5 Implementierung

In dem Kapitel der Implementierung werden zunächst die Datenstrukturen der Knotentabelle, der Node Save Area und der Integer-Tabelle dargestellt. Danach wird auf die Implementierung des Branch-and-Bound-Prozesses und der Heuristiken eingegangen. Die Implementierungen werden durch Pseudo Codes veranschaulicht, die einen vereinfachten Algorithmus der jeweiligen Implementierung darstellen.

5.1 Speicherung der Knoten

Ein Knoten wird in der Knotentabelle und in der Node Save Area (NSA) abgespeichert. Ist die Knotentabelle bzw. die NSA hinreichend voll, dann werden Knoten in das Node File geschrieben.

5.1.1 Knotentabelle

Die Knotentabelle umfasst sieben Spalten und *xmaxno* Knoten. In der Knotentabelle werden alle Informationen abgespeichert, auf die beim Laden eines Knotens direkt zugegriffen werden und die bei allen Knoten vorkommen. In der Abb. 5-1 sind die Spalten der Knotentabelle dargestellt.

xnodfc	xnodif	xnodrc	xnodem	xnodbj	xnodst	xnodle

Abb. 5-1: Knotentabelle

Jede Zeile der Knotentabelle repräsentiert einen Knoten und es werden pro Knoten folgende Informationen abgespeichert:

- *xnodfc* beinhaltet den Funktionswert eines Knotens.
- *xnodif* speichert die Summe der Unzulässigkeiten eines Knotens.
- *xnodrc* ist der Pointer auf die Node Save Area. Wenn der Pointer negativ ist, dann ist der Knoten gelöscht worden.
- *xnodem* speichert die Estimation eines Knotens.
- *xnodbj* speichert die Branchingvariable des Knotens. Bei einer negativen Branchingvariable wird der Lower Branch und bei einer positiven der Upper Branch als erstes ausgeführt.
- *xnodst* gibt den Status des Knotens an. Ist der Status gleich Eins, dann wurde der Knoten noch nicht betrachtet, ist er gleich minus Eins, dann wurde der erste Branch ausgeführt und ist er gleich Null, dann ist der Knoten abgearbeitet.
- *xnodle* gibt die Knotenlänge des Knotens in der Node Save Area an.

5.1.2 Node Save Area

In der Node Save Area *xnodei* werden alle weiteren Informationen zu einem Knoten abgespeichert. Die NSA ist in words (4 Byte) unterteilt, die wiederum jeweils in 32 Bits unterteilt werden können. Die Abb. 5-2 zeigt die Speicherbereiche eines Knotens in der NSA.

…	Branchingwert	Modifizierte Bounds	Stack	Basis	Fraktionelle Werte	…

Abb. 5-2: Node Save Area

- *Branchingwert:* Dieses Feld umfasst maximal ein word und speichert den Branchingwert $\lfloor x_j \rfloor$ einer Nichtbinär-Branchingvariablen x_j ab.
- *Modifizierte Bounds:* Das Feld kann maximal *4*(xmxj-xmxj1)* words umfassen und speichert die durch das Branchen modifizierten Bounds von nicht fixierten Integervariablen. Für eine Integervariable mit veränderter Upper und Lower Bound werden vier words nach der dargestellten Reihenfolge belegt.

...	Spaltenindex	Upper Bound	-Spaltenindex	Lower Bound	...

Abb. 5-3: Modifizierte Bounds

  Eine veränderte Upper Bound wird durch ein positives und eine veränderte Lower Bound durch ein negatives Vorzeichen des Spaltenindex gekennzeichnet.
- *Stack:* Der Stackbereich speichert alle fixierten Integer-Variablen eines Knotens. Er umfasst ein word für die Anzahl der gespeicherten Variablen und maximal *xmxj* words für die Spaltenindizes. Ein Spaltenindex ist negativ, wenn die Variable an der Lower Bound fixiert ist und positiv bei Fixierung an der Upper Bound.
- *Basis:* Der Basisbereich umfasst ein word für die Anzahl der Zeilen an dem Knoten und *(2*xn+xm)/32* words für die Basisinformationen. Die Abspeicherung erfolgt durch das Setzen der Bits in Reihenfolge der Spaltenindize. Für jede der *xm* Basisvariablen wird das Bit in der NSA gesetzt und für die *xn* Nichtbasisvariablen erfolgt keine Bitsetzung. Nach jeder Nichtbasisvariablen wird durch eine Bitsetzung in der NSA angegeben, dass diese an der Upper Bound liegt, und bei keiner Bitsetzung liegt die Nichtbasis-Variable an der Lower Bound.
- *Fraktionelle Werte:* Werden im Branch-and-Bound-Prozess Pseudo Kosten benutzt, dann werden in maximal *2*xmxj* words die fraktionellen Werte aller freien Integer-Variablen mit den dazugehörigen Spaltenindizes abgespeichert.

5.1.3 Partitionierung und Komprimierung

Die Knoten in der Knotentabelle können in Kandidatenknoten und wartende Knoten partitioniert werden. Die Kandidatenknoten werden an den Anfang der Knotentabelle geschrieben und haben den Zähler *xnnode*. Die wartenden Knoten werden an das Ende der Knotentabelle geschrieben, und der Zähler ist *xwnode*.

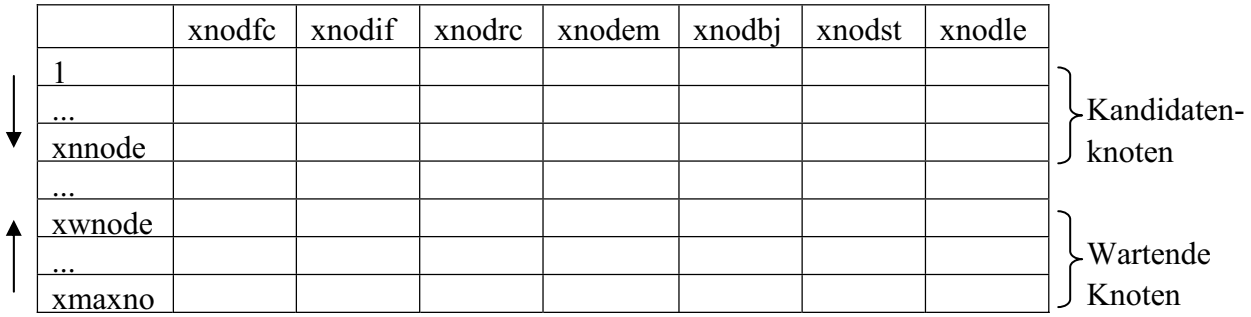


Abb. 5-4: Partitionierung der Knotentabelle

Die Knotentabelle ist voll, wenn *xnnode+1=xwnode* ist. In der Knotentabelle sind demzufolge *node=xnnode+(xmaxno+1-xwnode)* Knoten enthalten.

Beim Abspeichern eines Knotens in der Routine *xstnot* wird der Partitionierungswert *parval=xrtfc*xfunct+xrtem*xesti* für diesen Knoten berechnet, wobei *xfunct* der Funktionswert und *xesti* die Estimation des Knotens sind. Die beiden Faktoren *xrtfc* und *xrtem* mit *xrtfc+xrtem=1* werden durch den Partitionierungsparameter *xparnd* definiert und geben den Anteil des Funktionswerts und der Estimation in dem Partitionierungswert *parval* an. Die Ausprägungen von *xparnd* sind:

- *0:* keine Partitionierung
- *1:* Partitionierung nach dem Funktionswert (*xrtfc=1* und *xrtem=0*)
- *2:* Partitionierung nach dem Funktionswert und der Estimation (*xrtfc=0,5* und *xrtem=0,5*)
- *3:* Partitionierung nach der Estimation (*xrtfc=0* und *xrtem=1*)

Bei der Einstellung *xparnd<0* wird eine Partitionierung erst dann durchgeführt, wenn die Knotentabelle zur Hälfte gefüllt ist. Danach ergibt sich die Partitionierung aus |*xparnd*|.

Der Partitionierungswert *parval* wird mit dem kritischen Wert *xpkrit* verglichen. Wenn noch keine Integer-Lösung gefunden wurde und der Partitionierungswert von dem Funktionswert abhängt, dann ist *xpkrit=abs(xparlb)*(1+a*xparfa)*, ansonsten ist *xpkrit=xparlb+(xparub-xparlb)*xparfa*. *xparlb* gibt den kleinsten und *xparub* den größten Partitionierungswert aller Knoten an. Wenn die globale Lower Bound des Problems *xzlbnd≥0* ist, dann ist *a=1,* ansonsten *a=-1*. Der Partitionierungsfaktor *xparfa* gibt die Entfernung des kritischen Wertes zu dem kleinsten Partitionierungswert an und ist vom Benutzer steuerbar. Ist der Partitionierungswert *parval* an einem Knoten kleiner als der kritische Wert *xpkrit*, dann ist der Knoten ein Kandidatenknoten, ansonsten ist er ein wartender Knoten.

Die Lower und Upper Bound der Partitionierung *xparlb* und *xparub* werden bei der Komprimierung und gegebenenfalls durch den Partitionierungswert eines neuen Knotens aktualisiert. Die Knotenauswahl findet immer unter den Kandidatenknoten statt.

Die Knotentabelle muss komplett neu partitioniert werden, wenn

- das Knotenlimit *xparnl*, nachdem die letzte Partitionierung durchgeführt wurde, abgelaufen ist und der Benutzer durch *xparbb=1* angegeben hat, dass nach diesem Knotenlimit eine neue Partitionierung stattfinden soll oder

- keine aktiven Kandidatenknoten mehr vorhanden sind. Dann muss der kritische Partitionierungswert *xpkrit* neu berechnet werden, so dass die Menge der aktiven Kandidatenknoten nach der Partitionierung nicht mehr leer ist.

Die Partitionierung der Knotentabelle wird innerhalb der Komprimierung aufgerufen.

Die Komprimierung löscht alle abgearbeiteten Knoten oder Knoten mit einem schlechteren Funktionswert als der globalen Upper Bound *xzubnd* aus der Knotentabelle und der NSA. Der Parameter *xcopnd* legt fest, ob bei jedem Komprimierungsdurchlauf die Knotentabelle und die NSA (*xcopnd=0*) komprimiert werden sollen oder die NSA nur komprimiert wird, wenn sie hinreichend voll ist (*xcopnd=1*). Die Komprimierung findet in der Routine *xcomnd* statt.

Unabhängig von der Partitionierung wird die Komprimierung aufgerufen, wenn

- eine Integer-Lösung gefunden wurde, da sich dadurch die globale Upper Bound *xzubnd* verändert und somit möglicherweise Knoten gelöscht werden können,
- die Knotentabelle hinreichend voll ist oder
- die NSA hinreichend voll ist.

Die Komprimierung ist in zwei Teile aufgeteilt. Im ersten Teil erfolgt die Eliminierung aller gelöschten Knoten oder der Knoten mit einem schlechteren Funktionswert als *xzubnd* aus der Knotentabelle bzw. Node Save Area.

Dieser Vorgang wird im Pseudo Code 5-1 vereinfacht dargestellt.

Pseudo Code 5-1 Komprimierung Teil 1

```
1  for i=1 to 2
2    if i=1 then
3      Kandidatenknoten werden betrachtet
4    else
5      Wartende Knoten werden betrachtet
6    end if
7    for alle Knoten des zu betrachtenden Bereichs do
8      if Knoten kann gelöscht werden cycle
9      Verschieben des Knotens um die Anzahl der bisher gelöschten Knoten
10     Gegebenfalls Abspeicherung der Knotennummer und des Knotenzeigers für die
11     Anpassung der NSA
12   end for
13   if i=1 then
14     xnnode aktualisieren
15     if keine Partitionierung exit
16   else
17     xwnode aktualisieren
18   end if
19 end for
20 Gegebenfalls Aktualisierung der NSA
```

Wird keine Partitionierung durchgeführt (*xparnd=0*), dann werden nur die Knoten bis *xnnode* betrachtet. Ansonsten (*xparnd<>0*) wird die Komprimierung auch im Bereich der wartenden Knoten von *xwnode* bis *xmaxno* durchgeführt. Die aktiven Knoten jedes Bereichs verschieben sich in der Routine *xchpos* um die Anzahl der vorher gelöschten Knoten nach vorne. Die Zeiger *xnnode* und *xwnode* werden aktualisiert und zeigen jeweils auf den letzten aktiven Knoten des jeweiligen Knotenbereichs in der Knotentabelle.

Wenn nicht nur die Knotentabelle, sondern auch die NSA komprimiert werden soll, wird in der Routine *xconsa* bei jedem aktiven Knoten der Pointer und die Position des Knotens in der Knotentabelle in die zwei Arrays x*nodrp* und x*nodnr* gespeichert. Nachdem die Knotentabelle aktualisiert wurde, wird die NSA in der Routine *xupnsa* komprimiert. Dazu werden diese beiden Arrays in der Routine *xzsort* nach dem Pointer absteigend sortiert. Dann wird vom kleinsten Pointer ausgehend der erste gelöschte Knoten ermittelt. Ein Knoten ist gelöscht, wenn der Pointer plus der Knotenlänge ungleich dem nächsten Pointer ist. Ab diesem Knoten muss die NSA komprimiert werden. Es werden die words aller Knoten in x*nodnr* in dem Array *xnodei* (NSA) nacheinander verschoben, so dass alle Lücken geschlossen werden.

Nachdem die gelöschten Knoten aus der Knotentabelle entfernt wurden, wird im zweiten Teil der Komprimierung bei *xparnd<>0* die Knotentabelle neu partitioniert. Soll nur eine Partitionierung durchgeführt werden, dann wird der erste Teil der Komprimierung nicht ausgeführt. Wenn sich der Partitionierungswert aus der Estimation berechnet, werden in der Routine *xgtpco* die Estimation für alle Knoten aktualisiert und *xparlb* und *xparub* angepasst. Der kritische Wert *xpkrit* wird in der Routine *xcopnt* neu berechnet.
Der Pseudo Code 5-2 veranschaulicht den vereinfachten Ablauf der Partitionierung.

Pseudo Code 5-2 Komprimierung Teil 2

```
1  for alle Kandidatenknoten do
2    Berechnung des Partitionierungswertes krit des Knotens k
3    if krit>xpkrit then
4      Suche nach einem Kandidatenknoten in den wartenden Knoten
5      if Knoten gefunden then
6        Positionstausch der Knoten
7      else
8        Füge Knoten k vor den Bereich der wartenden Knoten ein
9        Schreibe letzten Knoten von Kandidatenknoten an Stelle k
10       Anpassung der Pointer
11     end if
12   end if
13 end for
```

```
14 if noch nicht betrachtete wartende Knoten then
15   for alle restlichen wartenden Knoten do
16     Berechnung des Partitionierungswertes krit des Knotens k
17     if krit≤xpkrit then
18       Füge Knoten k an den Bereich der Kandidatenknoten an
19       Schreibe letzten Knoten von wartenden Knoten an Stelle k
20       Anpassung der Pointer
21     end if
22   end for
23 end if
```

Zunächst werden alle Kandidatenknoten durchlaufen. Anhand des berechneten Partitionierungswertes *krit* wird überprüft, ob der Knoten ein Kandidatenknoten bleibt, d.h. *krit>xpkrit* ist. Falls diese Bedingung nicht eingehalten wird, muss der Knoten in den Bereich der wartenden Knoten verschoben werden. Dazu werden die wartenden Knoten von *xmaxno* an durchlaufen. Es wird überprüft, ob sich unter den wartenden Knoten ein Kandidatenknoten befindet, d.h. bei einem Knoten *krit≤xpkrit* ist. Falls so ein Knoten gefunden wird, werden die beiden Knoten in der Routine *xexpos* getauscht.
Wurde unter den wartenden Knoten kein Kandidatenknoten gefunden, dann wird der ehemalige Kandidatenknoten mit Hilfe der Routine *xchpos* vor die wartendenden Knoten geschrieben und der Pointer der wartenden Knoten um eins verringert. Um die Lücke in dem Bereich der Kandidatenknoten zu schließen, wird der letzte Kandidatenknoten mit der Routine *xchpos* an diese Stelle geschrieben und der Pointer der Kandidatenknoten um eins verringert. Nach der Betrachtung aller Kandidatenknoten werden die restlichen wartenden Knoten, die noch nicht überprüft wurden, durchlaufen. Befindet sich ein Kandidatenknoten unter diesen restlichen Knoten, dann wird dieser mit der Routine *xchpos* an den Bereich der Kandidatenknoten gefügt und der Pointer *xnnode* um eins erhöht. Die Lücke in den wartenden Knoten wird durch den letzten Knoten des Bereichs mit der Routine *xchpos* geschlossen. Der Pointer *xwnode* wird um eins erhöht.

Beim Betrachten aller Knoten werden die Partitionierungswerte *xparlb* und *xparub* aktualisiert und der beste Funktionswert aller Knoten ermittelt. Ist dieser Funktionswert besser als der beste Funktionswert im Node File, dann stellt dieser Funktionswert die neue globale Lower Bound *xzlbnd* dar.

5.1.4 Node File

Wenn durch die Komprimierung nicht genügend Knoten gelöscht werden können, so dass nach der Komprimierung die Knotentabelle oder die NSA immer noch hinreichend voll sind, dann werden bei einer vollen Knotentabelle *kk=rnode/2* und bei einer vollen NSA *kk=rnode/4* Knoten mit *rnode=xnnode+(xmaxno+1-xwnode)* in der Routine *xworno* in das Node File geschrieben.

Die Auswahl der Knoten, die in das Node File gespeichert werden, ist von der Knotenauswahlstrategie und der Partitionierung abhängig. Ist die Knotentabelle partitioniert, dann werden zunächst die wartenden Knoten betrachtet und danach folgen die Kandidatenknoten:

- *kk<Anzahl der wartenden Knoten (xmaxno–xwnode+1):* die *kk* schlechtesten wartenden Knoten werden mit der Knotenauswahlstrategie gewählt und in das Node File geschrieben
- *kk=(xmaxno–xwnode+1):* alle wartenden Knoten werden in das Node File geschrieben
- *kk>(xmaxno–xwnode+1):* alle wartenden Knoten und die fehlenden *(kk-(xmaxno-xwnode+1))* schlechtesten Knoten innerhalb der Kandidatenknoten werden in das Node File geschrieben.

Alle in das Node File geschriebene Knoten werden in der Knotentabelle gelöscht. Nach der Speicherung wird die Knotentabelle und gegebenenfalls die NSA erneut komprimiert.

Nach einer Komprimierung kann die Knotentabelle leer sein. Wenn im Node File Knoten abgespeichert sind, dann werden die besten *kk=xmaxno/3* Knoten in der Routine *xreano* anhand der Knotenauswahlstrategie aus dem Node File geladen und in der Knotentabelle und NSA abgespeichert.

Alle selektierten Knoten und die Knoten mit einem schlechteren Funktionswert als die globale Upper Bound *xzubnd* werden im Node File gelöscht.

5.2 Speicherung der erweiterten Modellklassen

Die erweiterten Modellklassen benötigen eine eigene Speicherstruktur. Das Kernstück dieser Struktur ist die Integer-Tabelle. Je nach Art der erweiterten Modellklassen sind noch weitere Datenstrukturen notwendig, in denen gegebene oder berechnete Informationen der Modellklassen abgespeichert werden.

5.2.1 Parameter und Datenstrukturen

Die Integer-Tabelle beinhaltet die Spaltenindizes und Typen aller Integer-Variablen. Die Spaltenindizes werden in der Datenstruktur *xivtab* gespeichert und kennzeichnen die Position der Integer-Variablen im Modell. In dem Array *xivtyp* sind die Nummern der jeweiligen Variablentypen der Integer-Variablen abgespeichert:

- *<0:* Die 01-Variablen von linearisierten Funktionen bekommen die negative Nummer ihrer linearisierten Funktion zugewiesen (L01-Variable).
- *0:* kontinuierliche Variable
- *1:* 01-Variable
- *2:* allgemeine Integer-Variable
- *3:* Semi-Continuous Variable (SC-Variable)
- *4:* Semi-Integer-Variable (SI-Variable)
- *5:* Partial-Integer-Variable (PI-Variable)
- *>5 und ≤5+xm:* Die SOS3-Variablen bekommen die Nummer plus fünf der SOS-Restriktion zugewiesen
- *>5+xm und ≤5+2*xm:* Die SOS1-Variablen erhalten die Nummer plus *5+xm* der SOS-Restriktion

- *>5+2*xm:* Die SOS2-Variablen bekommen die Nummer plus *5+2*xm* der SOS-Restriktion zugewiesen

Sowohl die Nummern einer linearisierten Funktion als auch die Nummern der SOS sind eindeutig.

Die Integer-Tabelle wird nach den Variablentypen partitioniert:

SC- und SI-Variable	Allg. Integer- und PI-Variable	L01-Variable	01-Variable	SOS3-Variable	SOS1- und SOS2-Variable

Abb. 5-5: Partitionierung der Integer-Tabelle

Beispiel einer Integer-Tabelle:

ivk	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*		*xntab*
xivtab	*34*	*12*	*4*	*11*	*67*	*34*	*6*	*78*	*80*	*...*	*45*
xivtyp	*3*	*4*	*3*	*5*	*2*	*-1*	*-1*	*-1*	*1*		*6*

z.B.: Die Integer-Variable an der Position 6 in der Integer-Tabelle ist x_{34} und ist eine L01-Variable aus der ersten Gruppe der linearisierten Funktionen.

Für die Integer-Tabelle und die weiteren Datenstrukturen werden folgende Parameter benötigt:

- *xmli:* Anzahl der linearisierten Funktionen
- *xmsos:* Anzahl der SOS-Restriktionen
- *xnjs3:* Anzahl der SOS3-Variablen
- *xnsc:* Anzahl der SC- und SI-Variablen
- *xntab:* Anzahl der Elemente in der Integer-Tabelle
- *xnvint:* Anzahl der Integer-Variablen ohne SOS-Variablen
- *xnvsos:* Anzahl der SOS-Variablen
- *xst01:* Startpointer der 01-Variablen
- *xstli:* Startpointer der L01-Variablen
- *xsts3:* Startpointer der SOS3-Variablen

Für die erweiterten Modellklassen werden folgende zusätzliche Datenstrukturen benötigt:

- *xcospi* speichert die Fixkosten der SC- und SI-Variablen, den Grenzwert der PI-Variablen und die Kosten der L01-Variablen. Während des Branch-and-Bounds werden die Branchingkosten von den SOS und Cliques in dem Array abgespeichert. Die Speicherung der Werte erfolgt an der Stelle j der betreffenden Variablen. Dieses Array umfasst *xnmax* Elemente.
- *xjivk* speichert für jede Integer-Variable an der Stelle j die Position dieser Variablen in der Integer-Tabelle ab. Demzufolge umfasst das Array maximal *xnmax* Elemente.

- *xlilen* speichert die Anzahl der Elemente pro linearisierter Funktion ab. Es umfasst *xmli* Elemente.
- *xliptr* speichert die Position des Anfangselements jeder linearisierten Funktion in der Integer-Tabelle ab. Das Array beinhaltet *xmli* Elemente.
- *xsolen* speichert die Anzahl der Elemente jeder SOS und weist *xmsos* Elemente auf.
- *xsoptr* speichert die Position des jeweiligen ersten Elements einer SOS in der Integer-Tabelle. Das Array enthält *xmsos* Elemente.
- *xsowsc* speichert die Gewichte der SOS-Variablen und den Grenzwert für die SC- und SI-Variablen. Die Gewichte werden an den Anfang und die Grenzwerte an das Ende des Arrays gespeichert. Die Reihenfolge der SOS- bzw. SC/SI-Variablen wird durch die Reihenfolge in der Integer-Tabelle gegeben. Das Array kann *xnmax* Elemente speichern.

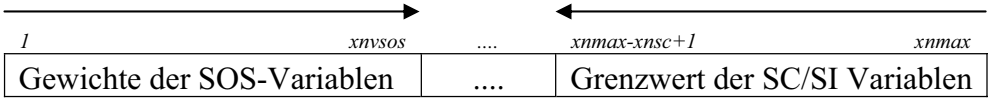


Abb. 5-6: Partitionierung von xsowsc

5.2.2 Eingabe der Werte

Über die DLL-Schnittstelle, das MPS- oder Triplet-File (siehe Anhang A) findet die Übergabe der Modelldaten an den IP-Solver statt. Bei der Angabe der Variablen legt der Benutzer den Variablentyp (siehe Kapitel 5.2.1) jeder Variablen fest.

Die Integer-Tabelle wird erst angelegt, nachdem das Anfangs-LP gelöst wurde. Da allerdings die Grenzwerte der PI-, SC- und SI-Variablen, die Fixkosten der SC- und SI-Variablen, die Gewichte der SOS-Variablen und die Variablentypen durch das Modell gegeben werden, müssen diese Werte vor dem Lösen des Anfangs-LPs abgespeichert werden. Demzufolge müssen die Arrays *xsowsc* und *xcospi* vor dem Einlesung der Daten angelegt werden, wenn bei einem MPS-File ein Scannen der Modell-Datei ergibt, dass die entsprechenden Variablentypen im Modell enthalten sind oder eine Eingabe im Triplet-Format das Vorhandensein erweiterter Modellklassen angibt. Die Variablentypen werden zunächst in dem Array *xintyp* gespeichert.

Der Benutzer hat die Wahl, in welcher Form er die Gewichte für die SOS-Variablen eingeben möchte. Die Eingabe erfolgt über die Lower Bound *xlb* der SOS-Variablen.

- *keine Eingabe:* Das Array *xlb* hat für jede SOS-Variable den Wert Null. Dann werden bei der Erstellung der Integer-Tabelle alle Elemente einer SOS in der Routine *xptwgh* absteigend gewichtet und das jeweilige Gewicht in *xsowsc* gespeichert. Bis dahin sind die Gewichte gleich Null.
- *Referenzzeile:* Der Benutzer gibt in *xlb* an der Stelle des ersten Elements der SOS den Zeilenindex der Referenzzeile an.
- *Gewichtangabe:* Für jedes Element einer SOS ist in *xlb* ein Gewicht angegeben.

Nach der Abspeicherung der Werte aus *xlb* in *xsowsc* werden die Werte in dem Array *xlb* mit Null überschrieben, da die Lower Bound jeder SOS-Variablen Null ist. Die Überprüfung, welche Variante der Benutzer für die jeweiligen SOS gewählt hat, findet in der Routine *xptref* statt. Es werden alle Werte in *xsowsc* betrachtet und überprüft, ob

- alle Gewichte gleich Null sind
- das erste Gewicht positiv und kleiner gleich *xm* ist und alle anderen Gewichte gleich Null sind. Dann wird die entsprechende Referenzzeile gesucht und die Koeffizienten der jeweiligen Variablen in *xsowsc* gespeichert. Sind Variablen der SOS nicht in der Referenzzeile enthalten, dann wird das Gewicht dieser Variablen auf Null gesetzt.
- mehrere Gewichte ungleich Null sind oder nur ein Gewicht, aber nicht an der ersten Position, ungleich Null ist. Die SOS behält die Gewichte in *xsowsc*.

In der Routine *xalitb* wird ermittelt, ob für das Problem eine Integer-Tabelle angelegt werden soll. Der Parameter *xnoitb* steuert die Anlegung. Ist der Parameter *xnoitb=1*, dann wird nur eine Integer-Tabelle angelegt, wenn sich eine Variable aus den erweiterten Modellklassen (siehe Kapital 3) im Modell befindet. Bei der Setzung von *xnoitb=0* wird die Integer-Tabelle bei Variablen aus den erweiterten Modellklassen und bei Modellen, die weniger Integer-Variablen als Zeilen enthalten, d.h. *xmxj<xm*, angelegt. Treffen die Kriterien für das Erstellen einer Integer-Tabelle zu, dann wird der Parameter *xintab=1* gesetzt. Soll keine Integer-Tabelle erstellt werden, dann ist *xintab=0*.

5.2.3 Erstellung einer Integer-Tabelle

Erst in der Initialisierung für den IP-Teil in der Routine *xiniti* wird die Integer-Tabelle angelegt. Dazu werden die Variablentypen in *xintyp* betrachtet und die Ausgangspunkte für den jeweiligen Variablenbereich in der Integer-Tabelle ermittelt.

Danach wird das Array *xintyp* durchlaufen. Jede aktive Integer-Variable j (*xintyp(j)<>0*) wird in den entsprechenden Variablenbereich der Integer-Tabelle eingetragen, *xjivk* wird mit den Positionen der Integer-Variablen in der Integer-Tabelle gefüllt und das Array *xsowsc* wird angepasst, falls durch das LP-Presolve SOS-Variablen bzw. SC/SI-Variablen inaktiviert wurden.

Außerdem werden für die SOS-Variablen und die L01-Variablen die Pointer (*xsoptr*, *xliptr*) und die Anzahl der Elemente (*xsolen*, *xlilen*) bestimmt. Dabei werden die Variablentypen neu nummeriert, damit eine zusammenhängende Nummerierung gegeben ist. Anhand des Pseudo Code 5-3 wird die Speicherung der Pointer und Elemente am Beispiel der SOS-Variablen vereinfacht dargestellt.

Pseudo Code 5-3 Erstellung von xsoptr und xsolen

```
1  xmsos=0
2  for alle Variablen do
3    if SC- oder SI-Variable then
4      Speichern der anderen Variablentypen
5    else if SOS-Variable then
6      if neue SOS-Variablengruppe then xmsos=xmsos+1
7      Speicherung der Variable in Integer-Tabelle an der Stelle k
8      if vorhergehende Variable war keine SOS-Variable then
9        Speicherung der Position in xsoptr(xmsos)
10       lens = 1
11     else if vorhergehende Var. war in der gleichen Gruppe then
12       lens=lens+1
13     else
14       xsolen(xmsos)=lens
15       Speicherung der Position in xsoptr(xmsos)
16     end if
17   end if
18   cycle
19   if vorhergehende Variable war SOS-Variable then xsolen(xmsos)=lens
20 end for
```

Bei jeder SOS-Variablen wird überprüft, ob die Variable das erste Element einer SOS ist. Das trifft zu, wenn der Variablentyp der aktuellen Variable verschieden ist von dem Variablentyp der vorangegangenen Variablen. Die Position des ersten Elements wird in *xsoptr* für die SOS an der Stelle *xmsos* abgespeichert. Ist die betrachtete Variable Nachfolger einer SOS-Variablen aus der gleichen Gruppe, dann wird nur die Anzahl der Elemente erhöht. Eine Gruppe ist vollständig aufgenommen, wenn auf eine SOS-Variable eine SOS-Variable aus einer anderen Gruppe oder keine SOS-Variable folgt. Dann wird die Anzahl der Elemente der Gruppe in *xsolen* an der Stelle *xmsos* gespeichert und gegebenenfalls eine neue SOS angelegt. Wenn die SOS-Variablen in der Integer-Tabelle nicht absteigend nach dem Gewicht sortiert sind, dann werden diese in der Routine *xssort* umsortiert

Die Abspeicherung von *xliptr* und *xlilen* erfolgt nach dem gleichen Prinzip.

Beispiel:

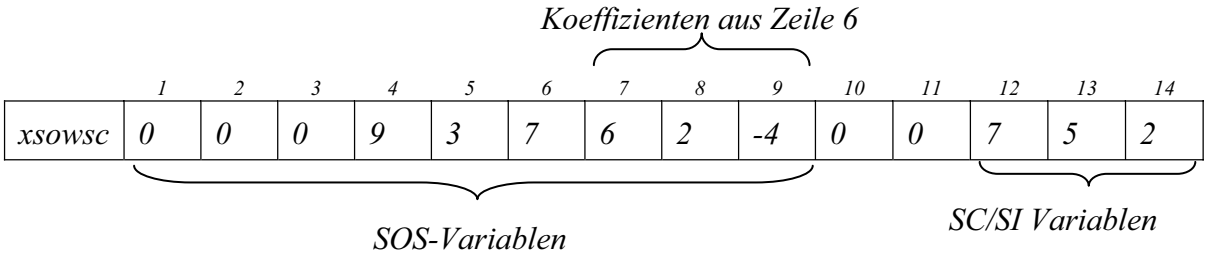
Anhand des folgenden Beispiels werden die Beziehungen der Arrays untereinander verdeutlicht.

Das Modell hat xn=14 Variablen und xm=7 Zeilen. Der Benutzer gibt über eine Schnittstelle u.a. die Variablentypen und die Lower Bound an, die in den Arrays xintyp und xlb abgespeichert sind.

SOS und SC/SI Variablen

j	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
xintyp	*6*	*6*	*6*	*3*	*3*	*14*	*14*	*14*	*4*	*0*	*22*	*22*	*22*	*0*
xlb	*0*	*0*	*0*	*2*	*5*	*9*	*3*	*7*	*7*	*0*	*6*	*0*	*0*	*0*

Beim Einlesen des Modells werden die Gewichte und die Grenzwerte der SC/SI-Variablen in dem Array xsowsc gespeichert:



Nach dem Lösen des Anfangs-LPs wird die Integer-Tabelle erstellt.

ivk	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
xivtab	*4*	*5*	*9*	*1*	*2*	*3*	*6*	*7*	*8*	*11*	*12*	*13*
xivtyp	*3*	*3*	*4*	*6*	*6*	*6*	*14*	*14*	*14*	*22*	*22*	*22*

	1	*2*	*3*
xsoptr	*4*	*7*	*10*

	1	*2*	*3*
xsolen	*3*	*3*	*3*

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
xsowsc	*2*	*1*	*0*	*9*	*7*	*3*	*6*	*2*	*-4*	*0*	*0*	*7*	*5*	*2*

Das Gewicht einer SOS-Variablen befindet sich in xsowsc an der Position ivk-xnvint und der Grenzwert einer SC/SI-Variable liegt an der Position xnmax-ivk+1 in dem Array xsowsc.

L01- und PI-Variablen

j	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
xintyp	*1*	*1*	*5*	*-1*	*-1*	*-1*	*0*	*0*	*0*	*0*	*5*	*2*	*5*	*2*
xlb	*0*	*0*	*1000*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1200*	*0*	*1300*	*0*

Beim Einlesen des Modells werden die Grenzwerte der PI-Variablen in dem Array xcospi gespeichert:

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
xcospi	*0*	*0*	*1000*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1200*	*0*	*1300*	*0*

Nach dem Lösen des Anfangs-LPs wird die Integer-Tabelle erstellt.

ivk	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
xivtab	*3*	*10*	*11*	*12*	*13*	*4*	*5*	*6*	*1*	*2*
xivtyp	*5*	*5*	*2*	*5*	*2*	*-1*	*-1*	*-1*	*1*	*1*

	1
xliptr	*6*

	1
xlilen	*3*

5.2.4 Update der Integer-Tabelle

Im Supernode Processing können Variablen fixiert oder Variablengrenzen reduziert werden. Alle fixierten Integer-Variablen bleiben im Branch-and-Bound-Prozess fixiert und werden folglich bei der Auswahl der Branchingvariablen nicht betrachtet. Demzufolge können diese in der Routine *xupint* aus der Integer-Tabelle entfernt werden. Durch die Reduktion von Variablengrenzen können Variablen ihren Typ wechseln. So sind

- allgemeine Integer-Variablen nach einer Reduktion mit den Bounds Null und Eins 01-Variablen,
- SC-Variablen mit einer reduzierten Lower Bound, die größer ist als der Grenzwert, kontinuierliche Variablen und werden aus der Integer-Tabelle entfernt,
- SI-Variablen mit einer reduzierten Lower Bound, die größer ist als der Grenzwert, allgemeine Integer-Variablen,
- L01-Variablen als einziges Element in der Gruppe 01-Variablen und
- PI-Variablen mit einer reduzierten Upper Bound, die kleiner als der Grenzwert ist, allgemeine Integer-Variablen.

Das Update ist sinnvoll, wenn eine hinreichend große Anzahl an Variablen fixiert wurde oder Variablen ihren Bereich geändert haben. Der Parameter *xudint* gibt den Anteil an, den die fixierten an den freien Integer-Variablen haben müssen, damit die Integer-Tabelle aktualisiert wird.

Beim Update der Integer-Tabelle wird jeder Variablentypbereich (siehe Kapitel 5.2.3) einzeln betrachtet. Die Variablentypbereiche sind so angeordnet, dass der Bereich, in dem die Variable mit dem neuen Variablentyp wechselt, nach dem betrachteten Bereich kommt. So werden alle Elemente von jedem Bereich nacheinander betrachtet, fixierte Variablen gelöscht und Variablen, die den Bereich wechseln, in dem Array *xbdstk* zwischengespeichert und erstmal aus der Integer-Tabelle gelöscht.

Pseudo Code 5-4 Update eines Variablentypbereichs

```
1  for k = Bereichsanfang to Bereichsende do
2    if Variable ist fixiert cycle
3    if Eigenschaften der Variable aufgehoben then
4      if Variable ist weiterhin integer then Speicherung Variable in xbdstk
5    else
6      Verschieben der Variablen an die neue Position und Änderung von xjivk
7      if SC-, SI- oder SOS-Variablen then Anpassung von xsowsc
8    end if
9  end for
```

Nachdem der Bereich abgearbeitet wurde und der Bereich des neuen Variablentyps an die Reihe kommt, werden die Elemente aus *xbdstk* in die Integer-Tabelle mit dem neuen Variablentyp geschrieben und der Pointer dieses Bereichs und *xjivk* angepasst.

Bei den L01-Variablen werden beim Durchlaufen des Bereichs die aktiven Elemente in der Gruppe gezählt. Ist nur ein Element aktiv, dann wird die Gruppe aus *xliptr* und *xlilen* gelöscht und die Variable in den Bereich der 01-Variablen geschrieben.

5.2.5 Berechnung der Kosten für die L01-Variablen

Für die L01-Variablen müssen nicht zwingend Kosten angegeben werden, da sich die Kosten auch aus den dazugehörigen kontinuierlichen Variablen berechnen lassen. Beim Erstellen der Integer-Tabelle wird überprüft, ob Kosten für die L01-Variablen angegeben wurden. Sind für mindestens eine L01-Variable Kosten angegeben, dann wird der Parameter *xlico=1* gesetzt und die Kosten werden direkt in das Array *xcospi* an der Stelle der Variablen abgespeichert.

Ansonsten werden diese in der Routine *xlicos* berechnet (siehe Pseudo Code 5-5). Ist eine L01-Variable gleich Eins, dann muss die dazugehörige kontinuierliche Variable an der Upper Bound liegen und alle vorhergehenden L01-Variablen und deren kontinuierlichen Variablen auch. Demzufolge fallen beim Setzen der L01-Variablen auf Eins die kumulierten Kosten aus den vorhergehenden L01- und kontinuierlichen Variablen an.

Pseudo Code 5-5 Berechnung der Kosten von L01-Variablen

```
1  for alle linearisierten Funktionen do
2    prcost=0.d0
3    for li=Anfang der Gruppe to Ende der Gruppe do
4      j=xivtab(li)
5      prcost=prcost+xcost(j)*xdscal(j)
6      Ermittlung der entsprechenden kontinuierlichen Variablen kj
7      prcost=prcost+xcost(kj)*xdscal(kj)
8      xcospi(j)=prcost
9    end for
10 end for
```

5.3 Implementierung des Branch-and-Bound-Prozesses

Der im Kapitel 2.2 beschriebene Branch-and-Bound-Prozess wird in der Routine *xbboun* (siehe Pseudo Code 5-6) durchgeführt. In der Routine *xslvlp* wird das relaxierte LP gelöst und alle fraktionellen Integer-Variablen ermittelt. Das Reduced Cost Fixing in der Routine *xbddjs* wird aufgerufen, wenn das relaxierte LP zulässig und nicht integer ist und schon mindestens eine Integer-Lösung gefunden wurde. Ist der Probing-Parameter *xprobl>0*, dann wird in der Routine *xprob* das Probing durchgeführt. Ist das Problem weiterhin zulässig, dann ermittelt die Routine *xbrvar* die Branchingvariable, und der Knoten wird in *xputnd* abgespeichert. Nach der Abspeicherung bzw. nach einer Unzulässigkeit oder Integer-Lösung, wird ein neuer Knoten in der Routine *xgetnd* geladen. Wenn das Knotenlimit *xmxnod*, das Zeitlimit *xmxmin*, der Gap *xglgap* erreicht wurde, kein offener Knoten mehr vorhanden ist oder kein Knoten mehr abgespeichert werden kann, dann wird der Branch-and-Bound-Prozess beendet. Ansonsten wird mit dem Lösen des relaxierten LPs des neugeladenen Knotens fortgefahren.

Pseudo Code 5-6 Branch-and-Bound-Prozess

```
1  do
2    call xslvlp
3    if gap erreicht exit
4    if Lösung ist zulässig und nicht integer then
5      if IP-Lösung vorhanden call xbddjs
6      if Problem ist zulässig then
7        call xprob
8        if Problem ist zulässig then
9          call xbrvar
10         call xputnd
11       end if
12     end if
13   end if
14   call xgetnd
15   if Abbruchkriterien erreicht exit
16 enddo
```

Die Implementierung der vier Kernbereiche Bound Reduction mit Probing und Reduced Cost Fixing, die Ermittlung der fraktionellen Variablen, die Wahl der Branchingvariablen, das Laden eines Knotens in Verbindung mit der Knotenwahl und dem Branching werden in diesem Kapitel behandelt.

5.3.1 Anwendung der Bound Reduction

Das Verfahren der Bound Reduction wurde im Kapitel 4.1 erläutert. Zunächst wird die Implementierung der Bound Reduction beschrieben, bevor es zu den Implementierungen des Reduced Cost Fixings und des Probings geht. Die Bound Reduction wird nur aufgerufen, wenn *xlotst>0* ist.

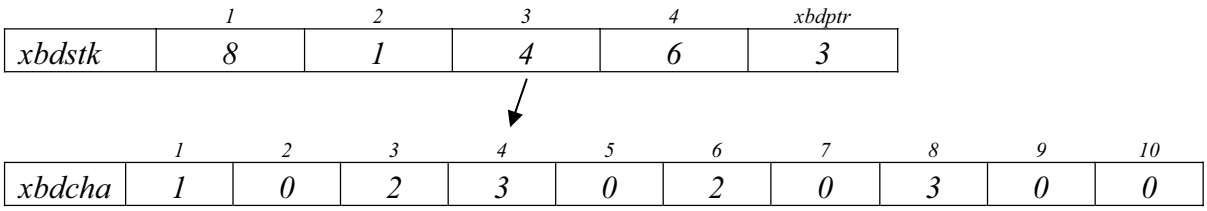
Bound Reduction

Wenn im Branch-and-Bound-Prozess ein Wertebereich reduziert bzw. eine Variable fixiert wird, dann wird für diese Variable die Bound Reduction in der Routine *xbdred* aufgerufen.

Alle eingeschränkten bzw. fixierten Variablen, für die die Bound Reduction ausgeführt werden soll, werden in dem Stack *xbdstk* gespeichert. Der Pointer für dieses Array ist *xbdptr*. Das Array *xbdcha* an der Stelle j gibt an, welche Bound der Variablen j reduziert wurde:

- *0:* keine Bound wurde reduziert
- *1:* die Lower Bound wurde reduziert
- *2:* die Upper Bound wurde reduziert
- *3:* die Lower und Upper Bound wurden reduziert

Beispiel:

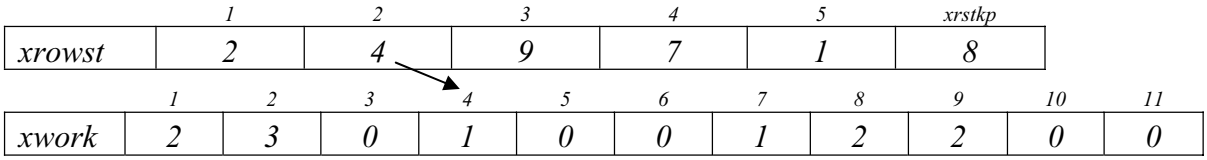


Die Variable x_4 wurde an der Lower und Upper Bound reduziert.

Für jede im Stack *xbdstk* gespeicherte Variable werden die Zeilen ermittelt, in denen diese Variable vorkommt und in den Stack *xrowst* aufgenommen. Der Pointer für diesen Stack ist *xrstkp*. In dem Array *xwork* an der Stelle i wird gespeichert, welche Zeilensumme der Zeile i sich durch die Einschränkung verändert hat:

- *0:* keine Zeilensumme hat sich verändert
- *1:* die minimale Zeilensumme hat sich verändert
- *2:* die maximale Zeilensumme hat sich verändert
- *3:* die minimale und die maximale Zeilensummen haben sich verändert

Beispiel:

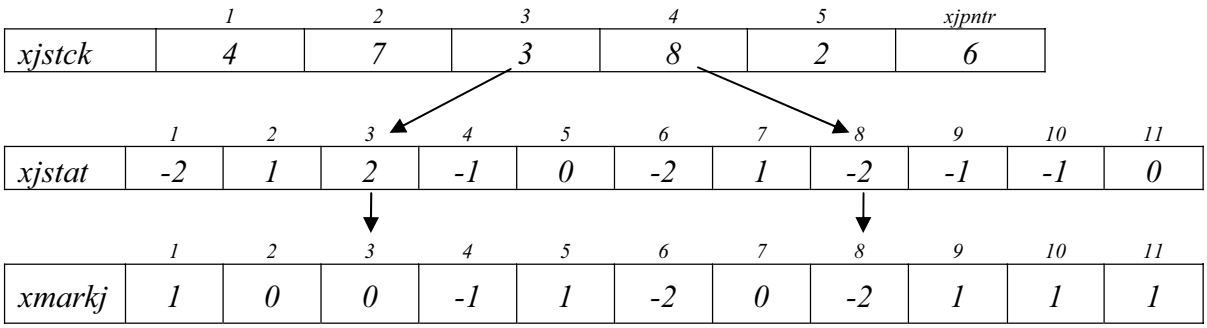


In der Zeile 4 wurde die minimale Zeilensumme verändert.

Bei einer Einschränkung bzw. Fixierung einer Variablen wird diese Variable in dem Stack *xjstck* gespeichert. Dieser Stack umfasst alle Variablen mit einem veränderten Wertebereich im Vergleich zu dem Ausgangsknoten. Der Pointer für diesen Stack ist *xjpntr*. Wird eine Variable fixiert, dann verändert sich der Status dieser Variable, der in *xjstat* gespeichert ist. Das Array *xmarkj* speichert für jede Variable in *xjstck* den alten Status, so dass zum einen *xmarkj* kennzeichnet, welche Variablen im Stack sind und zum anderen beim Zurücksetzen der Variablen *xjstat* wieder den ursprünglichen Status aus *xjmark* zugewiesen bekommt. Somit kann das Array x*markj* folgende Ausprägungen haben:

- *1:* Variable ist nicht im Stack
- *0:* 01-Variable ist im Stack
- *-1:* kontinuierliche Variable ist im Stack
- *-2:* allgemeine Integer-Variable ist im Stack

Beispiel:



Die Variable x_3 ist eine auf die Upper Bound fixierte 01-Variable mit dem Status 2. Die Variable x_8 ist eine allgemeine Integer-Variable mit eingeschränktem Wertebereich.

Der Aufruf der Bound Reduction kann für eine einzelne Variable oder für mehrere Variablen erfolgen. Soll für mehrere Variablen die Bound Reduction ausgeführt werden, dann können die Variablen entweder in *xjstck* (z.B. Aufruf nach dem Branching-Prozess) oder in *xbdstk* (Aufruf beim Reduced Cost Fixing) an die Bound Reduction übergeben werden. Beim Aufruf der Bound Reduction für eine einzelne Variable wird der Variablenindex an die Routine übergeben.

Für die übergebenden Variablen werden die entsprechenden Zeilen in der Routine *xptstk* ermittelt und in dem Array *xrowst* gespeichert. Die jeweils zu untersuchende Zeilensumme wird in dem Array *xwork* gekennzeichnet.

Da eine Reduktion von kontinuierlichen Variablen zum Teil nur weitere kontinuierliche Variablen geringfügig einschränken und keine Einschränkungen der Integer-Variablen zur Folge haben, steuert der Parameter *xlotst* die Reduktion der kontinuierlichen Variablen:

- *1:* Alle kontinuierlichen Variablen werden reduziert.
- *2:* Nur die Zeilen einer kontinuierlichen Variable mit mindestens einer freien Integer-Variable werden betrachtet. Alle anderen Zeilen werden nur überprüft, ob die Einschränkung der kontinuierlichen Variablen diese Zeile unzulässig gemacht hat oder die komplette Zeile fixiert werden kann. Diese Zeilen werden durch das Negativsetzen von dem Wert in *xwork* gekennzeichnet.
- *3:* Nur die kontinuierlichen Variablen mit einer Differenz von mindestens 0,05 zur alten Bound werden auf die neue Bound reduziert.

Bei der Bound Reduction wird für jede Zeile in *xrowst* in der Routine *xcosum* die zu betrachtende Zeilensumme berechnet. Für jede freie Variable dieser Zeile wird eine reduzierte Bound (Kapital 4.1) ermittelt. Wenn diese Zeile eine Variable mit unendlicher Bound aufweist, so dass die Zeilensumme mit dieser Variablen unendlich ist, dann wird bei der Bound Reduction nur diese Variable betrachtet. Die Berechnung der reduzierten Bound erfolgt nach der Formel der erweiterten Bound Reduction (Kapitel 4.1). Die erweiterte Bound Reduction wird nur ausgeführt, falls der Parameter *xbndrd=2* ist. Falls die berechnete Bound die alte Bound einschränkt, wird die Bound in der Routine *xbdub* bei einer neuen Upper Bound oder in der Routine *xbdlb* bei einer neuen Lower Bound auf den reduzierten Wert gesetzt. Bei Gleichheit der Variablengrenzen wird die Variable fixiert. Eine Zeile mit einer endlichen Zeilensumme wird solange durchlaufen, bis keine Vari-

ablenschranke mehr reduziert werden kann. Danach werden zu allen Variablen mit reduzierten Bounds die Zeilen in der Routine *xbndi* in den Array *xrowst* gespeichert. Ist eine Zeile unzulässig, dann wird der Branch-and-Bound Algorithmus beendet.

Der Pseudo Code 5-7 stellt diesen vereinfacht beschriebenen Ablauf der Bound Reduction für den Fall, dass sich die minimale Zeilensumme verändert hat, dar.

Pseudo Code 5-7 Bound Reduction

```
1  do xrstkp > 0
2    xrstkp = xrstkp - 1
3    Berechnung der Zeilensumme
4    if Zeile ist unzulässig exit
5    if Zeile wurde fixiert cycle
6    do
7      if minimale Zeilensumme hat sich verändert then
8        for jede Variable der Zeile i oder nur für Variable mit unendlicher Bound do
9          if Koeffizient > 0 then
10           Berechnung der UB
11           if berechnete UB schränkt alte UB ein then Bound reduzieren
12         else
13           Berechnung der LB
14           if berechnete LB schränkt alte LB ein then Bound reduzieren
15         end if
16         if Variable wurde fixiert then
17           Neuberechnung der Zeilensumme und Überprüfung
18         end if
19       end for
20     end if
21     if maximale Zeilensumme hat sich verändert then
22       Dieser Fall verläuft analog zur veränderten minimalen Zeilensumme
23     end if
24     if keine Bounds reduziert exit
25   enddo
26   if Bounds wurden reduziert then Zeilen werden in das Array xrowst geschrieben
27 enddo
```

Im Folgenden werden die Punkte der Bound Reduction detaillierter erläutert, die den Algorithmus effizienter machen.

- *Berechnung der Zeilensummen (Routine xcosum)*

  Für eine Zeile i werden in Abhängigkeit von dem jeweiligen Wert in *xwork* die entsprechenden Zeilensummen (*lrsum* und/oder *ursum*) für alle aktiven Elemente berechnet. Beinhaltet die Zeilensumme eine unendliche Bound, dann wird der Variablenindex in dem Parameter *infk1* für die minimale und in *infk2* für die maximale Zeilensumme gespeichert. Mit Hilfe dieser Parameter wird die Bound Reduction in der Zeile nur für die Variable mit unendlicher Bound ausgeführt. Die

Zeilensummen werden ohne die unendliche Bound berechnet. Falls die Zeilensumme mehr als eine unendliche Bound beinhaltet, wird diese Zeile bei der Bound Reduction nicht betrachtet. Von der rechten Seite werden jeweils die fixierten Elemente (Koeffizient*Bound) abgezogen.

Wurde die minimale Zeilensumme *lrsum* berechnet und ist die Upper Bound der betrachteten Zeile *rhsu* gleich dieser, dann kann die komplette Zeile fixiert werden. Ist allerdings *lrsum>rhsu*, dann ist die Zeile durch die Einschränkung unzulässig geworden und die Bound Reduction wird abgebrochen.

Bei der Berechnung der maximalen Zeilensumme *ursum* und einer Gleichheit mit der Lower Bound der Zeile *rshl*, können ebenfalls alle Variablen dieser Zeile fixiert werden. Wenn *ursum<rhsl* ist, dann ist die Zeile unzulässig und die Bound Reduction wird beendet.

Bei *lrsum<rhsu*, *ursum>rhsl* und einem positiven Wert in *xwork* wird mit der Bound Reduction der Zeile fortgefahren.

- *Fixieren einer Zeile (Routine xfixro)*

  Wenn *ursum=rshl* oder *lrsum=rshu* ist, dann werden alle freien Variablen dieser Zeile fixiert. Jede fixierte Variable wird in *xbdstk*, der Indikator für die veränderte Bound in *xbdcha*, die entsprechenden Zeilen in *xrowst* und der Indikator für die veränderte Zeilensumme in *xbdcha* gespeichert. Eine fixierte Variable j bzw. eine zu untersuchende Zeile i wird nur in dem Stack aufgenommen, wenn *xbdcha(j)=0* bzw. *xwork(i)=0* ist. Wenn eine Variable oder eine Zeile schon im Stack vorhanden ist, aber eine andere Bound bzw. Zeilensumme verändert wurde, dann werden die jeweiligen Indikatoren auf drei gesetzt. Die Pointer *xbdptr* und *xrstkp* werden bei neuen Einträgen erhöht.

- *Reduzieren der Bound (Routine xbdlb bzw. xbdub)*

  Die Neuberechnung einer Variablengrenze hat ergeben, dass die bestehende Bound reduziert werden kann. Wenn die Variable eine Integer-Variable ist, dann kann bei einer reduzierten Lower Bound l' die veränderte Bound zum nächsten Integerwert aufgerundet und bei einer reduzierten Upper Bound u' abgerundet werden. Falls bei der Reduction eine Unzulässigkeit entstanden ist, d.h. u'<l oder l'> u ist, dann wird die Bound Reduction beendet. Ist allerdings u'=l oder l'=u, dann kann die Variable auf den reduzierten Wert fixiert werden. Ansonsten wird die entsprechende Bound auf den reduzierten Wert gesetzt, die Variable, falls noch nicht vorhanden, in die Stacks *xbdstk* und *xjstck* aufgenommen, der Indikator für die veränderte Bound *xbdcha* angepasst und die Pointer *xbdptr* und *xjpntr* gegebenenfalls erhöht. Falls *xwork(i)=3* ist, d.h. sowohl *lrsum* als auch *ursum* untersucht werden sollen, wird die entsprechende Zeilensumme aktualisiert.

- *Neuberechnung der Zeilensumme und Überprüfung bei Fixierung einer Variablen (Routine xcrows)*

  Eine durch die Reduktion entstandene Fixierung einer Variablen auf den Wert *bound* führt zu den Änderungen der Zeilensummen *lrsum=lrsum-Koeffizient*bound* und *ursum=ursum-Koeffizeint*bound* und der rechten Seiten *rhsl=rhsl-Koeffizient*bound* und *rhsu=rhsu-Koeffzient*bound*.

Wurde bei der Reduktion die minimale Zeilensumme betrachtet, dann kann durch die Reduktion *ursum=rhsl* sein, und die komplette Zeile wird fixiert. Ist *ursum<rhsl* dann ist die Zeile unzulässig, und die Bound Reduction wird beendet.

Ist bei der Betrachtung der maximalen Zeilensumme *lrsum=rhsu*, dann wird die Zeile fixiert, und bei *lrsum>rhsu* ist die Zeile unzulässig.

Falls die komplette Zeile nicht fixiert werden kann und die Zeile durch die Fixierung auch nicht unzulässig wurde, wird die Bound Reduction bei der gleichen Zeile mit den veränderten Werten fortgesetzt.

- *Zeilen werden in xrowst geschrieben (Routine xbndi)*

  Nachdem keine Bound mehr für eine Zeile reduziert werden kann, werden für alle Variablen im Stack *xbdstk* die entsprechenden Zeilen in den Stack *xrowst* gespeichert. Anhand der veränderten Bounds, den jeweiligen Koeffizienten und dem Typ der jeweiligen Zeile wird ermittelt, welche Zeilensumme auf weitere Reduktionen untersucht werden soll. Der entsprechende Indikator wird in *xwork* gesetzt. Falls die Zeile i schon im Stack enthalten ist und die zu betrachtende Zeilensumme noch nicht gekennzeichnet wurde, dann wird *xwork(i)=3* gesetzt. Am Ende werden die Werte von *xbdcha* und der Pointer *xbdptr* zurückgesetzt.

Reduced Cost Fixing

Das Reduced Cost Fixing wird nach dem Lösen des relaxierten LPs in der Routine *xbdcos* aufgerufen. Voraussetzung dafür ist, dass schon mindestens eine Integer-Lösung gefunden wurde oder eine obere Schranke für den IP-Zielfunktionswert existiert.

Bei dem Reduced Cost Fixing werden alle freien Nichtbasis-Variablen mit reduzierten Kosten *xdjsc* ungleich Null betrachtet. Anhand der in Kapitel 4.1 beschriebenen Formel werden die reduzierten Bounds berechnet. Falls der bestehende Wertebereich eingeschränkt werden kann, werden die Variablengrenzen auf die berechneten Werte reduziert. Wenn die Variable eine Integer-Variable ist, dann kann bei einer reduzierten Lower Bound auf den nächsten Integerwert aufgerundet und bei einer reduzierten Upper Bound abgerundet werden. Falls durch die Reduktion die Upper Bound gleich der Lower Bound ist, wird die Variable fixiert. Ist allerdings die Upper Bound kleiner als die Lower Bound, dann ist der Knoten unzulässig. Die Variable mit dem veränderten Wertebereich wird in *xbdstk* und *xjstck* gespeichert, falls sie noch nicht in den Stacks enthalten ist. Der Indikator *xbdcha* wird für die veränderte Bound gesetzt. Nachdem alle Nichtbasis-Variablen betrachtet wurden, wird die Bound Reduction für alle Variablen in *xbdstk* aufgerufen. Können in der Bound Reduction weitere Variablen reduziert werden, wird für alle nicht fixierten Nichtbasis-Variablen in *xjstck* das Reduced Cost Fixing erneut durchgeführt. Wenn das betrachtete Modell ein reines oder gemischtes Binär-Modell ist, wird das Reduced Cost Fixing nach dem ersten Durchlauf beendet.

Pseudo Code 5-8 Reduced Cost Fixing

```
1  do
2    for alle zu betrachteten Variablen do
3      if Variable ist fixiert, ist Basisvariable oder hat Kosten gleich Null cycle
4      if Variable liegt an der Upper Bound then
5        Berechnung der reduzierten Lower Bound l'
6        if l<l' then Reduktion der Lower Bound und Abspeichern in xbdstk
7      else
8        Berechnung der reduzierten Upper Bound u'
9        if u<u' then Reduktion der Upper Bound Abspeichern in xbdstk
10     end if
11   end for
12   if Bounds wurden reduziert then
13     Bound Reduction für alle Variablen in xbdstk
14     if Reduktion ist zulässig, Variablen wurden reduziert und Modell hat allg.
15     Integer-Variablen cycle
16   end if
17   exit
18 end do
```

Probing

Das Probing kann innerhalb des Supernode Processings und im Branch-and-Bound-Prozess nach dem Lösen des relaxierten LPs aufgerufen werden. Das Probing mit Integer-Tabelle wird in der Routine *xprbit* und ohne in der Routine *xprobb* durchgeführt. Die Routine *xprobb* kann zusätzlich das Probing im Supernode Processing durchführen. Beim Probing im Supernode Processing werden alle 01-Variablen betrachtet und gleichzeitig die Implikationen gebildet, während bei der Anwendung im Branch-and-Bound nur ein Teil der Integer-Variablen betrachtet werden. Der hier beschriebene Algorithmus richtet sich auf die Anwendung des Probings im Branch-and-Bound-Prozess. Die Menge der betrachteten Variablen ist abhängig von dem Parameter *xprobl*:

- *0:* kein Probing im Branch-and-Bound
- *1:* Probing mit allen Integertypen
- *2:* Probing nur mit den 01-Variablen

und dem Parameter *xprobf*. Ein Probing wird nur für die Variablen ausgeführt, bei denen die Fraktionalität f bzw. (1-f) größer ist als *xprobf*, wobei $0 \leq xprobf \leq 0.5$ ist.

Das Probing-Verfahren wurde in Kapitel 4.1 dargestellt. Zur Implementierung dieses Verfahrens werden zusätzliche Arrays benötigt. Da während des Probings Variablen global für den Knoten und lokal durch das Branchen einer Variable in der Bound Reduction eingeschränkt werden können, gibt es ohne die zusätzlichen Arrays Probleme bei der Zurücksetzung der Variablen nach jedem Branch. Demzufolge muss es zwei Variablenstacks geben, die einmal die global (*xjstck*) und einmal die lokal (*xhstck*) eingeschränkten Variablen speichern und zwei Statusarrays, die den veränderten Status global (*xjmark*) und lokal

(*xpmark*) abspeichern. Dazu kommen zwei Arrays (*xsavlb, xsavub*), die den Wertebereich der Variablen (*xlb*, *xub*) vor dem Probing zugewiesen bekommen.

Wird ein Branch rückgängig gemacht, dann werden alle Variablen aus *xhstck* zurückgesetzt und bekommen den Typ aus *xpmark* und die gespeicherten Bounds aus *xsavlb* und *xsavub* zugewiesen.

Kann eine Variable beim Probing fixiert werden und werden dadurch weitere Variablen in der Bound Reduction eingeschränkt, dann werden diese Variablen, falls noch nicht vorhanden, in den globalen Arrays *xjstck* und *xmarkj* gespeichert. Die Arrays *xsavlb* und *xsavub* bekommen die neuen Bounds zugewiesen, und die lokalen Arrays werden zurückgesetzt.

5.3.2 Ermittlung der fraktionellen Variablen

Nach dem Lösen des relaxierten LPs werden im Fall einer Integer Tabelle in der Routine *xtlpin* ansonsten in *xteslp* alle fraktionellen Basis-Integervariablen betrachtet und in das Array *xlink* gespeichert. In dem Array *xfrac* werden die fraktionellen Teile der Variablen und in *xxint* der unskalierte Lösungswert jeder Variablen gespeichert.

Wie in Kapitel 4.2.1 beschrieben, wird das Branchen nach den erweiterten Modellklassen in mehrere Ebenen eingeteilt. In der Routine *xtlpin* wird die nach den Variablentypen partitionierte Integer-Tabelle durchlaufen und die einzelnen Ebenen untersucht. Die Reihenfolge ist wie folgt:

- *SOS-Variablen*: Es werden alle *xmsos* SOS-Restriktionen betrachtet und überprüft, ob sie fraktionell sind. Anhand des Variablentyps werden die Restriktionen in SOS1- und SOS3-Restriktionen und in SOS2-Restriktionen unterteilt. Bei den SOS1- und SOS3-Restriktionen müssen mindestens zwei SOS-Variablen fraktionell sein, damit die SOS fraktionell ist. Die Überprüfung findet statt, indem alle Variablen der SOS betrachtet und die fraktionellen Variablen gezählt werden. Ist die Anzahl *nfraso* größer als Eins, dann ist die SOS fraktionell.

  Die SOS2-Restriktionen sind fraktionell, wenn mehr als drei Variablen fraktionell sind oder zwei fraktionelle Variablen nicht nebeneinander liegen. Die Überprüfung erfolgt, indem die SOS durchlaufen wird. Falls eine Variable fraktionell ist, wird überprüft, ob die vorhergehende Variable auch fraktionell ist. Falls zwei Variablen hintereinander fraktionell sind, wird der Parameter *nachb=1* gesetzt. Alle fraktionellen Variablen werden gezählt und die Anzahl in *nfraso* gespeichert. Eine SOS2-Restriktion ist somit fraktionell, wenn *nfraso>2* ist oder *nfraso=2* und *nachb=0*.

  In dem Array *xlink* werden die Indizes der fraktionellen SOS-Restriktionen gespeichert. Der Parameter *xnfrso* gibt die Anzahl der fraktionellen SOS-Restriktionen an.
- *L01-Variablen:* Es werden alle L01-Variablen in der Integer-Tabelle von *xstli* bis *xst01-1* durchlaufen. Der Zähler *xnfrli* speichert die Anzahl der fraktionellen L01-Variablen.
- *01-Variablen:* Alle 01-Variablen von *xst01* bis *xnvint* werden in der Integer-Tabelle betrachtet. Der Zähler *xnfr01* wird auf die Anzahl der fraktionellen 01-Variablen gesetzt.

- *SC-und SI-Variablen:* Die SC-und SI-Variablen befinden sich am Anfang der Integer-Tabelle von Eins bis *xnsc*. Eine SC- bzw. SI-Variable an der Stelle k in der Integer-Tabelle ist fraktionell, wenn der Wert aus der relaxierten LP-Lösung zwischen Null und dem Grenzwert der Variablen liegt, welcher in *xsowsc(xnmax-k+1*) abgespeichert ist. Eine SI-Variable ist zusätzlich fraktionell, wenn sie einen fraktionellen Wert oberhalb des Grenzwertes aufweist.
- *Allgemeine Integer- und PI-Variablen:* Eine PI-Variable ist fraktionell, wenn sie einen fraktionellen Wert unterhalb des Grenzwertes in *xcospi* aufweist. Der Zähler *xnfrin* zählt die fraktionellen PI- und allgemeinen Integer-Variablen, aber auch die fraktionellen SC-und SI-Variablen.

Ohne erweiterte Modellklassen werden nur die Variablentypen 01- und allgemeine Integer-Variable unterschieden. Somit erfolgt die Wahl der Branchingvariablen je nach Strategie auf maximal zwei Ebenen. Demzufolge werden in der Routine *xteslp* die fraktionellen 01-Variablen mit dem Zähler *xnfr01* und die fraktionellen allgemeinen Integer-Variablen mit dem Zähler *xnfrin* gezählt.

5.3.3 Wahl der Branchingvariablen

Ausgehend von den Variablen in *xlink* wird die Branchingvariable nach der Branching-Strategie *xbrheu* ermittelt:

- *0:* Die fraktionelle Variable mit den größten Kosten wird gewählt. Der Lower Branch ist der erste Branch.
- *1:* Die fraktionelle Variable mit den kleinsten Kosten wird gewählt. Der Upper Branch ist der erste Branch.
- *2:* Die fraktionelle Variable mit der größten Unzulässigkeit wird gewählt. In einer Patt-Situation ist die Variable mit den kleinsten Kosten Branchingvariable. Wenn der fraktionelle Teil der Variablen j $xfrac(j) \leq 0.5$ ist, dann wird als erstes der Lower Branch ausgeführt, ansonsten der Upper Branch.
- *3:* Die fraktionelle Variable mit den größten Kosten wird gewählt. Gibt es eine Patt-Situation, dann ist die Variable mit der größten Fraktionalität Branchingvariable. Es wird immer der Lower Branch als erstes ausgeführt.
- *4:* Die fraktionelle Variable mit der größten absoluten Priorität wird als Branchingvariable gewählt. Die Prioritäten sind in *xprior* abgespeichert. Ist die Priorität negativ, dann wird der Lower Branch als erstes ausgeführt, ansonsten der Upper Branch.
- *5:* Die fraktionelle Variable mit der größten gewichteten Summe aus den Lower bzw. Upper Pseudo Kosten *xpcl* und *xpcu* wird gewählt. Ist für die fraktionelle Variable j $xpcl(j)*xfrac(j) \leq xpcu(j)*(1-xfrac(j))$, dann ist die Summe $xwpcsm*xpcl(j)*xfrac(j)+xwpcla*xpcu(j)*(1-xfrac(j))$, ansonsten $xwpcla*xpcl(j)*xfrac(j)+xwpcsm*xpcu(j)*(1-xfrac(j))$. Die Gewichte *xwpcsm* und *xwpcla* sind positive reelle Zahlen mit $xwpcsm<xwpcla$. Ist für die Branchingvariable k $xpcl(k)*xfrac(k) \leq xpcu(k)*(1-xfrac(k))$, dann wird der Lower Branch als erstes ausgeführt, ansonsten der Upper Branch.
- *6:* Für jede fraktionelle Variable wird $min(xpcl(j)*xfrac(j),xpcu(j)*(1-xfrac(j)))$ bestimmt, und die Variable mit dem größten Wert wird als Branchingvariable ge-

wählt. Bei einer Patt-Situation ist die Variable mit der größten Fraktionalität Branchingvariable. Ist für die Branchingvariable k *xpcl(k)*xfrac(k)≤ xpcu(k)*(1-xfrac(k))*, dann wird der Lower Branch als erstes ausgeführt, ansonsten der Upper Branch.

- *7:* Die fraktionelle Variable oder Clique mit den größten Kosten wird zum Branchen gewählt. Ist keine Clique in dem Problem vorhanden, dann wird die Branching-Strategie *xbrheu=3* angewandt. In einer Patt-Situation ist die Variable mit den größten Kosten die nächste Branchingvariable. Bei einer einzelnen Variablen wird der Lower Branch als erstes durchgeführt, und bei einer Clique ist die Branching-Reihenfolge davon abhängig, in welchem Teil der Clique die maximalen Kosten sind.
- *9:* Die Branchingvariable wird mit dem Logical Branching gewählt.
- *10:* Die Branchingvariable wird mit dem Strong Branching gewählt.

Ausführlich sind die Konzepte der Branchingstrategien in dem Kapitel 4.2.1 beschrieben. Der Aufruf der jeweiligen Branching-Strategie findet in der Routine *xgtvar* statt.

Die Branching-Strategien 5, 6, 7, 9 und 10 und das Branchen nach den erweiterten Modellklassen erfordern eine komplexere Implementierung und werden im Folgenden detaillierter erläutert.

Branchen mit den erweiterten Modellklassen

Vor der Wahl der Branchingvariable muss bei fraktionellen SOS (*xnfrso>0*) in der Routine *xgetwr* der Branchingpunkt bestimmt werden.

Pseudo Code 5-9 Bestimmung des Branchingpunkts einer SOS

```
1  for alle fraktionelle SOS do
2    for alle Variablen der SOS do
3      if Variable ist fixiert cycle
4      if Variable ist fraktionell then Branchinggewicht aktualisieren
5    end for
6    for alle Variablen der SOS do
7      if Variable ist fixiert cycle
8      if Gewicht des Elements ist größer als Branchinggewicht then
9        Fraktionalität des linken Teils der SOS aufaddieren
10       Maximale Kosten des linken Teils bestimmen
11     else
12       if Branchingpunkt wr noch nicht ermittelt then wr ist vorangegangene Variable
13       Fraktionalität des rechten Teils der SOS aufaddieren
14       Maximale Kosten des rechten Teils bestimmen
15     end if
16   end for
```

17 **if** SOS2 und Kosten der linken Seite kleiner als rechte Seite **then** *wr=wr+1*
18 Speicherung des Branchingpunkts in dem Array *xlink*
19 Nach den gewählten Branching-Strategien den ersten Branch bestimmen
20 Maximale Kosten in *xcospi* und Fraktionalität in *xxint* speichern
21 **enddo**

Für jede fraktionelle SOS wird ausgehend von den Gewichten in *xsowsc* das Branchinggewicht bestimmt, indem die Gewichte multipliziert mit den unskalierten Werten der Variablen in der relaxierten LP-Lösung *xx* für alle freien SOS-Variablen dieser SOS aufsummiert werden. Wenn durch das Reduced Cost Fixing keine Variable mehr fraktionell oder bei einer SOS2-Restriktion nur eine Variable fraktionell ist, dann sind die Bedingungen einer SOS nicht mehr verletzt. Der Wert in *xlink* wird auf Null gesetzt, so dass beim Branching diese Restriktion nicht mehr untersucht wird.

Zur Ermittlung des Branchingpunkts wird die SOS ein zweites Mal durchlaufen. Der Branchingpunkt liegt an der Stelle, wo das Gewicht einer SOS-Variablen gerade noch größer oder gleich dem Branchinggewicht ist. Wenn bei einer SOS2-Restriktion die Kosten des rechten Teils der SOS größer sind als die des linken Teils, dann wird der Branchingpunkt an die Stelle gelegt, wo das Gewicht das erste Mal kleiner als das Branchinggewicht ist.

Für jeden der beiden Teile werden die Fraktionalitäten und die maximalen Kosten berechnet. Der erste Branch einer SOS wird anhand der Branching-Strategie gewählt. So ist z.B. bei der Branching-Strategie *xbrheu=2* und einer größeren Fraktionalität im ersten Teil der SOS der Lower Branch der erste Branch.

In dem Array *xlink* werden die Branchingpunkte abgespeichert, in dem Array *xxint* die kumulierte Fraktionalität und in dem Array *xfrac* die Fraktionalität des linken Teils. Diese Fraktionalität wird bei der Bestimmung der Pseudo Kosten benötigt. Wird der Lower Branch als erstes ausgeführt, dann wird der Wert in *xlink* negativ gesetzt.

Nachdem für alle fraktionellen SOS der Branchingpunkt bestimmt wurde, wird die entsprechende Branching-Strategie aufgerufen.

Die Branching-Strategie 2, 3, 7, 8 bestimmen die Branchingvariablen nach den vier Ebenen SOS-, L01-, 01- und SC-, SI-, PI- und allgemeine Integer-Variablen, während die Strategien 0, 1, 5, 6 nur die Ebenen SOS und alle anderen Variablen betrachten. Die Variablenauswahl findet insgesamt auf der höchsten Ebene in der jeweiligen Branching-Strategie statt, bei der der Zähler der fraktionellen Variable dieser Ebene ungleich Null ist. Ist z.B. der Zähler der L01-Variablen *xnfrli* der erste Zähler ungleich Null, dann wird die Branchingvariable innerhalb der L01-Variablen gewählt.

Wurde keine Integer-Tabelle angelegt, dann erfolgt die Wahl der Branchingvariablen nur auf den Ebenen der 01- und der allgemeinen Integer-Variablen. Die Zähler der erweiterten Modellklassen *xnfrso* und *xnfrli* werden auf Null gesetzt.

Pseudo Kosten

Die Berechnung der Pseudo Kosten für eine Variable findet erst nach dem Branchen dieser Variablen statt (siehe Kapitel 4.2.1). Demzufolge müssen die Pseudo Kosten geeignet initialisiert werden.

Anhand des Parameters *xpcost* kann der Benutzer angeben, wie die Lower Pseudo Kosten *xpcl* bzw. die Upper Pseudo Kosten *xpcu* initialisiert werden sollen:

- *0:* Für jede Variable werden die Pseudo Kosten *xpcu* und *xpcl* gleich Null gesetzt
- *1:* Für jede Variable werden die Pseudo Kosten *xpcu* und *xpcl* gleich den Kosten der Variablen gesetzt
- *2:* Durch das teilweise Lösen eines relaxierten LPs werden die Pseudo Kosten bestimmt

Das Setzen der Pseudo Kosten auf Null oder auf die Kosten wird in der Routine *xinipc* durchgeführt. Die Initialisierung der Pseudo Kosten durch das teilweise Lösen von relaxierten LPs für jede fraktionelle Variable wird nach dem Lösen des LPs in der Routine *xslpco* aufgerufen. Der Pseudo Code 5-10 zeigt den Ablauf der Initialisierung nach *xpcost=2*.

Pseudo Code 5-10 Initialisieren der Pseudo Kosten

```
1  Speicherung der Basis
2  Berechnung der Anzahl an Iterationen in der dualen Phase 2
3  for alle fraktionellen Variablen do
4    if Variable fixiert oder wurden schon Pseudo Kosten berechnet cycle
5    for zwei Branches do
6      Erster oder zweiter Branch wird ausgeführt
7      Teilweises Lösen des relaxierten LPs
8      Setzungen rückgängig machen
9      if Problem ist zulässig then
10       Berechnung der Pseudo Kosten
11     else
12       Fixierung von Variable auf den jeweils anderen Branch und Bound Reduction
13       if Problem ist unzulässig then Initialisierung beenden
14       exit
15     end if
16   end for
17 end for
18 Laden der Basis
```

Bei der Initialisierung wird zunächst die Basis zwischengespeichert, da bei der Wahl der Branchingvariablen das LP aufgerufen und somit die Basis überschrieben wird. Die Variablengrenzen werden in die Arrays *xsavlb* und *xsavub* gespeichert, damit beim Zurücksetzen des Branches die Variablen die ursprünglichen Bounds zugewiesen bekommen.

Die Anzahl der Iterationen für das teilweise Lösen der relaxierten LPs bestimmen sich nach 20% der durchschnittlichen Iterationen, die für das Modell zum Lösen der relaxierten LPs im Laufe des Branch-and-Bounds benötigt werden.

Für jede fraktionelle nicht fixierte Variable j mit dem Indikator *xpcind(j)=0* werden die beiden Branches ausgeführt. Alle Variablen mit veränderten Bounds werden in das Array *xhstck* geschrieben. Der Marker *xpmark* speichert den Status der Variablen ab. Das rela-

xierte LP wird in der dualen Phase 2 für die Anzahl der berechneten Iterationen teilweise gelöst. Danach werden die Setzungen rückgängig gemacht, d.h. für jede Variable in dem Array *xhstck* werden die Bounds aus *xsavlb* und *xsavub* zugewiesen und der Status *xjstat* gleich dem Status aus *xpmark* gesetzt. Ist das relaxierte LP zulässig, dann werden mit dem Funktionswert die Pseudo Kosten *xpcl* bzw. *xpcu* berechnet (Formel siehe Kapitel 4.2.1). Bei einer Unzulässigkeit wird die Variable auf den anderen Branch global für den Knoten gültig eingeschränkt, d.h. alle Variablen mit veränderter Bound werden in *xjstck* gespeichert, der Marker *xjmark* auf den entsprechenden Status gesetzt und die Bound Reduction aufgerufen. Die veränderten Bounds werden in *xsavlb* und *xsavub* gespeichert. Ergibt die Bound Reduction wiederum eine Unzulässigkeit, dann ist der Knoten unzulässig und die Initialisierung wird abgebrochen. Nachdem die Pseudo Kosten für eine Variable j initialisiert wurde, wird der Indikator *xpcind(j)=1* gesetzt. Bevor die Routine verlassen wird, wird die abgespeicherte Basis geladen.

Die Initialisierung der Pseudo Kosten wird nur aufgerufen, wenn der Parameter *xpccal>0* ist, d.h. es mindestens eine fraktionelle Variable j gibt, bei der *xpcind(j)=0* ist.

Mit den berechneten Pseudo Kosten werden die Branchingvariablen in der Strategie fünf und sechs gewählt. Des Weiteren berechnet sich die Estimation eines Knotens aus den Pseudo Kosten, welche für die Partitionierung *|xparnd|>1* und die Knotenauswahlstrategien *xnodse≥5* benutzt wird.

Branchen nach Cliques

Das Verzweigen nach den Cliques findet nur statt, wenn mindestens eine Clique mindestens zehn Elemente aufweist. Demzufolge werden in der Routine *xtstcl* nach dem Supernode Processing die Anzahl der Elemente jeder Clique betrachtet. Die Variablenauswahl *xbrheu* wird auf drei gesetzt, wenn keine Clique oder nur Cliques mit weniger als zehn Elementen in dem Modell vorhanden sind.

Beim Ermitteln der fraktionellen Variablen in den Routinen *xtlpin* bzw. *xteslp* wird überprüft, ob mindestens eine fraktionelle 01-Variable in einer Clique enthalten ist, wobei Clique-Variablen durch die Bit-Setzung in *xkey* an der Stelle *xinsos* gekennzeichnet sind. Bei fraktionellen Clique-Variablen wird die Anzahl der fraktionellen 01-Variablen *xnfr01* negativ gesetzt.

Bei jeder fraktionellen Clique muss in der Routine *xgetcl* der Branchingpunkt bestimmt werden, der dadurch gekennzeichnet ist, dass in jedem der beiden Teile der Clique mindestens eine Variable fraktionell sein muss. Eine Clique ist nur zum Verzweigen geeignet, wenn sie mindestens zehn Elemente mit mindestens zwei fraktionellen Variablen aufweist.

Der Pseudo Code 5-11 stellt das Ermitteln des Branchingpunkts aller fraktionellen Cliques dar.

Pseudo Code 5-11 Bestimmung des Branchingpunkts einer Clique

```
1  for alle fraktionellen Clique-Variablen do
2    if Variable ist fixiert cycle
3    for alle Cliques der Variablen do
4      if Clique wurde schon betrachtet cycle
5      if Clique hat weniger als 10 Elemente cycle
6      for alle Elemente der Clique do
7        if Variable ist fixiert cycle
8        if Variable ist fraktionell then Branchinggewicht der Clique aktualisieren
9      end for
10     if Clique ist besser als vorherige Cliques then Clique wählen
11   end for
12   for alle Elemente der gewählten Clique do
13     if Variable ist fixiert cycle
14     if Gewicht des Elements ist größer als Branchinggewicht then
15       Fraktionalität und Kosten des linken Teils der Clique aktualisieren
16     else
17       if Branchingpunkt wr noch nicht ermittelt then vorhergehende Variable ist wr
18       Fraktionalität und Kosten des rechten Teils der Clique aktualisieren
19     end if
20   end for
21   Anhand der Kosten den ersten Branch bestimmten
22   Maximale Kosten in xcospi und die entsprechende Fraktionalität in xfrac speichern
23 end do
```

Da eine fraktionelle Variable in mehreren Cliques enthalten sein kann, muss für diese Variable eine fraktionelle Clique ausgewählt werden. Dazu wird jede Clique, die noch nicht betrachtet wurde, durchlaufen. Die Clique mit den meisten freien Variablen und in Patt-Situation mit der größten Fraktionalität wird gewählt. Beim Durchlaufen der Cliques werden die Variablen absteigend nach der Reihenfolge der Elemente in der Clique gewichtet. Das Branchinggewicht einer Clique ergibt sich aus der Kumulierung der Gewichte multipliziert mit dem fraktionellen Wert jeder Variablen.

Mit dem Branchinggewicht wird für die ausgewählte Clique der Branchingpunkt bestimmt. Dieser ergibt sich an der Stelle der Clique, wo das Gewicht der Variablen gerade noch größer bzw. gleich dem Branchinggewicht ist. Für jeden der beiden Teile einer Clique werden die maximalen Kosten bestimmt und die Fraktionalität berechnet. Die maximalen Kosten werden in dem Array *xcospi* und die Fraktionalität in *xfrac* gespeichert. Der Stack *xrowst* und der Marker *xwork* werden zweckentfremdet und speichern die Nummer der Clique bzw. den Branchingpunkt. Wurde für die fraktionelle Clique-Variable ermittelt, dass sie in keiner zum Branchen geeigneten Clique liegt, dann wird diese Variable als herkömmliche 01-Variable betrachtet. Eine Clique-Variable wird durch einen negativen Wert in *xlink* gekennzeichnet.

Nachdem jede fraktionelle Clique untersucht wurde, werden alle fraktionellen 01-Variablen und Cliques betrachtet. Die Variable bzw. Clique mit den größten Kosten wird gewählt. Für die Cliques werden die Kosten aus *xcospi* und für die 01-Variablen aus *xcost* ermittelt. Wurde eine Clique zum Verzweigen gewählt, dann erfolgt der erste Branch nach dem Teil, wo die Kosten am geringsten sind. Anstelle des Branchingwerts wird in der Variablen *xnodbb* der negative Index der Clique, nach der gebrancht werden soll, gespeichert und zur Kennzeichnung des Cliquebranchings wird der Branchingpunkt in *xjbvar* um *xn* erhöht. Soll nach einer 01-Variablen gebrancht werden, ist der Lower Branch der erste Branch.

Strong Branching

Beim Strong Branching wird, wie in Kapitel 4.2.1 beschrieben, für eine Teilmenge der fraktionellen Variablen das relaxierte LP teilweise gelöst.

Pseudo Code 5-12 Strong Branching

```
1  Speicherung der Basis
2  Berechnung des Branching-Intervalls und der Anzahl der Variablen im Intervall
3  Absteigende Sortierung der Variablen nach Fraktionalität
4  Berechnung der Anzahl an Iterationen in der dualen Phase 2
5  for alle fraktionellen Variablen do
6    if Variable fixiert cycle
7    Erster Branch ausführen mit teilweisen Lösen des relaxierten LPs
8    Setzungen rückgängig machen
9    if Problem ist unzulässig then
10     Fixierung von Variablen auf den zweiten Branch und Bound Reduction
11     if Problem ist unzulässig exit
12   end if
13   Zweiter Branch ausführen mit teilweisen Lösen des relaxierten LPs
14   Setzungen rückgängig machen
15   if Problem ist unzulässig then
16     Fixierung von Variablen auf den ersten Branch und Reduction
17     if Problem ist unzulässig exit
18   end if
19   if Summe der Funktionswerte < beste Summe then Summe ist neue beste Summe
20   if Abbruchkriterien erreicht exit
21 end for
22 Laden der Basis
```

Die Basis wird in Arrays abgespeichert, da durch das teilweise Lösen des LPs diese überschrieben wird. Die aktuellen Wertebereiche werden in die Arrays *xsavlb* und *xsavub* geschrieben. Alle fraktionellen Variablen werden absteigend nach der Fraktionalität sortiert, d.h. als erstes wird die Variable mit der höchsten Fraktionalität betrachtet. Die Auswahl findet unter den fraktionellen Variablen statt, die eine Fraktionalität von f bzw. bei f>0.5 von 1-f von mindestens 80% des größten fraktionellen Teils aller Variablen aufweisen.

Wird innerhalb dieser Variablen keine geeignete Branchingvariable gefunden, dann werden die anderen Variablen betrachtet. Der Auswahlprozess wird beendet, wenn alle Variablen im ausgewählten Intervall abgearbeitet sind bzw. eine geeignete Variable gefunden wurde.

Die Anzahl der Iterationen für das teilweise Lösen der relaxierten LPs bestimmen sich nach 10% der durchschnittlichen Iterationen, die für das Modell zum Lösen der relaxierten LPs im Laufe des Branch-and-Bounds benötigt werden.

Es wird für jede betrachtete Variable ein Lower und ein Upper Branch durchgeführt. Es wird jeweils das LP solange gelöst, bis die am Anfang berechnete Anzahl an Iterationen in der dualen Phase 2 erreicht wurde. Danach werden die Setzungen rückgängig gemacht. Wurde das LP beim Lösen unzulässig, dann kann die Variable auf den anderen Branch global fixiert werden. Tritt dabei ebenfalls eine Unzulässigkeit auf, dann ist der Knoten unzulässig. Nachdem beide Branches zulässig gelöst wurden, wird aus den beiden Funktionswerten der kleinste gewählt. Die Variable, bei der der kleinste Funktionswert maximal ist, ist die Branchingvariable für den Knoten.

Logical Branching

Nach dem Supernode Processing und bevor das Modell erneut skaliert wird, werden in der Routine *xdetli* alle Logicals überprüft, ob sie Integer sind. Eine Logical ist Integer, wenn alle Variablen in der Zeile Integer-Variablen mit ganzzahligen Koeffizienten sind. Es werden nur die Logicals betrachtet, bei der die Anzahl der Elemente in der Zeile kleiner gleich vier ist. Für diese Logicals wird in *xkey* das Bit *xinteg* gesetzt. Nach der Skalierung werden in der Routine *xsvlog* die Bounds der Logicals abgespeichert. Nachdem das LP im Branch-and-Bound gelöst wurde, werden die Logicals überprüft, ob sie fraktionell sind. Fraktionelle Logicals werden in *xlink* gespeichert und der Zähler *xnfrlo* erhöht. Ist der Zähler *xnfrlo* größer als Null, dann wird innerhalb der Logicals nach der größten Fraktionalität eine Branchingvariable gewählt. Sind keine Logicals fraktionell, dann wird die Branching-Strategie drei angewandt.

Das Logical Branching wird immer ohne Integer-Tabelle ausgeführt.

5.3.4 Knotenauswahl und Branching-Prozess

Das Branching erfolgt in der Routine *xgetnd*, nachdem ein Knoten aus der Knotentabelle geladen wurde.

Knotenauswahl

Der Status eines Knotens ist in der Spalte *xnodst* in der Knotentabelle gespeichert. Ist der Wert in *xnodst* für den aktuellen Knoten gleich minus Eins, dann wurde erst ein Branch bei dem Knoten durchgeführt und somit wird kein neuer Knoten geladen, da der zweite Branch noch ausgeführt werden muss. Ist allerdings der Status gleich Null, dann ist der Knoten abgearbeitet und ein neuer Knoten muss geladen werden.

Wird die Knotentabelle partitioniert, d.h. ist *xparnd <>0*, dann findet die Knotenauswahl immer unter den Kandidatenknoten statt. Es werden somit alle noch nicht abgearbeiteten Knoten, d.h. alle Knoten mit einem positiven Wert in der Spalte *xnodrc*, von Anfang der Knotentabelle bis zu dem Zeiger *xnnode* in die Knotenauswahl einbezogen.

Die Knotenauswahlstrategien, die im Kapitel 4.2.2 erläutert sind, werden in der Routine *xselnd* angewandt und sind durch den Parameter *xnodse* gegeben:

- *<0*: Mixed LIFO Strategie
- *0*: Einfache LIFO Strategie
- *1*: Der Knoten mit dem besten Funktionswert in *xnodfc* wird gewählt
- *2*: Der Knoten mit der geringsten Unzulässigkeit in *xnodif* wird gewählt
- *3*: Bis eine IP-Lösung gefunden wird, gilt *xnodse=2*. Danach wird der Knoten nach dem Best Projektion Prinzip nach Beale gewählt.
- *4*: Der Knoten wird nach dem originalen Best Projektion Prinzip gewählt.
- *5*: Der Knoten mit der besten Estimation basierend auf den Pseudo Kosten in *xnodem* wird gewählt.
- *6*: Bis eine IP-Lösung gefunden wird, gilt *xnodse=2.* Danach wird *xnodse=5* ausgeführt. Ist *abs(xzlbnd-xzubnd)/(1.d0+abs(xzlbnd))<0.05*, dann wird der Knoten nach dem minimalen prozentualen Fehler gewählt.
- *7*: Bis eine IP-Lösung gefunden wird, gilt *xnodse=2* und danach ist *xnodse=5.*

Ist die Knotenauswahl von der Estimation abhängig, dann wird nach 20 Knoten für alle aktiven Knoten in der Tabelle die Estimation in der Routine *xgtpco* aktualisiert, indem für alle fraktionellen Variablen an einem Knoten die Pseudo Kosten multipliziert mit der Fraktionalität aufaddiert und auf den Funktionswert in *xnodfc* gerechnet werden.

Zusätzlich kann durch den Parameter *xnodbf=1* angegeben werden, ob nach jeweils 100 Knoten der nächste Knoten mit der Best-First-Strategie gewählt werden soll. Die restlichen Knoten werden nach der Strategie *xnodse<>1* selektiert.

Beim LIFO-Verfahren wird immer der letzte Knoten in der Knotentabelle gewählt. Ist der Knoten abgearbeitet, d.h. ist der Wert des Knotens in *xnodrc* negativ, dann wird die Knotentabelle von hinten aufgearbeitet, bis ein Knoten mit einem positiven Pointer in *xnodrc* gefunden wurde. Dieser Knoten ist der Ausgangspunkt für den neuen LIFO-Strang. Sind alle Knoten abgearbeitet, dann ist der Branch-and-Bound beendet. Im LIFO-Verfahren muss die Basis nicht neu geladen werden, weil sich ein Nachfolgeknoten zum Vorgängerknoten nur durch eine weitere Einschränkung einer Variablen unterscheidet.

Das Mixed LIFO Verfahren ist abhängig von dem Parameter *xlifo* (Kapitel 4.2.2.3). Der LIFO Durchlauf startet bei *xlifo≤1* nach mindestens *xnlbab* Knoten mit der Knotenstrategie *|xnodse|*, wenn *dabs(xfunct-xzlbnd)/dabs(xzbest-xzlbnd)<xhlsgp* ist. Bei *xlifo=2* wird LIFO bei einem Gap von *dabs(xfunct-xzlbnd)/dabs(xzbest-xzlbnd)<xhligp* gestartet und der kritische Wert *xhlicv* berechnet. In der Regel ist *xhligp* kleiner bemessen als *xhlsgp*. Der Abbruch des LIFO Durchlaufs erfolgt je nach Ausprägung von *xlifo*:

- *0:* Der LIFO-Durchlauf wird beendet, wenn der betrachtete Knoten vollständig abgearbeitet ist. Es findet somit kein Backtrack statt.
- *1:* Der LIFO-Durchlauf wird abgebrochen, wenn *xnlstr* Knoten abgearbeitet sind oder der letzte Knoten in der Knotentabelle der abgearbeitete Ausgangsknoten ist.
- *2:* Wenn der Funktionswert des letzten Knotens in der Knotentabelle *xfunct* größer ist als der kritische Wert *xhlicv,* wird der LIFO-Durchlauf beendet.

Die Überprüfung, ob ein aktiver LIFO-Durchlauf beendet oder ein LIFO-Durchlauf gestartet werde soll, wird in der Routine *xbbstr* nach der Abspeicherung des Knotens durchgeführt.

Branching-Verfahren

Das Branching-Verfahren wird mit Integer-Tabelle in der Routine *xassbi* und ohne in der Routine *xassbr* durchgeführt.

Die Branchingvariable eines Knotens ist in der Spalte *xnodbj* gespeichert. In der Node Save Area im ersten *word* jedes Knotens ist der Branchingwert *xnodbb* einer Nicht-01-Variablen abgespeichert. Das Zerlegen einer Variablen bzw. Variablengruppe in zwei Teile ist abhängig von dem Variablentyp der Branchingvariablen. Bei einer Integer-Tabelle wird der Variablentyp durch das Array *xivtyp* gegeben und ohne Integer-Tabelle ergibt sich der Typ aus dem Status *xjstat* oder dem Setzen von Bits in *xkey*. Im Folgenden ist die Zerlegung der Branchingvariablen abhängig vom Variablentyp dargestellt:

- *01-Variablen*: Die Upper Bound wird beim Lower Branch auf Null und die Lower Bound beim Upper Branch auf Eins gesetzt.
- *Allgemeine Integer- und PI-Variable:* Die Upper Bound wird beim Lower Branch auf *xnodbb* und die Lower Bound beim Upper Branch auf *xnodbb+1* gesetzt.
- *SOS-Variablen:* Die Branchingvariable gibt den Branchingpunkt der SOS wieder. Das Branching wird in der Routine *xstsos* durchgeführt. Aus dem Variablentyp in *xivtyp* wird die Position *sosnr* der SOS in *xsoptr* und *xsolen* berechnet. Beim Lower Branch werden alle freien Variablen bis zum Branchingpunkt, bzw. bei einer SOS2-Restriktion alle freien Variablen vor dem Branchingpunkt und beim Upper Branch werden alle freien Variablen nach dem Branchingpunkt auf Null gesetzt.

Pseudo Code 5-13 Branchen nach einer SOS

```
1  Bestimmung der SOS-Nummer sosnr
2  if Lower Branch then
3    start=xsoptr(sosnr)
4    ende=Branchingpunkt
5    if SOS2-Variable then ende=ende-1
6  else
7    start=Branchingpunkt + 1
8    ende=xsoptr(sosnr)+xsolen(sosnr)-1
9  end if
10 for k=start to ende do
11   if Variable ist fixiert cycle
12   Variable auf Null setzen und in Stack speichern
13 end for
```

- *L01-Variablen:* Die Aufteilung der Gruppe erfolgt in der Routine *xstl01*. Der absolute Betrag des Variablentyps in *xivtyp* der L01-Variable ergibt die Gruppennummer *linr,* in der sich die L01-Variable befindet. Mit dieser Nummer werden aus *xliptr* und *xlilen* die Anfangs- und Endelemente der Gruppe ermittelt. Im Lower

Branch werden alle freien Variablen bis zur Branchingvariable auf Null und im Upper Branch alle freien Variablen ab der Branchingvariablen auf Eins gesetzt.

Pseudo Code 5-14 Branchen nach einer SOS

```
1  Bestimmung der LI-Nummer linr
2  if Lower Branch then
3    start=Branchingvariable
4    ende=letztes Element von linr
5  else
6    start=erstes Element von linr
7    ende=Branchingvariable
8  end if
9  for k=start to ende do
10   if Variable ist fixiert cycle
11   if Upper Branch then
12     Variable wird auf Eins setzen
13   else
14     Variable wird auf Null setzen
15   end if
16   Variable wird in Stack speichern
17 end for
```

- *SC- und SI-Variable:* Bei der SC- und bei der SI-Variablen mit der Position *k* in der Integer-Tabelle und einem Branchingwert *xnodbb* kleiner als der Grenzwert in *xsowsc(xnmax-k+1*) wird die Upper Bound im Lower Branch auf Null und im Upper Branch die Lower Bound auf den Grenzwert *xsowsc(xnmax-k+1)* gesetzt. Ist der Branchingwert der SI-Variable größer als der Grenzwert, dann wird im Lower Branch die Upper Bound auf *xnodbb* und im Upper Branch die Lower Bound auf *xnodbb+1* gesetzt.
- *Clique-Variable:* Der Branchingwert ist bei einer Clique-Variablen ungleich Null und gibt die Nummer der Clique an, nach der gebrancht werden soll. Ob eine Clique-Variable als 01-Variable oder als Branchingpunkt einer Clique beim Branching verwendet werden soll, hängt zusätzlich von dem Variablenindex ab. Wird nach einer Clique gebrancht, dann wurde bei der Wahl der Branchingvariablen auf den Variablenindex *xn* addiert. Die Aufteilung der Clique findet in der Routine *xstclq* statt. Beim Lower Branch werden vom Anfang der Clique bis einschließlich Branchingpunkt alle Variablen mit negativen Index auf Eins und mit positiven Index auf Null fixiert. Beim Upper Branch erfolgt das Setzen vom Ende der Clique bis zur Variablen vor dem Branchingpunkt.

Pseudo Code 5-15 Branchen nach einer Clique

```
1  if Lower Branch then
2    start=Anfang der Clique
3    ende=Ende der Clique
4    step=1
5  else
6    start=Ende der Clique
7    ende=Anfang der Clique
8    step=-1
9  end if
10 for k=start to ende do step
11   if |j|=Branchingvariable und Upper Branch exit
12   if Variable |j| ist fixiert cycle
13   if j<0 then
14     Variable wird auf Eins gesetzt
15   else
16     Variable wird auf Null gesetzt
17   end if
18   Variable wird in Stack gespeichert
19   if |j|=Branchingvariable und Lower Branch exit
20 end for
```

- *Logical Variable:* Eine Logical Variable wird durch einen Index größer als *xn* gekennzeichnet und hat die gleichen Eigenschaften wie eine Integer-Variable. Demzufolge wird die Upper Bound beim Lower Branch auf *xnodbb* und die Lower Bound beim Upper Branch auf *xnodbb+1* gesetzt.

5.4 Heuristiken

Die Heuristiken werden, wie in Kapitel 4.3 beschrieben, vor oder während des Branch-and-Bounds eingesetzt.

5.4.1 Heuristik vor dem Branch-and-Bound

Beim Einsetzen einer Heuristik vor dem Branch-and-Bound können folgende Parameter vom Benutzer definiert werden:

- *xbbheu:* Strategie zur Wahl der Branchingvariablen in der Heuristik
- *xhchbd:* Veränderung der Rundungsparameter *xhrdlb* und *xhrdub*
- *xheutp:* Angabe der Heuristik, die ausgeführt werden soll
  - 0: Keine Heuristik
  - 1: Totales Runden
  - 2: Local Branching
  - 3: Relaxation Induced Neighborhood Search

  xvrins: Variante der Relaxation Induced Neighborhood Search

0: erst Total Rounding bis IP-Lösung gefunden
1: Start der Heuristik ohne IP-Lösung

4: Relaxation-based Search Space Heuristik mit LIFO
5: Relaxation-based Search Space Heuristik
6: Relaxation-based Search Space Heuristik mit Totalem Runden

- *xhgap:* Heuristik-Gap
- *xhmxit:* Anzahl der maximalen Rundungsdurchläufe
- *xhnbfi:* Fixierung der Nicht-Basisvariablen
  0: Es werden keine Nicht-Basisvariablen fixiert
  1: Es werden Nicht-Basisvariablen fixiert
- *xhnbpt:* Prozentuale Angabe der zu fixierenden Nicht-Basisvariablen
- *xhnods:* Knotenauswahlstrategie in der Heuristik
- *xhrddi:* Angabe der Rundungsrichtung
  0: alle Variablen im unteren Rundungsintervall werden abgerundet und im oberen Rundungsintervall werden aufgerundet
  1: alle Variablen im unteren Rundungsintervall werden aufgerundet und im oberen Rundungsintervall werden abgerundet
- *xhrdlb:* Rundungsparameter für das untere Intervall
- *xhrdty:* Angabe des zu rundenden Variablentyps
  0: nur 01-Variablen im Rundungsintervall werden gerundet
  1: alle Integer-Variablen im Rundungsintervall werden gerundet
  2: alle 01-Variablen im Rundungsintervall und alle restlichen Variablen werden gerundet
- *xhrdub:* Rundungsparameter für das obere Intervall
- *xhtlim:* Maximale Zeit in der Heuristik
- *xmnheu:* Maximale Anzahl der Knoten in der Heuristik

Für die jeweiligen Heuristiken werden die Default-Werte in der Routine *xhedef* gespeichert.

Die Hauptroutine der Heuristik vor dem Branch-and-Bound ist *xheuri*. In dieser Routine werden als erstes bei *xhndfi=1* die Nicht-Basisvariablen in der Routine *xfixnb* fixiert. In dieser Prozedur werden die reduzierten Kosten *xdjsc* ungleich Null deskaliert und in *xxint* gespeichert. Der dazugehörige Spaltenindex wird in dem Stack *xhstck* zwischengespeichert. Alle Nicht-Basisvariablen im Stack werden nach den reduzierten Kosten absteigend sortiert und die *xhnbpt* Prozent der Nichtbasis-Variablen mit den größten reduzierten Kosten werden auf die Upper Bound fixiert, wenn das Bit *xatub* in *xkey* für diese Variable gesetzt wurde oder auf die Lower Bound beim Nichtsetzen dieses Bits. Nach jeder Fixierung wird die Bound Reduction aufgerufen. Führt die Setzung zu einer Unzulässigkeit, dann werden alle Fixierungen rückgängig gemacht, der Parameter *xhnbpt* um den Parameter

xhchbd reduziert und die Fixierung erneut durchgeführt. Wenn nach der Fixierung keine Unzulässigkeit besteht oder *xhnbpt* kleiner gleich Null ist, dann wird die Routine beendet.

Nach den Fixierungen werden in der Routine *xstnbb* in *xnodei* die Indizes der durch die Fixierung veränderten Variablen, deren Status aus *xmarkj* und die ursprünglichen Bounds *xlbsav* und *xubsav* abgespeichert. Die Arrays *xlbsav* und *xubsav* bekommen die veränderten Bounds *xlb* und *xub* zugewiesen.

Die jeweiligen durch *xheutp* bestimmten Heuristiken werden in der Routine *xheust* aufgerufen. Wurden in der ausgewählten Strategie Variablen gerundet bzw. eine Local Branching Restriktion eingefügt und ist das Problem danach noch zulässig, dann wird in der Routine *xbboun* der Branch-and-Bound-Algorithmus ausgeführt bis der eingeschränkte Lösungsraum durchsucht ist, das Knotenlimit *xmnheu*, das Zeitlimit *xhtlim* oder der Heuristik-Gap *xhgap* erreicht wurde. Die Änderungen werden in *xbdclr* zurückgesetzt. Falls die Abbruchkriterien nicht erreicht wurden, wird in *xgtlpn* die Ausgangsbasis neu geladen und in der Routine *xheupa* die Rundungsparameter angepasst, um dann einen neuen Heuristikdurchlauf zu starten. Die Veränderungen der Heuristikparameter in *xheupa* werden durch die jeweilige Heuristik bestimmt. Die Heuristik wird so oft durchlaufen, bis die Abbruchkriterien erreicht werden.

In der Routine *xrenbb* werden nach der Heuristik die Fixierungen der Nichtbasis-Variablen rückgängig gemacht, indem der Status und die alten Bounds für jede veränderte Variable aus *xnodei* geladen werden.

Pseudo Code 5-16 Struktur von xheuri

```
1  Lösen des relaxierten LPs
2  if xhnbfi=1 then call xfinob
3  do
4    call xheust
5    if Problem zulässig und Änderungen vorhanden then
6      call xbboun
7    end if
8    call xbdclr
9    if Abbruchkriterien erreicht exit
10   call xgtlpn
11   call xheupa
12 end do
13 if xhnbfi=1 then call xrstnb
```

5.4.2 Local Search während des Branch-and-Bounds

Für den Einsatz der Local Search während des Branch-and-Bounds sind folgende Parameter von Bedeutung:

- *xhfiva:* Gibt an, ob Variablen in den Local Search Strategien fixiert wurden bzw. aktive Nichtnull-Elemente in der Local Branching Restriktion vorhanden sind.
- *xhgap:* Heuristik-Gap
- *xhlsgp:* Gap zur Ermittlung einer guten LP-Lösung

- *xloc:* Gibt die Aktivität der Local Search im Branch-and-Bound-Algorithmus an.
  - -1: Local Search wird nicht ausgeführt, d.h. wenn xlocs=0 ist oder ein Abbruchkriterium erreicht wurde
  - 0: Local Search ist nicht aktiv
  - 1: Local Search ist aktiv
- *xlocs:* Angabe, welche Local Search ausgeführt werden soll
  - 0: keine Local Search
  - 1: Relaxation Induced Neighborhood Search
  - 2: Local Branching
  - 3: Local Rounding
  - 4: Local Total Rounding
- *xnlbab:* Knotenlimit im ursprünglichen Branch-and-Bound bevor die Local Search startet
- *xnlstr:* maximales Knotenlimit der Local Search bevor der ursprüngliche Branch-and-Bound erneut startet
- *xnostr:* Anzahl der Integer-Lösungen bevor die Local Search Heuristik abgebrochen wird

Wenn der Parameter *xloc>-1* ist, dann wird nach jedem Abspeichern eines Knotens in der Routine *xbbstr* überprüft, ob eine inaktive Local Search gestartet, eine aktive Local Search lokal oder global beendet werden kann.

Eine inaktive Local Search wird gestartet, wenn das Knotenlimit *xnlbab* erreicht wurde und ein Knoten eine LP-Lösung hat, die einen kleineren Gap als *xhlsgp* aufweist. Bevor die jeweilige durch *xlocs* gesteuerte Heuristik aufgerufen wird, werden die wichtigsten Parameter, wie z.B. der Knotentabellenpointer, abgespeichert. In der Local Search erfolgt eine Speicherung der Knoten in der Knotentabelle im Anschluss an die Kandidatenknoten aus dem ursprünglichen Branch-and-Bound. Es findet keine Partitionierung in der Local Search statt. Die Komprimierung und die Knotenwahl erfolgt nur in dem Bereich der Local Search Knoten. Dieser Bereich wird durch den Parameter *xnst* und *xnnode* gegeben, wobei *xnst* der erste Knoten aus der Local Search ist.

Falls durch die Local Search Strategie das Modell nicht eingeschränkt wurde, d.h. keine Fixierungen stattfanden oder die Local Branching Restriktion keine aktiven Elemente besitzt, wird mit dem ursprünglichen Branch-and-Bound fortgefahren, bis wieder ein Aktivierungskriterium erreicht wurde.

Eine aktive Local Search wird lokal beendet, wenn das Knotenlimit *xnlstr* erreicht wurde, der Lösungsraum durchsucht wurde, der Gap erreicht wurde oder die Knotentabelle bzw. Node Save Area voll ist. Beim Beenden der Local Search werden alle Parameter zurückgesetzt, so dass beim Laden eines neuen Knotens auch ein Knoten aus dem ursprünglichen Branch-and-Bound gewählt werden kann. Bis auf das Local Branching können alle Knoten, die in der Local Search ermittelt werden, auch in dem ursprünglichen Branch-and-Bound weiterbearbeitet werden. Die Fixierungen werden beim Laden des neuen Knotens rückgängig gemacht. Beim Local Branching wird die Local Branching Restriktion in der

Routine *xinloc* inaktiviert. Falls während der Local Search eine neue Integer-Lösung gefunden wurde, wird die Knotentabelle neu komprimiert bzw., wenn *xparnd>0* ist, neu partitioniert.

Ein globales Beenden der Local Search findet statt, wenn der Heuristik-Gap *xhgap* erreicht wurde oder mehr Integer-Lösungen als *xnostr* gefunden wurden. Dann wird der Parameter *xloc=-1* gesetzt, so dass die Local Search nicht mehr aufgerufen wird. Beim Local Branching wird die Local Branching Restriktion in *xdeloc* gelöscht.

5.4.3 RSS Heuristik

Das Einschränken des Modells durch die RSS Heuristik findet in der Routine *xrssdi* (mit Integer-Tabelle) oder in der Routine *xrssd* (ohne Integer-Tabelle) statt. Wurde eine Integer-Tabelle angelegt, dann werden alle Basis-Integer-Variablen bis zu den SOS1- und SOS2-Variablen betrachtet. Ansonsten werden alle Basis-Integer-Variablen betrachtet. Das Runden ist von dem jeweiligen Variablentyp abhängig. Der genaue Rundungsvorgang ist in Kapitel 4.3.2.1 erläutert und in dem nachfolgenden Pseudocode dargestellt.

Pseudo Code 5-17 Vereinfachter Algorithmus von xrssd

```
1  do
2    for alle freien Basisvariablen in Abhängigkeit von xhrdty do
3      if Variable liegt im Rundungsintervall then
4        Runden der Variablen und Bound Reduction
5        if Rundung unzulässig then
6          Alle Fixierungen rückgängig machen
7          if mind. zweiter Schleifendurchlauf then beende Heuristik
8          Rundungsintervalle verkleinern
9        end if
10     end if
11   end for
12   if Unzulässig bei erstem Schleifendurchlauf cycle
13   if keine Variablen wurden fixiert then
14     Rundungsintervalle vergrößern
15     if xhrdlb<0 und xhrdub>1 then beende Heuristik
16     cycle
17   end if
18   if keine weitere Variable wurde fixiert exit
19   LP lösen
```

```
20   if LP ist unzulässig then
21     Alle Fixierungen rückgängig machen
22     if mind. zweiter Schleifendurchlauf then
23       Beende Heuristik
24     else
25       Rundungsintervalle verkleinern
26       cycle
27     end if
28   end if
29   if maximale Durchlaufzahl erreicht exit
30 end do
```

In dem ersten Rundungsdurchlauf werden die Werte der relaxierten LP-Lösung nach dem Supernode Processing in *xlpsnp* betrachtet. Nach dem Runden aller infrage kommenden Basis-Variablen wird das LP gelöst. Bei den nächsten Durchläufen wird die relaxierte LP-Lösung *xx* betrachtet.

In dem Stack *xhstck* werden alle Änderungen eines Durchlaufs und in dem Array *xpmark* der Status der veränderten Variablen gespeichert, so dass bei einer Unzulässigkeit alle Änderungen eines Durchlaufs in der Routine *xhchse* rückgängig gemacht werden können. Besteht die Unzulässigkeit schon beim ersten Durchlauf, dann werden die Rundungsparameter *xhrdlb* durch *xhchbd* verkleinert und *xhrdub* durch *xhchbd* vergrößert. Sind keine Integer-Variablen im ersten Durchlauf fixiert worden, dann werden die Rundungsintervalle vergrößert, indem *xhrdlb* durch *xhchbd* vergrößert und *xhrdub* durch *xhchbd* verkleinert werden.

Ist ein Rundungsdurchlauf zulässig, dann werden in der Routine *xhbdsv* die veränderten Bounds der vorhergehenden Durchläufe gespeichert. Die Variablen aus *xhstck* werden in dem Stack *xjstck* und der lokale Status aus *xpmark* in *xjstat* gespeichert.

Das Runden der RSS-Heuristik wird beendet, wenn mindestens ein Rundungsdurchlauf ausgeführt wurde und eine Unzulässigkeit entstanden ist, keine zusätzlichen Variablen fixiert werden konnten, die Rundungsparameter so verändert wurden, dass xhrdlb<0 und xhrdub>1 oder xhrdlb > xhrdub sind oder die maximale Anzahl der Rundungsdurchläufe *xhmxit* erreicht wurde.

5.4.4 Totales Runden

Das Totale Runden (Kapitel 4.3.2.2) setzt sich ohne Integer-Tabelle aus den Routinen *xhrall* und *xhroun* und mit Integer-Tabelle aus *xhralli* und *xhroui* zusammen. Beim Runden mit Integer-Tabelle wird diese durchlaufen und ohne Integer-Tabelle werden alle *xn* Variablen betrachtet. Die jeweilige erstgenannte Routine führt das dynamische Runden durch.

Das dynamische Runden basiert immer auf der aktuellen relaxierten LP-Lösung *xx*. Alle noch nicht gerundeten Integer-Variablen werden in jedem Durchlauf in den Stack *xhstck* gespeichert. Ab dem zweiten Durchlauf werden nur die Variablen im Stack betrachtet. Können in einem Rundungsdurchlauf keine Variablen gerundet werden, dann wird der Rundungsparameter um *xhchbd* erhöht. Nach jedem erfolgreichen Rundungsdurchlauf

wird das relaxierte LP gelöst. Das Runden endet, wenn das Problem unzulässig oder integer ist.

Die Routinen *xhroun* und *xhroui* führen die Gap Rounding Strategien durch. Das Runden erfolgt immer ausgehend von der relaxierten LP-Lösung nach dem Supernode Processing *xlpsnp*. In einem Rundungsdurchgang werden alle freien Integer-Variablen gerundet bis eine Unzulässigkeit in der Bound Reduction oder durch das LP festgestellt wird oder das LP integer ist. Danach werden die Rundungsparameter *xhrdlb* um *xhchbd* erhöht bzw. *xhrdub* um *xhchbd* verringert. Nach jedem Rundungsdurchlauf werden durch die Routine *xclrbd* die Fixierungen rückgängig gemacht. Es werden insgesamt *xhmxit* Rundungsdurchläufe durchgeführt bzw. die Heuristik beendet, wenn *xhrdlb>xhrdub* ist.

5.4.5 RSS-Heuristik mit Totalem Runden

Die Kombination aus RSS-Heuristik und Totalem Runden wurde im Kapitel 4.3.2.3 beschrieben. Das Aufrufen der Rundungsheuristik während der Suche im durch die RSS-Heuristik eingeschränkten Lösungsraum erfolgt durch die Local Search-Strategie *xlocs=4*. Die Implementierung dieser Local Search Strategie wird im Kapitel 5.4.9 beschrieben. Damit es keine Korrelationen mit der eventuell auszuführenden Local Search im eigentlichen Branch-and-Bound gibt, müssen vor der Heuristik alle betreffenden Parameter (*xlocs*, *xnlbab*, *xnlstr*, *xhlsgp*) gerettet werden.

5.4.6 Relaxation Induced Neighborhood Search (RINS)

Die RINS wird vor dem Branch-and-Bound (Kapitel 4.3.2.3) in der Routine *xhrins* und während des Branch-and-Bounds (Kapitel 4.3.3) in der Routine *xrinsh* aufgerufen.

Bei dem Aufruf vor dem Branch-and-Bound mit der Variante *xvrins=0* werden zunächst die Total Rounding Strategien aufgerufen. Sobald eine IP-Lösung gefunden wurde, wird diese im Array *xintso* der RINS Routine übergeben. Bei der Variante *xvrins=1* wird der RINS Routine die originale LP-Lösung *xlpsol* übergeben.

Die übergebende Lösung wird mit der relaxierten LP-Lösung nach dem Supernode Processing *xlpsnp* verglichen. Wenn für eine Integer-Variable der Wert aus *xlpsnp* im Umkreis von *xhrdlb* um den Wert einer ganzzahligen Integer-Lösung liegt, dann kann die Variable auf diesen Wert gerundet werden. Ergibt die Bound Reduction oder das LP, nachdem alle möglichen Variablen gerundet wurden, dass das Problem unzulässig ist, dann wird der Rundungsparameter *xhrdlb* um *xhchbd* verkleinert und alle Veränderungen werden rückgängig gemacht. Die Rundungen mit dem neuen Rundungsparameter *xhrdlb* werden erneut durchgeführt. Die Heuristik endet, wenn nach allen möglichen Rundungen das Problem zulässig ist oder *xhrdlb<0* ist.

Wird der eingeschränkte Lösungsraum vorzeitig durchsucht und keine Integer-Lösung gefunden, dann wird in der Routine *xheupa* der Rundungsparameter *xhrdlb* um *xhchbd* vergrößert. Sobald allerdings eine Integer-Lösung gefunden wird, wird der Rundungsparameter auf seinen ursprünglichen Wert zurückgesetzt und diese Integer-Lösung als Vergleichslösung gewählt.

Wenn die RINS als Local Search Strategie während des Branch-and-Bounds aufgerufen wird, dann wird bei der Existenz einer Integer-Lösung der Routine *xrinsh* die IP-Lösung

xintso, ansonsten die relaxierte LP-Lösung nach dem Supernode Processing *xlpsnp* übergeben. Diese Lösung wird mit der aktuellen relaxierten LP-Lösung verglichen. Nur wenn für eine Integer-Variable beide Lösungen den gleichen ganzzahligen Wert aufweisen, wird die Variable auf diesen Wert fixiert. Deckt die Bound Reduction eine Unzulässigkeit auf, dann wird die Local Search inaktiviert und alle Änderungen rückgängig gemacht.

5.4.7 Local Branching

Wenn *xheutp=2* oder *xlocs=2* ist, wird am Ende des Supernode Processing, bevor die Skalierung stattfindet, in der Routine *xlocon* die Local Branching Restriktion angelegt. Der Zeilenindex der Restriktion wird in *xlocro* gespeichert. Die Restriktion beinhaltet alle freien Integer-Variablen mit einem Koeffizienten von 1*/(xub-xlb)* und ist nicht aktiv.

Das Local Branching wird sowohl vor als auch während des Branch-and-Bounds aufgerufen. Wenn ein dualer Simplex-Algorithmus zum Lösen des LPs verwendet wird, dann muss die Routine *xlbrdu*, ansonsten *xlbrac* aufgerufen werden. Wurde eine neue Integer-Lösung vor dem Aufruf des Local Branching gefunden, dann wird der jeweiligen Routine eine Eins zugewiesen und die Vergleichslösung ist die IP-Lösung. Ansonsten wird der Routine eine Null übergeben und als Vergleichslösung wird beim Local Branching vor dem Branch-and-Bound *xlpsnp* und beim Einsatz im Branch-and-Bound die aktuelle relaxierte LP-Lösung *xx* gewählt.

Beim Aufruf des Local Branchings wird die Local Branching Restriktion angepasst. Dazu werden alle Variablen dieser Restriktion betrachtet. Ausgehend von der jeweiligen Lösung wird eine Variable in den inaktiven Teil der Restriktion verschoben, wenn sie fixiert oder ihr Wert fraktionell ist. Eine Variable ist im aktiven Teil der Restriktion, wenn sie frei und nicht fraktionell ist. Bei einem aktiven Element muss der Koeffizient neu berechnet werden. Dieser berechnet sich aus den unskalierten Bounds *lb* und *ub* und bei einem skalierten Problem zusätzlich aus dem Skalierungsfaktor der Variablen *xdscal(j)* und der Restriktion *xdscal(xlocro+xn)*: *val=1/(ub-lb)*(xdscal(xlocro+xn)/xdscal(j))*. Liegt der Wert der Variablen an der Upper Bound, dann ist der Koeffizient *val* negativ, und auf den Parameter der rechten Seite *xlrhs* wird *ub/(ub-lb)* addiert. Ansonsten ist er positiv, und von *xlhrs* wird *lb/(ub-lb)* subtrahiert.

Eine Veränderung des Koeffizienten einer Variablen und eine Aktivierung bzw. Inaktivierung einer Variablen muss sowohl in der zeilenweisen, in der spaltenweisen und im Falle eines dualen Simplex-Algorithmus in der internen zeilenweisen Darstellung durchgeführt werden. Die spaltenweise Anpassung erfolgt beim Durchlaufen der Restriktion. Zu jeder Variablen wird die Position der Local Branching Restriktion in der spaltenweisen Darstellung ermittelt. Wechselt die Variable den Bereich oder wechselt das Vorzeichen der Variablen, dann erfolgt in der Routine *xpcloc* eine Änderung in der Spalte. Bei dem Positionstausch werden sowohl der Koeffizient und der jeweilige Zeilenindex getauscht.

- *Variable j wechselt vom aktiven in den inaktiven Bereich*: Der aktive Bereich einer Spalte ist in einen positiven und negativen Bereich partitioniert. Wenn der Koeffizient des letzten Elements des aktiven Bereichs das gleiche Vorzeichen hat, wie der Koeffizient der Zeile *xlocro*, dann werden die beiden Positionen getauscht. Sind die Vorzeichen verschieden, dann müssen die Elemente der Zeile *xlocro* erst die Position mit den letzten Elementen des positiven Bereichs tauschen. Danach wird die

Position mit den letzten aktiven Elementen getauscht. Wenn der Koeffizient von der Zeile *xlocro* positiv ist, dann verringert sich der Pointer auf den negativen Teil der Zeile um Eins. Die Anzahl der aktiven Elemente in der Spalte nimmt um Eins ab.

- *Variable j wechselt vom inaktiven in den aktiven Bereich:* Hat der Koeffizient der Zeile *xlocro* einen positiven Koeffizienten in der Spalte j, dann wechseln die Elemente der Zeile *xlocro* in den positiven Bereich. Als erstes tauschen diese Elemente die Position mit den ersten inaktiven Elementen der Spalte. Danach wird die Position mit der Position der ersten Elemente des negativen Bereichs getauscht. Der Pointer des negativen Teils erhöht sich um Eins. Weist der Koeffizient der Zeile *xlocro* einen negativen Index auf, dann wird die Position der Zeile mit der an der ersten Stelle des inaktiven Bereichs getauscht. Die Anzahl der aktiven Elemente in der Spalte wird um Eins erhöht.
- *Koeffizient der Variable j wechselt das Vorzeichen:* Wechselt die Zeile *xlocro* vom positiven in den negativen Bereich, dann wird die Position der Zeile mit der Position der letzten Zeile im positiven Bereich getauscht. Der Pointer auf den negativen Bereich wird um Eins erhöht. Weist der Koeffizient der Zeile *xlocro* einen positiven Index auf, dann tauscht die Variable die Position mit der ersten Zeile im negativen Bereich. Der Pointer wird um Eins erhöht.

Die Koeffizienten in der zeilenweisen Darstellung werden angepasst und inaktive Elemente durch einen negativen Spaltenindex gekennzeichnet. Nachdem die Restriktion durchlaufen wurde, wird die Zeile in der Routine *xprloc* in aktive und inaktive Bereiche partitioniert. Wenn der duale Simplex-Algorithmus angewendet werden soll, dann erfolgt die Partitionierung auch für die dualen Arrays. Nach der Partitionierung wird die Zeile nach den Koeffizienten absteigend sortiert.

Die Upper Bound der Restriktion wird aus der Anzahl der aktiven Elemente *nz* berechnet. Für die Anwendung des Local Branchings vor dem Branch-and-Bound wird der Parameter *kloc* auf $\lceil xhrdlb*nz \rceil$ und während des Branch-and-Bounds auf $\lceil 2*xhrdlb*nz \rceil$ gesetzt. Die Upper Bound ergibt sich dann aus *kloc – xlrhs* und wird gegebenenfalls skaliert.

Bei dem Einsatz des Local Branchings vor dem Branch-and-Bound wird in der Routine *xheupa* bei einem zu kleinen Lösungsraum und keiner neuen Integer-Lösung bei gleich bleibender Local Branching Restriktion *kloc* halbiert. Wurde eine neue Integer-Lösung gefunden, wird die Local Branching Restriktion mit dieser Lösung neu strukturiert.

Beim Einsatz während des Branch-and-Bounds wird bei jedem Aufruf des Local Branchings die Restriktion an die Lösung angepasst.

5.4.8 Local Rounding

Das Local Rounding (siehe Kapitel 4.3.3) wird als Local Search Strategie während des Branch-and-Bounds eingesetzt. In der Routine *xbbstr* wird bei dem Erreichen der Aktivierungskriterien die Routine *xlorou* aufgerufen.

Die Rundungen im Local Rounding finden auf Basis der aktuellen relaxierten LP-Lösung *xx* statt. Der Rundungsparameter *xpastr* wird vom Benutzer gesetzt und gibt den Grenzwert der Fraktionalität an, bis zu der eine freie Integer-Basis-Variable in *xh* mit einem Index

kleiner gleich *xn* gerundet werden darf. Wurden keine Variablen gerundet bzw. fixiert, dann wird der Rundungsparameter *xparst* um 0.1 erhöht. Das Runden ist beendet, wenn alle möglichen Variablen gerundet wurden, durch die Bound Reduction eine unzulässige Rundung identifiziert wurde oder *xparstr>0.5* ist. Nach dem Runden wird *xparst* auf den ursprünglichen Wert zurückgesetzt. Bei einer Unzulässigkeit oder wenn keine Variable gerundet wurde, wird die Local Search wieder inaktiviert (*xloc=0*) und alle Änderungen rückgängig gemacht.

5.4.9 Local Total Rounding

Das Local Total Rounding (siehe Kapitel 4.3.3) wird an einem ausgewählten Knoten in der Routine *xlsdfi* aufgerufen. Da nur jeweils ein Knoten in der Local Search betrachtet wird, muss der Parameter *xnlstr=1* gesetzt werden.

Die Anzahl der Knoten bevor die Local Search (*xnlbab*) aufgerufen wird, ist abhängig von der Anzahl der bis jetzt betrachteten Knoten. Je fortgeschrittener die Suche im Branch-and-Bound desto weniger soll diese Local Search aufgerufen werden. Demzufolge kann der Benutzer einen Startwert für *xnlbab* angeben, der eventuell im Laufe des Branch-and-Bounds auf *xnlbab=max(xnlbab,0.01*xnodes)* angepasst wird. Die Rundungsheuristik wird bei jedem gewählten Knoten ausgeführt. Demzufolge muss der Gap *xhlsgap=1.d0* gesetzt werden.

Der folgende Pseudo Code beschreibt das Rundungsverfahren.

Pseudo Code 5-18 Vereinfachter Algorithmus von xlsdfi

```
1  do
2    for alle noch nicht betrachteten freien Basisvariablen do
3      if Variable liegt in den Rundungsintervallen then
4        Runden der Variablen und Bound Reduction
5        if Rundung unzulässig return
6      end if
7    end for
8    if alle Basis-Variablen wurde gerundet exit
9    Rundungsintervalle werden um 0,1 vergrößert
10 end do
11 for alle freien Nichtbasis-Variablen do
12   Fixierung der Variablen auf die entsprechende Bound und Bound Reduction
13   if Rundung unzulässig return
14 end for
15 Lösung des relaxierten LPs
16 if Integer-Lösung mit besseren Funktionswert als xzubnd then
17   Aktualisierung von xzubnd und Postsolve der IP-Lösung
18 end if
```

Zunächst werden die Basis-Variablen betrachtet und anhand der Fraktionalität gerundet. Am Anfang sind die Rundungsintervalle klein, so dass die Rundung für die weniger fraktionellen bzw. nicht fraktionellen Variablen durchgeführt wird. Sobald eine Unzulässigkeit

in der Bound Reduction festgestellt wird, wird der Rundungsprozess beendet. Alle Variablen, die noch nicht gerundet werden konnten, werden in den Stack *xhstck* gespeichert. Wurden alle Variablen gerundet, dann wird der Rundungsprozess für die Basis-Variablen beendet. Ansonsten werden die Rundungsintervalle um 0,1 vergrößert, und alle Variablen in dem Stack *xhstck* werden erneut durchlaufen.

Die Fixierung der Nichtbasis-Variablen auf deren Bounds erfolgt in einem zweiten Schritt. Die Bound Reduction überprüft erneut die Fixierung auf Zulässigkeit. Sind alle Setzungen zulässig, dann wird das relaxierte LP gelöst. Ist das Ergebnis integer und der Zielfunktionswert besser als die bisherige globale Upper Bound *xzubnd*, dann stellt diese Lösung die neue globale Upper Bound dar.

Es ist keine Speicherung der Basis und kein Zurücksetzen der Wertebereiche nach dem Runden nötig, da im nächsten Schritt ein neuer Knoten mit einer neuen Basis geladen wird und die Wertebereiche zurückgesetzt werden.