Aufgabe 1: Flohmarkt in Langdorf

Teilnahme-ID: 56905
Chuyang Wang

28. Mai 2021

Inhaltsverzeichnis

1 Lösungsidee

Die Aufgabe ist ein *0-1-Integer-Linear-Programming (ganzzahlige lineare Optimierung)* Problem. Generelle LP Probleme lassen sich mit der Simplex Methode lösen. Für ILP muss dazu noch der branch-and-bound oder branch-and-cut Algorithmus eingesetzt werden. D. h. die LP wird zunächst für alle rationalen Zahlen gelöst (LP-Relaxation). Dann wird es für nicht-ganzzahlige Lösungen in zwei Teilprobleme zerlegt und separat gelöst.

Das vorliegende Problem lässt sich wie Folgendes zu modellieren:

Zu jeder Voranmeldung i wird eine Entscheidungsvariable x_i erzeugt, die nur einen Wert von 0 oder 1 nehmen kann. Wird diese Voranmeldung akzeptiert, so hat x_i einen Wert von 1 und vice versa.

Das (lineare) Ungleichungssystem besteht am Anfang aus zehn Ungleichungen, die jeweils eine Zeitspanne von einer Stunde repräsentieren (z. B. das erste Intervall wäre für 8 bis 9 Uhr).

Sei $c_{i,j}$ die vom Anbieter i in dem Intervall j benötigte Standlänge, dann kann man die Ungleichungen so darstellen:

$$\sum_{i=0}^{n-1} c_{i,j} x_i \leq 1000 \qquad \forall j \in \{0, 1, ..., 9\} \tag{1}$$

Da jede Entscheidungsvariable nur einen Wert von 0 oder 1 nehmen kann, muss dies auch berücksichtigt werden. Die Simplex Methode impliziert bereits, dass alle Variablen nicht negativ sind. Für die Obergrenze gibt es zwei Möglichkeiten: Entweder schreibt man sie als explizite Ungleichungen und setzt sie in die Tableau ein, oder, man nutzt *Simplex method with bounded variables*. D. h. die Obergrenze der Variablen wird bei dem Suchen nach Pivotelementen direkt berücksichtigt und es benötigt damit keine expliziten Ungleichungen. Die beiden Vorgehensweise wurden implementiert, jedoch wird hier nur die Erste präsentiert, da die Zweite keinen besonderen zeitlichen Vorteil aufweist, während dadurch das Programm schwer verkompliziert wird. Also haben wir noch folgende Ungleichungen:

$$x_i \leq 1 \qquad \forall i \in \{0, 1, ..., n-1\} \tag{2}$$

Die Zielfunktion, die maximiert werden soll, ist die Summe der Einnahme P. Hier gilt:

$$P = \sum_{i=0}^{n-1} c_i x_i \qquad \text{für} \quad c_i = \text{Intervallänge}_i \cdot \text{Standlänge}_i \tag{3}$$

Setzt man die Schlupfvariablen ein und schreibt in das Simplextableau:

$$
\begin{array}{c|cccccccccccc|c}
 & x_0 & x_1 & ... & x_{n-1} & s_0 & s_1 & ... & s_9 & s_{10} & ... & s_{n+9} & P & RHS \\
R_P & -c_0 & -c_1 & ... & -c_{n-1} & 0 & 0 & ... & 0 & 0 & ... & 0 & 1 & 0 \\
R_0 & c_{0,0} & c_{1,0} & ... & c_{n-1,0} & 1 & 0 & ... & 0 & 0 & ... & 0 & 0 & 1000 \\
R_1 & c_{0,1} & c_{1,1} & ... & c_{n-1,1} & 0 & 1 & ... & 0 & 0 & ... & 0 & 0 & 1000 \\
... & & & ... & & & & ... & & & ... & & & 1000 \\
R_9 & c_{0,9} & c_{1,9} & ... & c_{n-1,9} & 0 & 0 & ... & 1 & 0 & ... & 0 & 0 & 1000 \\
R_{10} & 1 & 0 & ... & 0 & 0 & 0 & ... & 0 & 1 & ... & 0 & 0 & 1 \\
... & & & ... & & & & ... & & & ... & & & 1 \\
R_{n+9} & 0 & 0 & ... & 1 & 0 & 0 & ... & 0 & 0 & ... & 1 & 0 & 1
\end{array}
\tag{4}
$$

Die Simplex-Methode gilt als allgemein bekannt und wird hier nicht sehr detailliert beschrieben. Grundsätzlich müssen alle negativen Elemente in R_P eliminiert oder in einen positiven Wert durch Zeilenoperation gebracht werden. Pivotspalte ist die Spalte, deren Element in Zeile R_P am niedrigsten und negativ ist. Das Pivotelement der Pivotspalte ist das Element mit dem niedrigsten nicht negativ Quotient von RHS und dem Element.

Ist das optimale Tableau aus (4) erreicht, i.e. gibt es keine negativen Werte in Zeile R_P mehr, wird nun das Problem zerlegt, sofern es Entscheidungsvariablen gibt, die nicht-ganzzahlig sind. Der Wert von RHS in der Zeile R_P entspricht der Obergrenze des jetzigen Ergebnisses. Die Untergrenze bekommt man, indem man alle nicht-ganzzahlige Variablen zu 0 setzt und den Gewinn neu berechnet. Ist die jetzige Untergrenze höher als die globale Untergrenze, ersetzt man die Globale durch die Jetzige. Ist die jetzige Obergrenze niedriger als die globale Untergrenze, wird das Problem nicht in weiteren Zweigen zerlegt, weil es bereits bessere Lösung gibt. Alle Möglichkeiten werden in ein Heap eingefügt und es wird immer diejenige zuerst bearbeitet, deren Obergrenze am höchsten ist. Wird damit eine ganzzahlige Lösung gefunden, so muss nicht weiter gesucht werden, denn das gefundene Ergebnis den höchstmöglichen Wert hat.

Bei der Zerlegung wird die nicht-ganzzahlige Variable x_i zu 0 oder 1 gesetzt und mithilfe der Dual-Methode reoptimiert. Da es einfacher ist, mit Ungleichungen zu arbeiten als mit Gleichungen, wird hierbei entweder $x_i <= 0$ oder $-x_i <= -1$ in das Tableau eingefügt, anstatt $x_i = 0$ oder $x_i = 1$.

Theoretisch sollte alles erledigt sein. Leider musste ich feststellen, dass die Implementierung für manche gegebenen Beispiele zu langsam ist. Dazu gibt es mehrere Möglichkeiten: Entweder verwende ich eine schnellere Variante zum Lösen von LP, oder kombiniere ich den jetzigen Algorithmus mit einem anderen, der für solche Beispiele schnell genug ist, oder, verwende ich ein Lösungsprogramm, das von Dritten geschrieben ist. Die zweite und dritte Möglichkeiten werden hier praktiziert.

Der kombinierte Algorithmus führt eine Depth-first search mit Backtracking durch. Bei jeder Verzweigung wird ein Anbieter abgelehnt und den jetzt noch zu erwartenen Gewinn berechnet. Ist dieser Gewinn niedriger oder gleich als die globale Untergrenze, wird diese Knoten abgeschnitten. Im anderen Fall wir geprüft, ob jetzt die räumliche Bedingung (1000m/Stunde) passt. Wenn ja, dann ersetzt man die globale Untergrenze mit der jetzigen Kombination; wenn nein, dann wird eine weitere Verzweigung durchgeführt.

Für größere n Wert wird am Anfang den höchstmöglichen Gewinn P_h berechnet. Das Programm geht davon aus, dass es bereits eine Lösung mit dem Gewinn $P_h - 1$ gibt. Alle Knoten, deren möglichen Gewinn niedriger als diesen Wert liegt, werden abgeschnitten. Findet das Programm eine Lösung höher als $P_h - 1$, terminiert das Programm. Ist alle Möglichkeiten hingegen vollständig gesucht, dann fängt das Programm erneut mit der Annahme von $P_h - 2$ an und so weiter und so fort.

Die beiden vorgestellten Algorithmen laufen unabhängig nebeneinander. Wird eine optimale Lösung von einem der beiden Algorithmen gefunden, werden die beiden Algorithmen terminiert. Wenn in der gegebenen Zeitfrist keine optimale Lösung gefunden, wird die lokale optimale Lösung gegeben, i.e. die beste Auswahl, die bisher entdeckt wurde.

Die Zeitkomplexität dieses Verfahrens wird am Ende der Umsetzung erläutert.

2 Umsetzung

Das Programm wird in C# .Net Core 3.1 umgesetzt. Der Name der Test-Datei muss beim Durchführen des Programms eingegeben werden. Für Kommandozeile-Argumente gibt es folgende Möglichkeiten:

```
// Eine positive Ganzzahl in Milisekunde
// Default: Unbegrenzt (--time-limit=int.MaxValue)
--time-limit=INT32
// Ob das Programm das Lösenprogramm von SCIP nutzen darf
// Default: --use-google=false
--use-google=true|false
// Ob das Programm nur den Simplex Algorithmus verwenden darf
// Default: --force-simplex=false
--force-simplex=true|false
```

In der `Main()` Methode wird zunächst die Datei eingelesen (durch Methode `ReadInput()`). Danach werden zwei Threads initialisiert für den Simplex (Klasse `LinearSolver`) und Backtracking (Klasse `CompleteSearch`) Algorithmus. Nachdem eine optimale Lösung gefunden wird, wird das Ergebnis mithilfe der Methode `PrintResult()` in Console ausgedruckt.

Jede Zeile der Test-Datei wird in Form von `int[] { ANFANG, ENDE, LAENGE, ID }` durch die Methode `ReadInput()` umgewandelt und in eine Liste gespeichert. ID fängt von 0 an und entspricht der Zeile in der Test-Datei. Für den Simplex Algorithmus müssen sie noch in Ungleichungen und Zielfunktion umgeschrieben werden. Diese gelingt durch `GetConstraints()`.

Die Methode `Solve()` instanziiert die Klasse `LinearSolver` und löst das Problem. Die Klasse `CompleteSearch` wird in der `Main()` Methode ebenfalls instanziiert und läuft nebeneinander. Da es sich um Multithreading handelt, müssen sie abgebrochen werden können, wenn eine optimale Lösung durch eine der Methoden gefunden wird oder die Zeit abgelaufen ist, weswegen `CancellationToken` ins Spiel kommt.

Im Falle, dass die Zeit abgelaufen ist und es keine optimale Lösung gefunden wurde, werden die Werte von den beiden Algorithmen vergleicht und die Höhere davon ausgedruckt. Dem Nutzer wird darüber informiert, dass es möglicherweise nicht die optimale Lösung ist.

Der bisherige Aufrufstapel sieht grundsätzlich so aus:

```
Main()
  ParseArgs()
  ReadInput()
      |
      |----------------------|----------------------|
  sw.Start()       simplexThread.Start()      searchThread.Start()
  Thread.Sleep() // Warte
  PrintResult() // optimale Loesung gefunden | TLE
```

Hierbei sind die lokalen Variablen so deklariert:

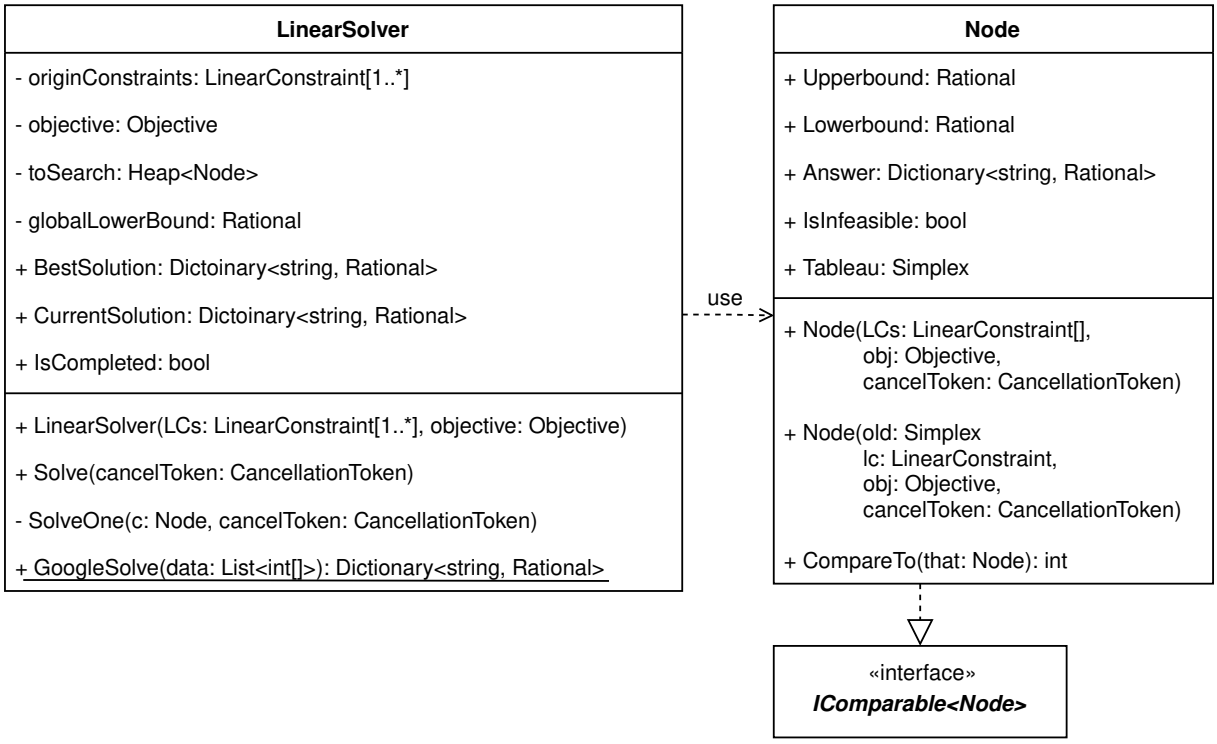
```
// Aufgabe1.FlohmarktManagement.Main() #Aufgabe1.cs
Tuple<bool, int, List<int[]>> input = ReadInput();

Stopwatch sw = new Stopwatch();

var simplexThread = new Thread(
    () => res = Solve(input.Item3, sToken.Token));

CompleteSearch c = new CompleteSearch(input.Item3);
var searchThread = new Thread(() => c.Process(cToken.Token));
```

Die Klasse `LinearSolver` wird wie Folgendes strukturiert:

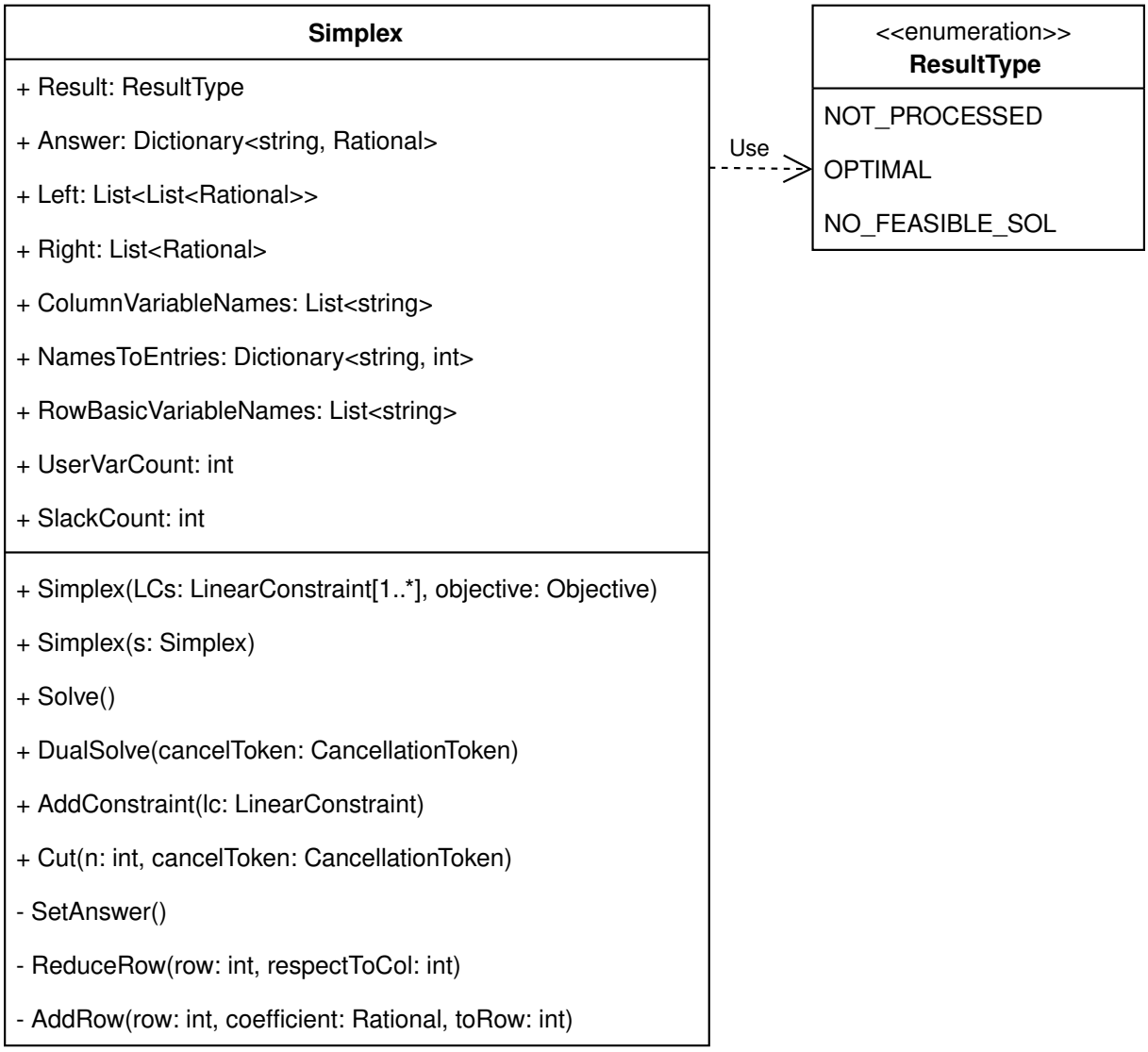


Die Methode `Solve()` ist der Kern des Branch-and-Bound Algorithmus. Dort wird eine Schleife angelegt und solange, bis es kein Element mehr in dem Heap gibt oder eine optimale Lösung, i.e. eine Lösung, deren Upper- und Lowerbound gleich sind, gefunden wird. Es wird immer die Knote zur Methode `SolveOne()` weitergegeben, deren Upperbound am höchsten ist (vgl `toSearch: Heap<Node>`). Das garantiert eine optimale Lösung, wenn dabei eine ganzzahlige Lösung gefunden wird. Diese Methode `LinearSolver.Solve()` ist nicht mit `FlohmarktManagement.Solve()` zu verwechseln.

`BestSolution` existiert nur, wenn eine optimale Lösung gefunden wird. In diesem Fall hat `IsComplete` auch den Wert `true`. `CurrentSolution` speichert die bisherige beste Lösung, welche auch der globalen Untergrenze `globalLowerBound` entspricht.

Die Methode `SolveOne()` dient zur Verzweigung. Der eingegebene Parameter `c: Node` speichert Ober- und Untergrenze des Ergebnisses und das Tableau von der Simplex Methode. Für jede Variable in `Answer`, die nicht ganzzahlig ist, wird diese zu 0 oder 1 gesetzt und das Problem wird verzweigt, i.e. ein neues Objekt von Node wird instanziiert mit dieser neuen Ungleichung und in den Heap `toSearch` eingefügt. Bei der Instanziierung der Klasse `Node` wird eine neue Beschränkung (Ungleichung) in das Tableau eingefügt und das Tableau wird durch die Dual-Methode reoptimiert. Danach wird die Ober- und Untergrenze berechnet. Die Obergrenze ist der Zielfunktionswert in dem Tableau der LP-Relaxation und die Untergrenze bekommt man, indem man alle nicht-ganzzahlige Variablen zu 0 setzt.

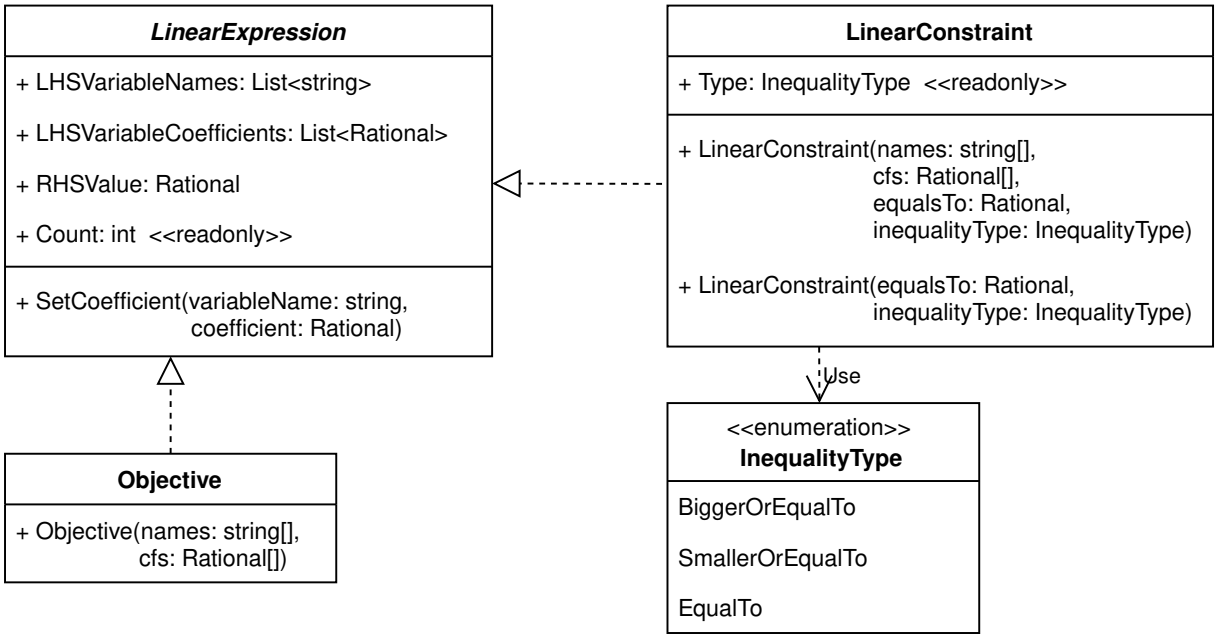
Die Klasse `Simplex` implementiert den Simplex Algorithmus (Der Klassendiagramm ist möglicherweise nicht vollständig, nur die wichtigsten Elemente werden hier dargestellt):



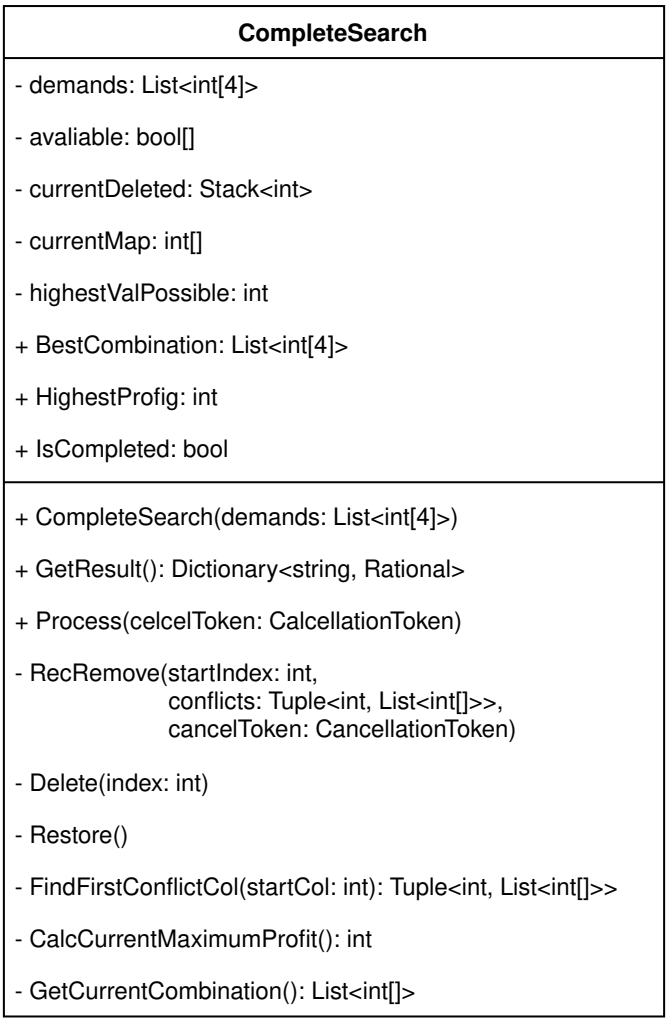
`Answer` speichert die optimale Lösung beim Lösen des Tableaus in Form von Zuordnung `VariableName: string -> VariableWert: Rational`. `Left` und `Right` sind die linke und rechte Seite des Tableaus bzw. der Matrix. `UserVarCount` ist die Anzahl der Entscheidungsvariablen und `SlackCount` die Anzahl der Schlupfvariablen.

Die Methode `Solve()` löst das Tableau, wenn das Objekt durch den ersten Konstruktor instanziiert wird, wobei `DualSolve()` dazu dient, das Tableau zu reoptimieren, nachdem eine neue Ungleichung durch `AddConstraint()` eingefügt wurde. Die Methode `Cut()` implementiert *Gomory Cut*, welche bei jeder Verzweigung in `LinearSolver` durchgeführt wird. Nachdem das optimale Simplextableau erreicht wird, wird die Methode `SetAnswer()` aufgerufen und wandelt die Matrix in besser lesbare Form, i.e. `Answer`, um. Die Methode `ReduceRow()` führt die Zeilenoperation $R_{row} = R_{row}/c_{row,respectToCol}$ durch und `AddRow()` die Zeilenoperation $R_{toRow} = R_{toRow} + R_{row} * coefficient$.

Die Klassen `LinearConstraint` und `Objective` dienen zur Speicherung von Ungleichungen und Zielfunktionen. Sie sind von der abstrakten Klasse `LinearExpression` abgeleitet:



Wie vorher erwähnt muss noch ein Backtracking Algorithmus eingesetzt werden (vgl. Diagramm unten). Die Liste `demands` speichert die Daten der Anbieter, die von der Datei eingelesen werden, in Form von Array: `int [4] {Anfang, Ende, Laenge, ID}`. Die Methode `Process()` ruft die rekursive Methode `RecRemove()` auf und wartet, bis eine optimale Lösung gefunden wird. Für kleinere Data (hier n $<$ 50) wird die `HighestProfit` am Anfang zu 0 gesetzt und es wird erwartet, dass alle Möglichkeiten durchgesucht werden (Abschneiden von Knoten bleibt natürlich, wie in der Lösungsidee beschrieben). Für größeres Beispiel setzt man `HighestProfit` am Anfang zu `highestValPossible - 1`. Das erlaubt schnellere Beschneidung von Knoten. Wird jedoch keine Lösung gefunden, so muss nun `HighestpProfit` ↪ `--` durchgeführt und das obige Verfahren wiederholt werden. Bei jeder Verzweigung wird ein Anbieter abgelehnt, indem man den Wert `avaliable[index dieses Anbieters]` zu `false` setzt und diesen Index in `currentDeleted` einkellert. Das gelingt durch die Methode `Delete()` und die Methode `Restore()` nimmt die letzte Ablehnung zurück, indem der Index des Anbieters von `currentDeleted` ausgekellert und alle andere Operationen, die in der Methode `Delete()` durchgeführt werden, rückgängig gemacht werden. Das Array `currentMap` speichert die jetzige gebuchte Länge für die jeweiligen Zeitspannen. Diese hilft zu erkennen, ob eine mögliche Lösung bereits gefunden wird. Es ist der Fall, wenn alle Werte in diesem Array kleiner oder gleich 1000 sind.



2.1 Externe Bibliotheken

Bei der Umsetzung werden folgende Bibliotheken verwendet:

- Rationals von Tomáš Pažourek : Datenstruktur für rationale Zahlen
- OR-Tools von Google : Lösenprogramm für ILP. **Wird NUR verwendet, WENN durch Kommandozeilen explizit aktiviert wird. Sollte nur zum Überprüfen beim Debuggen dienen.**

2.2 Zeitkomplexität

Der Branch-and-Bound Algorithmus hat im schlimmsten Fall exponentielles Zeitkosten. Der andere Backtraking Algorithmus ist dann $O(n!)$.

3 Beispiel

Für jedes gegebene Beispiel wird folgende Kommandozeilen-Argumente verwendet:

```
.\Aufgabe1.exe --time-limit=5000 --use-google=false
2                --force-simplex=false
```

Um Vermüllung dieses Abschnittes zu vermeiden, wird hier nur Teil des ausgedruckten Ergebnisses gezeigt. **Eine .txt Datei mit allen vollständigen Ausgaben findet man im Ordner**: `Aufgabe1\AusfuehrbaresProgramm\ausgabe.txt`

Hier scheint es zu sein, dass die Klasse `CompleteSearch` überwiegend verwendet wird. Das liegt jedoch daran, dass die Simplex-Methode zur Initialisierung etwas mehr Zeit benötigt. Allermeisten Beispiele können ohne `CompleteSearch` auch gelöst werden. Um das zu testen, kann man die Kommandozeilen-Argumente `--force-simplex=true` setzen.

3.1 flohmarkt1.txt

```
Jetzige Konfiguration:
2         --time-limit=5000
          --use-google=False
4         --force-simplex=False

6   Bitte Name der Test-Datei eingeben...
    flohmarkt1.txt
8
    Die beste Auswahl ist:
10  (1 bedeutet akzeptiert, 0 hingegen abgelehnt)
    ......
12  --------------------
    Zusammenfassung:
14
    Akzeptierte Anbieter: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
     ↪ 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
     ↪ 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62
     ↪ 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
     ↪ 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104
     ↪ 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
     ↪ 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136
     ↪ 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152
     ↪ 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168
     ↪ 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184
     ↪ 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200
     ↪ 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216
     ↪ 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232
     ↪ 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248
     ↪ 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264
     ↪ 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280
     ↪ 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296
     ↪ 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312
     ↪ 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328
```

```
 ↪ 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344
 ↪ 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360
 ↪ 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376
 ↪ 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392
 ↪ 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408
 ↪ 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424
 ↪ 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440
 ↪ 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456
 ↪ 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472
 ↪ 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488
 ↪ 489

Abgelehnte Anbieter:

Diese Auswahl besteht aus 490 Anbietern.
Die Mieteinnahme betraegt 8028 Euro.

Diese Antwort stammt aus: 'CompleteSearch'
Gesamt verwendete Zeit: 489
```

Der Anbieter werden von 0 an nummeriert. D. h. der Anbieter, der in der Zeile n der Test-Datei steht, hat die Nummer bzw. ID $n - 1$.

`Akzeptierte Anbieter` und `Abgelehnte Anbieter` werden nur bei diesem Beispiel gezeigt, um die Struktur der Ausgabe klar zu machen. Bei den folgenden Beispielen wird nur eine kleine Zusammenfassung hier dargestellt.

3.2 flohmarkt2.txt

```
Jetzige Konfiguration:
        --time-limit=5000
        --use-google=False
        --force-simplex=False

Bitte Name der Test-Datei eingeben...
flohmarkt2.txt

Die beste Auswahl ist:
(1 bedeutet akzeptiert, 0 hingegen abgelehnt)
......
Abgelehnte Anbieter: 1 6 7 14 27 28 48 50 54 67 106 108 109 118 120
 ↪ 132 133 146 152 165 167 170 171 172 179 190 191 193 200 209 211
 ↪ 215 218 219 221 225 229 240 246 261 270 289 299 322 327 335 340
 ↪ 346 353 363 368 371 381 382 384 385 388 389 393 395 399 401 405
 ↪ 408 414 415 422 423 426 432 443 447 449 451 467 469 470 471 472
 ↪ 480 482 483 491 494 498 504 508 509 520 524 527 529 535 536 537
 ↪ 542 545 547 549 550 557 559 561 566 567 568 573 575 578 582 589
 ↪ 594 598

Diese Auswahl besteht aus 490 Anbietern.
Die Mieteinnahme betraegt 9077 Euro.

```

```
Diese Antwort stammt aus: 'Simplex'
Gesamt verwendete Zeit: 1374
```

3.3 flohmarkt3.txt

```
Abgelehnte Anbieter: 2 7 9 10 14 15 17 21 22 23 24 25 26 27 34 39 43
 ↪ 46 50 51 52 53 54 62 66 67 74 75 77 83 87 94 95 96 98 99 101 105
 ↪ 108 109 115 117 118 124 128 130 132 137 143 144 147 148 152 154
 ↪ 155 156 160 162 163 165 172 174 179 182 185 189 190 191 192 197
 ↪ 198 199 201 202 203 204 209 210 212 213 214 216 218 225 230 231
 ↪ 236 240 241 242 243 248 249 250 253 255 257 259 260 261 266 272
 ↪ 274 280 284 288 289 292 296 297 302 307 309 315 331 348 350 352
 ↪ 356 364 366 369 371 373 385 390 414 449 470 480 489 531

Diese Auswahl besteht aus 603 Anbietern.
Die Mieteinnahme betraegt 8778 Euro.

Diese Antwort stammt aus: 'CompleteSearch'
Gesamt verwendete Zeit: 519
```

3.4 flohmarkt4.txt

```
Jetzige Konfiguration:
        --time-limit=5000
        --use-google=False
        --force-simplex=False

Bitte Name der Test-Datei eingeben...
flohmarkt4.txt

Die beste Auswahl ist:
(1 bedeutet akzeptiert, 0 hingegen abgelehnt)
x0: 1
x1: 1
x2: 0
x3: 1
x4: 1
x5: 1
x6: 0
--------------------
Zusammenfassung:

Akzeptierte Anbieter: 0 1 3 4 5

Abgelehnte Anbieter: 2 6

Diese Auswahl besteht aus 5 Anbietern.
Die Mieteinnahme betraegt 7370 Euro.

```

```
Diese Antwort stammt aus: 'CompleteSearch'
Gesamt verwendete Zeit: 131
```

3.5 flohmarkt5.txt

```
Zusammenfassung:
Akzeptierte Anbieter: 1 2 5 6 8 13 18 19

Abgelehnte Anbieter: 0 3 4 7 9 10 11 12 14 15 16 17 20 21 22 23 24

Diese Auswahl besteht aus 8 Anbietern.
Die Mieteinnahme betraegt 8705 Euro.

Diese Antwort stammt aus: 'CompleteSearch'
Gesamt verwendete Zeit: 137
```

3.6 flohmarkt6.txt

```
Zusammenfassung:
Akzeptierte Anbieter: 0 1 2 3 4 5 6 7 8

Abgelehnte Anbieter:

Diese Auswahl besteht aus 9 Anbietern.
Die Mieteinnahme betraegt 10000 Euro.

Diese Antwort stammt aus: 'CompleteSearch'
Gesamt verwendete Zeit: 116
```

3.7 flohmarkt7.txt

```
Abgelehnte Anbieter:

Diese Auswahl besteht aus 566 Anbietern.
Die Mieteinnahme betraegt 10000 Euro.

Diese Antwort stammt aus: 'CompleteSearch'
Gesamt verwendete Zeit: 385
```

4 Quellcode

Unwichtige Quellcode-Elemente werden hier nicht gezeigt und werden durch ..... ersetzt. Der vollständige Quelltext ist in dem Ordner `Aufgabe1/Quelltext/` zu finden.

Trotz meines Versuchs kann der Quellcode nicht weiter reduziert werden. Der Code für `CompleteSearch` findet man unter `Aufgabe1/Quelltext/CompleteSearch.cs`. Der wird hier wegen der Längenbeschränkung nicht eingefügt.

```
// Aufgabe1.cs
.....

using Aufgabe1.LinearProgramming;
using rat = Rationals.Rational;

namespace Aufgabe1
{
  public class FlohmarktManagement
  {
    public const int HEIGHT = 1000;
    public const int START_TIME = 8;
    public const int END_TIME = 18;
    public const int INTERVAL_LENGTH = END_TIME - START_TIME;

    public static void Main(string[] args)
    .....

    // Liest die Test-Datei
    // IsSuccess, demandCount, data
    private static Tuple<bool, int, List<int[]>> ReadInput()

    // Zeigt das Ergebnis in Console
    private static void PrintResult(Dictionary<string, rat> result,
    ↪ string source)

    // Loesen mit Simplex (b-a-c)
    public static Tuple<bool, Dictionary<string, rat>> Solve(List<int[]>
    ↪  data, CancellationToken cancelToken)
    .....

    // Bearbeitet die originelle Data und gibt sie als Beschraenkung
    ↪ zueurck
    private static Tuple<LinearConstraint[], Objective> GetConstraints(
    ↪ List<int[]> data)
    {
      List<LinearConstraint> constraints = new List<LinearConstraint>();
      // 10 Zeitspannen, Laenge sollte kleiner oder gleich als 1000
      for (int i = 0; i < INTERVAL_LENGTH; i++)
      {
        var c = new LinearConstraint(HEIGHT, LinearConstraint.
    ↪ InequalityType.SmallerOrEqualTo);
        var colItems = GetItemsInCol(i, data);
        foreach (var item in colItems)
          c.SetCoefficient($"x{item}", data[item][2]);
        constraints.Add(c);
      }
```

```
        // Obergrenze von Entscheidungsvariablen 0 <= x <= 1
44      for (int i = 0; i < data.Count; i++)
        {
46        constraints.Add(new LinearConstraint(new string[] { $"x{i}" },
    ↪ new rat[] { 1 }, 1, LinearConstraint.InequalityType.
    ↪ SmallerOrEqualTo));
        }
48
        // Zielfunktion
50      var objective = new Objective(Enumerable.Range(0, data.Count)
                                      .Select(i => $"x{i}")
52                                    .ToArray(),
                                    Enumerable.Range(0, data.Count)
54                                    .Select(i => (rat)data[i].GetSize
    ↪ ())
                                      .ToArray());
56      return new Tuple<LinearConstraint[], Objective>(constraints.
    ↪ ToArray(), objective);
       }
58   }
}
60

62 // LinearSolver.cs
...
64 using Aufgabe1.DataStructure;
using rat = Rationals.Rational;
66
// Wird NUR genutzt wenn --use-google=true explizit gesetzt wird
68 using Google.OrTools.LinearSolver;

70 namespace Aufgabe1.LinearProgramming
{
72   // Branch and Cut
     public class LinearSolver
74   {
       // Beschraenkung/Ungleichungen
76     private readonly LinearConstraint[] originConstraints;
       // Die Zielfunktion
78     private readonly Objective objective;
       private readonly Heap<Node> toSearch;
80     // Das jetzige beste Ergebnis
       // Ist ein UpperBound kleiner als diesen Wert
82     // wird es nicht mehr in zwei Zweigen geteilt
       private rat globalLowerBound;
84     // Die Loesung
       public Dictionary<string, rat> BestSolution { get; private set; }
86     // Die temporale beste Loesung, entspricht dem globalLowerBound
       public Dictionary<string, rat> CurrentSolution { get; private set; }
88     // Falls der Algorithmus durch CancellationToken unterbrochen wird
       // anstatt selbst damit fertig ist, dann false
90     public bool IsCompleted { get; private set; }

92     public LinearSolver(LinearConstraint[] LCs, Objective objective)
       ......
94
       // Loest die LP durch Branch-And-Cut
96     public void Solve(CancellationToken cancelToken)
```

```
    {
98    var c = new Node(originConstraints, objective, cancelToken);
      // Teilt das jetzige Ergebnis in zwei Zweigen
100   // sofern die Werte nicht alle ganzzaehlig sind
      SolveOne(c, cancelToken);
102   // Solange es weitere Zweige zum Rechnen gibt
      while (toSearch.Count != 0)
104   {
        SolveOne(toSearch.Pop(), cancelToken);
106   }
      ......
108 }

110 // Unterteilt das jetzige Ergebnic c in Unterzweigen
    // Fuer die Variablen, die nicht ganzzahlig sind
112 private void SolveOne(Node c, CancellationToken cancelToken)
    {
114   ......
      if (!c.IsInfeasible)
116   {
        // Falls das jetzige Ergebis schlechter als die Untergrenze,
118     // dann muss es nicht in Unterzweigen mehr geteilt werden
        if (globalLowerBound > c.Upperbound) return;
120     // Falls das Lowerbound des jetizgen Ergebnises besser
        // als das Gespeicherte, dann ersetz das Gespeicherte
122     // durch das Jetzige
        if (globalLowerBound < c.Lowerbound)
124       ......

126     // Falls eine ganzzahlige Loesung (alle Variablen ganzzahlig)
    ↪ gefunden
        // dann kann man die ganze Suchreihe loeschen
128     // denn dort wird die Ergebnis nach Upperbound sortiert
        // d.h. dieses Ergebnis 'c' hat den hoechstwahrscheinlichen Wert
130     // innerhalb aller Moeglichkeiten
        if (c.Lowerbound.Equal(c.Upperbound))
132     {
          toSearch.Clear();
134       // Die endgueltige Loesung
          BestSolution = new Dictionary<string, rat>();
136       foreach (var kvp in c.Answer)
            BestSolution.Add(kvp.Key, kvp.Value.CanonicalForm);
138       return;
        }
140
        // Falls das jetizge Ergebnis nicht-ganzzahlige Variablen
    ↪ enthaelt
142     // werden jede nicht-ganzzahlige Variable in zwei Zweigen
    ↪ unterteilt
        // naemlich x_i = 0 oder x_i = 1
144     foreach (var pair in c.Answer)
        {
146       // der Zielwert 'P' kann uebersprungen werden
          if (pair.Key.Equals(objective.LHSVariableNames[0])) continue;
148       if (!(pair.Value.FractionPart == 0))
          {
150         LinearConstraint zero = new LinearConstraint(new string[] {
    ↪ pair.Key }, new rat[] { 1 }, 0, LinearConstraint.InequalityType.
```

```
    ↪ SmallerOrEqualTo);
            toSearch.Add(new Node(c.Tableau, zero, objective,
    ↪ cancelToken));
152         LinearConstraint one = new LinearConstraint(new string[] {
    ↪ pair.Key }, new rat[] { -1 }, -1, LinearConstraint.InequalityType.
    ↪ SmallerOrEqualTo);
            toSearch.Add(new Node(c.Tableau, one, objective, cancelToken
    ↪ ));
154       }
        }
156   }
  }
158
  // Loesen des ILP mithilfe von SCIP integiert in Google OrTools
160 // Diese Methode wird NUR aufgerufen, wenn
  // --use-google=true gesetzt wird
162 public static Dictionary<string, rat> GoogleSolve(List<int[]> data)
  ......
164
  // Klasse zum Speichern von geloeste Simplex Tableau
166 public class Node : IComparable<Node>
  {
168   public rat Upperbound { get; }
    public rat Lowerbound { get; }
170   // Das jetizge Ergebnis
    public Dictionary<string, rat> Answer { get; }
172   public bool IsInfeasible;
    // Das jetizge Tableau
174   public Simplex Tableau { get; }

176   public Node(LinearConstraint[] LCs, Objective obj,
    ↪ CancellationToken cancelToken)
      ......
178
      // Neuloesen mit einer weiterer Beschraenkung
180   public Node(Simplex old, LinearConstraint lc, Objective obj,
    ↪ CancellationToken cancelToken)
      {
182     Simplex s = new Simplex(old);
        // Addiert das neu ergaenzte Ungleichung
184     // Also, wenn man x_i <= 0 oder -x_i <= -1 gesetzt hat
        // diese ist identisch wie x_1 = 0 oder = 1 zu setzen
186     s.AddConstraint(lc);
        // Reoptimieren mit Dual Methode
188     s.DualSolve(cancelToken);

190     if (s.Result == Simplex.ResultType.OPTIMAL)
        {
192       // Gomory Cut
          s.Cut(5, cancelToken);
194       if (cancelToken.IsCancellationRequested) return;
          Answer = s.Answer;
196       Tableau = s;

198       // Rechnet Upper- und Lowerbound
          rat maximalVal = s.Answer[obj.LHSVariableNames[0]];
200       rat lowerBound = 0;
          for (int i = 1; i < s.Answer.Count; i++)
```

```
202           {
                rat ans = s.Answer[obj.LHSVariableNames[i]];
204             if (ans.FractionPart == 0)
                  lowerBound += obj.LHSVariableCoefficients[i] * s.Answer[
 ↪ obj.LHSVariableNames[i]];
206           }
              Upperbound = maximalVal;
208           Lowerbound = lowerBound * -1;

210         }
            ......
212       }

214       // Das Objekt ist 'groesser',
          // wenn sein Upperbound groesser ist
216       // ist es gleich, dann vergleicht man den Lowerbound
          public int CompareTo(Node that)
218     }
    }
220
    // einfach Simplex
222 // Big M Methode, in diesem Kontext wird dies jedoch
    // nicht gebraucht
224 public class Simplex
    {
226   // Fuer BigM Methode
      // Hier wird es NICHT gebraucht
228   public const int M = 100000;
      // Wenn die abgeschnittene Flaeche zu klein ist,
230   // dann mach es lieber nicht
      // Grenzwert:
232   public const int CUT_D_LIMIT = 150;
      // NOT_PROCESSED | OPTIMAL | NO_FEASIBLE_SOL
234   public ResultType Result { get; private set; }
      // Die Loesung
236   public Dictionary<string, rat> Answer { get; private set; }

238
      public List<List<rat>> Left { get; }
240   public List<rat> Right { get; }
      public List<string> ColumnVariableNames { get; }
242   public Dictionary<string, int> NamesToEntries { get; }
      public List<string> RowBasicVariableNames { get; }
244   // Speichert, wie viele Entscheidungsvariablen es gibt
      // Also die Anzahl von Variablen, die nicht Slack sind
246   public int UserVarCount { get; }
      // Also die Anzahl von Schlupfenvariablen
248   public int SlackCount { get; private set; }

250   public Simplex(LinearConstraint[] LCs, Objective objective)
      {
252     Result = ResultType.NOT_PROCESSED;
        Answer = new Dictionary<string, rat>();
254
        // Alle 0-1 Variablen
256     UserVarCount = objective.Count;
        int colCount = UserVarCount;
258     // Anzahl von Slack, Surplus und Artificial Variables
```

```
    foreach (LinearConstraint lc in LCs)
260 {
      switch (lc.Type)
262   {
        // Slack
264     case LinearConstraint.InequalityType.SmallerOrEqualTo:
          colCount += 1;
266       break;
        ......
268   }
    }
270 // Constraints + Zielfunktion
    int rowCount = LCs.Length + 1;
272 // Ptr (col entry) fuer slack, surplus und artificial vars
    int colPtr = objective.Count;
274 Left = new List<List<rat>>(rowCount);
    ColumnVariableNames = new List<string>(new string[colCount]);
276 NamesToEntries = new Dictionary<string, int>();
    Right = new List<rat>(new rat[rowCount]);
278 RowBasicVariableNames = new List<string>(new string[rowCount]);

280 // Zielfunktion
    Left.Add(new List<rat>(new rat[colCount]));
282 for (int i = 0; i < objective.Count; i++)
    {
284   Left[0][i] = objective.LHSVariableCoefficients[i];
      ColumnVariableNames[i] = objective.LHSVariableNames[i];
286   NamesToEntries.Add(objective.LHSVariableNames[i], i);
    }
288 Right[0] = objective.RHSValue;
    RowBasicVariableNames[0] = objective.LHSVariableNames[0];
290

292 // Fuer Constraints
    SlackCount = 0; int artificialCount = 0;
294 for (int row = 1; row <= LCs.Length; row++)
    {
296   Right[row] = LCs[row - 1].RHSValue;
      Left.Add(new List<rat>(new rat[colCount]));
298   for (int j = 0; j < LCs[row - 1].Count; j++)
      {
300     if (NamesToEntries.TryGetValue(LCs[row - 1].LHSVariableNames[j
↪ ], out int entry))
        {
302       Left[row][entry] = LCs[row - 1].LHSVariableCoefficients[j];
        }
304   }

306   if (LCs[row - 1].Type == LinearConstraint.InequalityType.
↪ SmallerOrEqualTo)
      {
308     AddVar($"s{SlackCount++}", ref colPtr);
      }
310   ......

312   // Funktion, um eine neue Variable in den Matrix zu setzen
      void AddVar(string name, ref int entry)
314 }
```

```
      if (!HasFeasibleSol()) Result = ResultType.NO_FEASIBLE_SOL;
316   }

318   // Kopiert von einem vorhandenen Objekt
      public Simplex(Simplex s)
320
      // Loesen mit traditionellem Simplex
322   public void Solve()
      {
324     // i.e. Spalte, deren 'reduced cost' am niedrigsten und negativ
    ↪ ist
        int FindEnteringVariable()
326     {
          int c = 1;
328       List<int> allCols = new List<int> { c };
          for (int i = 2; i < Left[0].Count; i++)
330       {
            if (Left[0][c] > Left[0][i])
332         {
              c = i;
334           allCols.Clear();
              allCols.Add(i);
336         }
            else if (Left[0][c].Equal(Left[0][i])) allCols.Add(i);
338       }
          if (Left[0][c] >= 0) return -1;
340       return allCols[0];
        }
342
        // i.e. Zeile mit kleinstem Quotient in dieser Spalte
344     int FindLeavingVariable(int inCol)
        {
346       int r = 1;
          while (Left[r][inCol] <= 0) r++;
348       for (int i = r + 1; i < Left.Count; i++)
          {
350         if (Left[i][inCol] > 0 && Right[i] / Left[i][inCol] >= 0
                && Right[r] / Left[r][inCol] > Right[i] / Left[i][inCol])
352         {
              r = i;
354         }
          }
356       return r;
        }
358
        int col = FindEnteringVariable();
360     // Solange es noch negative Werte in P Zeile gibt
        while (col != -1)
362     {
          int row = FindLeavingVariable(col);
364       RowBasicVariableNames[row] = ColumnVariableNames[col];
          ReduceRow(row, col);
366       for (int i = 0; i < Left.Count; i++)
          {
368         if (i == row) continue;
            // Eliminiert diese Variable in anderen Zeilen
370         if (!Left[i][col].IsZero)
              AddRow(row, Left[i][col] * -1, i);
```

```
372         }
            // Such nach einer neuen Variable
374         // deren reduced cost negativ ist
            col = FindEnteringVariable();
376       }

378       if (IsFeasibleSol())
          {
380         Result = ResultType.OPTIMAL;
            SetAnswer();
382       }
          // Unter diesem Kontext sollte nicht passieren
384       else Result = ResultType.NO_FEASIBLE_SOL;
        }
386

388     // Dual Methode zur Reoptimierung
        // Nachdem eine neue Beschraenkung eingefuegt wurde
390     public void DualSolve(CancellationToken cancelToken)
        {
392       // Zeile mit kleinsten (und nagativ) reduced cost
          // Sie ist in diesem Fall Right[]
394       // Denn der Matrix wird transposed
          int FindEnteringVariable()
396       // ..... Aehnlich wie in Solve()

398       // Spalte mit niedrigsten Betrag von Quotient
          int FindLeavingVariable(int r)
400       // ..... Aehnlich wie in Solve()

402       int row = FindEnteringVariable();
          // Solange es noch weiter reduziert werden
404       while (row != -1 && !cancelToken.IsCancellationRequested)
          {
406         int col = FindLeavingVariable(row);
            if (col == -1)
408         {
              Result = ResultType.NO_FEASIBLE_SOL;
410           return;
            }
412         RowBasicVariableNames[row] = ColumnVariableNames[col];
            ReduceRow(row, col);
414         for (int i = 0; i < Left.Count; i++)
            {
416           if (i == row) continue;
              if (!Left[i][col].IsZero)
418             AddRow(row, Left[i][col] * -1, i);
            }
420         row = FindEnteringVariable();
          }
422
          if (IsFeasibleSol())
424         SetAnswer();
          else
426         Result = ResultType.NO_FEASIBLE_SOL;
        }
428
        // Fuegt eine neue Beschraekung
```

```
430     // DualSolve() sollte aufgerufen werden um das zu reoptimieren
        public void AddConstraint(LinearConstraint lc)
432     {

434       Right.Add(lc.RHSValue);
          for (int i = 0; i < Left.Count; i++) Left[i].Add(0);
436       Left.Add(new List<rat>(new rat[Left[0].Count]));
          for (int i = 0; i < lc.LHSVariableCoefficients.Count; i++)
438         Left[^1][NamesToEntries[lc.LHSVariableNames[i]]] = lc.
    ↪ LHSVariableCoefficients[i];

440       // neue slack Var
          ......
442
          // Falls manche neue eingesetzte Koeffiziente Basic ist
444       // muessen sie dementsprechen in dieser neu eingefuegte Zeile
          // eliminiert werden
446       ......
        }
448
        // Gomory Cut und reoptimieren mit Dual Methode
450     public void Cut(int n, CancellationToken cancelToken)
        {
452       // Iteriert man durch die Zeilen und
          // sucht nach nicht-ganzzahlige Variablen,
454       // deren Bruchteil am groessten ist,
          // um moeglich groesse Flaeche abzuschneiden
456       for (int count = 0; count < n && !cancelToken.
    ↪ IsCancellationRequested; count++)
          {
458         int r = 1;
            while (Right[r].FractionPart == 0)
460         {
              r++;
462           if (r == Right.Count) return;
            }
464         // Such nach weitere nicht ganzzahligen Werte
            for (int row = 2; row < Right.Count; row++)
466         {
              if (Right[row].FractionPart != 0)
468           {
                // Falls die abgeschnittene Flaeche zu klein ist
470             // dann sollte das nicht durchgefuehrt werden
                if (Right[row].Denominator > CUT_D_LIMIT) continue;
472             if (Right[r].FractionPart < Right[row].FractionPart) r = row
    ↪ ;
              }
474         }

476         // Aendert von >= zu <=
            // indem man (* -1) auf beiden Seiten durchfuehrt
478         LinearConstraint newLc = new LinearConstraint(Right[r].
    ↪ FractionPart * -1, LinearConstraint.InequalityType.
    ↪ SmallerOrEqualTo);
            for (int col = 1; col < Left[0].Count; col++)
480           newLc.SetCoefficient(ColumnVariableNames[col], Left[r][col].
    ↪ FractionPart * -1);
```

```
482             AddConstraint(newLc);
                DualSolve(cancelToken);
484           }
        }
486
        // Liest die Loesung von dem Tableau
488     private void SetAnswer()

490     // Zeilenoperation: R_row = R_row / c_row,respectToCol
        private void ReduceRow(int row, int respectToCol)
492
        // Zeilenoperation: R_toRow = R_row * coefficient + R_toRow
494     private void AddRow(int row, rat coefficient, int toRow)

496     public bool HasFeasibleSol()
        ......
498
        public bool IsFeasibleSol()
500     ......

502     public enum ResultType
        ......
504   }

506   // LinearExpression ist, emmm, Linear Expression
      // Als Basisklasse fuer die Ungleichungen und die Zielfunktion
508   public abstract class LinearExpression
      {
510     ......
        public void SetCoefficient(string variableName, rat coefficient)
512   }

514   // Lineare Ungleichung
      public class LinearConstraint : LinearExpression
516   {
        public enum InequalityType
518     ......
      }
520
      // Angenommen fuer Maximazation
522   // Die Zielfunktion
      public class Objective : LinearExpression
524 }
```