# CHOCO

Choco ist eine Open-Source Java-Bibliothek zum Lösen von Constraint-Problematiken (Constraint Programming). Choco erlaubt es, in einer deklarativen Art und Weise das Problem zu modellieren, indem einzelne Variablen (Integer, Boolean, Set und Real), Domänen und Constraints festgelegt werden können. Anschließend wird mithilfe von Propagierungs- und Suchalgorithmen nach einer, mehreren oder allen möglichen Lösungen gesucht.

- Choco
  - Homepage: http://www.choco-solver.org/
  - JavaDoc: http://www.choco-solver.org/apidocs/index.html
  - Tutorials: http://choco-tuto.readthedocs.io/en/latest/
  - User Guide: https://choco-solver.readthedocs.io/en/latest/
- Allgemein
  - Global Constraint Catalog: http://sofdem.github.io/
  - Sammlung bekannter Constraint-Probleme: http://csplib.org/
  - Wettbewerb zwischen CP-Solvern: http://www.minizinc.org/challenge.html

Es existiert eine Default-Suchstrategie, die man aber anpassen bzw. durch eine eigene Variante ersetzen kann, um die Suche zu optimieren. Der Standard ist eine einfache Tiefensuche über einen Zustandsgraphen. Die Zustände (also die Knoten des Graphen) enthalten alle Variablen und die entsprechend zugewiesenen Domänen. Mit jeder Zustandsänderung werden den Variablen feste Werte aus den jeweiligen Domänen zugewiesen. Anschließend werden die Constraints überprüft. Sollte der neue Zustand gegen ein Constraint verstoßen, wird vom letzten Zustand aus weiter gesucht (Backtracking). Wurden Werte erfolgreich zugewiesen, können meist mithilfe der Constraints die Domänen anderer Variablen verkleinert werden, bevor zum nächsten Zustand übergegangen wird. In einem vollständigen Suchbaum bilden die Blätter alle möglichen Lösungen ab.

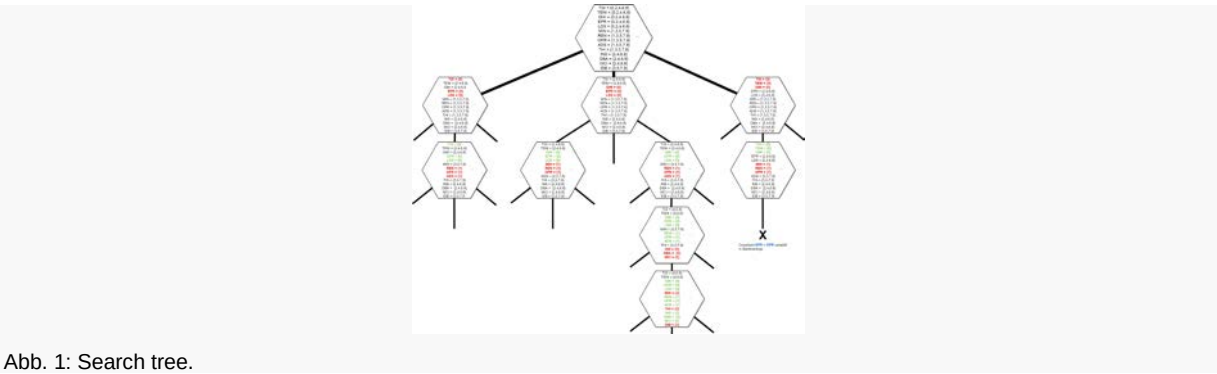

Abb. 1: Search tree.

Dies ist eine Beispielsuche, bei der bis zur Lösung 16 Knoten durchlaufen werden müssen, wobei es zu 4 Backtracks (Rückzügen) kommt:

```
.... [1/2] TMP_1064 = {1}  // TGI = {0,1,2,3,4,5...,6} TENI = {0,1,2,3,4,5...,8} GMI = {0,1,2,3,4} EPR = {0,1,2,3,4} LDS =
    /!\ CONTRADICTION (PropIntCstNotMemberSet(scheduledCourses[9]), scheduledCourses[9] = [{27}, {26, 27}]) :
.... [2/2] TMP_1064 \ {1}  // TGI = {0,1,2,3,4,5...,6} TENI = {0,1,2,3,4,5...,8} GMI = {0,1,2,3,4} EPR = {0,1,2,3,4} LDS =
..... [1/2] TMP_365 = {1}  // TGI = {0,1,2,3,4} TENI = {0,1,2,3,4,5...,6} GMI = {0,1,2,3,4} EPR = {0,1,2,3,4} LDS =
...... [1/2] not(TMP_163) = {1}  // TGI = {0,1,2,3,4} TENI = {0,1,2,3,4,5...,6} GMI = {0,1,2,3,4} EPR = 0 LDS = {0,1,2,3,4}
    /!\ CONTRADICTION (PropScale(TMP_15, TMP_16), TMP_15 = 0) : new upper bound is lesser than lower bound
....... [2/2] not(TMP_163) \ {1}  // TGI = {0,1,2,3,4} TENI = {0,1,2,3,4,5...,6} GMI = {0,1,2,3,4} EPR = 0 LDS = {0,1,2,3,4}
........ [1/2] TMP_15 = {0}  // TGI = {0,1,2,3,4} TENI = {0,1,2,3,4,5...,6} GMI = {0,1,2,3,4} EPR = 0 LDS = {0,2,3,4} MIN =
......... [1/2] not(TMP_699) = {1}  // TGI = {0,1,2,3,4} TENI = {0,1,2,3,4,5...,6} GMI = {0,1,2,3,4} EPR = 0 LDS = 0 MIN
.......... [1/2] not(TMP_1061) = {1}  // TGI = {0,1,2,3,4} TENI = {0,1,2,3,4,5...,6} GMI = {0,1,2} EPR = 0 LDS = 0 MIN =
........... [1/2] not(TMP_446) = {1}  // TGI = {0,1,2,3,4} TENI = {0,1,2,3,4,5...,6} GMI = {0,1,2} EPR = 0 LDS = 0 MIN =
............ [1/2] |TMP_24| = {1}  // TGI = {0,1,2,3,4} TENI = {0,1,2,3,4,5...,6} GMI = {0,1,2} EPR = 0 LDS = 0 MIN =
............. [1/2] not(TMP_974) = {1}  // TGI = 4 TENI = {4,5,6} GMI = 0 EPR = 0 LDS = 0 MIN = {1,3,4,5} REN = {1,3,4,5} OPR
.............. [1/2] TMP_704 = {1}  // TGI = 4 TENI = {4,5,6} GMI = 0 EPR = 0 LDS = 0 MIN = {1,3,4,5} REN = {1,3,4,5} OPR = 1
............... [1/2] not(TMP_541) = {1}  // TGI = 4 TENI = {4,5,6} GMI = 0 EPR = 0 LDS = 0 MIN = {1,3,4,5} REN = {1,3,4,5}
................ [1/2] not(TMP_572) = {1}  // TGI = 4 TENI = {4,5,6} GMI = 0 EPR = 0 LDS = 0 MIN = {1,3,4,5} REN = {1,3,4,5}
................. [1/2] not(TMP_553) = {1}  // TGI = 4 TENI = 4 GMI = 0 EPR = 0 LDS = 0 MIN = {1,3,5} REN = {1,3,4,5} OPR = 1
    /!\ CONTRADICTION (WM2 =/= 5, WM2 = 5) : remove last value
.................. [2/2] not(TMP_553) \ {1}  // TGI = 4 TENI = 4 GMI = 0 EPR = 0 LDS = 0 MIN = {1,3,5} REN = {1,3,4,5} OPR = 1
................... [1/2] not(TMP_982) = {1}  // TGI = 4 TENI = 4 GMI = 0 EPR = 0 LDS = 0 MIN = {1,3} REN = {1,3} OPR = 1 ADS =
    /!\ CONTRADICTION (PropSumOfElements(scheduledCourses[1], TMP_0[1]), TMP_0[1] = {15,16}) : new lower bound is greater
.................. [2/2] not(TMP_982) \ {1}  // TGI = 4 TENI = 4 GMI = 0 EPR = 0 LDS = 0 MIN = {1,3} REN = {1,3} OPR = 1 ADS =
................... [1/2] |TMP_39| = {0}  // TGI = 4 TENI = 4 GMI = 0 EPR = 0 LDS = 0 MIN = {1,3} REN = {1,3} OPR = 1 ADS = 1
```

# Implementierungsversuche: Studienverlaufsplan

Das Ziel dieses Testlaufs ist es, anhand verschiedener Kriterien einen Studienverlaufsplan automatisch erstellen zu lassen.

**Annahme:**
Der Nutzer kann maximal 3 Module pro Semester absolvieren (z.B. `maxCoursesPerTerm = 3`). Wie viele Semester muss er studieren und wie sieht ein möglicher Plan aus?

# 1. Versuch

## Variablen & Domänen

Der erste Versuch, das Problem zu modellieren, ist der offensichtlichste. Jedes Modul ist eine Variable. Die Domänen entsprechen den Indizes des Verlaufsplans (s. Abb. 2). Da die Domänen eine obere Begrenzung benötigen, muss eine maximale Anzahl an zu studierenden Fachsemestern vorgegeben werden (z.B. `maxTerms = 10`).

## Constraints

Es muss zunächst sichergestellt werden, dass die Verlaufsplanindizes jeweils nur einmal an die Module vergeben werden (`alldiff(V)`). Weiterhin können sehr leicht Modulabhängigkeiten sowie SS/WS-Einschränkungen festgelegt werden. Dabei liefert die Formel `Index/coursesPerTerm` das Fachsemester, das dem Modul zugeordnet wird (z.B. `(OPR/coursesPerTerm) > (EPR/coursesPerTerm)`). Ungerade Fachsemester können als Sommersemester und gerade als Wintersemester interpretiert werden (z.B. `(OPR/coursesPerTerm) mod 2 = 1`).

Die Berechnung der Credit-Points stellt mit diesem Modell ein Problem dar (s. nächste Versuche).

## Optimierung

Eine zu berechnende Lösung sollte dahingehend optimiert werden, dass der Studienverlauf möglichst kompakt geplant wird, sodass nur das Minimum an Fachsemestern zu den gegebenen Kriterien absolviert werden muss.

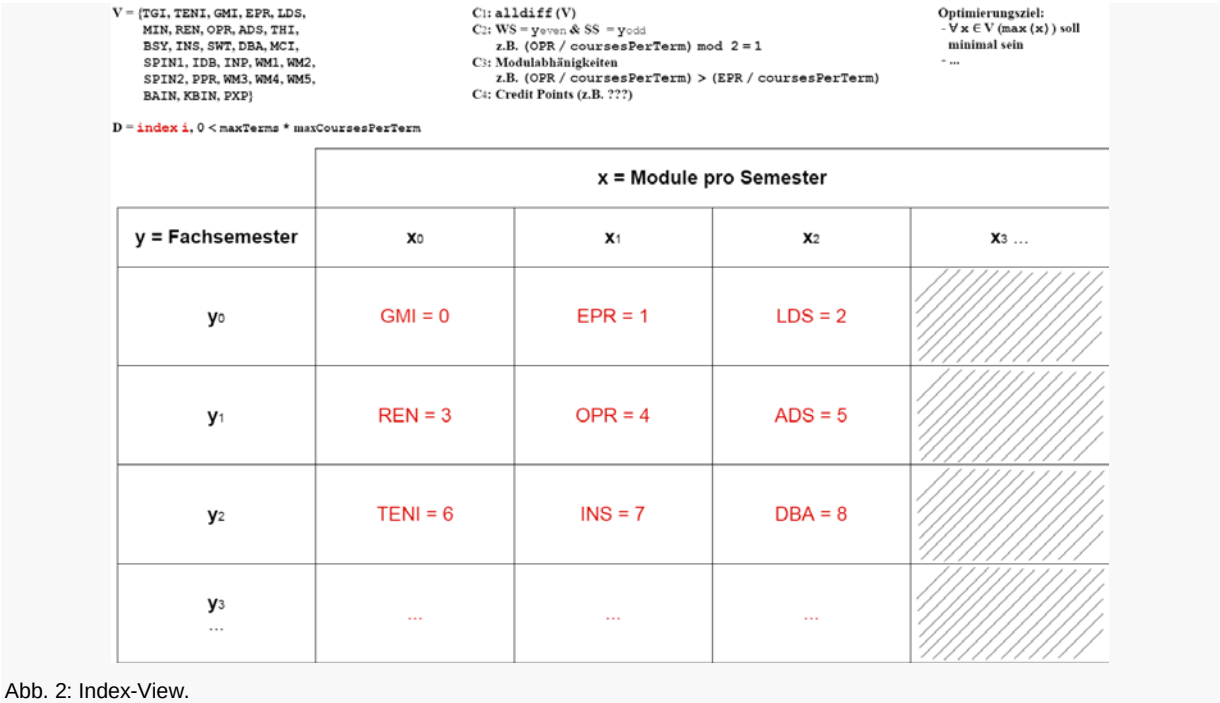

Abb. 2: Index-View.

## Eine mögliche Lösung

```
Model[CourseScheduling_IndexBased], 1 Solutions, Resolution time 0,051s, 42 Nodes (823,7 n/s),
13 Backtracks, 13 Fails, 0 Restarts
Variables: 324
Constraints: 435
 1. Sem:   LDS     EPR     GMI
 2. Sem:   OPR     ADS     MIN
 3. Sem:   DBA     MCI     SWT
 4. Sem:   SPIN1   INP     WM1
 5. Sem:   SPIN2   TGI     INS
 6. Sem:   IDB     REN     THI
 7. Sem:   TENI    BSY     PPR
 8. Sem:   PXP     WM2     WM3
 9. Sem:   WM4     WM5
10. Sem:   BAIN    KBIN
```

Dass die CP-Voraussetzungen in dieser Lösung eingehalten werden, ist reiner Zufall. Es dürfte auch Lösungen geben, bei denen diese Voraussetzungen nicht eingehalten werden. Ein Problem stellt zudem die Ungleichverteilung des Arbeitsaufwands dar. Module sind anhand des Arbeitsaufwandes bzw. der Credit-Points unterschiedlich gewichtet. Im 8. Semester muss der Student nach dem oben berechneten Verlaufsplan die Module PXP, WM2 und WM3 absolvieren. Das ergibt in Summe 28 Credit-Points, was dem durchschnittlichen Arbeitsaufwand von vier bis fünf Modulen entspricht.

**Umstellung 1:**
Beschränkung sollte statt Module/Semester anhand der CP erfolgen. Drei Module pro Semester entsprechen genau 18 Credit-Points pro Semester. Da das nicht immer unmöglich ist, erfolgt eine Annäherung: 15-21 Credit-Points (wahlweise 15-18 CP).

Eine perfekte Optimierung ist in diesem Modell nicht möglich, da nicht alle Lösungen berechnet werden können (der Java-Heap läuft selbst mit über 3GB über). Das liegt zum einen an den fehlenden CP-Constraints und zum anderen an der Tatsache, dass die Domänen unnötig groß sind (die Reihenfolge unter den Modulen pro Semester ist durch die einzelnen Indizes nicht irrelevant). Beide Sachverhalte sorgen dafür, dass es zu viele mögliche Lösungen gibt. [Anmerkung: Theoretisch wäre eine Optimierung der ersten 100.000 Lösungen möglich.]

**Umstellung 2:**
Statt Indizes sollten die Domänen nur die Zeilennummern bzw. die Nummer des Fachsemesters enthalten und nicht die einzelnen Platzierungen im Semester.

# 2. Versuch

## Variablen & Domänen

Jedes Modul ist eine Variable. Die Domänen entsprechen den Fachsemesternummern (angefangen mit 0). Da die Domänen eine obere Begrenzung benötigen, muss eine maximale Anzahl an zu studierenden Fachsemestern vorgegeben werden (z.B. `maxTerms = 10`).

## Constraints

Da die Berechnung von Credit-Points auch in dieser Lösung zunächst zurückgestellt wird, bleibt es bei der Begrenzung auf max. 3 Module pro Semester (`count(yi, V, coursesPerTerm)`). Die Modulabhängigkeiten (z.B. `(OPR > EPR)` sowie die SS/WS-Einschränkungen können auch in dieser Lösung sehr leicht festgelegt werden. Ungerade Fachsemester können als Sommersemester und gerade als Wintersemester interpretiert werden (z.B. `(OPR mod 2 = 1)`.

## Optimierung

Eine zu berechnende Lösung sollte dahingehend optimiert werden, dass der Studienverlauf möglichst kompakt geplant wird, sodass nur das Minimum an Fachsemestern zu den gegebenen Kriterien absolviert werden muss.

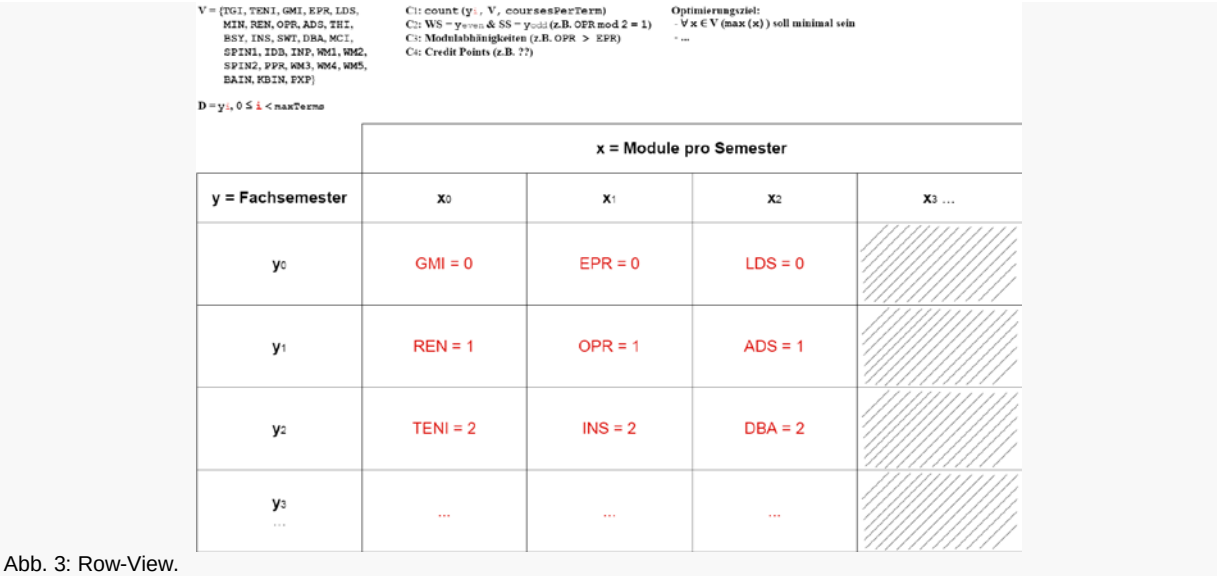

Abb. 3: Row-View.

## Eine mögliche Lösung

```
Model[CourseScheduling_RowBased], 1 Solutions, Resolution time 0,233s, 388 Nodes (1.664,3
n/s), 176 Backtracks, 89 Fails, 0 Restarts
Variables: 145
Constraints: 259
 1. Sem:   GMI     EPR     LDS
 2. Sem:   REN     OPR     ADS
 3. Sem:   DBA     MCI
 4. Sem:   WM1     WM2     WM3
 5. Sem:   TGI     INS     SWT
 6. Sem:   MIN     THI     SPIN1
 7. Sem:   TENI    BSY     SPIN2
 8. Sem:   IDB     INP     PXP
 9. Sem:   PPR     WM4     WM5
10. Sem:   BAIN    KBIN
```

Neben den zuvor bei Versuch 1 sichtbaren und hier ebenfalls auftretenden Problemen, können auch hier nicht alle möglichen Lösungen gefunden werden. Es muss daher unbedingt eine Lösung für die CP-Problematik gefunden werden.

# 3. Versuch

## Variablen & Domänen

Für den dritten Versuch wurden konträr zu Versuch 1 die Indizes im Verlaufsplan als Variablen betrachtet (`ti`) und die Modulnummern als Domänen. Da nicht alle Plätze im Verlaufsplan Module zugewiesen bekommen, muss auch ein mehrfach setzbares Leermodul in den Domänen vorhanden sein (Modulnummer = 0).

Für jede gesetzte Stelle im Verlaufsplan gibt es eine eigene Credit-Points-Variable (`pi`), die zum gesetzten Modul korrespondierende CP-Anzahl enthält. Die CP-Variablen, die wiederum zu ein und demselben Semester gehören können jeweils in einer separaten Variable akkumuliert werden (`ai`).

## Constraints

Es muss sichergestellt werden, dass die Verlaufsplanplätze jeweils unterschiedliche Module zugewiesen bekommen, mit der Ausnahme des Leermoduls als Platzhalter für eine Nicht-Belegung (`allDiffExcept0(V1)`). Die Credit-Points werden durch ein Constraint berechnet, das für alle Verlaufsplanindizes die Modul mit den CP-Variablen verbindet und die passende Credit-Points-Anzahl aus einer separaten Tabelle mithilfe der Modulnummer ausliest (`∀ ti,pi (pi = CP(ti))`). Auch die akkumulierte Anzahl an CP pro Semester lässt sich über ein solches Constraint berechnen (`a0 = sum(p1,p2,p3)`) und anschließend mit einer Bedingung versehen (z.B. `ai >= 15`). Diese Beschränkung hat durch die Bindung zu den Verlaufsplan-Variablen den rückwirkenden Effekt, dass Lösungen mit zu wenigen Modulen pro Semester verworfen werden. Eine anderes Constraint verhindert beispielsweise, dass Module mit CP-Voraussetzungen nicht in den frühen Semestern gesetzt werden können, wenn zuvor noch nicht die benötigten Credit-Points erreicht werden konnten (z.B. `if (t6 == BSY) { a0 + a1 ≥ 30 }`).

Modulabhängigkeiten und SS/WS-Einschränkungen werden aufgrund der umgekehrten Variablen und Domänen zunächst nicht betrachtet.

## Optimierung

Eine zu berechnende Lösung sollte dahingehend optimiert werden, dass der Studienverlauf möglichst kompakt geplant wird, sodass nur das Minimum an Fachsemestern zu den gegebenen Kriterien absolviert werden muss.

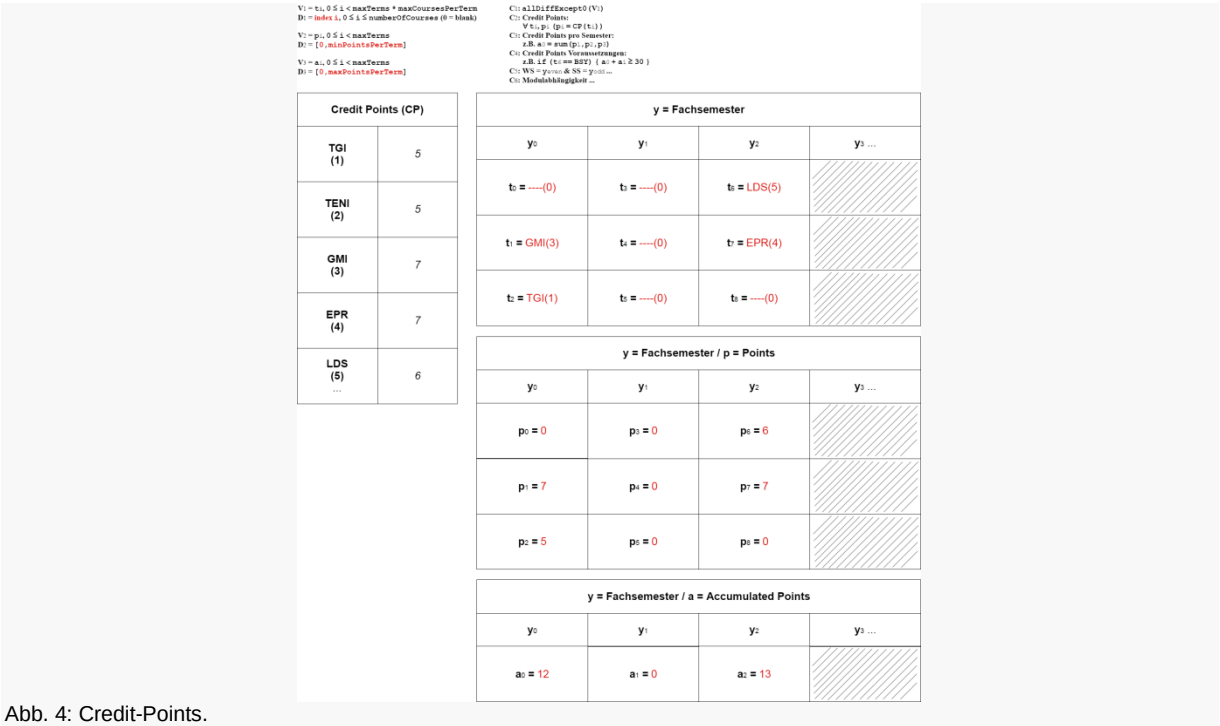

Abb. 4: Credit-Points.

## Eine mögliche Lösung

```
Model[CourseScheduling_CreditPoints], 1 Solutions, Resolution time 0,228s, 440 Nodes (1.927,4
n/s), 722 Backtracks, 362 Fails, 0 Restarts
Variables: 101
Constraints: 61
 1. Sem:   ADS     WM5     LDS
 2. Sem:   SPIN1   MCI     MIN
 3. Sem:   WM3     SWT     INP
 4. Sem:   WM4     BSY     GMI
 5. Sem:   WM1     DBA     IDB
 6. Sem:   ____    BAIN    KBIN
 7. Sem:   REN     TENI    TGI
 8. Sem:   PXP     ____    ____
 9. Sem:   PPR     INS     THI
10. Sem:   SPIN2   WM2     EPR
```

Die Nicht-Beachtung der Modulabhängigkeiten führt natürlich zu einem vollkommen unbrauchbaren Verlaufsplan. Das Ziel dieses Versuches war es jedoch lediglich, eine Möglichkeit für die Berechnung der Credit-Points und die Umsetzung der damit verbundenen Constraints zu beweisen.

**Umstellung:**
Die einzige Möglichkeit, die Credit-Points zu berechnen, ist die Festlegung der Fachsemester bzw. der Verlaufsplanplatzierungen als Variablen. Anderseits müssen die Module als Variablen betrachtet werden, um die Modulabhängigkeiten richtig umsetzen zu können. Die richtige Implementierung muss also eine Kombination aus beiden Wegen enthalten.

# 4. Versuch

## Variablen & Domänen

Für den vierten Versuch wurde nun eine Implementierung gesucht, die es ermöglicht, alle benötigten Constraints umzusetzen. Zudem sollte aber der bereits aufgezeigte Nachteil der Verlaufsplanindizes elemeniert werden. Neben Integer und Boolean- Variablen bietet Choco auch die Möglichkeit, Mengen als Variablen zu betrachten (`SetVar`). Für diesen Versuch wurde jedes Semester als eine Menge von Modulen definiert. Die Fachsemester bilden also die Variablen und die Modulnummern die Domänen. Jedes Set kann zunächst alle Modulnummern beinhalten. Mithilfe der richtigen Constraints sollen die Domänen nach und nach verkleinert werden, sodass zum Schluss eine korrekte Lösung gebildet werden kann. Zusätzlich gibt es Variablen für die Credit-Points, die in jedem einzelnen Semester errungen wurden.

## Constraints

Damit alle Module auch gesetzt werden, stellt ein Constraint sicher, dass die Vereinigung aller Semester-Mengen einer Menge entspricht, die alle Module enthält (`union(V1, {TGI, TENI, …, BAIN, KBIN, PXP})`). Auf der anderen Seite, darf das gleiche Modul nicht mehrfach in verschiedenen Semestern gesetzt sein (`allDisjoint(V1)`). Eine `SetVar`-Funktion erlaubt es, die Mengen-Domänen mit einer Gewichtung zu versehen und anschließend die Gewichte der gesetzten Elemente einer Menge zu addieren. Man braucht also lediglich für jedes Modul in einer separaten Tabelle die Credit-Points als Gewichtswerte festzuhalten. Anschließend können die Credit-Points aller Module innerhalb eines Semesters berechnet werden (`∀ si,ai (ai = sum(si, weights) mit weights aus CP)`).

Weiterhin ist es im Gegensatz zum letzten Versuch mit den Mengen sehr einfach, die CP-Voraussetzungen und die SS/WS-Einschränkungen umzusetzen (z.B. `notMember(TGI, s1)`, `notMember(TGI, s3)` usw.). Auch die Modulabhängigkeiten sind implementierbar, indem für jedes in Abhängigkeiten stehende Modul ein Constraint pro Modul/Semester-Paar erstellt wird, das die abhängigen Module aus den nachfolgenden Semestern ausschließt (z.B. `∀ si (if (member(OPR, si) { notMember(EPR, si+1), notMember(EPR, si+2), usw. }))`.

## Optimierung

Eine zu berechnende Lösung sollte dahingehend optimiert werden, dass der Studienverlauf möglichst kompakt geplant wird, sodass nur das Minimum an Fachsemestern zu den gegebenen Kriterien absolviert werden muss.

$V_1 = s_i, 0 \leq i < maxTerms$
$D_1 = sets(LB, UB), LB = \emptyset \wedge UB = numberOfCourses$

$V_2 = a_i, 0 \leq i < maxTerms$
$D_2 = [minPointsPerTerm, maxPointsPerTerm]$

C1: union ($V_1$, {TGI(0), TENI(1), … , BAIN(25), KBIN(26), PXP(27)})
C2: allDisjoint ($V_1$)
C3: WS = $y_{even}$ & SS = $y_{odd}$
z.B. notMember (TGI(0), $s_1$) usw. …
C4: Credit Points pro Semester:
$\forall s_i, a_i$ ($a_i$ = sum($s_i$, weights) mit weights aus CP)
C5: Credit Points Voraussetzungen …
C6: Modulabhänigkeiten …

Optimierungsziel:
- $\forall s_i$ mit $s_i = \emptyset$ (min (i)) soll minimal sein
- …

| y = Fachsemester<br>s = Scheduled Courses | | y = Fachsemester<br>a = Accumulated Points | | Credit Points (CP) | |
|---|---|---|---|---|---|
| $y_0$ | $s_0$ = {GMI(2), EPR(3), LDS(4)} | $y_0$ | $a_0$ = 20 | TGI (0) | 5 |
| $y_1$ | $s_1$ = {REN(6), OPR(7), THI(9)} | $y_1$ | $a_1$ = 18 | TENI (1) | 5 |
| $y_2$ | $s_2$ = {INS(11), DBA(13), MCI(14)} | $y_2$ | $a_2$ = 18 | GMI (2) | 7 |
| $y_3$ | $s_3$ = {ADS(8), IDB(16), WM1(18)} | $y_3$ | $a_3$ = 18 | EPR (3) | 7 |
| $y_4$ … | $s_4$ = {TGI(0), TENI(1), SWT(12)} | $y_4$ … | $a_4$ = 16 | LDS (4) … | 6 |

Abb. 5: Set-View.

## Eine mögliche Lösung

```
Model[CourseScheduling_SetBased], 1 Solutions, Resolution time 0,232s, 22 Nodes (94,9 n/s), 5
Backtracks, 5 Fails, 0 Restarts
Variables: 8510
Constraints: 8288
 1. Sem:   GMI     EPR     LDS
 2. Sem:   REN     OPR     THI
 3. Sem:   INS     DBA     MCI
 4. Sem:   ADS     IDB     WM1
 5. Sem:   TGI     TENI    SWT
 6. Sem:   MIN     INP     WM4
 7. Sem:   BSY     WM2     WM3
 8. Sem:   SPIN1   PXP
 9. Sem:   SPIN2   PPR     WM5
10. Sem:   BAIN    KBIN
```

Aufgrund der nun deutlich erhöhten Einschränkungen, kann die Constraint-Propagierung noch vor der eigentlichen Suche die Domänen sehr stark verkleinern. Die anschließende Suche nach der ersten Lösung fällt dadurch deutlich schneller aus (`22 Nodes, 5 Backtracks, 5 Fails`). Zwar macht das zeitlich gesehen bei nur einer Lösung keinen großen Unterschied - möchte man aber alle Lösungen finden, ist dies signifikant (abgesehen von der Tatsache, dass zuvor nicht alle möglichen Lösungen „richtig“ waren). Aber auch mit dieser Implementierung ist es momentan nicht möglich, alle Lösungen zu berechnen, ohne an Speichergrenzen zu stoßen, was an der großen Anzahl an Variablen und Constraints liegt.

# 5. Versuch

Statt es bei der vierten Implementierung zu belassen, wurden die zahlreichen Erkenntnisse dieses Versuches dazu genutzt, den zweiten Vesuch „CourseScheduling_RowBased“ so auszubauen, dass alle entscheidenden Constraints umgesetzt werden konnten. Während die meisten Constraints zum Aufbau des Studienverlaufsplans, der Modulabhängigkeiten und SS/WS-Einschränkungen weiterhin mit der „RowBased“-Methode funktionieren (d.h. die Module stellen die Variablen dar), wurden nun noch die Sets aus dem letzten Versuch in das neue Modell übernommen (d.h. auch die Fachsemester sind Variablen). Die Fachsemester-Variablen werden an die „RowBased“-Constraints gebunden und lediglich zur Berechnung der Credit-Points und zur Einhaltung der auf den CP basierenden Einschränkungen verwendet. Dadurch lässt sich eine deutliche Verringerung der Variablen und Constraints erreichen.

```
Model[CourseScheduling_RowBased], 1 Solutions, Resolution time 0,074s, 16 Nodes (216,0 n/s), 4
Backtracks, 4 Fails, 0 Restarts
Variables: 2200
Constraints: 2011
 1. Sem:   GMI     EPR     LDS
 2. Sem:   MIN     OPR     ADS
 3. Sem:   INS     SWT     DBA
 4. Sem:   REN     THI     IDB
 5. Sem:   TGI     TENI    MCI
 6. Sem:   SPIN1   WM1     WM2
 7. Sem:   BSY     SPIN2   WM3
 8. Sem:   INP     PXP
 9. Sem:   PPR     WM4     WM5
10. Sem:   BAIN    KBIN
```

Diese Kombination führt nicht nur zu einer schnelleren Lösung, sondern damit auch erstmals zu einem kompletten Durchlauf und der Berechnung aller möglichen Lösungen, wobei der Java-Heap eine Größe von etwas mehr als 2GB erreicht. Das bedeutet, dass der vollständige Suchbaum verkleinert wurde. Es ist jedoch sehr wahrscheinlich immer noch nicht das optimale Modell.

```
Number of all solutions: 28.069
Estimated time: 18 min
```

# What to do next?

- *Unterscheidung zwischen bereits erhaltenen CPs und im Semester zu erhaltenden CPs (BAIN+KBIN+PXP in einem Sem.)*
- *Unterscheidung zwischen Bounded/Enumerated Domains (schnellere Lösungen)*
- Such-Optimierung (Lösungen ohne Lücken) für 4 und 5 Module pro Semester
- eigene Suchstrategie (schnellere Lösungen)
- mit anderen Bibliotheken vergleichen
- Verbindung mit interaktivem App-Ansatz
- weitere Studiengänge (vllt. Fachbereiche)
- …