
  int v = bestSucc_[0];
  while (v ≠ 0) {
    out() << ' ' << v;
    v = bestSucc_[v];
  }
  out() << endl;
}
```

This code is used in section 56.

59. The function *newSubTours* increments the counter for the generated subtour elimination constraints.

Arguments:

n
: The number of new generated subtour elimination constraints.

⟨`tspmaster.cc` 25⟩ +≡

```
void TSPMASTER::newSubTours(int n)
{
  nSubTours_ += n;
}
```

60. The function *updateBestTour*( ) replaces the tour stored in *bestSucc_* by extracting it from an incidence vector.

Arguments:

xVal
: An array storing an incidence vector of a tour. The length of this array must be the number of edges of the complete graph.

⟨`tspmaster.cc` 25⟩ +≡

```
void TSPMASTER::updateBestTour(double *xVal)
{
  ⟨local variables (TSPMASTER::updateBestTour()) 62⟩;
  ⟨find the two neighbors of each node 63⟩;
  ⟨assign the successor of each node 64⟩;
}
```

61. The arrays *neigh1* and *neigh2* store for each node the first and the second neighbor, respectively. A node does not have a first or second neighbor so far, if the corresponding component of these arrays is −1.

62. ⟨ local variables (**TSPMASTER** :: *updateBestTour* ( )) 62 ⟩ ≡

```
ARRAY⟨int⟩ neigh1(this, nNodes_, −1);
ARRAY⟨int⟩ neigh2(this, nNodes_, −1);
double oneMinusEps = 1.0 − machineEps( );
int edge = 0;
```

This code is used in section 60.

63. ⟨ find the two neighbors of each node 63 ⟩ ≡

```
for (int t = 0; t < nNodes_ − 1; t++)
  for (int h = t + 1; h < nNodes_; h++) {
    if (xVal[edge] > oneMinusEps) {
      if (neigh1[t] ≠ −1) neigh2[t] = h;
      else neigh1[t] = h;
      if (neigh1[h] ≠ −1) neigh2[h] = t;
      else neigh1[h] = t;
    }
    else if (xVal[edge] > machineEps( )) {
      err( ) ≪ "TSPMASTER::update():␣x␣is␣not␣incidence␣vector␣of␣a␣tour" ≪ endl;
      exit(Fatal);
    }
    edge++;
  }
```

This code is used in section 60.

64. ⟨ assign the successor of each node 64 ⟩ ≡

```
int v, w;
bestSucc_[0] = neigh1[0];
v = 0;
for (int i = 0; i < nNodes_ − 1; i++) {
  w = bestSucc_[v];
  if (neigh1[w] ≡ v) bestSucc_[w] = neigh2[w];
  else bestSucc_[w] = neigh1[w];
  v = w;
}
```

This code is used in section 60.

65. The function *nNodes* ( ).

Return Value:

The number of nodes of the problem instance.

⟨ tspmaster.cc 25 ⟩ +≡

```
int TSPMASTER::nNodes( ) const
{
  return nNodes_;
}
```

66. The function *initializeParameters* ( ) redefines a virtual dummy function of the base class **MASTER**. This function is called at the beginning of the optimization. We show how a parameter for the traveling salesman problem can read using the ABACUS parameter reading facilities.

The function *readParamters*( ) reads the file `.tsp` that must be in the ABACUS parameter file format (see the user's guide). All parameters together with their values are added to an internal parameter table. From this table parameters can be read with the function *getParameter*( ) that is overloaded for most basic data types.

⟨ tspmaster.cc 25 ⟩ +≡

```
void TSPMASTER::initializeParameters()
{
  readParameters(".tsp");
  int status = getParameter("ShowBestTour", showBestTour_);
  if (status) {
    err() << "Parameter ShowBestTour not in configuration file .tsp." << endl;
    exit(Fatal);
  }
}
```

67. TSPSUB.

The class **TSPSUB** is derived from the abstract class **SUB** and implements the problem specific functions for the optimization of a subproblem (node of the branch-and-bound tree). In particular, this is the feasibility test for a solution of the LP-relaxation and the separation of cutting planes.

```
⟨tspsub.h 67⟩ ≡
#ifndef TSPSUB_H
#define TSPSUB_H
#include "sub.h"
  class TSPMASTER;
  class EDGE;
  class TSPSUB : public SUB {
  public:
    TSPSUB(MASTER *master, SUB *father, BRANCHRULE * branchRule);
    TSPSUB(MASTER *master);
    virtual ~TSPSUB( );
    virtual bool feasible( );
    virtual SUB *generateSon(BRANCHRULE * rule);
    virtual int separate( );
    double minCut(int &nCutNodes, int *cutNodes);
  private:
    TSPMASTER *tspMaster( );
    EDGE *edge(int i);
  };
#endif    /* ¬TSPSUB_H */
```

68. All member functions are defined in the file `tspsub.w`.

```
⟨tspsub.cc 68⟩ ≡
#include "tspsub.h"
#include "tspmaster.h"
#include "fastset.h"
#include "edge.h"
#include "subtour.h"
#include "lpsub.h"
  extern "C"
  {
#include "PadbergRinaldi.h"
   }
```

See also sections 69, 70, 71, 72, 77, 78, 82, 87, and 88.

69. The constructor for the root node of the enumeration tree.

Arguments:

master
  A pointer to the corresponding master of the optimization.

Calling the constructor of the base class **SUB** we indicate that 10% additional storage space for constraints, no additional storage space for variables, and 50% additional space for the nonzeros of the constraint matrix in the LP-solver should be allocated.

```
⟨tspsub.cc 68⟩ +≡
TSPSUB::TSPSUB(MASTER *master):
  SUB(master, 10, 0, 50)
  { }
```

70. The constructor for a son of an existing node.

Arguments:

master
A pointer to the corresponding master of the optimization.

father
A pointer to the father in the enumeration tree.

branchRule
The rule defining the subspace of the solution space associated with this node.

```
⟨tspsub.cc 68⟩ +≡
TSPSUB::TSPSUB(MASTER *master, SUB *father, BRANCHRULE *branchRule):
  SUB(master, father, branchRule)
  { }
```

71. The destructor.

```
⟨tspsub.cc 68⟩ +≡
  TSPSUB::~TSPSUB()
  { }
```

72. The LP-solution of a traveling salesman problem is feasible if all variables have values zero or one and no subtour elimination constraint is violated. Instead of solving the separation problem for the subtour elimination constraint, we check first if there is no fractional value. In this case the LP-solution is the incidence vector of a tour if the graph induced by the edges having value one is connected. Remember, the degree constraints hold for each node.

We check if the graph induced by the edges with value 1 is connected with the help of a disjoint set data structure (ABACUS class **FASTSET**). Initially each node is the representative of a set containing only this node. We scan all edges. If there is a edge with value one we check if the two end nodes are contained in disjoint sets. If this is the case unit the two sets and continue. Otherwise the LP-solution is not the incidence vector of a tour except if the edge between the two nodes is the last edge closing the tour.

Return Value:

true
If the LP-solution is the incidence vector of a tour,

false
otherwise.

```
⟨tspsub.cc 68⟩ +≡
  bool TSPSUB::feasible()
  {
    ⟨local variables (TSPSUB::feasible()) 73⟩;
    ⟨initialize the connected components with the single nodes of the graph 74⟩;
    ⟨check if LP-value is integer for each edge and no subtour is induced 75⟩;
  }
```

73. ⟨local variables (TSPSUB::feasible()) 73⟩ ≡

```
  FASTSET conComp(master_, tspMaster()→nNodes());
    /* each set represents a connected component */
  double x;    /* the LP-value of a variable */
  int t;    /* the tail node associated with this variable */
  int h;    /* the head node associated with this variable */
  double eps = master_→machineEps();
  double oneMinusEps = 1.0 − eps;
```

This code is used in section 72.

74. ⟨ initialize the connected components with the single nodes of the graph 74 ⟩ ≡

```
for (int i = 0; i < tspMaster()→nNodes(); i++) conComp.makeSet(i);
```

This code is used in section 72.

75. If the member function *unionSets*( ) of the class **FASTSET** returns *false*, then t and h are contained already in the same set. Only if there are as many edges with value 1.0 as nodes in the graph and there is only one connected component, then the solution is feasible as the degree constraints hold for each node.

If there are no fractional edges, but there is a subtour, then we we find this subtour already before the last edge is added.

⟨ check if LP-value is integer for each edge and no subtour is induced 75 ⟩ ≡

```
int nEdges = 0;    /* the number of edges with value 1; */
for (int i = 0; i < nVar(); i++) {
  x = xVal_[i];
  if (x > oneMinusEps) {
    t = edge(i)→tail();
    h = edge(i)→head();
    if (++nEdges ≡ tspMaster()→nNodes()) {
      ⟨ LP-solution is incidence vector of a tour 76 ⟩;
    }
    if (¬conComp.unionSets(t, h)) return false;
  }
  else if (x ≥ eps) return false;
}
return false;
```

This code is used in section 72.

76. If this tour is shorter than the best known one, then we update it in the associated object of the class **TSPMASTER**. It is not necessary, but although no error, to update the value primal bound. This task is performed by ABACUS.

⟨ LP-solution is incidence vector of a tour 76 ⟩ ≡

```
if (master_→betterPrimal(lp_→value())) tspMaster()→updateBestTour(xVal_);
return true;
```

This code is used in section 75.

77. The function *generateSon*( ) redefines a pure virtual function of the base class **SUB**. While the function **TSPMASTER**::*firstSub*( ) initializes the root node of the branch and bound tree with a problem specific subproblem, this function generates a problem specific son of a subproblem.

Usually in all applications this function is defined like this one line function.

Return Value:

A pointer to a son of this subproblem, which is generated according to the branching rule *rule*.

Arguments:

rule

The branching rule for the generation of the son.

⟨ tspsub.cc 68 ⟩ +≡

```
SUB *TSPSUB::generateSon(BRANCHRULE *rule)
{
  return new TSPSUB (master_, this, rule);
}
```

78. The function *separate*( ) generates violated subtour elimination constraint. Although the number of subtour elimination constraints is exponential in the number of nodes of the graph, this separation problem

can be solved in polynomial time by determining the minimum cut in the support graph, i.e., the graph induced by the variables (edges) having non-zero value in the LP-solution. According to the definition of a subtour elimination constraint, a subtour elimination constraint is violated if and only if the value of the minimum cut in the support graph is less or equal than 2. The corresponding subtour elimination constraint is given by one of the node sets defining a shore of this minimum cut.

Return Value:
The number of generated inequalities.

```
⟨ tspsub.cc 68 ⟩ +≡
  int TSPSUB::separate()
  {
    ⟨ compute the minimum cut in the support graph 79 ⟩;
    ⟨ set up the induced subtour elimination constraint 80 ⟩;
    ⟨ clean up and return number of constraints 81 ⟩;
  }
```

79. ⟨ compute the minimum cut in the support graph 79 ⟩ ≡

```
  int nCutNodes;    /* the number of nodes of one shore of the cut */
  int *cutNodes = new int [tspMaster()→nNodes()];    /* the nodes of a shore of the cut */
  double cutValue;    /* the value of the minimum cut */
  cutValue = minCut(nCutNodes, cutNodes);
  master_→out() << "mincut value = " << cutValue << " on " << nCutNodes << " nodes" << endl;
```

This code is used in section 78.

80. If the value of the minimum cut is less than 2, we generate the constraint. A better implementation would check if the cardinality of the complement of the nodes defining the minimum cut is smaller. In this case one would prefer this equivalent cut induced by the complement.

The function *addCons*( ) adds the generated constraint to the default cutting plane pool and the buffer of new constraints. This constraint is added to the current relaxation at the beginning of the next iteration.

```
⟨ set up the induced subtour elimination constraint 80 ⟩ ≡
  int nGen;    /* the number of generated constraints */
  if (cutValue < 2.0 − master_→eps()) {
    nGen = 1;
    BUFFER⟨CONSTRAINT *⟩ constraint(master_, 1);
    SUBTOUR *subTour = new SUBTOUR (master_, nCutNodes, cutNodes);
    constraint.push(subTour);
    int nAdded = addCons(constraint);
    if (nAdded ≠ 1) {
      master_→err() << "Addition of constraint failed." << endl;
      exit(Fatal);
    }
  }
  else nGen = 0;
```

This code is used in section 78.

81. ⟨ clean up and return number of constraints 81 ⟩ ≡

```
  delete [ ]cutNodes;
  return nGen;
```

This code is used in section 78.

82. The function *minCut*( ) computes the minimcum cut in the support graph.

Return Value:
The value of the minimum cut.

Arguments:
nCutNodes
Holds the number of nodes stored in the array *cutNodes* after the function call.
cutNodes
Stores one shore of the minimum cut. This array must have the length at least the number of nodes of the graph minus 1.

```
⟨ tspsub.cc 68 ⟩ +≡
  double TSPSUB::minCut(int &nCutNodes, int *cutNodes)
  {
    ⟨ initialize the support graph 83 ⟩;
    ⟨ initialize the node induced subgraph 84 ⟩;
    ⟨ call the Padberg-Rinaldi algorithm 85 ⟩;
    ⟨ clean up and return (TSPSUB::mincut()) 86 ⟩;
  }
```

83. The function we will use for solving the minimum cut problems requires that the nodes are numbered beginning with 1 and the first used component of an array has the number 1.

```
⟨ initialize the support graph 83 ⟩ ≡
  int *tail = new int [nVar() + 1];   /* the tail of the edges */
  int *head = new int [nVar() + 1];   /* the head of the edges */
  double *x = new double [nVar() + 1];   /* the weight of the edges */
  int nEdges = 0;   /* the number of edges of the graph */
  for (int i = 0; i < nVar(); i++)
    if (xVal_[i] > master_→machineEps()) {
      ++nEdges;
      tail[nEdges] = edge(i)→tail() + 1;
      head[nEdges] = edge(i)→head() + 1;
      x[nEdges] = xVal_[i];
    }
```

This code is used in section 82.

84. The function *PadbergRinaldi*( ) can compute the minimum cut of a subgraph induced by a set of nodes. Therefore, we store all nodes of the graph in the array *node*. However, a better implementation for the solution of the separation problem would first compute the connected components of the support graph. Each connected component induces already a subtour elimination constraint. Then in addition, one would solve the minimum cut problem in each subgraph induced by the nodes of a connected component. There are further techniques to accelerate the solution of this separation problem (see [PR90]).

```
⟨ initialize the node induced subgraph 84 ⟩ ≡
  int *node = new int [tspMaster()→nNodes() + 1];   /* the nodes inducing the graph */
  for (int i = 1; i ≤ tspMaster()→nNodes(); i++) node[i] = i;
```

This code is used in section 82.

85. For the solution of the minimum cut problem, we use the Padberg-Rinaldi algorithm that is part of a package for the solution of minimum cut problems [JRT96b] and turned out to be very fast for solving the separation problem for the subtour elimination constraints [JRT96a].

The function *PadbergRinaldi*( ) stores the nodes defining a shore of minimum the cut in the components 1,. . . , *nCutNodes* of an array and numbers the nodes beginning at 1. Therefore, we require an extra array for calling this function and have to transform the node set.

⟨ call the Padberg-Rinaldi algorithm 85 ⟩ ≡
```
double cutValue;
int *prCutNodes = new int [tspMaster()->nNodes() + 1];
PadbergRinaldi(tspMaster()->nNodes(), nEdges, tspMaster()->nNodes(), 1, node, tail, head, x,
    &nCutNodes, prCutNodes, &cutValue);
for (int i = 0; i < nCutNodes; i++) cutNodes[i] = prCutNodes[i + 1] - 1;
```
This code is used in section 82.

86. ⟨ clean up and return (**TSPSUB** :: *mincut*()) 86 ⟩ ≡
```
delete []prCutNodes;
delete []tail;
delete []head;
delete []node;
delete []x;
return cutValue;
```
This code is used in section 82.

87. The function *tspMaster*().

Return Value:
A pointer to the corresponding object of the class **TSPMASTER**.

⟨ tspsub.cc 68 ⟩ +≡
```
TSPMASTER *TSPSUB::tspMaster()
{
  return (TSPMASTER *) master_;
}
```

88. The function *edge*().

Return Value:
A pointer to an object of the class **EDGE** corresponding to the *i*-th variable.

Arguments:
i
The number of a variable.

⟨ tspsub.cc 68 ⟩ +≡
```
EDGE *TSPSUB::edge(int i)
{
  return (EDGE *) variable(i);
}
```

89. MAIN PROGRAM.

The main program creates an object of the class **TSPMASTER** and invokes the optimization.

Return Value:

0
: If the optimum solution has been computed,

1
: otherwise.

Arguments:

argc
: The number of arguments of the command line.

argv
: The command line. The second string is the name of the file storing the problem instance that should be solved.

```
⟨ tspmain.cc 89 ⟩ ≡
#include <iostream.h>
#include "tspmaster.h"
  int main(int argc, char **argv)
  {
    ⟨ analyze command line arguments 90 ⟩;
    ⟨ generate a tsp and optimize 91 ⟩;
    ⟨ clean up and return (main( )) 92 ⟩;
  }
```

90. ⟨ analyze command line arguments 90 ⟩ ≡

```
if (argc ≠ 2) {
  cerr ≪ "usage:␣" ≪ argv[0] ≪ "␣<file>" ≪ endl;
  return 1;
}
```

This code is used in section 89.

91. ⟨ generate a tsp and optimize 91 ⟩ ≡

```
TSPMASTER *tsp = new TSPMASTER (argv[1]);
MASTER::STATUS status = tsp→optimize( );
delete tsp;
```

This code is used in section 89.

92. ⟨ clean up and return (main( )) 92 ⟩ ≡

```
if (status) return 1;
else return 0;
```

This code is used in section 89.

93. REFERENCES.

[JRT94] M. Jünger, G. Reinelt, and S. Thienel (1994), Provably good solutions for the traveling salesman problem, *Zeitschrift für Operations Research* **40**, 183–217.

[JRT96a] M. Jünger, G. Rinaldi, and S. Thienel (1996), Practical performance of minimum cut algorithms, technical report, Universität zu Köln, to appear.

[JRT96b] M. Jünger, G. Rinaldi, and S. Thienel (1996), MINCUT, software package, Universität zu Köln, to appear.

[KL93] D.E. Knuth and S. Levy (1993), The CWEB system of structured documentation, technical report and software package, `ftp://labrea.stanford.edu:/pub/cweb`.

[PR90] M.W. Padberg and G. Rinaldi (1990), Facet identification for the symmetric traveling salesman polytope, *Mathematical Programming* **47**, 219–257.

[PR91] M.W. Padberg and G. Rinaldi (1991), A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems, *SIAM Review* **33**, 60–100.

[Rei91] G. Reinelt (1991), TSPLIB—A traveling salesman problem library, *ORSA Journal on Computing* **3**, 376–384, `http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html`.

[Thi95] S. Thienel (1995), ABACUS—A Branch-And-CUt System, doctoral thesis, Universität zu Köln, 1995.

94. INDEX AND SECTION NAMES.

addCons: 80.
argc: 89, 90.
argv: 89, 90, 91.
bestSucc_: 24, 26, 31, 33, 47, 58, 60, 64.
betterPrimal: 76.
Binary: 6.
bool: 72.
branchRule: 67, 70.
BRANCHRULE: 67, 70, 77.
cerr: 90.
coeff: 10, 14, 15, 19.
compress: 15, 23.
conComp: 73, 74, 75.
constraint: 80.
CONSTRAINT: 12, 17.
coordSectionFound: 37, 38.
CSENSE: 12, 17.
cutNodes: 67, 79, 80, 81, 82, 85.
cutValue: 79, 80, 85, 86.
DEGREE: 10, 12, 13.
DEGREE_H: 10.
degreeConstraints: 45, 46.
dimensionFound: 37, 38.
dist: 24, 44, 52, 54, 55.
double: 14, 19, 82.
EDGE: 4, 6, 7, 67.
edge: 14, 62, 63, 67, 75, 83, 88.
EDGE_H: 4.
endl: 28, 30, 35, 37, 38, 39, 40, 43, 57, 58, 63, 66, 79, 80, 90.
eps: 73, 75, 80.
Equal: 12.
err: 28, 30, 35, 37, 38, 39, 40, 63, 66, 80.
exit: 28, 30, 35, 37, 38, 39, 40, 63, 66, 80.
expand: 15, 22, 23.
expanded_: 18, 19.
false: 6, 12, 17, 21, 22, 26, 37, 49, 72, 75.
Fatal: 28, 30, 35, 37, 38, 39, 40, 63, 66, 80.
father: 67, 70.
fclose: 40.
feasible: 67, 72.
fgets: 37.
fileName: 24, 27, 29, 32, 34, 35, 38, 40.
firstSUB: 41.
firstSub: 24, 41, 77.
floor: 55.
fopen: 35.
front: 49, 50, 52, 53, 54.
fscanf: 39.
generateSon: 67, 77.
getenv: 30.
getParameter: 66.
h: 21, 24, 44, 55, 63, 73.
head: 4, 6, 9, 14, 20, 21, 75, 83, 85, 86.
head_: 4, 6, 9.
hFound: 21.
i: 17, 39, 45, 47, 51, 64, 67, 74, 75, 83, 84, 85, 88.
initializeOptimization: 24, 42.
initializeOptSense: 26.
initializeParameters: 24, 66.
initializePools: 46.
int: 8, 9, 48, 55, 65, 78.
INT_MAX: 52.
j: 52.
length: 47, 49, 50, 53, 54.
Less: 17.
lineBuf: 37.
lp_: 76.
machineEps: 62, 63, 73, 83.
main: 89.
makeSet: 74.
marked: 49, 50, 52, 53.
marked_: 15, 17, 18, 20, 22, 23.
MASTER: 26.
master: 4, 6, 10, 12, 15, 17, 67, 69, 70.
master_: 22, 73, 76, 77, 79, 80, 83, 87.
Max: 26.
maxCharPerLine: 37.
Min: 26.
minCut: 67, 79, 82.
minDist: 49, 52, 53.
n: 22, 24, 59.
nAdded: 80.
nCutNodes: 67, 79, 80, 82, 85.
nearestNeighbor: 24, 47, 48.
nEdges: 44, 46, 75, 83, 85.
neigh1: 61, 62, 63, 64.
neigh2: 61, 62, 63, 64.
newSubTours: 24, 59.
next: 49, 52, 53.
nGen: 80, 81.
nNodes: 15, 17, 22, 24, 45, 65, 73, 74, 75, 79, 84, 85.
nNodes_: 24, 26, 31, 34, 37, 39, 44, 45, 46, 47, 49, 51, 52, 62, 63, 64, 65.
nNodesSubTour: 22.
node: 84, 85, 86.
node_: 10, 12, 14.
nodeNumber: 39.
nodes: 15, 17, 19.
nodes_: 15, 17, 21, 22.
nSubTours_: 24, 26, 57, 59.

nVar: 75, 83.
obj: 4, 6.
oneMinusEps: 62, 63, 73, 75.
operator: 15, 24.
optimize: 91.
OPTSENSE: 26.
out: 43, 57, 58, 79.
output: 24, 56.
PadbergRinaldi: 84, 85.
prCutNodes: 85, 86.
primalBound: 47.
problemName: 24, 26, 28, 29.
push: 44, 45, 80.
readParameters: 66.
readParamters: 66.
readTsplibFile: 24, 27, 34.
rhs: 15, 24.
rule: 67, 77.
separate: 67, 78.
ShowBestTour: 58.
*showBestTour_*: 24, 26, 58, 66.
size: 21, 22.
sprintf: 29.
sqrt: 55.
sscanf: 37.
status: 66, 91, 92.
strlen: 29, 37.
strncmp: 37.
SUB: 69, 70.
subTour: 80.
SUBTOUR: 15, 17, 18.
SUBTOUR_H: 15.
succ: 24, 47, 48, 53, 54.
t: 21, 24, 44, 55, 63, 73.
tail: 4, 6, 8, 14, 20, 21, 75, 83, 85, 86.
*tail_*: 4, 6, 8.
tFound: 21.
true: 12, 17, 21, 22, 24, 26, 37, 50, 53, 58, 72, 76.
tsp: 91.
tspFile: 35, 37, 39, 40.
tsplib: 29, 30.
TSPLIB_DIR: 26, 30.
tspMaster: 67, 73, 74, 75, 76, 79, 84, 85, 87.
TSPMASTER: 24, 26, 33, 41, 67.
TSPMASTER_H: 24.
TSPSUB: 67, 69, 70, 71, 77, 87, 88.
TSPSUB_H: 67.
typeFound: 37, 38.
unionSets: 75.
updateBestTour: 24, 60, 76.
v: 10, 12, 14, 15, 19, 21, 22, 58, 64.
value: 76.
variable: 88.
VARIABLE: 6.
variables: 44, 46.
VARTYPE: 6.
void: 22, 23, 34, 42, 56, 59, 60, 66.
w: 64.
x: 73, 83.
*xCoor_*: 24, 26, 33, 34, 39, 55.
xd: 55.
xVal: 24, 60, 63.
*xVal_*: 75, 76, 83.
*yCoor_*: 24, 26, 33, 34, 39, 55.
yd: 55.

⟨ LP-solution is incidence vector of a tour 76 ⟩ Used in section 75.
⟨ add *next* to the partial tour 53 ⟩ Used in section 51.
⟨ allocate further memory for class **TSPMASTER** 31 ⟩ Used in section 26.
⟨ analyze command line arguments 90 ⟩ Used in section 89.
⟨ assign the successor of each node 64 ⟩ Used in section 60.
⟨ call the Padberg-Rinaldi algorithm 85 ⟩ Used in section 82.
⟨ check if LP-value is integer for each edge and no subtour is induced 75 ⟩ Used in section 72.
⟨ check if *problemName* is not 0 28 ⟩ Used in section 27.
⟨ check the problem type and read the dimension 37 ⟩ Used in section 36.
⟨ clean up and return (**TSPSUB** :: *mincut*( )) 86 ⟩ Used in section 82.
⟨ clean up and return (*main*( )) 92 ⟩ Used in section 89.
⟨ clean up and return number of constraints 81 ⟩ Used in section 78.
⟨ clean up **TSPMASTER** :: **TSPMASTER**( ) 32 ⟩ Used in section 26.
⟨ close the TSPLIB file 40 ⟩ Used in section 34.
⟨ close the tour and return its length 54 ⟩ Used in section 48.
⟨ collect the other nodes 51 ⟩ Used in section 48.
⟨ compute a nearest neighbor tour 47 ⟩ Used in section 42.
⟨ compute coefficient for subtour in compressed format 21 ⟩ Used in section 19.
⟨ compute coefficient for subtour in expanded format 20 ⟩ Used in section 19.
⟨ compute the minimum cut in the support graph 79 ⟩ Used in section 78.
⟨ `degree.cc` 11, 12, 13, 14 ⟩
⟨ `degree.h` 10 ⟩
⟨ determine the complete file name 29 ⟩ Used in section 27.
⟨ `edge.cc` 5, 6, 7, 8, 9 ⟩
⟨ `edge.h` 4 ⟩
⟨ find the location of the TSPLIB 30 ⟩ Used in section 29.
⟨ find the node *next* having minimal distance to *front* 52 ⟩ Used in section 51.
⟨ find the two neighbors of each node 63 ⟩ Used in section 60.
⟨ generate a tsp and optimize 91 ⟩ Used in section 89.
⟨ generate the degree constraints 45 ⟩ Used in section 42.
⟨ generate the variables 44 ⟩ Used in section 42.
⟨ have all required keywords been found in the file? 38 ⟩ Used in section 36.
⟨ initialize the connected components with the single nodes of the graph 74 ⟩ Used in section 72.
⟨ initialize the node induced subgraph 84 ⟩ Used in section 82.
⟨ initialize the partial tour with node 0 50 ⟩ Used in section 48.
⟨ initialize the pools 46 ⟩ Used in section 42.
⟨ initialize the support graph 83 ⟩ Used in section 82.
⟨ local variables (**TSPMASTER** :: *nearestNeighbor*( )) 49 ⟩ Used in section 48.
⟨ local variables (**TSPMASTER** :: *updateBestTour*( )) 62 ⟩ Used in section 60.
⟨ local variables (**TSPSUB** :: *feasible*( )) 73 ⟩ Used in section 72.
⟨ open the input file 35 ⟩ Used in section 34.
⟨ output a banner 43 ⟩ Used in section 42.
⟨ output best tour 58 ⟩ Used in section 56.
⟨ output statistics on constraint generation 57 ⟩ Used in section 56.
⟨ read the coordinates of the nodes 39 ⟩ Used in section 36.
⟨ read the input data 27 ⟩ Used in section 26.
⟨ read the problem 36 ⟩ Used in section 34.
⟨ set up the induced subtour elimination constraint 80 ⟩ Used in section 78.
⟨ `subtour.cc` 16, 17, 18, 19, 22, 23 ⟩
⟨ `subtour.h` 15 ⟩
⟨ `tspmain.cc` 89 ⟩
⟨ `tspmaster.cc` 25, 26, 33, 34, 41, 42, 48, 55, 56, 59, 60, 65, 66 ⟩

⟨`tspmaster.h` 24⟩
⟨`tspsub.cc` 68, 69, 70, 71, 72, 77, 78, 82, 87, 88⟩
⟨`tspsub.h` 67⟩

ANGEWANDTE MATHEMATIK UND INFORMATIK
UNIVERSITÄT ZU KÖLN

Report 96.245

A Simple TSP-Solver: An ABACUS Tutorial

Version 1.0, August 1996

Stefan Thienel

	Section	Page
TSP	1	1
EDGE	4	3
DEGREE	10	5
SUBTOUR	15	7
TSPMASTER	24	10
TSPSUB	67	21
MAIN PROGRAM	89	27
REFERENCES	93	28
INDEX AND SECTION NAMES	94	29

Copyright © 1996 by Stefan Thienel

Permission is granted to copy and modify this document and the corresponding source files for the development of ABACUS applications. However, we ask all users to keep the original files of this example unmodified to keep it consistent everywhere in the world. Therefore, modification is only allowed if the modified file receives a new name.

This work has been partially supported by ESPRIT LTR Project no. 20244 (ALCOM-IT) and H.C.M. Institutional Grant no. ERBCHBGCT940710 (DONET).

1991 Mathematics Subject Classification: 68-04, 90c11, 90c27
Keywords: Traveling Salesman Problem, Mixed Integer Programming