CVRPSEP: A package of separation routines for the Capacitated Vehicle Routing Problem

Jens Lysgaard
Department of Management Science and Logistics,
Aarhus School of Business, Denmark
e-mail: lys@asb.dk

Working paper 03-04

Abstract

This report is a technical documentation of the package of routines that were used in the paper "A new branch-and-cut algorithm for the Capacitated Vehicle Routing Problem", by Lysgaard, Letchford, and Eglese, to appear in *Mathematical Programming.* All routines are written in the C programming language.

This documentation describes how to call the separation routines that we used. The package contains all of our separation routines for capacity inequalities, homogeneous multistar inequalities, generalized large multistar inequalities, framed capacity inequalities, strengthened comb inequalities, and hypotour inequalities. In addition, the package contains a routine for generating candidate sets for branching.

Preface

During 1998-2003, Adam N. Letchford, Richard W. Eglese and I collaborated on a project concerned with developing a branch-and-cut algorithm for the Capacitated Vehicle Routing Problem (CVRP).

It was my experience as the programmer of our algorithm that the development of a full branch-and-cut algorithm for the CVRP requires considerable effort. As such, the availability of our code may be beneficial to many others working on branch-and-cut algorithms for the CVRP. It is my hope that the package will be of use to many others working in this field.

Jens Lysgaard
December 11, 2003

1 The modules

The package contains several modules, as shown in the following table.

File name	Comment
MEMMOD	General routines for memory allocation
BASEGRPH	Management of network data structures
SORT	Routines for sorting
CNSTRMGR	Management of data structures for storing constraints
STRNGCMP	Routine for computing the strong components of a directed graph
CUTBASE	Routines for computing left and right hand sides of a capacity cut
COMPRESS	Routine for shrinking of support graph for capacity separation
COMPCUTS	The connected components separation procedure for capacity cuts
MXF	Routine for solving a max-flow problem in a directed graph
FCAPFIX	The capacity separation procedure based on max-flow computations
GRSEARCH	Greedy search procedures for separating capacity cuts
CAPSEP	The main module for capacity separation
HPMSTAR	Separation procedure for homogeneous multistar cuts
MSTARSEP	The main module for homogeneous multistar separation
BINPACK	Routine for exact solution of the Bin Packing Problem
FCITS	Tree search procedure for separating FCIs
FCISEP	The main module for FCI separation
TWOMATCH	Routine for exact separation of 2-matching inequalities
STRCOMB	Routine for separation of strengthened comb inequalities
COMBSEP	The main module for separating strengthened comb inequalities
GLMSEP	The main module for separating Generalized Large Multistar inequalities
BLOCKS	Routine for computing the blocks (biconnected components) in a graph
INTAP	Routine for solving the Assignment Problem
NEWHTOUR	Routine for separating hypotour inequalities
HTOURSEP	The main module for separating hypotour inequalities
BRNCHING	Routine for generating candidate branching sets

Table 1: The files.

For each of the file names in table 1 the package contains two files, a "`.h`" file and a "`.cpp`" file, as is conventional for C programming. The whole package should be compiled and linked into the program.

2 Capacity inequalities

2.1 Calling the separation routine

To separate rounded capacity inequalities, the following routine, which is located in the "`CAPSEP.CPP`" file, must be called (include the "`CAPSEP.H`" file in your calling module):

```
void CAPSEP_SeparateCapCuts(int NoOfCustomers,
                            int *Demand,
                            int CAP,
                            int NoOfEdges,
                            int *EdgeTail,
                            int *EdgeHead,
                            double *EdgeX,
                            CnstrMgrPointer CMPExistingCuts,
                            int MaxNoOfCuts,
                            double EpsForIntegrality,
                            char *IntegerAndFeasible,
                            double *MaxViolation,
                            CnstrMgrPointer CutsCMP);
```

Table 2 explains the individual parameters.

Before calling the separation routine, you should prepare the `CutsCMP` structure, e.g., you code might look like this:

```
#include "cnstrmgr.h"

int Dim;
CnstrMgrPointer MyCutsCMP;

  Dim = 100;
  CMGR_CreateCMgr(&MyCutsCMP,Dim);

  /* Now call the separation routine */
```

The `Dim` parameter is the number of cuts for which memory is reserved initially. Don't worry about reserving insufficient memory. The `MyCutsCMP` structure expands itself if neccessary in order to store all of the cuts.

2.2 Retrieving the cuts

Suppose that you have these declarations:

```
int i,j,Listsize;
double RHS;
int *List;
```

After allocating memory to the `List` vector (allocate memory for $n + 1$ integers), you can read the cuts like this (assuming that you have separated capacity inequalities only):

```
for (i=0; i<MyCutsCMP->Size; i++)
{
  ListSize=0;
  for (j=1; j<=MyCutsCMP->CPL[i]->IntListSize; j++)
  {
    List[++ListSize] = MyCutsCMP->CPL[i]->IntList[j];
  }

  /* Now List contains the customer numbers defining the cut. */
  /* The right-hand side of the cut, */
  /* in the form x(S:S) <= |S| - k(S), is RHS. */

  RHS = MyCutsCMP->CPL[i]->RHS;

  /* Add the cut to the LP. */
}
```

On subsequent calls to capacity separation, you can pass the previously found cuts as input. Your code, using repeated calls of the separation routine, and using all previously found cuts as input, could contain the following:

```
#include "cnstrmgr.h"
#include "capsep.h"

  char IntegerAndFeasible;
  int NoOfCustomers,CAP,NoOfEdges,MaxNoOfCuts;
  double EpsForIntegrality,MaxViolation;
  int *Demand, *EdgeTail, *EdgeHead;
  double *EdgeX;
  CnstrMgrPointer MyCutsCMP,MyOldCutsCMP;

  CMGR_CreateCMgr(&MyCutsCMP,100);
  CMGR_CreateCMgr(&MyOldCutsCMP,100); /* Contains no cuts initially */

  /* Allocate memory for the three vectors EdgeTail, EdgeHead, and EdgeX */
  /* Solve the initial LP */

  EpsForIntegrality = 0.0001;

  do
  {
    /* Store the information on the current LP solution */
    /* in EdgeTail, EdgeHead, EdgeX. */

    /* Call separation. Pass the previously found cuts in MyOldCutsCMP: */
    CAPSEP_SeparateCapCuts(NoOfCustomers,
                           Demand,
                           CAP,
                           NoOfEdges,
                           EdgeTail,
                           EdgeHead,
                           EdgeX,
                           MyOldCutsCMP,
                           MaxNoOfCuts,
                           EpsForIntegrality,
                           &IntegerAndFeasible,
                           &MaxViolation,
                           MyCutsCMP);

    if (IntegerAndFeasible) break; /* Optimal solution found */
    if (MyCutsCMP->Size == 0) break; /* No cuts found */

    /* Read the cuts from MyCutsCMP, and add them to the LP */
    /* Resolve the LP */

    /* Move the new cuts to the list of old cuts: */
    for (i=0; i<MyCutsCMP->Size; i++)
    {
      CMGR_MoveCnstr(MyCutsCMP,MyOldCutsCMP,i,0);
    }

    MyCutsCMP->Size = 0;
  } while (1);
```

2.3 LP forms

A capacity cut for a customer set S can be written in different forms. It is important for the speed of the LP reoptimization to choose a sparse form (i.e., a form with a relatively small number of nonzero coefficients) of the cut. Given the degree equations for all customers, the following three forms of a capacity cut are equivalent ($\bar{S}$ denotes the set of customers outside S, and 0 denotes the depot):

(i) $x(S : S) \leq |S| - k(S)$;

(ii) $x(\delta(S)) \geq 2k(S)$;

(iii) $x(\bar{S} : \bar{S}) + \frac{1}{2}x(\{0\} : \bar{S}) - \frac{1}{2}x(\{0\} : S) \leq |\bar{S}| - k(S)$.

Roughly speaking, form (i) is the sparsest when $|S| \leq n/2$, and form (iii) is the sparsest otherwise. With respect to LP reoptimization time, it is a significant advantage to choose the form according to this rule when representing a capacity cut in the LP, compared to using the same form for all capacity cuts.

3 Separating multiple classes of cuts

Before dealing with other classes of cuts, this section tells how to manage the list of cuts when multiple classes of cuts have been separated before LP reoptimization.

The parameter `CutsCMP` is passed for every separation routine (except for the separation of Generalized Large Multistar inequalities). If you call multiple separation routines, the cuts from the individual separation routine are appended to the list of cuts already stored in `CutsCMP`. When retrieving the cuts you will need to identify the type of each cut. The following piece of code shows how to identify each type of cut after calling multiple separation routines:

```
for (i=0; i<MyCutsCMP->Size; i++)
{
  if (MyCutsCMP->CPL[i]->CType == CMGR_CT_CAP)
  {
    /* Capacity cut */
  }
  else
  if (MyCutsCMP->CPL[i]->CType == CMGR_CT_MSTAR)
  {
    /* Multistar or partial multistar cut */
  }
  else
  /* and so on through all possible types of cuts */
}
```

4 Homogeneous multistar inequalities

4.1 Calling the separation routine

To separate (homogeneous) multistar and partial multistar inequalities, the following routine, which is located in the "MSTARSEP.CPP" file, must be called (include the "MSTARSEP.H" file in your calling module):

```
void MSTARSEP_SeparateMultiStarCuts(int NoOfCustomers,
                                    int *Demand,
                                    int CAP,
                                    int NoOfEdges,
                                    int *EdgeTail,
                                    int *EdgeHead,
                                    double *EdgeX,
                                    CnstrMgrPointer CMPExistingCuts,
                                    int MaxNoOfCuts,
                                    double *MaxViolation,
                                    CnstrMgrPointer CutsCMP);
```

The parameters are the same as those passed to the separation routine for capacity inequalities, except that two parameters on integrality tolerance and feasibility are omitted. The parameter `MaxNoOfCuts` is the maximum number of separated multistar and partial multistar cuts. The violation, in the form (1) given in subsection 4.2, of the cut with the largest violation is returned in `MaxViolation`. The following example shows what the previous "do-loop" might look like, if you first call capacity separation and then also call multistar separation if tailoff occurs wrt. capacity separation. The example builds on that shown for capacity separation.

```
do
{
  /* Store the information on the current LP solution */
  /* in EdgeTail, EdgeHead, EdgeX. */

  /* Call separation. Pass the previously found cuts in MyOldCutsCMP: */
  CAPSEP_SeparateCapCuts(NoOfCustomers,
                         Demand,
                         CAP,
                         NoOfEdges,
                         EdgeTail,
                         EdgeHead,
                         EdgeX,
                         MyOldCutsCMP,
                         MaxNoOfCapCuts,
                         EpsForIntegrality,
                         &IntegerAndFeasible,
                         &MaxCapViolation,
                         MyCutsCMP);

  if (IntegerAndFeasible) break; /* Optimal solution found */

  if (MaxCapViolation < 0.1) /* Tailoff rule */
  {
    /* Double the maximum total number of cuts */
    MaxNoOfMStarCuts = 2 * MaxNoOfCapCuts - MyCutsCMP->Size;

    MSTARSEP_SeparateMultiStarCuts(NoOfCustomers,
                                   Demand,
                                   CAP,
                                   NoOfEdges,
                                   EdgeTail,
                                   EdgeHead,
                                   EdgeX,
                                   MyOldCutsCMP,
                                   MaxNoOfMStarCuts,
                                   &MaxMStarViolation,
                                   MyCutsCMP);
  }

  /* Read the cuts from MyCutsCMP, and add them to the LP */

  /* Resolve the LP */

  /* Move the new cuts to the list of old cuts: */
  for (i=0; i<MyCutsCMP->Size; i++)
  {
    CMGR_MoveCnstr(MyCutsCMP,MyOldCutsCMP,i,0);
  }

  MyCutsCMP->Size = 0;
} while (1);
```

4.2 Retrieving the cuts

The data that are stored for a multistar cut are the three node sets (nucleus N, satellites T, and connectors C) as well as three integers A, B, and L. The three integers make it possible to represent the multistar cut without having to use floating-point coefficients, which is an advantage in terms of precision. In the following form of a multistar (or partial multistar) cut:

$$x(\delta(N)) \geq \lambda + \sigma x(C:T), \tag{1}$$

the constant λ is stored as L/B, and the constant σ is stored as A/B, where A, B and L are integers. The cut can be written using only integer coefficients as follows:

$$Bx(\delta(N)) - Ax(C:T) \geq L. \tag{2}$$

The data for a multistar cut can be retrieved as shown by the following piece of code.

```
int A,B,L;
int *NList, *TList, *CList;

/* Allocate memory for n+1 integers in each of */
/* the vectors NList, TList, CList */

for (i=0; i<MyCutsCMP->Size; i++)
if (MyCutsCMP->CPL[i]->CType == CMGR_CT_MSTAR)
{
  /* Nucleus: */
  for (j=1; j<=MyCutsCMP->CPL[i]->IntListSize; j++)
  NList[j] = MyCutsCMP->CPL[i]->IntList[j];

  /* Satellites: */
  for (j=1; j<=MyCutsCMP->CPL[i]->ExtListSize; j++)
  TList[j] = MyCutsCMP->CPL[i]->ExtList[j];

  /* Connectors: */
  for (j=1; j<=MyCutsCMP->CPL[i]->CListSize; j++)
  CList[j] = MyCutsCMP->CPL[i]->CList[j];

  /* Coefficients of the cut: */
  A = MyCutsCMP->CPL[i]->A;
  B = MyCutsCMP->CPL[i]->B;
  L = MyCutsCMP->CPL[i]->L;
  /* Lambda=L/B, Sigma=A/B */

  /* Add the cut to the LP */
}
```

4.3 LP forms

A multistar cut can be written in three forms, each of which may be the sparsest, depending on $|N|$, $|T|$, and $|C|$. The three forms are the following.

(i) $Bx(\delta(N)) - Ax(C : T) \geq L$;

(ii) $2Bx(N : N) + Ax(C : T) \leq 2B|N| - L$;

(iii) $2Bx(\bar{N} : \bar{N}) + Bx(\{0\} : \bar{N}) - Bx(\{0\} : N) + Ax(C : T) \leq 2B|\bar{N}| - L$.

The easiest way to choose the sparsest form might be to calculate the number of nonzeros for each form of the individual cut, before representing it in the LP.

5 Generalized large multistar inequalities

5.1 Calling the separation routine

To separate generalized large multistar inequalities (GLMs), the following routine, which is located in the "`GLMSEP.CPP`" file, must be called (include the "`GLMSEP.H`" file in your calling module):

```
void GLMSEP_SeparateGLM(int NoOfCustomers,
                        int *Demand,
                        int CAP,
                        int NoOfEdges,
                        int *EdgeTail,
                        int *EdgeHead,
                        double *EdgeX,
                        int *CustList,
                        int *CustListSize,
                        double *Violation);
```

Unlike the separation of the other classes of inequalities, the separation routine for GLMs is exact. At most one cut is identified, namely the cut with the largest violation (if a violated GLM exists). The following piece of code shows how to call the separation routine.

```
int CustListSize;
double Violation;
int *CustList;

/* Allocate memory for n+1 integers in CustList */

/* Store the information on the current LP solution */
/* in EdgeTail, EdgeHead, EdgeX. */

/* Call separation. */
GLMSEP_SeparateGLM(NoOfCustomers,
                   Demand,
                   CAP,
                   NoOfEdges,
                   EdgeTail,
                   EdgeHead,
                   EdgeX,
                   CustList,
                   &CustListSize,
                   &Violation);

if (CustListSize > 0)
{ /* A violated GLM is identified.           */
  /* The nucleus consists of the customers   */
  /* CustList[1],...,CustList[CustListSize]. */
}
```

The GLM can be written as:

$$Qx(N:N) + \sum_{j \in \bar{N}} q_j x(N:j) \leq Q|N| - \sum_{i \in N} q_i, \tag{3}$$

where N is the nucleus, $\bar{N}$ denotes the set of customers outside N, q_i is the demand of customer i, and Q is the vehicle capacity. Using the degree equations, the GLM can also be written as:

$$x(\delta(N)) \geq 2 \sum_{i \in N} \frac{q_i}{Q} + 2 \sum_{j \in \bar{N}} \frac{q_j}{Q} x(N:j). \tag{4}$$

The `Violation` that is returned is the violation of the GLM in the form (4).

6 Framed capacity inequalities

6.1 Calling the separation routine

To separate framed capacity inequalities (FCIs), the following routine, which is located in the "`FCISEP.CPP`" file, must be called (include the "`FCISEP.H`" file in your calling module):

```
void FCISEP_SeparateFCIs(int NoOfCustomers,
                         int *Demand,
                         int CAP,
                         int NoOfEdges,
                         int *EdgeTail,
                         int *EdgeHead,
                         double *EdgeX,
                         CnstrMgrPointer CMPExistingCuts,
                         int MaxNoOfTreeNodes,
                         int MaxNoOfCuts,
                         double *MaxViolation,
                         CnstrMgrPointer CutsCMP);
```

The parameter `MaxNoOfCuts` is the maximum number of separated FCIs. The parameter `MaxNoOfTreeNodes` is the maximum number of nodes explored in the search tree. The violation, in the form (5), of the cut with the largest violation is returned in `MaxViolation`.

6.2 Retrieving the cuts

The FCI, with *frame* $S \subseteq \{1, \ldots, n\}$ and partition $\Omega = \{S_1, \ldots, S_p\}$ of S, is:

$$x(\delta(S)) + \sum_{i=1}^{p} x(\delta(S_i)) \geq 2r(S, \Omega) + 2\sum_{i=1}^{p} k(S_i). \qquad (5)$$

The data that are stored for an FCI is the frame, information on the partition (the p subsets), and the right-hand side. The stored partition covers all customers in the frame, i.e., singletons are included.

The data for an FCI can be retrieved as shown by the following piece of code. The `RHS` that is read for each cut is the right-hand side in (5).

```
int *Label;

/* Allocate memory for n+1 integers in the Label vector */

for (i=0; i<MyCutsCMP->Size; i++)
if (MyCutsCMP->CPL[i]->CType == CMGR_CT_FCI)
{
  for (j=1; j<=n; j++) Label[j] = 0;

  MaxIdx = 0;
  for (SubsetNr=1;
       SubsetNr<=MyCutsCMP->CPL[i]->ExtListSize;
       SubsetNr++)
  {
    /* (subset sizes are stored in ExtList) */
    MinIdx = MaxIdx+1;
    MaxIdx = MinIdx + MyCutsCMP->CPL[i]->ExtList[SubsetNr] - 1;

    for (j=MinIdx; j<=MaxIdx; j++)
    {
      k = MyCutsCMP->CPL[i]->IntList[j];
      Label[k] = SubsetNr;
    }
  }

  /* Now Label[j] is the number of the subset that customer j */
  /* is a member of. The frame is the union of all subsets. */
  /* Label[j] == 0 means that customer j is not in the frame. */

  RHS = MyCutsCMP->CPL[i]->RHS;

  /* Add the cut to the LP */
}
```

6.3 LP forms

An FCI can be viewed as a sum of capacity inequalities (with the capacity inequality of the frame as a special *conditional* inequality). Since each of these capacity inequalities can be written in various forms, there exist several possible forms of an FCI.

One particular form would be a reasonable choice in many cases, namely to use the cutset form for the frame and the *inside form* for each subset:

$$-\frac{1}{2}x(\delta(S)) + \sum_{i=1}^{p} x(S_i : S_i) \leq -r(S,\Omega) + \sum_{i=1}^{p}(|S_i| - k(S_i)). \tag{6}$$

This form is relatively sparse if the frame contains all customers (which is the case if the support graph contains only one connected component) and all subsets are small (all singletons disappear in this form). In other cases the form of each capacity inequality may be chosen individually for each subset.

7 Strengthened comb inequalities

7.1 Calling the separation routine

To separate strengthened comb inequalities, the following routine, which is located in the "`COMBSEP.CPP`" file, must be called (include the "`COMBSEP.H`" file in your calling module):

```
void COMBSEP_SeparateCombs(int NoOfCustomers,
                           int *Demand,
                           int CAP,
                           int QMin,
                           int NoOfEdges,
                           int *EdgeTail,
                           int *EdgeHead,
                           double *EdgeX,
                           int MaxNoOfCuts,
                           double *MaxViolation,
                           CnstrMgrPointer CutsCMP);
```

The parameter `MaxNoOfCuts` is the maximum number of separated strengthened combs.

The violation, in the form (8), of the cut with the largest violation is returned in `MaxViolation`.

The parameter `QMin` is a lower bound on the demand that must be delivered on any route in a feasible CVRP solution. Without loss of generality, an upper bound K can be assumed to be given on the number of routes (if the number of routes is free, K can be set equal to n). As such, a valid value of `QMin` can be calculated as:

$$QMin := \sum_{i=1}^{n} q_i - (K-1)Q, \tag{7}$$

where q_i is the demand of customer i and Q is the vehicle capacity. If any vehicle delivers less than `QMin` on a route, the remaining demand exceeds the total capacity on the remaining $K-1$ routes.

Note that if K is not fixed at the minimum possible, or if the minimum possible number of routes exceeds the trivial lower bound $\lceil \sum_{i=1}^{n} q_i/Q \rceil$, then `QMin` becomes nonpositive, if it is calculated according to (7). Any value of `QMin` (negative, positive, or zero) is a valid input to the procedure.

7.2 Retrieving the cuts

The strengthened comb inequality, with handle H and teeth $T_1, \ldots, T_t$, can be written in the following form:

$$x(\delta(H)) + \sum_{j=1}^{t} x(\delta(T_j)) \geq RHS. \tag{8}$$

The data for a strengthened comb inequality can be retrieved as shown by the following piece of code. The RHS that is read for each cut is the right-hand side in (8).

```
char **InTooth;

/* Allocate memory for n+2 rows and TMAX+1 columns in InTooth, */
/* where TMAX is an upper bound on the number of teeth in any  */
/* of the separated strengthened comb inequalities. */

for (i=0; i<MyCutsCMP->Size; i++)
if (MyCutsCMP->CPL[i]->CType == CMGR_CT_STR_COMB)
{
  NoOfTeeth = MyCutsCMP->CPL[i]->Key;

  for (Node=1; Node<=NoOfCustomers+1; Node++)
  for (Tooth=0; Tooth<=NoOfTeeth; Tooth++)
  InTooth[Node][Tooth] = 0;

  for (k=1; k<=MyCutsCMP->CPL[i]->IntListSize; k++)
  {
    j = MyCutsCMP->CPL[i]->IntList[k];
    InTooth[j][0] = 1; /* Node j is in the handle */
  }

  for (t=1; t<=NoOfTeeth; t++)
  {
    MinIdx = MyCutsCMP->CPL[i]->ExtList[t];

    if (t == NoOfTeeth)
    MaxIdx = MyCutsCMP->CPL[i]->ExtListSize;
    else
    MaxIdx = MyCutsCMP->CPL[i]->ExtList[t+1] - 1;

    for (k=MinIdx; k<=MaxIdx; k++)
    {
      j = MyCutsCMP->CPL[i]->ExtList[k];
      InTooth[j][t] = 1; /* Node j is in tooth t */
    }
  }

  /* Now InTooth[j][t] == 1 if and only if    */
  /* node j is in tooth number t. The depot   */
  /* is node number NoOfCustomers+1, and the  */
  /* handle is represented as tooth number 0. */

  RHS = MyCutsCMP->CPL[i]->RHS;

  /* Add the cut to the LP */
}
```

8 Hypotour inequalities

8.1 Calling the separation routine

To separate *2-edges extended hypotour inequalities* (2EHs), the following routine, which is located in the "`HTOURSEP.CPP`" file, must be called (include the "`HTOURSEP.H`" file in your calling module):

```
void HTOURSEP_SeparateHTours(int NoOfCustomers,
                             int *Demand,
                             int CAP,
                             int NoOfEdges,
                             int *EdgeTail,
                             int *EdgeHead,
                             double *EdgeX,
                             CnstrMgrPointer CMPExistingCuts,
                             int MaxNoOfCuts,
                             double *MaxViolation,
                             CnstrMgrPointer CutsCMP);
```

The parameter `MaxNoOfCuts` is the maximum number of separated 2EHs.

The violation, in the form (10), of the cut with the largest violation is returned in `MaxViolation`.

8.2 Retrieving the cuts

The 2EH, for a given $W \subset \{1, \ldots, n\}$ and two distinct edges $e_1, e_2 \in \delta(W)$, can be written as:

$$x(\delta(W)) + 2x(F) \geq 2x_{e_1} + 2x_{e_2}. \tag{9}$$

Using the degree equations, it can be rewritten as:

$$x(W:W) - x(F) + x_{e_1} + x_{e_2} \leq |W|. \tag{10}$$

Unlike the storage of the other classes of cuts, a 2EH is not stored in any particular compact form. Instead, a 2EH is simply stored as a list of those edges and coefficients which appear on the left-hand side in (10). Correspondingly, the `RHS` that is read for each cut is the right-hand side in (10).

The data for a 2EH inequality can be retrieved as shown by the following piece of code.

```
int *Tail, *Head;
double *Coeff;

/* Allocate memory for the three vectors Tail, Head, and Coeff. */
/* An upper bound on the number of entries for which memory     */
/* should be allocated in each of Tail, Head, and Coeff, is     */
/* (n(n+1)/2)+1, i.e., the number of edges in the problem plus  */
/* one for index zero. */

for (i=0; i<MyCutsCMP->Size; i++)
if (MyCutsCMP->CPL[i]->CType == CMGR_CT_TWOEDGES_HYPOTOUR)
{
  for (j=1; j<=MyCutsCMP->CPL[i]->IntListSize; j++)
  {
    Tail[j]  = MyCutsCMP->CPL[i]->IntList[j];
    Head[j]  = MyCutsCMP->CPL[i]->ExtList[j];
    Coeff[j] = MyCutsCMP->CPL[i]->CoeffList[j];
  }

  RHS = MyCutsCMP->CPL[i]->RHS;

  /* The cut is: */
  /* Coeff[1] * x(Tail[1],Head[1]) +
     Coeff[2] * x(Tail[2],Head[2]) +
     ...
     <= RHS. */

  /* Add the cut to the LP */
}
```

9 Branching

One possible way of branching is to choose a customer set S for which $2 < x^*(\delta(S)) < 4$ and impose, by branching, the disjunction $(x(\delta(S)) = 2) \vee (x(\delta(S)) \geq 4)$.

The following routine can be used for generating candidates for S. The routine is contained in the file "`BRNCHING.CPP`" (include the "`BRNCHING.H`" file in your calling module):

```
void BRNCHING_GetCandidateSets(int NoOfCustomers,
                               int *Demand,
                               int CAP,
                               int NoOfEdges,
                               int *EdgeTail,
                               int *EdgeHead,
                               double *EdgeX,
                               CnstrMgrPointer CMPExistingCuts,
                               double BoundaryTarget,
                               int MaxNoOfSets,
                               CnstrMgrPointer SetsCMP);
```

The parameter `MaxNoOfSets` is the maximum number of identified sets. The parameter `BoundaryTarget` is the target value of $x^*(\delta(S))$. The identified sets are returned in the same type of data structure as that used for storing cuts. The following piece of code shows how to retrieve the candidate sets:

```
int MaxNoOfSets,SetNr;
double BoundaryTarget;
CnstrMgrPointer SetsCMP;

BoundaryTarget = 3.0; /* Specify a value between 2.0 and 4.0 */
MaxNoOfSets = NoOfCustomers;
CMGR_CreateCMgr(&SetsCMP,MaxNoOfSets);

BRNCHING_GetCandidateSets(NoOfCustomers,
                          Demand,
                          CAP,
                          NoOfEdges,
                          EdgeTail,
                          EdgeHead,
                          EdgeX,
                          MyOldCutsCMP,
                          BoundaryTarget,
                          MaxNoOfSets,
                          SetsCMP);

for (i=0; i<SetsCMP->Size; i++)
{
  ListSize=0;
  for (j=1; j<=SetsCMP->CPL[i]->IntListSize; j++)
  {
    List[++ListSize] = SetsCMP->CPL[i]->IntList[j];
  }

  /* Now List contains the numbers of the customers in */
  /* the i'th candidate set for S. */

  /* The boundary x^*(\delta(S)) of this S is RHS. */
  RHS = SetsCMP->CPL[i]->RHS;
}

CMGR_FreeMemCMgr(&SetsCMP); /* Deallocate the memory  */
                            /* for the candidate sets */
```

The sets in SetsCMP are listed in nondecreasing order of $|x^*(\delta(S)) - T| / \sum_{i \in S} q_i$, where T is the value of BoundaryTarget.

Parameter	Comment
`NoOfCustomers`	The number of customers in the problem. The n customers are assumed to be numbered $1, \dots, n$.
`Demand`	A vector containing the customer demands, i.e., `Demand[i]` is the demand of customer i for $i = 1, \dots, n$.
`CAP`	The capacity of each vehicle.
`NoOfEdges`	The number of edges for which information is passed by the following three parameters.
`EdgeTail` `EdgeHead` `EdgeX`	These three vectors give information on the current LP solution. Only information on those edges e with $x^*_e > 0$ should be passed in the three vectors. `EdgeTail[e], EdgeHead[e]` are the two end vertices of edge e. `EdgeX[e]` is x^*_e. The depot is assumed to be numbered $n+1$. Note that edge numbers are 1-based ($e = 1, \dots, \mathit{NoOfEdges}$).
`CMPExistingCuts`	This is a pointer to a data structure containing previously generated rounded capacity inequalities. It is used as input to our fourth heuristic. It is explained in the text how to use this data structure.
`MaxNoOfCuts`	The maximum number of cuts to be returned. Note that this maximum applies only if the connected components heuristic fails. That is, the number of cuts found by the connected components heuristic cannot be controlled through this parameter. The heuristic based on max-flows finds at most half of the maximum number of cuts. The rest are allowed to be found by the greedy search and the add/drop (the fourth) heuristic. The actual number of cuts found is returned in `CutsCMP->Size`.
`EpsForIntegrality`	This is the tolerance used when checking whether the input solution is integer (use, e.g., `EpsForIntegrality = 0.0001`). If the connected components heuristic fails, and each x^*_e deviates no more than `EpsForIntegrality` from the nearest integer, the entire routine returns `IntegerAndFeasible = 1`, otherwise it returns `IntegerAndFeasible = 0`.
`IntegerAndFeasible`	See the comment to `EpsForIntegrality`.
`MaxViolation`	This is the violation of the cut with the largest violation, in the form $x(S:S) \leq \|S\| - k(S)$.
`CutsCMP`	This is a pointer to the data structure containing the cuts found by the separation routine. It is explained in the text how to read the actual cuts from this structure.

Table 2: Parameters for `CAPSEP_SeparateCapCuts`.