

Benutzerhandbuch

Software-Engineering

**Logistik-Optimierungsbeispiel**

eingereicht an der

**Montanuniversität Leoben**

|  |  |
| --- | --- |
| **Vorgelegt von** | **Betreuer/Gutachter** |
| Hoffelner Mario, m01614788 Nehl Stefan, m00935188 | Univ.-Prof. Dipl.-Ing. Dr.techn. Peter Auer |

Leoben, 28. Juli 2022

# Inhaltsverzeichnis

Inhaltsverzeichnis II

Abbildungs- und Tabellenverzeichnis III

1 Aufbau der SRC 4

1.1 Contracts 4

1.2 Entities 5

1.3 Services 7

2 Anleitungen 8

2.1 Einlesen von Dateien 8

2.2 Erzeugung einer neuen Factory 8

2.3 Start einer Optimierung 9

2.4 Main-Klasse 9

3 Enumeration Calculation Algorithmus 10

3.1 Erstellen der Prozessschritte 10

3.1.1 Acquire 10

3.1.2 Produce 11

3.1.3 Deliver 11

3.2 Bounding, Cutting planes und Transporterauswahl 11

3.2.1 Einschränkung der Laufzeit 11

3.2.2 Ausschließen von unmöglichen Reihenfolgen 11

3.2.3 Verwendung von verschiedenen Transportern 12

4 Ergebnisse 15

4.1 Durchmischte Liste 15

4.2 Liste mit ähnlichen Produkten 16

4.3 Liste mit verschiedenen Prozessen 17

4.4 Zusammenfassen der Materialien 18

4.5 Zusammenfassung 18

# Abbildungs- und Tabellenverzeichnis

Abbildung 1: Ordnerstruktur der src 4

Abbildung 2: Contracts-Ordner 4

Abbildung 3: Entities-Ordner 5

Abbildung 4: FactoryStep 6

Abbildung 5: Komplette Liste von FactoryStep‘s für die Produktion von 94 Stahlblech 6

Abbildung 6: Services-Ordner 7

Abbildung 7: Vereinfachte FactorySteps-Liste 12

Abbildung 8: Auswahl der Transporter Bsp. 1 13

Abbildung 9: Auswahl der Transporter Bsp. 2 13

Abbildung 10: Vergleich der Transporter 14

[Tabelle 1: Durchmischte Auftragsliste 15](#_Toc109851499)

[Tabelle 2: Ergebnisse durchmischte Auftragsliste 16](#_Toc109851500)

[Tabelle 3: Ähnliche Produkte 16](#_Toc109851501)

[Tabelle 4: Ergebnis ähnliche Produkte 17](#_Toc109851502)

[Tabelle 5: Verschiedene Prozesse 17](#_Toc109851503)

[Tabelle 6: Ergebnis verschiedene Prozesse 17](#_Toc109851504)

[Tabelle 7: Ergebnis Zusammenfassen der Materialien 18](#_Toc109851505)

# Aufbau der SRC

Wie in der Abbildung 1 ersichtlich, besteht der *Source-Ordner* nur aus dem *logistikoptimierung-Ordner.* Dieser Ordner unterteilt sich dann in 3 weitere Ordner, namentlich *Contracts*, *Entities* und *Services,* zusätzlich ist in diesem Ordner auch die Main Klasse abgelegt.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung : Ordnerstruktur der src

## Contracts

Wie in der Abbildung 2 abgebildet, unterteilt sich der Ordner *Contracts* in zwei Klassen, der *IDataService-* und *IOptimizationService-Klasse*. Diese sind zwei Interfaces, welche für den Optimierungsservice und Datenservice implementiert werden müssen.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung : Contracts-Ordner

## Entities

Wie aus Abbildung 3 ersichtlich, unterteilt sich der Ordner *Entities* in die Ordner *FactoryObjects*, *WarehouseItems* und in die Klasse *Instance*. In diesen Ordnern befinden sich alle Klassen, welche zur Objekterzeugung benötigt werden. In *FactoryObjects* sind Klassen, welche die aktiven Teile der Simulation darstellen wie z.B. *Driver, Factory, FactoryStep* oder auch *Transporter*. Die *WarehouseItems* sind alle Klassen welche die passiven Teile, wie z.B. Material oder Produkt simulieren. Die genaueren Beschreibungen zu den Klassen bzw. einzelnen Methoden können im *Javadoc* nachgelesen werden.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung : Entities-Ordner

**FactoryStep**

Wir verwenden für unsere Simulation *FactoryStep*‘s. Diese *Steps* beinhalten alle Informationen, was mit welchem Objekt, zu welchem Zeitpunkt passieren soll. Ein

*FactoryStep* sieht wie in Abbildung 4 aus. Dabei ist das *itemToManipulate* das passive

Item oder *WarehouseItem* welches manipuliert werden soll. (Zum Beispiel vom Lager

in den Buffer der Produktion gelegt werden soll) das *factoryObject* das aktive Objekt

welches die Manipulation durchführt, hier ein Transporter, der *stepType* welche

Manipulation durchgeführt wird, (diese sind im Enum *FactoryStepTypes* hinterlegt),

das Object *factory* stellt die Fabrik also das übergeordnete Objekt wo die

Manipulation ausgeführt werden soll, *amountOfItems* die Anzahl an Items welche

manipuliert werden sollen, *doTimeStep* wann die Aktion durchgeführt werden soll

und *factoryStepsToDoBefore* welche Schritte ausgeführt werden soll, bevor der aktuelle schritt ausgeführt wird.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung : FactoryStep

In Abbildung 5 sieht man Beispielhaft die *FactoryStep*‘s welche für den Auftrag von 94 Stahlblech benötigt werden. Hierbei starten die *Step’*s mit dem Holen der Rohstoffe und endet mit der Auslieferung der fertigen Produkte an den Kunden.

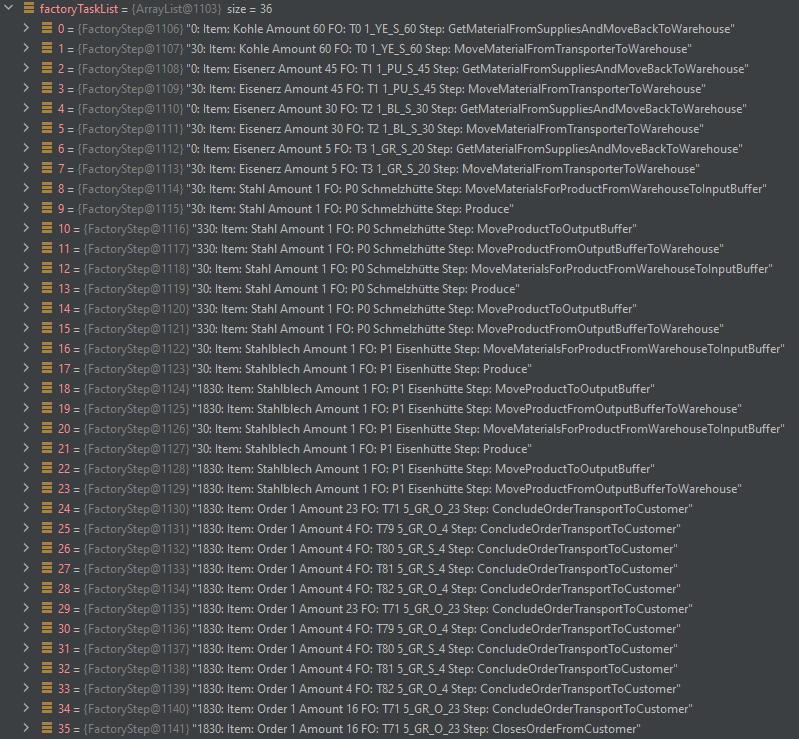


Abbildung : Komplette Liste von FactoryStep‘s für die Produktion von 94 Stahlblech

## Services

Wie in Abbildung 6 ersichtlich, besteht der Ordner *Services* aus den zwei Unterordnern *EnumeratedCalculation* und *FirstComeFirstServeOptimizer* als auch aus der Klasse *CSVDataImportService*.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung : Services-Ordner

**EnumeratedCalculation - Ordner**

Alle Klassen in diesem Ordner sind selbsterklärend bzw. in der Javadoc beschrieben. In dem nächsten Absatz wird noch auf die Klasse *EnumeratedCalculationMain* genauer eingegangen.

In der *EnumeratedCalculationMain* Klasse wird ein Objekt erzeugt, welches dann durch einen rekursiven Aufbau, alle möglichen Kombinationen der Arbeitsschritte durchmischt und berechnet. Die Berechnung erfolgt in unserem Programm durch die Methode *startFactory*. In dieser Methode greifen wir auf die Klasse *logistikoptimierung* 🡪 *Entities* 🡪 *FactoryObjects* 🡪 *Factory* zurück. In dieser, versucht unsere Factory die eingegebene Abfolge der Arbeitsschritte durchzuführen. Am Ende wird entweder die benötigte Zeit zum Abschluss aller Arbeitsschritte zurückgegeben oder die Methode abgebrochen, falls sie eine gewisse Zeit überschreitet.

**FirstComeFirstServeOptimizer - Ordner**

In der Klasse *FirstComeFirstServeOptimizerMain* wird der FirstComeFirstServe Algorithmus verwendet, um auf ein Simulationsergebnis zu kommen. Hierfür werden *FactorySteps* erzeugt und diese nach dem FirstComeFirstServe Prinzip in die Simulation gepackt. Wir verwenden diese Methode, um einen Startwert zu erhalten, den wir dann mit der Enumeration versuchen zu verbessern.

**CSVDataImportService - Klasse**

Diese Klasse wird verwendet, um alle Parameter aus der CSV-Datei einzulesen. Die CSV-Dateien sind im *data* Ordner hinterlegt und beinhalten alle Daten, welche zum Erzeugen einer Instanz notwendig sind.

# Anleitungen

In diesem Kapitel werden bestimmte Anwendungsfälle erklärt, um das Verwenden des Programmes einfacher zu gestalten.

## Einlesen von Dateien

Das Einlesen von Dateien erfolgt über die *CSVDataImportService* Klasse. Diese befindet sich in *logistikoptimierung* 🡪 *Services.* Der Konstruktor dieser Klasse erzeugt ein Objekt, welche das Importieren von CSV Dateien ermöglicht. Als Parameter übergibt man dem Konstruktor die Anzahl zur Verfügung stehenden Fahrer und die Kapazität des Warenlagers. Beide Werte sind als integer hinterlegt. Die Methode *loadData* ladet dann die Datei mit dem Namen, welcher der Methode als Input gegeben wird.

Beispiel:

String contractList = CSVDataImportService.*CONTRACT\_3*;

var dataService = new CSVDataImportService(7, 1000);  
var instance = dataService.loadData(contractList);

Hierbei definieren wir mit dem String *contractList*, welche Liste im Anschluss geladen werden soll. In der nächsten Zeile erzeugen wir das Objekt der Klasse *CSVDataImportService* und mit den übergebenen Parametern haben wir festgelegt, dass es 7 Fahrer gibt, welche zur Verfügung stehen und die Lagerkapazität auf 1000 beschränkt ist. Mit der letzten Zeile laden wir dann alle Aufträge der ausgewählten Spalte.

## Erzeugung einer neuen Factory

Die Factory beinhaltet alle Informationen über die aktuelle Simulation. Sie beinhaltet alle Informationen über die Produktionen, die Transporter, die verfügbaren Rohstoffe und auch die Order Liste. Eine Factory kann erst nach dem Einlesen der Daten erstellt werden. Mit der Methode *startFactory* führt man eine Simulation für die aktuellen Parameter und die übergebene *FactoryStep’s* Liste durch. Diese Methode gibt einen long Wert zurück, der entweder die maximale RunTime welche, der Methode als Input gegeben wird oder die Laufzeit, welche bis zum Abschluss aller *FactoryStep’s* benötigt wurde.

## Start einer Optimierung

Nachdem man eine Instanz mit Fabrik und Auftragsliste erstellt hat, kann man einen Optimierungsservice erstellen. Hierstehen die zwei zuvor vorgestellten Services zur Verfügung. Der *FirstComeFirstServeOptimizer* und der EnumeratedCalculation Optimierungsservice. Zum Erstellen dieser Services muss das davor erzeugte Instanzobjekt im Konstruktor übergeben werden. Die Methode *optimize* des jeweiligen Optimierungsservice kann dann aufgerufen werden und ein Integer-Wert mit der Anzahl an Aufträgen übergeben werden. Dieser Parameter der Methode *optimize* ist bestimmt die Anzahl wie viele Aufträge der in der Instanz gespeicherten Liste optimiert werden sollen. Die Optimierungsmethode des EnumeratedCalculation Services erstellt alle möglichen Anordnungen der *FactoryStep‘s*. Hierbei beachten wir, dass nur Listen in die Simulation gepackt werden, welche auch logisch sind. Ein Beispiel hierzu wäre: Wir simulieren keine Listen, wo die Auslieferung der Endprodukte vor der Anlieferung der Materialien stattfindet. Im Ablauf der Optimierung werden nun alle möglichen Kombinationen in der Fabrik simuliert und die Ergebnisse in der Konsole ausgegeben und das beste Ergebnis in Form einer Liste mit *FactorySteps* zurückgegeben.

## Main-Klasse

Zu Beginn der Klasse können wir festlegen, welche Messages wir in der Konsole angezeigt bekommen möchten. Die Einstellungen dafür werden im Record FactoryMessageSettings gespeichert und als Parameter im Konstruktor übergeben.

var factoryMessageSettings = new FactoryMessageSettings(  
 false,  
 false,  
 false,  
 false,  
 false,  
 false,  
 false,  
 false,  
 false,  
 false  
);

Der nächste Parameter, der verändert werden kann, ist die Anzahl an Aufträgen, die wir optimieren möchten. Hierbei empfiehlt es sich, eine Zahl ≥ 1 zu wählen. Die String Variable gibt unserem Projekt an, welche Auftragsliste wir optimieren möchten. Der letzte Parameter, der hier abgeändert werden kann, ist die maximale Laufzeit einer Simulation. Diese ist deswegen relevant, da unsere Simulation mit Zeitschritten arbeitet und in jedem Zeitschritt versucht alle Aufträge, der Reihe nach abzuarbeiten. Diese *maxRuntimeInSeconds* ist jedoch nur für den ersten Durchlauf relevant. Mehr hierzu finden Sie unter dem Kapitel 3.2.

int nrOfOrderToOptimize = 4;  
String contractList = CSVDataImportService.*CONTRACT\_3*;  
long maxRuntimeInSeconds = 100000000;

Möchte man nun einen neuen Algorithmus verwenden muss dieser einfach im gleichen Schema erzeugt wie die anderen Tests, indem vorher die Daten geladen werden und dann die Instanz dem jeweiligen Algorithmus übergeben werden.

# Enumeration Calculation Algorithmus

Der von uns angewandte Algorithmus erstellt für die einzelnen Aufträge eine Prozessliste für das Holen des Materials, das Produzieren eines Produktes und die Lieferung zum Kunden. Diese werden dann rekursive zu der verschiedenen Kombination zusammengefügt und simuliert. Da die Anzahl der Auswahlmöglichkeiten faktoriell mit der Anzahl an Prozessschritten steigt haben wir versucht Lösungen wegzuschneiden umso schneller auf eine Lösung zu kommen.

## Erstellen der Prozessschritte

Für die Erstellung der Prozessschritte wurde die Klasse *PlanningItems* erzeugt. Diese hat drei verschiedene Typen, *Acquire* für das Holen von Materialen, *Produce* für das Produzieren eines Loses und *Deliver* für das ausliefern.

### Acquire

Für das Holen von Materialen werden zuerst die Prozessschritte berechnet, welche notwendig sind, um ein Produkt zu produzieren. Dies geschieht unter Berücksichtigung der Losgröße der einzelnen Produktionsprozesse. Aus diesen Prozessschritten werden dann die benötigten Materialen abgeleitet. Die Zuweisung zu einem Transporter geschieht dann in der Transporterauswahl welche im Punkt 3.2.3 näher besprochen wird.

### Produce

Für das Produzieren eines benötigten Produktes wird die vorher erstellte Prozessliste wieder herangezogen und ein *PlanningItem* für jedes Los, welches produziert werden muss, erstellt.

### Deliver

Für das Ausliefern wurde für jede Lieferung ein *PlanningItem* erzeugt.

## Bounding, Cutting planes und Transporterauswahl

Um die Laufzeit unseres Programmes zu reduzieren, haben wir versucht so viele vernünftige Boundings und Cutting planes einzufügen, wie möglich.

### Einschränkung der Laufzeit

In der *main*-Methode übergeben wir zuerst eine *maxRuntimeInSeconds*, diese wird für den ersten Durchlauf der Simulation als Abbruchkriterium herangezogen. Im Anschluss wird diese Laufzeit auf die Laufzeit der aktuellen besten Lösung begrenzt. Diese Begrenzung haben wir eingeführt, da es nicht möglich ist, eine bessere Lösung zu erhalten, wenn die Laufzeit der aktuellen besten Lösung überschritten ist.

### Ausschließen von unmöglichen Reihenfolgen

Da wir in unserem Programm, alle FactorySteps durchmischen, kann es zu sehr unlogischen Reihenfolgen kommen. Ein Beispiel hierzu wäre, wir möchten, dass unsere Factory die Endprodukte ausliefert, bevor diese überhaupt produziert wurden. Solche Varianten verfolgen wir nicht weiter. Um unser Vorgehen leichter zu verstehen, haben wir drei sehr vereinfachte Beispiele in Abbildung 7 dargestellt.



Abbildung : Vereinfachte FactorySteps-Liste

### Verwendung von verschiedenen Transportern

Um eine möglichst breite Lösungsmenge abzudecken, haben wir uns bei der Auswahl der Transporter folgendes überlegt. Da wir es nicht sinnvoll fanden, für jeden Branch, alle Transporter durchzutesten, haben wir für jeden einzelnen Branch, den optimalen Transporter bestimmt und somit eine möglichst hohe Variation bei der Auswahl der Transporter, für verschiedene Transportschritte zu gewährleisten. Um unsere Auswahl leichter zu verstehen, finden Sie im Anschluss ein Beispiel hierzu.

Wir suchen für unsere Transporte immer den optimalen Transporter, diese Suche bedeutet für uns, wir suchen einen Transporter, der die geringste Differenz zur benötigten Kapazität besitzt. Sollte es zwei Transporter geben, welche die gleiche Differenz aufweisen, jedoch einmal negativ und einmal positiv, wird der Transporter gewählt, welcher eine größere Kapazität hat, damit wir den Transport mit einem Transporter abwickeln können. Wie in der Abbildung 8 zu sehen, wird zuerst für den *FactoryStep* Holen von Rohstoff R1 ein Transporter gesucht, hierzu gehen wir unsere Liste der Transporter durch und finden den Transporter T1, dieser hat eine Differenz von +10, nun gehen wir die Liste weiter, bis wir den letzten Transporter finden, der eine positive oder genau die benötigte Kapazität besitzt. Wir halten unsere Transporter-Listen immer sortiert. Als letzten Transporter in unserer Liste, finden wir den Transporter T4, welcher genau der benötigten Kapazität entspricht und deshalb ausgewählt wird. Im nächsten Schritt suchen wir für den Rohstoff R2 einen Transporter, hier finden wir gleich mit dem ersten Transporter, T1 einen perfekten, passenden Transporter.



Abbildung : Auswahl der Transporter Bsp. 1

Um zu zeigen, wie die unterschiedliche Reihenfolge der *FactorySteps* die Auswahl der Transporter beeinflussen kann, haben wir in Abbildung 9 eine andere Reihenfolge gewählt und das gleiche Verfahren angewendet.



Abbildung : Auswahl der Transporter Bsp. 2

Um die unterschiedlichen Ergebnisse darzustellen, haben wir die Ergebnisse in Abbildung 10 nochmals gegenübergestellt.

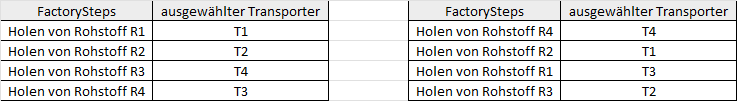


Abbildung : Vergleich der Transporter

# Ergebnisse

Unser *Branch and Cutting* Algorithmus ist stark abhängig von den Aufträgen und den möglichen Kombinationen, da die Anzahl der Auswahlmöglichkeiten faktoriell mit der Anzahl an *PlanningItems* steigt. Diese Anzahl hängt von der Tiefe einer Produktion, also wie viele Produktionsschritte notwendig sind, um ein Produkt für die Order herzustellen, von der Anzahl der verschiedenen Materialen und die Anzahl an Produktionen die parallele abgearbeitet werden können. Um dies Abzubilden haben wir verschiedene Auftragslisten erstellt, um unseren Algorithmus zu testen. Eine Liste versucht eine große Variation der Aufträge darzustellen, eine die viele ähnliche Produkte produziert und eine Liste mit verschiedenen Prozessen welche parallel abgearbeitet werden können. Die maximale Anzahl an Aufträgen, welche wir optimieren wollen, wurde von 1 auf 5 gesteigert, um die Änderungen in der Laufzeit zu beobachten. Weiters wurde die Kapazität des Lagers auf 1000 gesetzt und die Anzahl an Fahrer auf 6.

## Durchmischte Liste

Wie schon vorhin erwähnt wurde die durchmischte Liste so erzeugt, um möglichst viele Produkte mit unterschiedlicher Prozesstiefe zu haben. Couch hat eine Prozesstiefe von 3, Nägel und Eisenpulver von 2, Kohle und Stahl, welches wir immer vom Hersteller holen, eine Prozesstiefe von 1. Die Auftragsliste ist in Tabelle 1 zu sehen.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Areal | Produkte | Menge | Ertrag | Col | Engine | Zeit |
| 1 | Couch | 236 | 4202 | GR | x | 60 |
| 5 | Stahl | 101 | 3638 | BL | x | 30 |
| 5 | Kohle | 40 | 702 | YE | x | 3 |
| 4 | Nägel | 105 | 1563 | GR | x | 60 |
| 5 | Eisenpulver | 80 | 2493 | YE | x | 15 |

Tabelle : Durchmischte Auftragsliste

In Tabelle 2 ist gut zusehen, wie die Laufzeit mit der Anzahl an Kombinationen bzw. der Anzahl an *PlanningItems* steigt. Zwar wird die Anzahl an Simulation stark herunter gebrochen und ist nur ein Bruchteil der Anzahl an Kombination jedoch wächst die Anzahl an möglichen Kombinationen sehr schnell an und damit auch unsere Zeit, mit der unser Algorithmus das Problem optimal löst. Auch gut zu sehen ist, dass sich der *FirstComeFirstServe* Algorithmus bei den Aufträgen 4 sehr schwertut und die Simulationszeit sehr stark ansteigt. Unser Algorithmus hingegen kommt in kurzer Zeit auf ein besseres Ergebnis. Mit 5 Aufträgen steigt die Zeit und die Anzahl an Simulationen wieder sehr stark an, das liegt daran, dass mit Auftrag 5 ein weiteres Produkt auf der Eisenhütte dazu kommt. Damit steigen die möglichen Kombinationen auf der Eisenhütte wieder an was zu einer längeren Rechenzeit führt. Das beste Ergebnis wurde jedoch schon nach 8637 Simulationen gefunden. Die Ergebnisfindung wurde hier nach 30 Minuten abgebrochen.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Aufträge | Zeit | Planning Items | Kombinationen | Simulationen | Ergebnis | Ergebnis FCFS |
| 1 | 00:00:00 | 7 | 5040 | 29 | 21600 | 22800 |
| 2 | 00:00:00 | 9 | 362880 | 245 | 21600 | 22800 |
| 3 | 00:00:01 | 11 | 39916800 | 823 | 21600 | 22800 |
| 4 | 00:00:49 | 18 | 6,40237E+15 | 73678 | 22140 | 99900 |
| 5 | 00:30:00 | 25 | 1,55112E+25 | 2172073 | 22350 | 99900 |

Tabelle : Ergebnisse durchmischte Auftragsliste

## Liste mit ähnlichen Produkten

In der Liste mit ähnlichen Produkten haben wir Couch, Stahl und Nägel hinzugefügt. Hier tritt ein ähnlicher Fall wie in der durchmischten Liste auf. Da die Produktion der Couch, Nägel braucht und Nägeln Stahl brauchen steigt auch hier die Anzahl der Kombinationen an einer Maschine, was zu einer hohen Laufzeit führt. Die Auftragsliste ist in Tabelle 3 abgebildet. Tabelle 4 zeigt die Ergebnisse mit dieser Liste.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Areal | Produkte | Menge | Ertrag | Col | Engine | Zeit |
| 1 | Couch | 236 | 4202 | GR | x | 60 |
| 5 | Stahl | 101 | 3638 | BL | x | 30 |
| 5 | Couch | 80 | 3688 | x | S | 15 |
| 4 | Nägel | 105 | 1563 | GR | x | 60 |
| 3 | Stahl | 189 | 2284 | PU | x | 60 |

Tabelle : Ähnliche Produkte

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Aufträge | Zeit | Planning Items | Kombinationen | Simulationen | Ergebnis | Ergebnis FCFS |
| 1 | 00:00:00 | 7 | 5040 | 29 | 21600 | 22800 |
| 2 | 00:00:00 | 9 | 362880 | 245 | 21600 | 22800 |
| 3 | 00:00:16 | 16 | 2,09228E+13 | 29990 | 22680 | 22800 |
| 4 | 00:30:00 | 23 | 2,5852E+22 | 2439402 | 23400 | 99990 |
| 5 | 00:30:00 | 25 | 1,55112E+25 | 1589384 | 26040 | 99990 |

Tabelle : Ergebnis ähnliche Produkte

In Tabelle 4 ist gut zu sehen, dass bei einer hohen Anzahl an Produktionen auf einer Maschine bzw. dass so bald mehr als 20 *PlanningItems* zum Lösen benötigt werden, sodass die Rechenzeit des Algorithmus schneller steigt. Hier wurde auch ein Abbruch nach 30 Minuten gemacht.

## Liste mit verschiedenen Prozessen

In der Liste mit verschiedenen Prozessen haben wir Aufträge welche parallel abgearbeitet werden können. Die Liste ist in Tabelle 5 abgebildet. Wir haben uns dazu für die Produkte Eisenpulver, Benzin, Mehl und Kunststoffmöbel entschieden, da diese nur wenige Überschneidungen der Produktionsprozesse haben.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Areal | Produkte | Menge | Ertrag | Col | Engine | Zeit |
| 5 | Eisenpulver | 80 | 2493 | YE | x | 15 |
| 5 | Benzin | 45 | 2377 | GR | x | 30 |
| 5 | Mehl | 21 | 1011 | BL | x | 30 |
| 5 | Kunststoffmöbel | 96 | 4024 | BL | x | 30 |
| 5 | Eisenpulver | 60 | 1172 | GR | x | 15 |

Tabelle : Verschiedene Prozesse

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Aufträge | Zeit | Planning Items | Kombinationen | Simulationen | Ergebnis | Ergebnis FCFS |
| 1 | 00:00:00 | 7 | 5040 | 132 | 3810 | 3810 |
| 2 | 00:00:00 | 10 | 3628800 | 1863 | 5700 | 5700 |
| 3 | 00:00:01 | 13 | 6227020800 | 3732 | 5700 | 5700 |
| 4 | 00:08:58 | 19 | 1,21645E+17 | 1562685 | 9390 | 11000 |
| 5 | 00:00:12 | 24 | 6,20448E+23 | 28129 | 9930 | 11100 |

Tabelle : Ergebnis verschiedene Prozesse

Hier kommt ein sehr ähnliches Ergebnis zum Vorschein. Sobald über 20 verschiedene Kombinationen möglich sind, wird der Algorithmus langsam.

## Zusammenfassen der Materialien

In unseren Algorithmus wurde auch die Möglichkeit eingebaut, das Holen der Materialen zu verdichten damit weniger *PlanningItems* zur Planung entstehen. Wir haben diese Möglichkeit mit der Auftragsliste mit den verschiedenen Prozessen getestet. In Tabelle 7 ist gut zusehen, wie sich die Laufzeit verändert.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Aufträge | Zeit | Planning Items | Kombinationen | Simulationen | Ergebnis | Ergebnis FCFS |
| 1 | 00:00:00 | 5 | 120 | 0 | 3810 | 3810 |
| 2 | 00:00:00 | 8 | 40320 | 48 | 5700 | 5700 |
| 3 | 00:00:01 | 11 | 39916800 | 3732 | 5700 | 5700 |
| 4 | 00:02:17 | 15 | 1,30767E+12 | 399008 | 9390 | 11000 |
| 5 | 00:00:12 | 18 | 6,40237E+15 | 28129 | 9900 | 11100 |

Tabelle : Ergebnis Zusammenfassen der Materialien

Für 5 Aufträge ist die Laufzeit auch deswegen schneller, da durch das Zusammenfassen der Aufträge die Kombinationsmöglichkeiten geringer werden und es eher der Fall ist, dass zum Beispiel das Holen der Materialen nicht vor der dem eigentlichen Produzieren ist. Hier greifen dann auch die Cutting Planes, also Lösungen, welche wir wegschneiden.

## Zusammenfassung

Unser Algorithmus hat gewisse Probleme mit der Laufzeit so bald viele Entscheidungen zu treffen sind. Zwar greifen die Cutting Planes in manchen Fällen, jedoch liegt hier sicher noch Verbesserungspotential. Auch eine Parallelisierung für das Branchen würde die Laufzeit des Algorithmus verbessern.