dieser Technik kann beträchtliche Laufzeitverkürzungen bei der Optimierung von CSP zur Folge haben, da nutzlose, mehrfache Traversierungen gleicher Teilbäume im Entscheidungsbaum vermieden werden.

Weiterhin sind die Optimierungsverfahren so implementiert, dass durch Iteration alle besten Lösungen aufzählbar sind und je nach Einstellung entweder *monoton* oder *dichotom* die Grenzen des besten Zielfunktionswerts einschränken (s. dazu insbesondere Abschnitt 14.3).

Alle von der Klasse AbstractLabel abgeleiteten Suchverfahren erben deren Konstruktor, der dazu dient, das zu betrachtende Constraint-System festzulegen und die Suchverfahren zu initialisieren (s. Listing 9.23): Initial ist die Tiefe des Entscheidungsbaums gleich 0 (Zeile 3). Die Suche setzt gerade nicht zurück (Zeile 4) und hat noch keine optimale Lösung gefunden (Zeile 6). Ferner wird der Rücksetzpunkt erzeugt (Zeile 7), jedoch erst beim Aufsetzen der Suche wird er mittels der Methode set() gesetzt.

```
1       public AbstractLabel(CS cs) {
2           this.cs = cs;
3           level = 0;
4           isBT = false;
5           isInconsistent = false;
6           isOptimal = false;
7           entry = new ChoicePoint(cs);
8       }
```

Listing 9.23. Konstruktor der Klasse AbstractLabel

Da bei der Optimierung sukzessive nach immer besseren Lösungen gesucht wird, bedarf es hier des Rücksetzens der gesamten Suche und des Aufsetzens einer neuen Suche. Da dabei jedoch das Wissen darüber, ob eine optimale Lösung bereits gefunden wurde, nicht verloren gehen darf, wurde hierfür die Methode smartResetSet() geschaffen, die dieses leistet (vgl. Listing 9.24).

```
1       /**
2        * Setze die Suche zurück und erneut auf,
3        * wobei die Information über bereits
4        * erreichte Optimalität erhalten bleibt.
5        */
6       public void smartResetSet() {
7           boolean icons = isInconsistent;
8           boolean opt = isOptimal;
9           reset();
10          set();
11          isOptimal = opt;
12          isInconsistent = icons;
13      }
```

Listing 9.24. Rücksetzen und Wiederaufsetzen der Suche in der Klasse AbstractLabel

Die Suche nach optimalen Lösungen erfolgt dann abhängig von der konkreten Implementierung der Methode nextSolution() in einer Unterklasse von AbstractLabel(). Ohne kompatible Implementierungen der beiden Methoden storeLastSolution() und restoreLastSolution() gelten die Implementierungen in der Oberklasse AbstractLabel, die jedoch ohne Wirkung sind, da ihre Methodenrümpfe leer sind. Die Suche ist dann nicht inkrementell.

Eine Implementierung eines Optimierungsverfahren ist z. B. die Methode nextMinimalSolution() in der Klasse AbstractLabel, die bei Lösbarkeit des Problems durch monotone Reduktion möglicher Zielfunktionswerte erst eine minimale Lösung findet und bei erneuten Aufrufen weitere minimale Lösungen bestimmt, sofern solche vorhanden sind. Sie ist in Listing 9.25 dargestellt und funktioniert folgendermaßen:

Zuerst wird geprüft, ob bei einem früheren Aufruf der Methode bereits Unlösbarkeit festgestellt wurde (vgl. Zeile 15); ist dies der Fall, wird die Methode durch Rückgabe von *false* beendet (Zeile 3). Ansonsten wird nach der Bestimmung des Maximalwerts der Zielfunktion unterschieden, ob bereits bei einem früheren Aufruf der Methode eine optimale Lösung gefunden wurde oder noch durch Verbesserung zu bestimmen ist (Zeile 6). Ist dies nicht der Fall (Zeile 7), wird nach irgendeiner Lösung gesucht. Existiert keine, so wird die Methode durch Rückgabe von *false* beendet (Zeile 9). Gibt es eine, so legt der aktuelle Wert der Zielfunktion die Grenze fest, mit der anschließend nach einer besseren Lösung gesucht wird (Zeile 10). Nach Speicherung dieses Grenzwerts wird die Lösung gespeichert und dann die Suche neu aufgesetzt (Zeilen 11 und 12).

Die nachfolgende Schleife wird dann solange durchlaufen, bis eine beste Lösung gefunden wurde. Dabei wird durch ein zusätzliches Constraint erzwungen, dass die aktuell gesuchte Lösung einen Zielfunktionswert hat, der kleiner ist als der Wert der zuletzt gefundenen Lösung (vgl. Zeilen 20 und 21). Führt bereits die Propagation zu einer Inkonsistenz oder wird keine solche Lösung gefunden, so ist folglich die zuletzt gefundene Lösung eine beste Lösung (Zeile 29 bzw. Zeile 23). Wird jedoch bei Fortsetzung der Suche an der zuletzt gefundenen Lösung (Zeile 22) eine weitere Lösung gefunden, so wird diese sowie der verbesserte Zielfunktionswert abgespeichert (Zeilen 25 und 26). Die Suche wird dann in jedem Fall neu aufgesetzt (Zeile 31).

Ist Optimalität erreicht, wird der Zielfunktionswert mit dem kleinsten gefundenen Grenzwert nach oben beschränkt (Zeilen 34 und 35) und eine beste Lösung zurückgegeben. Diese ist entweder die zuletzt gefundene und wieder hergestellte Lösung, wenn zuvor nach immer besseren Lösungen gesucht wurde (Zeilen 36 bis 39) oder eine nächste beste Lösung mit minimalem Zielfunktionswert (Zeile 40). Die Beschränkung des Zielfunktionswerts darf dabei keine Inkonsistenz verursachen, sonst wäre diese Implementierung der Optimierung oder von `firstcs` fehlerhaft (Zeilen 41 bis 43).

```
1  public boolean nextMinimalSolution(Variable objective) {
2      if (isInconsistent) {
3          return false;
```

```
4       }
5       limit = objective.max();
6       boolean isImproving = !isOptimal;
7       if (!isOptimal) {
8           // Berechne ggf. eine initiale Lösung:
9           if (nextSolution()) {
10              limit = objective.max();
11              storeLastSolution();
12              smartResetSet();
13          } else { // Problem unlösbar
14              isInconsistent = true;
15              return false;
16          }
17      }
18      while (!isOptimal) { // ... verbessere:
19          try { // Suche eine bessere Lösung:
20              objective.less(limit);
21              cs.activate();
22              restoreLastSolution();
23              isOptimal = !nextSolution();
24              if (!isOptimal) {
25                  storeLastSolution();
26                  limit = objective.max();
27              }
28          } catch (InconsistencyException e) {
29              isOptimal = true;
30          }
31          smartResetSet();
32      } // Ende von while (!isOptimal)
33      try {
34          objective.lessEqual(limit);
35          cs.activate();
36          if (isImproving) {
37              restoreLastSolution();
38              return true;
39          } // ... ansonsten:
40          return nextSolution();
41        catch (InconsistencyException e) {
42          throw new Error("Das sollte nicht passieren!");
43      }
44  } // Ende von nextMinimalSolution()
```

Listing 9.25. Finden einer nächsten minimalen Lösung in der Klasse AbstractLabel

9.2.5 Modellierung und Lösung eines Optimierungsproblems

Die Verwendung der Programmbibliothek `firstcs` zeigen wir exemplarisch anhand der Beschreibung und Lösung eines Constraint-Optimierungsproblems