Projektbericht zum Master-Projekt: "Prototypenhafte Implementierung eines qualitativ-optimierten Konzepts für die Zuordnung von Artikeln zu Lagerplätzen"

Oleksandr Tokunov

Hochschule Bonn-Rhein-Sieg,
Grantham-Allee. 20, 53757 Sankt Augustin, Deutschland

in Kooperation mit

Ehrhardt + Partner GmbH & Co. KG,
Alte Römerstraße 3, 56154 Boppard-Buchholz, Deutschland

Zusammenfassung. Folgender Projektbericht handelt von der Konzeption und Umsetzung eines qualitativ-optimierten Zuordnungsverfahrens von Artikeln zu Lagerplätzen. Das Verfahren basiert auf mathematischen Funktionen, welche zunächst in Form von entsprechenden Algorithmen abgebildet werden müssen, sowie einer entsprechend mathematisch beschriebenen Spezialisierungshierarchie. Die Umsetzung des Konzepts in Form einea lauffähigen Prototypen dient dazu, das Verfahren auf seine Korrektheit, Vollständigkeit und Optimalität zu untersuchen.

27. März 2014

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Der grundlegende Ablauf in einem modernen Lagersystem scheint auf den ersten Blick recht simpel zu sein: die angelieferte Ware wird meist nicht direkt benötigt und zunächst zur Seite gelegt, bis der Bedarf daran durch den Kunden signalisiert wird. Nun wird die Ware wieder hervorgeholt und an den Kunden übergeben. Der Basisprozess lässt sich also auf vier wesentliche Schritte reduzieren: *Empfangen, Lagern, Entnehmen* und *Versenden*.
Die Komplexität des zuerst simpel wirkenden Prozesses steigt jedoch in der Praxis durch Zeit-, Qualitäts-, und Kostenansprüche, sowie eine Verkettung äußerer Einflüsse. Es besteht also Optimierungsbedarf, um die Effizienz der Lagerhaltung zu steigern. Das Hauptziel dieses Masterprojekts ist die prototypenhafte Umsetzung einiger Vorschläge für die Optimierung der Lagerplatzvergabe.

Die Gesamtkosten für vorhersehbare Zugriffe auf die Lagereinheiten (bspw. bei einer Kommissionierung von Bestellungen) sollen durch die optimale Lagerplatzzuordnung möglichst gering gehalten werden. Bei der Vergabe des Lagerplatzes müssen jedoch einige Parameter berücksichtigt werden, welche sich sowohl aus den physischen Anforderungen des Lagergutes, der betriebstechnisch optimalen Lageroperation sowie aus der sicherheitstechnischen und rechtlichen Betrachtung ergeben. Einige Beispiele sind in der Abbildung 1 aufgelistet.

Parameter	Anforderungen
technische Anforderungen	Beachtung zulässiger Fach- und Feldlasten gleichmäßige Regalbelastung und Vermeidung einseitiger Belastungen, insbesondere in dynamischen Lagersystemen Lagerfachvolumen optimal nutzen
betriebliche Optimierung	Fahr- und Transportwege minimieren Umschlagleistung maximieren maximale Nutzung der vorhandenen Lagerkapazität hohe Verfügbarkeit, d. h. Zugriffssicherheit auch bei Ausfall einzelner Transport- oder Bediengeräte schnelles Finden und Identifizieren der Waren in manuellen Systemen
sicherheits-technische und rechtliche Vorgaben	Beachtung von Zusammenlagerungsverboten (Gefahrgutlagerung) Getrenntlagerung (Lebensmittelbereich) Chargengruppierung

Abb. 1. Einige Kriterien zur Lagerplatzvergabe [5]

Es existieren bereits einige Belegungsstrategien für die Bestimmung der Plätze und Zonen für die Artikel, um eine möglichst gute Platznutzung und kurze Wege für die Ein- und Auslagerung zu erreichen. Viele dieser Verfahren greifen ausschließlich auf Attribute der Stellplätze zurück und verfolgen teilweise unterschiedliche Ziele (s. Abbildung 2).

Die angelieferten Artikel besitzen bestimmte Merkmale, welche in deren Stammdaten aufgeführt sind. Hierzu zählen beispielsweise solche Merkmale, wie *Artikelnummer, -gewicht, -abmessungen, Prioritätsklassifikation, Produktions- und Verfalldatum, Lagerbedingungen (klimatische Verhältnisse, Gefährdungspotential)* usw. Die Entscheidung, ob ein Lagerplatz für die Lieferung geeignet ist oder nicht, hängt also von mehreren Artikelmerkmalen ab. Häufig tritt die Situation auf, dass mehrere Lagerplätze für die Lagerung der angelieferten Artikel geeignet sind. Grund hierfür ist, dass die meisten Artikelmerkmale im Wesentlichen unabhängig voneinander sind.

Bezeichnung	Strategie	Zielsetzung
Festplatzlagerung	feste Zuordnung eines Lagerplatzes für einen bestimmten Artikel	Zugriffssicherheit bei Ausfall des Verwaltungssystems schneller Zugriff in manuellen Kommissioniersystemen durch Übung (Verringerung der Suchzeiten)
freie Platzvergabe (chaotische Lagerung)	Artikel kann auf einem beliebigen freien Lagerplatz eingelagert werden	maximale Ausnutzung der Lagerkapazität
Zonung	Wahl des Lagerplatzes entsprechend der Umschlaghäufigkeit des Artikels	Erhöhung der Umschlagleistung durch Minimierung der mittleren Weglänge
Querverteilung	Lagerung mehrerer Einheiten eines Artikels verteilt über mehrere Lagergassen	Verfügbarkeit des Artikels bei Ausfall eines Regalförderzeuges Erhöhung der Umschlagleistung durch Parallelisierung
Pick-/Teilefamilien *Clustering*	benachbarte Lagerorte für häufig kombinierte Artikel	Erhöhung der Umschlag- bzw. Kommissionierleistung durch Minimierung der Anschlusswege
kürzester Fahrweg	Anfahrt des Lagerplatzes mit kürzestem Weg	Erhöhung der Umschlagleistung durch Minimierung der Anschlusswege
Vorpufferung	in Spitzenzeiten Einlagerung auf vorderen Lagerbereich	Vermeidung von Rückstau durch Erhöhung des Durchsatzes

Abb. 2. Operationelle Lagerplatzvergabestrategien [5]

Die Entscheidung über den optimalen Lagerplatz sieht meistens die exakte Übereinstimmung der Merkmalausprägungen für Lieferungen und Lagerplätzen vor. Allerdings besteht hier ebenfalls ein Optimierungsbedarf. So kann beispielsweise eine Lieferung mit Zugriffshäufigkeit (ABC-Status) A auch an einem Lagerplatz mit ABC-Status B eingelagert werden, falls kein Lagerplatz mit ABC-Status A verfügbar ist.

Das Ziel dieses Masterprojekts ist, qualitative Optimierung mittels ausgewählter Konzepte für die Lagerplatzvergabe zu eruieren und zu bewerten. Die Verfahren basieren auf mathematischen Funktionen sowie einer entsprechend

mathematisch beschriebenen Spezialisierungshierarchie für die Merkmalsausprägungen. Die qualitativ optimierte Zuordnung soll folgenden Anforderungen genügen:

- *Korrektheit*: Das Zuordnungsverfahren muss gewährleisten, dass für eine Lieferung nur solche Lagerplätze ausgewählt werden, die geeignet sind.
- *Vollständigkeit*: Wenn für eine Lieferung mindestens ein geeigneter Lagerplatz zur Verfügung steht, muss das Zuordnungsverfahren eine Zuordnung vornehmen.
- *Optimalität*: Wenn das Zuordnungsverfahren einer Lieferung L den Lagerplatz P zuweist, dann ist P in einem gewissen Sinne optimal.

Die Verwaltung großer Lagersysteme wird heutzutage mittels entsprechender Software automatisiert ausgeführt. Um die entsprechenden Zuordnungsalgorithmen in der Praxis einzusetzen, müssen diese zunächst prototypenhaft implementiert und getestet werden. Die Implementierung soll folgenden wesentlichen Anforderungen genügen:

- *Effektivität*: die Zuordnung ist berechenbar auf gängigen Systemen.
- *Flexibilität*: die Art der Zuordnung ist gleich für jede Anzahl von Merkmalen und für jede Anzahl von Ausprägungen der einzelnen Merkmale.
- *Nachvollziehbarkeit*: aus der berechneten Zuordnung für eine Lieferung können die für die Lieferung zur Berechnung der Zuordnung angegebenen Merkmale zurückgerechnet werden; sollte also z.B. eine Lieferung eine Zuordnung bekommen, für die es keinen entsprechenden Lagerplatz gibt, kann festgestellt werden, welche (möglicherweise fehlerhaften) Eingaben dazu geführt haben.
- *Effizienz*: Zuordnung und Rückrechnung sind praktisch effizient berechenbar.
- *Praktikabilität*: die vorgeschlagenen Funktionen für die Zuordnung sind unmittelbar implementierbar.

Anmerkung: Im Folgenden werden Begriffe *Gödelisierung* und *Gödelnummer* benutzt. Ursprünglich gehen diese auf Kurt Gödel (1906 - 1978) zurück, der als einer der bedeutensten Logiker der Neuzeit gilt. Er benutzte Nummerierungen zur Kodierung von formalen Objekten, wie z.B. logischen Ausdrücken, um fundamentale Aussagen über formale Systeme (Logiken) zu beweisen. Das optimierte Konzept, welches in diesem Master-Projekt implementiert und eruiert wird, benötigt ein Verfahren, welches einem numerischen Vektor eine eindeutige Kennzahl zuordnet. Die Begriffe Gödelisierung und Gödelnummer werden also im Zusammenhang mit den verschiedenen Verfahren der eindeutigen Kennzahlenzuordnung bzw. für die konkreten Kennzahlen verwendet.

2 Konzeption der benötigten Algorithmen für den Prototypen

2.1 Prototyp-Algorithmus

Der konzipierte Algorithmus, welcher im Folgenden detailliert vorgestellt wird, setzt die qualitativ-optimierte Lagerplatzvergabe mittels mathematisch beschriebener Spezialisierungshierarchie [1] um. Das Konzept wurde unabhängig von der Anzahl an konkreten Merkmalen bzw. Merkmalausprägungen aufgestellt und bietet somit die gewünschte Flexibilität. Das Algorithmusbeispiel in der Abbildung 3 dient als Codebeispiel, um die Funktionsweise des Prototypen anschaulich darzustellen.

Der Algorithmus benötigt als Input die Beschreibung der Merkmale einer Lieferung. Diese werden in einem Array $L = [x, y, z, ...]$ übergeben. Die jeweiligen Einträge des Arrays (x, y, z usw.) sind die Ausprägungen des jeweiligen Merkmals. Die Anzahl der Merkmale ist hier irrelevant, lediglich die Reihenfolge, in der die Merkmale gelistet sind, soll bekannt sein. Dies ist für den ersten Schritt entscheidend, denn die Einträge des Arrays L müssen zuächst numerisch abgebildet werden. Dies geschieht mittels der Funktion *LookUp()*, welche mit den Nachschlagetabellen in der Datenbank arbeitet und der jeweiligen Merkmalausprägung eine eindeutige Zahl zuordnet, entsprechend der Ordnung. Die hierfür notwendige Datenbankstruktur sowie deren konkrete Umsetzung wird im Kapitel 3.1 beschrieben. Als Ergebnis entsteht ein Array $L' = [x, y, z, ...]$ mit den numerischen Einträgen für die übergebenen Merkmalausprägungen der Lieferung.

Die Lagerplätze eines Lagers mit den selben Merkmalausprägungen werden zu Lagerplatzklassen zusammengefasst. Die Zugehörigkeit eines Lagerplatzes zu einer Lagerplatzklasse wird durch einen zusätzlichen Eintrag in der Datenbankzeile gekennzeichnet, welcher auf den entsprechenden Tabelleneintrag der Lagerplatzklasse verweist. Für die Lagerplatzklassen muss also eine separate Tabelle angelegt werden. Da das Konzept mit den numerischen Abbildungen arbeitet, müssen auch die Merkmalausprägungen der Lagerplatzklassen numerisch vorliegen (entsprechend der Ordnung). Um große Tabellen zu vermeiden (separater Eintrag pro Merkmal), wird aus den Zahlen eine eindeutige *Gödelnummer* gebildet und als Tabelleneintrag abgespeichert. Eine Abfrage, ob die jeweilige Lagerplatzklasse über freie Lagerplätze verfügt, soll ebenfalls möglich sein.

Die Funktion *Lagerplatzklassen()* soll eine Schnittstelle zu der Datenbank des Lagers bzw. des Systems repräsentieren, welche die Auskunft über aktuell verfügbare Lagerplatzklassen erteilt. Als Rückgabe erhält das Programm ein Array $P = [x, y, z, ..]$, dessen Eintrge (x, y, z usw.) Gödelnummern der verfügbaren Lagerplatzklassen sind. Diese werden nun mittels der Umkehrfunktion f'_g zurücktransformiert, sodass für jede Lagerplatzklasse wieder ein Array entsteht mit den numerischen Abbildungen ihrer Merkmalausprägungen. Schließlich werden diese

in einem zweidimensionalen Array $P_z = [[a, b, c, ...], [d, e, f, ...], ...]$ zusammengefasst, dessen Länge von der Anzahl der verügbaren Lagerplatzklassen abhängig ist. Die einzelnen Vektoren innerhalb des Arrays haben dieselbe Länge, wie das Array L'. Gilt diese Bedingung nicht, so haben die Lieferung und Lagerplatzklassen bzw. Lagerplätze unterschiedliche Anzahl von Merkmalen.

Die vorherigen Schritte gelten als Vorbereitung für die eigentliche Zuordnungsroutine. Diese basiert letztendlich auf einer Modulo-Operation, welche überprüft, ob die numerische Abbildung der jeweiligen Merkmalausprägung der Lieferung ein Teiler der numerischen Abbildung der Merkmalausprägung desselben Merkmals einer der verügbaren Lagerplatzklassen ist. Entscheidend und wichtig in diesem Punkt ist die Reihenfolge der Merkmale, in der die Ausprägungen im Array angegeben sind. Diese muss bei beiden Vektoren identisch sein (Lieferung und Lagerplatzklasse). Die Routine besteht aus zwei **for**-Schleifen und einer boolischen Hilfsvariable *bool*. Die erste Schleife wählt aus dem zweidimensionalen Array P_z eine verfügbare Lagerplatzklasse aus und setzt zunächst die Hilfsvariable auf *false*. Die zweite durchläuft alle Einträge des dazugehörigen Vektors. Da die Länge des Vektors einer Lagerplatzklasse mit der der Lieferung übereinstimmt (nach Definition), wird mit derselben Zählvariable auch das Array L' durchlaufen. Hier geschieht nun komponentenweise die Modulo-Überprüfung. Schlägt die Modulo-Überprüfung einmal fehl, so ist die ausgewählte Lagerplatzklasse nicht für die Lagerung der Lieferung geeignet. In diesem Fall wird die zweite Schleife abgebrochen und der Wert der Hilfsvariable bleibt auf *false*. Falls sich die mathematische Beziehung aller Merkmalausprägungen der Lieferung mit den Ausprägungen der gewählten Lagerplatzklasse bestätigt hat ($L'(j)$ ein Teiler von $P_z(i)(j)$ für alle j), so kann diese Lagerplatzklasse als für die Lieferung L geeignet bezeichnet werden. Der Wert der Hilfsvariable wird also auf *true* gesetzt und in der äußeren Schleife wird dieser abgefragt. Hiervon hängt die Entscheidung ab, ob die gewählte Lagerplatzklasse $P_z(i)$ zu der Menge der geeigneten Klassen P_{gz} zugeordnet wird oder nicht. Als Ergebnis dieser Routine entsteht also ein zweidimensionales Array $P_{gz} = [[a, b, c, ...], [d, e, f, ...], ...]$, welches alle für die Lagerung der Lieferung L geeigneten Lagerplatzklassen in Form von numerischen Vektoren beinhaltet.

Das Konzept der qualitativ-optimierten Lagerplatzvergabe sieht vor, die Gesamtmenge der geeigneten Lagerplatzklassen nur auf die speziellsten Lagerplatzklassen zu reduzieren [1]. Diese reduzierte Menge ist bekannt als *Pareto-Menge*. In dem Algorithmus wird hierfür eine Funktion *Pareto* aufgerufen, welche das zweidimensionale Array P_{gz} auf die besagte Pareto-Menge reduziert. Der entsprechende Algorithmus wird im Kapitel 2.3 veranschaulicht und erläutert. Die ermittelte Pareto-Menge wird als zweidimensionales Array P_{pz} zurückgegeben.

Die Lagerplatzklassen sind ursprünglich in Form von Gödelnummern in der Datenbanktabelle eingetragen. Für die Zuordnungsroutine des Prototypen wurden diese mittels der Umkehrfunktion transformiert. Nun müssen die Gödelnummern der speziellsten Lagerplatzklassen aus der Pareto-Menge P_{pz} erneut

bestimmt werden. Hierfür wird die entsprechende Funktion f_g benutzt. Als endgültige Ausgabe entsteht also ein Array $P_{pg} = [x, y, z, ...]$, dessen Einträge die Gödelnummern der speziellsten Lagerplatzklassen für die Lieferung L sind.

Input : $L = [x, y, z, ...]$ mit x, y, z - Ausprägungen der jeweiligen Merkmale
for $i = 0$ **to** $n = |L|$ **do**
  $L'(i) =$ `LookUp(`$L(i)$`);`
end
Output: $L' = [x, y, z, ...]$ mit $x, y, z \in \mathcal{N}$
Funktion `Lagerplatzklassen()`
  `/* Aktuell verfügare Lagerplatzklassen */`
  return $P = [x, y, z, ...]$ mit x, y, z - Gödelnummern
for $i = 0$ **to** $n = |P|$ **do**
  $P_z(i) = f'_g($`(P(i)) ;` `/* Umkehrfunktion der Gödelisierung */`
end
Output: $P_z = [[a, b, c, ...], [d, e, f, ...], ...]$ mit $a, b, c, d, e, f \in \mathcal{N}$
for $i = 0$ **to** $n = |P_z|$ **do**
  $bool = false$;
  for $j = 0$ **to** $n = |P_z(i)|$ **do**
    if $P_z(i)(j) \bmod L'(j) \neq 0$ **then**
      break;
    end
    $bool = true$;
  end
  if $bool == true$ **then**
    $P_{gz}(i) = P_z(i)$;
  end
end
Output: $P_{gz} = [[a, b, c, ...], [d, e, f, ...], ...]$ mit $a, b, c, d, e, f \in \mathcal{N}$
`/* Die Gesamtmenge der geeigneten Lagerplatzklassen */`
Funktion `Pareto(`P_{gz}`)`
  `/* Die Gesamtmenge auf die Prateo-Menge reduzieren */`
  return $P_{pz} = [[a, b, c, ...], [d, e, f, ...], ...]$ *mit* $a, b, c, d, e, f \in \mathcal{N}$
for $i = 0$ **to** $n = |P_{pz}|$ **do**
  $P_{pg} = f_g($`(`$P_{pz}(i)$`);`
end
Output: $P_{pg} = [x, y, z, ...]$ mit x, y, z - Gödelnummern
`/* Pareto-Menge mit Gödelnummern der Lagerplatzklassen */`

Abb. 3. Algorithmus zum Ablauf des Programms

2.2 Eindeutige Kennzahlenzuordnung (Gödelnummer)

Der im Kapitel 2.1 konzipierte Prototyp-Algorithmus besitzt eine Funktion, welche aus der eindeutig zugeordneten Kennzahl (sogenannte *Gödelnummer*) einer bestimmten Lagerplatzklasse den dazugehörigen numerischen Vektor bilden kann. Außerdem existiert eine inverse Funktion, welche die ursprüngliche Kennzahl wieder errechnen kann. Das Verfahren, welches hierfür benutzt wird, nennt sich *Gödelisierung* und kann mittels bestimmten mathematischen Algorithmen realisiert werden. Die Gödelisierung ermöglicht die Abspeicherung der Lagerplatzklassen, basierend auf deren Eigenschaften, lediglich in einer einzigen Spalte (als Gödelnummer). Die Auflistung jedes einzelnen Parameters des dazugehörigen numerischen Vektors in einer Datenbank wäre weniger effizient.
Eine geläufige Kodierung mehrerer Zahlenfaktoren zu einer eindeutigen Kennzahl stellt die Funktion g [2] dar. Hierbei werden die Ordnungsnummern der jeweiligen Merkmalausprägung aus dem Merkmalsvektor als Exponenten von Primfaktoren genutzt. Schließlich werden die errechneten Werte miteinander multipliziert. Somit entsteht eine Kennzahl, welche wiederum mittels Primzahlzerlegung eindeutig in die einzelnen Ordnungsnummern zerlegt werden kann.
Eine weitere Möglichkeit zur Umsetzung der Gödelisierung sowie deren Inverse stellt die *Zwiebelfunktion* $f_{m;n}$ dar [2], welche für $n = 2$ Merkmale und $m \in \mathbb{N}_0$ definiert ist durch

$$f(i,j) = \begin{cases} j^2 + i, & i \leq j \\ (i+1)^2 - j - 1, & i \geq j \end{cases}$$

Die optimale eineindeutige Zuordnung wird für den Fall, dass beide Merkmale dieselbe Anzahl m von Ausprägungen haben, mittels einer „schalenförmigen“ Abzählung realisiert:

	0	1	2	3	4	…
0	0	1	4	9	16	…
1	3	2	5	10	17	…
2	8	7	6	11	18	…
3	15	14	13	12	19	…
4	24	23	22	21	20	…
⋮	⋮	⋮	⋮	⋮	⋮	

Gleiches gilt für die Verallgemeinerung auf

$$f_{m;n} : [\,0, m\,]^n \to [\,0, (m+1)^n - 1\,]$$

Für den Fall $n \geq 3$ ist die Funktion rekursiv definiert durch

$$f_{m;n}(x_1, \ldots, x_{n-1}, x_n) = f_{m;n-1}(x_1, \ldots, x_{n-1}) + x_n \cdot (m+1)^{n-1}$$

oder durch folgende nach Auflösung der Rekursion äquivalente Darstellung:

$$f_{m;n}(x_1, \ldots, x_n) = f(x_1, x_2) + \sum_{i=3}^{n} x_i \cdot (m+1)^{i-1}$$

Dabei gilt $f_{m;2}(i, j) = f(i, j)$. Außer der Anzahl der Merkmale n benötigt die Funktion den Wert m, welcher die größtmögliche Merkmalausprägung repräsentiert. Der Algorithmus zur Bestimmung der Inversen dieser Funktion ermöglicht ebenfalls eine eindeutige Zerlegung in die ursprünglichen Ordnungszahlen. Der Algorithmus in Abbildung 4 beschreibt ein Verfahren zur Berechnung der Inversen der Funktionen $f_{m;n}$. Dieser Algorithmus benutzt einen Algorithmus zur Berechnung der Inversen von f, der in Abbildung 5 dargestellt ist.

```
algorithm ZwiebelInvers
        input:   n ∈ ℕ, n ≥ 2; m ∈ ℕ0; y ∈ ℕ0;
        output: f⁻¹m;n(y) = x = (x1, ..., xn) ∈ ℕ0^n mit fm;n(x1, ..., xn) = y;
        yy := y;
        for i := n to 3 by −1 do
            d := (m + 1)^(i−1);
            r := yy mod d;
            x[i] := yy div d;
            yy := r
        endfor;
            x[1, 2] := FInvers(yy)
    endalgorithm ZwiebelInvers
```

Abb. 4. Algorithmus ZwiebelInvers zur Berechnung der Inversen von $f_{m;n}$.

Beide Verfahren sind eindeutig, d.h. bei ihrer Verwendung können mögliche Fehlereingaben erkannt werden. Durch die Zerlegung der Kennzahl in die ursprüngliche Faktoren mittels der jeweiligen inversen Funktion kann die Fehlereingabe genau bestimmt werden. Der direkte Vergleich der beiden Funktionen ergibt einige Vor- und Nachteile [2]. So ist die erste Funktion g durch ihre Unabhängigkeit von der Anzahl n der Merkmale und von der Anzahl der Merkmalsausprägungen bzw. von der größtmöglichen Merkmalausprägung m dynamisch und flexibel. Allerdings können die ermittelten Kennzahlen sehr schnell von einer sehr großen Größenordnung werden. Im Gegensatz hierzu ist die Zwiebelfunktion $f_{m;n}$ um einiges effizienter. Die errechneten Werte werden deutlich kleiner als die von g. Andererseits ist die Funktion $f_{m;n}$ durch die Notwendigkeit eines zusätzlichen Parameters m weniger flexibel.

```
algorithm FInvers
    input:   y ∈ ℕ₀;
    output:  f⁻¹(y) = x = (x₁, x₂) ∈ ℕ₀² mit f(x₁, x₂) = y;
    l := ⌊√y⌋;
    gefunden := false;
    if y ≤ l · (l + 1) then
        for i := 0 to l by 1 while not gefunden do
            if y = f(i, l) then gefunden := true endif
        endfor;
        x[1, 2] := (i, l);
    else
        for j := 0 to l − 1 by 1 while not gefunden do
            if y = f(l, j) then gefunden := true endif
        endfor;
        x[1, 2] := (l, j);
    endif;
endalgorithm FInvers
```

Abb. 5. Algorithmus FInvers zur Berechnung der Inversen der Funktion f.

Im Laufe des Projekts werden beide Verfahren zur eindeutigen Kennzahlzuordnung sowie deren Inverse implementiert. Schließlich wird mittels mehrerer Tests mit verschiedenen Werten für n und m ermittelt werden, welches Verfahren für den endgültigen Algorithmus besser geeignet ist. Hierbei werden die erwähnten Vor- und Nachteile mitberücksichtigt.

2.3 Pareto-Algorithmus

Mittels des im Kapitel 2.1 vorgestellten Algorithmus wird zunächst eine Menge aller theoretisch geeigneten Lagerplatzklassen bestimmt. Abhängig von der aktuellen Verfübarkeit der Lagerplatzklassen kann diese sowohl die speziellsten, als auch weniger speziellen Klassen beinhalten. Die weniger speziellen Klassen sollen aus der Menge aussortiert werden, falls speziellere vorhanden sind. Es gilt also, die speziellste Klasse aus den geeigneten zu ermitteln. Existieren allerdings mehrere Klassen, welche in $n-1$ Kriterien spezieller sind, so entsteht eine Menge, in der nur die speziellsten Klassen enthalten sind. Diese wird nach dem Prinzip der Pareto-Optimierung ermittelt und entsprechend als Pareto-Menge bezeichnet [1].
Die Herausforderung bei der Umsetzung dieser Optimierung besteht darin, den Algorithmus möglichst effizient zu entwerfen. Grundsätzlich basiert dieser auf einem elementaren Vergleich der Ordnungsnummern (Ausprägungen) für das jeweilige Merkmal. Es wäre jedoch nicht effizient, alle möglichen Kombinationen der Ordnungsnummern auszuwerten. Aus diesem Grund wurde ein Algorithmus konzipiert, welcher die Pareto-Menge effizienter ermittelt. Im Folgenden wird der Ablauf des Algorithmus detailliert vorgestellt. Das Codebeispiel in der Abbildung 6 veranschaulicht seine Funktionsweise.

Der Funktion *Pareto* wird eine Menge aller aktuell verfügbaren, geeigneten Lagerplatzklassen übergeben. Diese wird mittels eines zweidimensionalen Arrays P_{gz} repräsentiert. Nun werden alle ermittelten Lagerplatzklassen überprüft. Für den weiteren Ablauf des Algorithmus werden Hilfsvariablen benötigt. Die Variablen *dom* und *count* sind Zählvariablen, und *tmp* ist ein temporärer Zwischenspeicher für die Ergebnisse der jeweiligen Iteration. Schließlich entsteht ein Array P_{pz}, welches nur die speziellsten Lagerplatzklassen beinhaltet.

Zu Beginn wird das erste Element des Arrays P_{gz} in das Ergebnisarray geschrieben. Diese Lagerplatzklasse dient nun als Ausgangswert für den weiteren Verlauf. Das Zwischenergebnis (die aktuellen Werte des Arrays P_{gz}) wird in *tmp* kopiert. Im nächsten Schritt folgt der Vergleich der aktuell im *tmp* vorhandenen Lagerplatzklassen mit einer weiteren Lagerplatzklasse aus dem Array P_{gz}. Hier soll ermittelt werden, ob und wie viele Parameter der untersuchten Klasse aus P_{gz} spezieller (oder gleichwertig) sind, als die der bereits zwischengespeicherten Klassen in *tmp* bzw. P_{pz}.

Laut Definition der Pareto-optimalen Lösungsmenge [1] soll es keine von X verschiedene Lösung Y existieren, die bezüglich allen Kriterien mindestens genau so gut wie X ist. Falls jedoch mehrere Lösungen in der Pareto-Menge vorhanden sind, so wird es immer zwei Kriterien geben, bezüglich derer X besser als Y ist bzw. umgekehrt. Hieraus folgt, dass nach dem Vergleich über alle Werte der untersuchten Lagerplatzklasse diese nur dann relevant ist, wenn sie im Bezug auf mindestens $n-1$ Kriterien spezieller (oder gleichwertig) ist als bereits ausgewählte Klassen. Die Zählvariable *count* dient hier als Indikator und wird im

Anschluss ausgewertet. Tritt der Fall auf, dass die untersuchte Lagerplatzklasse in allen Kriterien spezieller ist, als eine der bereits ausgewählten, so wird diese Klasse aus *tmp* entfernt. Somit werden Klassen aus der Gesamtmenge der für die Lieferung geeigneten Lagerplatzklassen eliminiert, welche allgemeinere Parameter besitzen, um die speziellsten möglichst sinnvoll auszunutzen.

Wie bereits erwähnt können mehrere speziellste Lagerplatzklassen in der Pareto-Menge enthalten sein. Enthält also *tmp* bereits mehrere Klassen, so soll die Untersuchung der nächsten Klasse im Bezug auf alle diese ausgewählten Klassen geschehen. Die Zählvariable *dom* dient als Indikator für die Anzahl der Klassen aus *tmp*, die weniger speziell sind als die aktuell untersuchte Klasse. Die Auswertung dieser Variable ist abschließend dafür entscheidend, ob die untersuchte Lagerplatzklasse zwischengespeichert bzw. der Pareto-Menge hinzugefügt wird oder nicht (falls diese also im Bezug auf nicht alle Klassen spezieller ist). Das Zwischenergebnis *tmp* wird so mit jeder Iteration aktualisiert, bis alle Klassen aus P_{gz} untersucht wurden. Das aktualisierte Array *tmp* wird in das Ergebnisarray P_{pz} kopiert, welches letzten Endes die Pareto-optimierte Menge der für die Lieferung geeigneten, aktuell verfügbaren Lagerplatzklassen repräsentiert.

```
Input  : P_gz = [[a,b,c,...],[d,e,f,...],...] mit a,b,c,d,e,f ∈ 𝒩
/* Die Gesamtmenge der geeigneten Lagerplatzklassen          */
for i = 0 to n = |P_gz| do
    dom = 0;
    count = 0;
    tmp = P_pz;
    if tmp == ∅ then
        P_pz = P_gz(i);
        continue;
    end
    for j = 0 to n = |tmp| do
        for q = 0 to n = |tmp(j)| do
            if P_gz(i)(q) ≤ tmp(j)(q) then
                count++;
            end
        end
        if count ≥ |P_gz(i)| − 1 then
            /* Wenn mind. n-1 Kriterien besser oder gleich gut
               sind                                           */
            if count == |P_gz(i)| then
                /* Wenn alle Kriterien besser sind           */
                tmp −= tmp(j);
                /* Klasse dominiert, also entfernen          */
            end
            dom++;
            /* Besser in mind. n-1 Kriterien im Bezug auf 1
               Klasse                                         */
            count = 0;
        end
    end
    if dom ≥ |tmp| then
        /* Wenn im Bezug auf alle Klassen mind. um n-1
           Kriterien besser bzw. wenn dom größer, dann wurden
           Klassen eliminiert                                 */
        tmp += P_gz(i);
    end
    P_pz = tmp;
end
Output: P_pz = [[a,b,c,...],[d,e,f,...],...] mit a,b,c,d,e,f ∈ 𝒩
/* Prateo-Menge der geeigneten Lagerplatzklassen             */
```

Abb. 6. Pareto-Algorithmus

3 Umsetzung

3.1 Datenbankstruktur

Für den konzipierten Prototypen, basierend auf dem im Kapitel 2.1 vorgestellten Algorithmus, wird eine zusätzliche Datenbankstruktur benötigt. Der Grund hierfür ist die effiziente und eindeutige Abbildung der Merkmalausprägungen auf numerische Vektoren, basierend auf der Ordnung der jeweiligen Ausprägungen. Diese Abbildung wird für die Zuordnungsroutine der für eine Lieferung geeigneten Lagerplatzklassen benötigt. Die Eigenschaften der Lagerplatzklassen sind ebenfalls mittels dieser numerischen Abbildung gekennzeichnet und schließlich gödelisiert in einer separaten Datenbanktabelle abgelegt. Die Datenbankstruktur soll eine gute Flexibilität bezüglich der Anzahl an Merkmalen bzw. deren Ausprägungen aufweisen. Im Folgenden wird eine Lösung präsentiert, welche vor allem diese wichtige Eigenschaft berücksichtigt. Diese basiert auf zwei Datenbanktabellen sowie einer zusätzlichen View. Die Abbildung 7 veranschaulicht die konzipierte Datenbankstruktur.

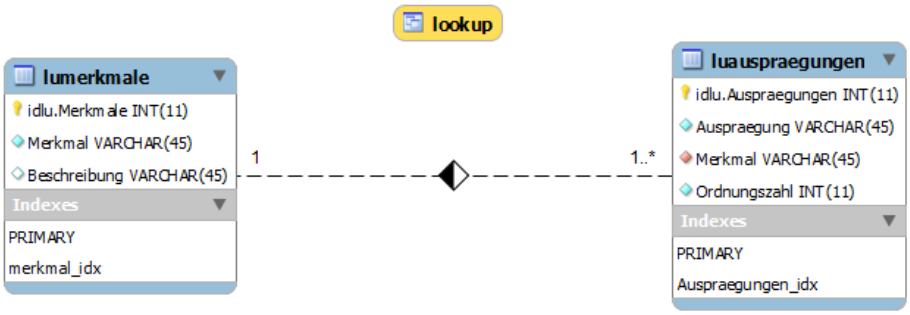

Abb. 7. Datenbankstruktur

3.1.1 Merkmal-Tabelle

Die Eigenschaften eines Lagerplatzes bzw. einer Lagerplatzklasse können abhängig von dem konkreten Lager variieren. Dementsprechend variieren auch die Stammdaten der angelieferten Artikel. Die verschiedene Merkmale, welche die Eigenschaften der Lagerplätze bzw. Bedingungen für die Lagerung einer Lieferung beschreiben, werden zunächst in einer Datenbanktabelle abgelegt. Der Zugriff auf diese kann entweder durch die eindeutige ID oder über den jeweiligen Namen des Merkmals erfolgen. Die Tabelle sieht außerdem eine Spalte für die Beschreibung des Merkmals vor. Die Erzeugung dieser Tabelle geschieht mittels des im Listing 1.1 abgebildeten SQL-Codes.

```
CREATE TABLE ‘lumerkmale‘ (
  ‘idlu.Merkmale‘ int(11) NOT NULL,
  ‘Merkmal‘ varchar(45) NOT NULL,
  ‘Beschreibung‘ varchar(45) DEFAULT NULL,
  PRIMARY KEY (‘idlu.Merkmale‘),
  KEY ‘merkmal_idx‘ (‘Merkmal‘)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Listing 1.1. Erstellen der Merkmal-Tabelle

Die Tabelle kann nun mit entsprechenden Merkmalen gefüllt werden. Die Abbildung 8 zeigt ein beispielhaftes Aussehen der Tabelle mit vier Merkmalen: *Zugriffshäufigkeit, Gewichtslimit, Fachgröße* und *Lademittel*.

idlu.Merkmale	Merkmal	Beschreibung
1	Zugriffshäufigkeit	ABC
2	Gewichtslimit	Angaben in kg
3	Fachgröße	Abmessungen in cm
4	Lademittel	Entsprechende Einheiten

Abb. 8. Merkmal-Tabelle

3.1.2 Ausprägungen

Jedes der Merkmale eines Lagerplatzes bzw. der Lieferung hat unterschiedliche Ausprägungen. Auch diese müssen zunächst in einer separaten Tabelle abgelegt werden. In dieser Tabelle werden alle möglichen Ausprägungen der Merkmale aufgelistet. Ein Fremdschlüssel verweist die Ausprägungen auf das jeweilige Merkmal aus der Merkmal-Tabelle. Dies erfolgt über die gleichnamige Spalte *Merkmal*. Die konzipierte Datenbankstruktur soll als Abhilfe für die Abbildung der Merkmalausprägungen auf numerische Vektoren dienen. Dementsprechend beinhaltet diese Tabelle eine Spalte, in welcher die zugehörige Ordnungsnummer eingetragen wird. Diese Nummer soll mittels des beschriebenen Verfahren der Codierung mit Primzahlen [1] im Vorfeld ermittelt werden, entsprechend der jeweiligen Ordnung der Merkmalausprägungen. Die Erzeugung dieser Tabelle geschieht mittels des im Listing 1.2 abgebildeten SQL-Codes.

```
CREATE TABLE ‘luauspraegungen‘ (
    ‘idlu.Auspraegungen‘ int(11) NOT NULL,
    ‘Auspraegung‘ varchar(45) NOT NULL,
    ‘Merkmal‘ varchar(45) NOT NULL,
    ‘Ordnungszahl‘ int(11) NOT NULL,
    PRIMARY KEY (‘idlu.Auspraegungen‘),
    KEY ‘Auspraegungen_idx‘ (‘Merkmal‘),
    CONSTRAINT ‘auspraegungen‘ FOREIGN KEY (‘Merkmal‘)
        REFERENCES ‘lumerkmale‘ (‘Merkmal‘)
        ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Listing 1.2. Erstellen der Tabelle für die Ausprägungen

Entsprechend dem vorherigen Beispiel wird diese Tabelle mit den jeweiligen Merkmalausprägungen sowie dazugehörigen Ordnungszahlen gefüllt. Diese ist in Abbildung 9 dargestellt.

idlu.Auspraegungen	Auspraegung	Merkmal	Ordnungszahl
1	A	Zugriffshäufigkeit	2
2	B	Zugriffshäufigkeit	6
3	C	Zugriffshäufigkeit	30
4	5	Gewichtslimit	2
5	10	Gewichtslimit	6
6	Inf	Gewichtslimit	30
7	klein	Fachgröße	2
8	flach	Fachgröße	3
9	mittel	Fachgröße	30
10	hoch	Fachgröße	210
11	KisteS	Lademittel	2
12	KisteM	Lademittel	6
13	KisteL	Lademittel	30
14	KisteXL	Lademittel	210
15	EU	Lademittel	2310

Abb. 9. Ausprägungen

3.1.3 LookUp-View

Wie bereits zu Beginn erwähnt variiert die Anzahl an Merkmalen bzw. deren Ausprägungen. Dennoch soll die konzipierte Datenbankstruktur unabhängig von den konkreten Werten bestehen und für die numerische Abbildung genutzt werden können. Diese Flexibilität wird durch eine zusätzliche View ermöglicht, welches beide vorherigen Tabellen vereinigt. Dies hat den Vorteil, dass hier alle Ausprägungen, deren Zugehörigkeit zu den jeweiligen Merkmalen und die entsprechende Ordnungsnummer dargestellt werden können, unabhängig von dem konkreten Aussehen der Merkmal- bzw. Ausprägungen-Tabelle. Die Vereinigung geschieht über die in beiden Tabellen gleichnamige Spalte *Merkmal*. Unabhängig von der konkreten Anzahl werden alle Merkmale in der View angezeigt sowie deren in der Ausprägungen-Tabelle gelisteten Ausprägungen. Weiterhin werden nur die nötigen Spalten aus beiden Tabellen in die View übernommen (*Ordnungszahl, Beschreibung*). Die Erzeugung dieser View geschieht mittels des im Listing 1.3 abgebildeten SQL-Codes.

```
CREATE
    ALGORITHM = UNDEFINED
    DEFINER = 'root'@'localhost'
    SQL SECURITY DEFINER
VIEW 'lookup' AS
    select
        'luauspraegungen'.'Auspraegung' AS 'Auspraegung',
        'lumerkmale'.'Merkmal' AS 'Merkmal',
        'luauspraegungen'.'Ordnungszahl' AS 'Ordnungszahl',
        'lumerkmale'.'Beschreibung' AS 'Beschreibung'
    from
        ('luauspraegungen'
        join 'lumerkmale')
    where
        ('luauspraegungen'.'Merkmal' = 'lumerkmale'.'Merkmal')
```

Listing 1.3. Erstellen der LookUp-View

Um das vorherige Beispiel fortzuführen wird die LookUp-View für die beiden bisher erstellten Tabellen erzeugt und in Abbildung 10 präsentiert.

Auspraegung	Merkmal	Ordnungszahl	Beschreibung
A	Zugriffshäufigkeit	2	ABC
B	Zugriffshäufigkeit	6	ABC
C	Zugriffshäufigkeit	30	ABC
5	Gewichtslimit	2	Angaben in kg
10	Gewichtslimit	6	Angaben in kg
Inf	Gewichtslimit	30	Angaben in kg
klein	Fachgröße	2	Abmessungen in cm
flach	Fachgröße	3	Abmessungen in cm
mittel	Fachgröße	30	Abmessungen in cm
hoch	Fachgröße	210	Abmessungen in cm
KisteS	Lademittel	2	Entsprechende Einheiten
KisteM	Lademittel	6	Entsprechende Einheiten
KisteL	Lademittel	30	Entsprechende Einheiten
KisteXL	Lademittel	210	Entsprechende Einheiten
EU	Lademittel	2310	Entsprechende Einheiten

Abb. 10. LookUp-View

3.1.4 LookUp-Aufruf

Der konzipierte Algorithmus, wie er im Kapitel 2.1 detailliert beschrieben wird, sieht für die numerische Abbildung der Merkmalausprägungen einer Lieferung eine Funktion *LookUp()* vor. Genau an dieser Stelle wird die LookUp-View aufgerufen, um für die gegebene Ausprägung, welche zunächst als Zeichenfolge vorliegt, die dazugehörige Ordnungszahl abzulesen. Eine beispielhafte Abfrage ist dem im Listing 1.4 abgebildeten SQL-Code zu entnehmen.

```
SELECT Ordnungszahl FROM lookup WHERE Auspraegung = 'EU'
```

Listing 1.4. Beispielhafte Abfrage

Hierbei wird die Ordnungszahl für die Ausprägung „EU“ des Merkmals „Lademittel“ abgefragt. Das zurückgelieferte Ergebnis ist in Abbildung 11 abgebildet. Somit erhält das Programm mittels dieser View sofort die nötigen Ordnungszahlen, woraus schließlich ein numerischer Vektor (Array) gebildet wird, der die Merkmalausprägungen der Lieferung repräsentiert.

Ordnungszahl
2310

Abb. 11. LookUp-Aufruf

3.2 Implementierung

Die Implementierung des konzipierten Prototypen sollte möglichst gute Plattformunabhängigkeit aufweisen sowie unabhängig von externen Komponenten geschehen. Als Programmiersprache wird C/C++ gewählt, da diese sowohl von den meisten Plattformen unterstützt wird, als auch als Basis in der Lagerverwaltungssoftware von Ehrhardt + Partner GmbH & Co. KG benutzt wird. Außerdem bietet C/C++ eine Vielzahl an Standard-Bibliotheken an, welche zur Implementierung der konzipierten Algorithmen verwendet werden können.
Ziel der Implementierung ist die Erstellung einer externen dynamischen Bibliothek, welche die Algorithmen zur Optimierung der Lagerplatzzuordnung in Form von Funktionen zur Verügung stellt. Diese kann dann in die Lagerverwaltungssoftware eingebunden und die Funktionen in den Programmablauf integriert werden.

3.2.1 Gödelisierung

Im Kapitel 2.2 wurden zwei Verfahren zur eindeutigen Kennzahlenzuordnung (auch Gödelisierung genannt) vorgestellt. Es soll nun entschieden werden, welche Funktion zur Kodierung der numerischen Vektoren (Parameter) der Lagerplatzklassen besser geeignet ist sowie höhere Effizienz aufweist. Hierzu wurden beide Verfahren gemäß deren formalen Beschreibung implementiert. Somit stehen vier Funktionen zur Verfügung:

```
unsigned long long z_goedel (int platzklasse[], int size, int max);
unsigned long long goedel (int platzklasse[], int size);
vector<int> z_degoedel(unsigned long long goedelnummer, int size, int max);
vector<int> degoedel (unsigned long long goedelnummer, int size);
```

Abb. 12. Gödelfunktionen

Die Funktionen *goedel()* bzw. *degoedel()* implementieren das geläufige Verfahren der Gödelisierung mittels der Funktion g, welche Ordnungsnummern als Potenz der Primzahlfaktoren benutzt [2], sowie die Berechnung ihrer Inversen mittels Primzahlzerlegung. Diese Funktionen benötigen lediglich die Platzklasse in numerischer Abbildungsform bzw. die jeweilige Gödelnummer als Übergabeparameter. Mit der Variable *size* wird die Größe des Vektors einer Lagerplatzklasse - also die Anzahl der Merkmale - übergeben und dient hier nur als Hilfsvariable. Außerdem existiert eine Funktion *prim()*, welche eine bestimmte Anzahl an Primzahlen generiert, die in der Funktion benötigt werden.
Die zwei weiteren Funktionen *z_goedel()* und *z_degoedel()* implementieren das alternative Verfahren der Gödelisierung mittels der Zwiebelfunktion $f_{m;n}$ [2] bzw. ihrer Inverse. Der Unterschied in der Flexibilität der beiden Verfahren lässt sich bereits an den Übergabeparametern erkennen: hier sind sowohl die Variable *size* (Anzahl der Merkmale) als auch eine weitere Variable *max* (größtmögliche Merkmalausprägung) notwendig, um die jeweilige Berechnung auszuführen. Die

Durchführung der Tests mit unterschiedenen Werten für *size* und *max* zeigen jedoch deutliche Vorteile des alternativen Verfahrens. Folgendes Beispiel zeigt den immensen Unterschied in der Größenordnung der errechneten Gödelnummer für ein und dieselbe Lagerplatzklasse $P_z = [2, 2, 2, 2310]$:

$$P_g = g(P_z) = g(2, 2, 2, 2310) \approx 1,35 \cdot 10^{1955}$$
$$P_g = f_{m;n}(P_z) = f_{2310;4}(2, 2, 2, 2310) \approx 3 \cdot 10^{13}$$

Es ist also offensichtlich, dass die Zwiebelfunktion hier besser geeignet ist. Die zusätzliche Abhängigkeit von den Variablen *size* und *max* stellt keine große Einschränkung dar. Beide Variablen können mittels einer einfachen Datenbankabfrage ermittelt werden.

3.2.2 Verfügbare Funktionen

Die externe dynamische Bibliothek stellt außer den Funktionen, welche die beiden Verfahren der Gödelisierung implementieren, weitere Funktionen zur Verfügung. Das Konzept sieht vor, eine eindeutige numerische Abbildung einer Lieferung zu generieren. Hierfür wird eine Datenbank angelegt, welche die Ordnungszahlen der jeweiligen Merkmalausprägungen beinhaltet (siehe Kapitel 3.1). Somit sind Funktionen nötig, mittels derer die Interaktion mit der Datenbank geschieht. Diese sind in der Abbildung 13 aufgelistet. Es besteht also die Möglichkeit, eine Verbindung zur Datenbank herzustellen bzw. zu trennen und diese zu testen. Hierfür ist eine externe Hilfsbibliothek notwendig, welche das Interagieren mit einer MySQL-Datenbank ermöglicht. Das Tool MySQL Connector/C liefert die nötige Bibliothek sowie dazugehörige Header-Dateien.

```
void dbconnect();
void dbdisconnect();
void dbtest();
int db_max();
int transform(string auspraegung);
```

Abb. 13. DB-Funktionen

Die Funktion *dbmax()* ermittelt die größtmögliche Ordnungszahl, welche für die Gödelisierung mittels Zwiebelfunktion (siehe Kapitel 3.2.1) benötigt wird. Es wird lediglich der größte Wert der Spalte „Ordnungszahl“ aus der Look-Up View abgefragt.

```
SELECT max(Ordnungszahl) FROM lookup
```

Listing 1.5. Ermitteln der größten Ordnungszahl

Schließlich liefert die Funktion *transform()* anhand der übergebenen Ausprägung die dazugehörige Ordnungszahl zurück. Die Datenbankabfrage besteht aus dem

bereits vordefinierten SQL-Code (siehe Kapitel 3.1.4), welcher durch die konkrete Ausprägung ergänzt wird. Eine fehlerhafte Eingabe (nicht aufgelistete Ausprägung) wird ebenfalls behandelt und liefert eine 0 sowie die entsprechende Kommandozeilenausgabe zurück.

Es folgt nun die Umsetzung des konzipierten Zuordnungsalgorithmus sowie die Implementierung des Pareto-Algorithmus. Beide Algorithmen werden ebenfalls als separate Funktionen zur Verfügung gestellt (siehe Abbildung 14). Die Funktionen sollen jeweils ein zweidimensionales Array zurück liefern, welches die numerischen Abbildungen der ausgewählten Lagerplatzklassen beinhalten. Dies erfordert das Anlegen von dynamischen Arrays, da die Anzahl der ermittelten Lagerplatzklassen nicht im Vorfeld bekannt ist. Leider ist das mittels typischer Array-Struktur in C/C++ nicht möglich oder recht umständlich und speicherineffizient. Aus diesem Grund bietet sich eine weitere Struktur aus den Standard-Bibliotheken von C/C++ sehr gut an, nämlich *vector*. Diese Struktur kann ihre Größe dynamisch verändern und ist somit sehr flexibel.

```
vector<vector <int>> auswahl(unsigned long long **availible, int lieferung[], int anzahl_klassen, int anzahl_merkmale);
vector<vector <int>> pareto(vector<vector <int>> auswahl);
```

Abb. 14. Algorithmen

Die Funktion *auswahl()* implementiert den Algorithmus, welcher zu der eingegebenen Lieferung die geeigneten, verfügbaren Lagerplatzklassen bestimmt. Es werden einige Übergabeparameter benötigt:

- ***availible*: ein Pointer auf ein zweidimensionales Array, welches alle aktuell verfügbaren Lagerplatzklassen in ihrer numerischen Vektor-Form beinhaltet. Dieses Array wird zuvor mittels der dafür vorgesehenen Funktion erstellt, welche eine inverse Gödelisierung der übergebenen Gödelnummern (Kennzahlen der verfgbaren Lagerplatzklassen) durchführt.
- *lieferung[]*: ein Array, welches die numerische Abbildung der eingegebenen Lieferung repräsentiert. Dieses wird zuvor erstellt, indem die jeweiligen Merkmalausprägungen über die Kommandozeile eingegeben und mittels der dafür vorgesehenen Funktion zu den dazugehörigen Ordnungsnummern transformiert werden.
- *anzahl_klassen*: die Anzahl der aktuell Verfügbaren Lagerplatzklassen muss im Vorfeld bestimmt und übergeben werden, da dies hier nicht mehr möglich ist (nur der Pointer wird übergeben, da das Array dynamisch angelegt wird).
- *anzahl_merkmale*: die Anzahl der Merkmale wird zu Beginn bestimmt (bzw. eingegeben) und ist entscheidend für die Vollständigkeit und Korrektheit der Algorithmenabläufe.

Die Implementierung des Algorithmus ist gemäß der im Kapitel 2.1 beschriebenen Zuordnungsroutine umgesetzt.
Die Funktion *pareto()* führt eine Pareto-Optimierung der zuvor ermittelten Wahl geeigneter Lagerplatzklassen durch und liefert einen zweidimensionalen Vektor

zurück, welcher nur die speziellsten Lagerplatzklassen (oder eine konkrete, speziellste Lagerplatzklasse) in ihrer numerischen Vektor-Form beinhaltet. Auch hier ist die Implementierung gemäß dem bereits im Kapitel 2.3 konzipierten Algorithmus durchgeführt. Die Pareto-optimalen Lagerplatzklassen können nun wieder gödelisiert und weiter verwendet werden.

3.2.3 Kapselung

Alle vorhin beschriebenen Funktionen werden an verschiedenen Stellen des erstellten Prototyp-Konzepts benötigt und können zu Testzwecken separat aufgerufen werden. Aus der praktischen Sicht ist es jedoch sinnvoller eine einzelne Funktion aufzurufen, welche das gewünschte Ergebnis sofort liefert. Für diesen Fall steht eine weitere Funktion *prototyp()* zur Verfügung, welche den gesamten optimierten Zuordnungsablauf gekapselt ausführt.

```
vector<unsigned long long> prototyp(unsigned long long availibleG[], int lieferung[], int anzahl_merkmale, int db_max);
```

Abb. 15. Funktion prototyp()

Es ist zu erkennen, dass hier nur wenige Parameter benötigt werden, welche in ihrer ursprünglichen Form übergeben und von der Funktion verwendet werden können:

- *availibleG[]*: ein Array, welches die Kennzahlen (Gödelnummern) der aktuell verfügbaren Lagerplatzklassen beinhält.
- *lieferung[]*: ein Array, welches die numerische Abbildung der eingegebenen Lieferung repräsentiert.
- *anzahl_merkmale*: die ermittelte/eingegebene Anzahl der Merkmale.
- *db_max*: die größtmögliche Ordnungszahl.

Für die Ermittlung der Variablen *db_max* und *anzahl_merkmale* sowie der numerischen Abbildung der Lieferung sind Datenbankinteraktionen notwendig. Die Lagerverwaltungssoftware besitzt in der Regel eine eigene, kundenspezifische Datenbank. Aus diesem Grund wird davon ausgegangen, dass die erforderten Datenbankoperationen bereits im Vorfeld ausgeführt werden. Hierfür stehen ebenfalls eigene Funktionen zur Verfügung (siehe Abbildung 13).
Die Funktion zerlegt zuächst die übergebenen Gödelnummern in die jeweilige numerische Vektor-Form. Als nächstes werden die für die Lieferung geeigneten Lagerplatzklassen ermittelt und zwecks Überprüfung ausgegeben. Es werden nun die Pareto-optimalen Klassen ausgesucht, erneut gödelisiert und als Vektor der Gödelnummern zurück geliefert.

4 Lokale Testläufe und Ausblick

Die im Rahmen dieses Master-Projekts erstellte dynamische Bibliothek bietet eine Implementierung aller im qualitativ-optimierten Konzept für die Zuordnung von Artikeln zu Lagerplätzen verwendeten Algorithmen. Diese stehen sowohl in Form mehrerer separaten Funktionen als auch einer gekapselten Funktion zur Verfügung. Die durchgeführten Testläufe beschränkten sich auf vier Merkmale sowie einige vordefinierte Lagerplatzklassen. Die Datenbank für die Look-Up Tabelle wurde ebenfalls lokal angelegt. Somit bestand das Ziel zunächst darin, den ersten lauffähigen Prototypen zu erstellen und mittels lokaler Testläufe seine Korrektheit, Vollständigkeit und Effizienz zu eruieren.

Im Laufe der Tests wurden Lieferungen mit verschiedenen Kombinationen von Merkmalausprägungen generiert. Sowohl die Ergebnisse der numerischen Abbildung, als auch die des Zuordnungsverfahren haben mit den theoretisch erwarteten Ergebnissen übereingestimmt. Eine endgültige Aussage über die Korrektheit der implementierten Algorithmen kann jedoch erst nach einer Reihe weiterer Tests unter realen Bedingungen getroffen werden. Dies soll im Rahmen der künftigen Integrationsphase durchgeführt werden, indem die erstellte Bibliothek in die vorhandene Lagerverwaltungssoftware eingebettet wird und die angebotenen Funktionen verwendet werden.

Die Zwiebelfunktion hat sich durch ihre höhere Effizienz etabliert (siehe Kapitel 3.2.1). Die „schalenförmige“ Abzählung ermöglicht eine optimale Werteausnutzung für den Fall dass alle Merkmale dieselbe Anzahl von Ausprägungen haben. Die Praxiserfahrungen zeigen jedoch, dass die meisten geläufigen Merkmale nur wenige Ausprägungen haben und nur einige Merkmale eine höhere Anzahl an Ausprägungen besitzen. Des Öfteren tritt der Fall auf, dass nur ein Merkmal aus n viele Ausprägungen und somit einen hohen Wert für die größtmögliche Ordnungszahl aufweisen. Dadurch wird der Wert m in der Zwiebelfunktion automatisch größer für alle Merkmale. Eine Verbesserung der Effizienz kann dadurch erreicht werden, dass die Funktion $f_{m;n}$ um zusätzliche Begrenzung der jeweiligen Dimension (größtmögliche Ordnungszahl für jedes Merkmal) erweitert und die Berechnung entsprechend angepasst wird.
Das implementierte Konzept zeigt durch die entsprechend angelegte Datenbankstruktur sowie die Abhängigkeit der Algorithmen von nur wenigen Parametern einen hohen Grad an Flexibilität auf. Das verwendete Verfahren der eindeutigen Kennzahlenzuordnung ist jedoch stark abhängig von der konkreten Anzahl der Merkmalen sowie ihrer größtmöglichen Ausprägungen. Dies wird durch die hohe Effizienz in den meisten Fällen kompensiert. Im Hinblick auf die Verbesserung der Flexibilität des Gesamtkonzepts würde sich eine separate Bibliothek anbieten, welche mehrere Verfahren der Kennzahlenzuordnung implementiert und diese als Funktionen zur Verfügung stellt. Abhängig von der konkreten Zusammensetzung der Merkmale kann entscheiden werden, welche der Funktionen besser geeignet ist.

5 Literaturverzeichnis

1. Becker, P; Deitelhoff, F; Witt, K-U.: "Qualitativ-optimierte Zuordnung von Artikeln zu Lagerplätzen.", Internes Papier, 2013.

2. Becker, P; Deitelhoff, F; Witt, K-U.: "Vorschläge für die Zuordnung von Lieferungen zu Lagerplätzen.", Internes Papier, 2013.

3. Gudehus, T.: "Logistik 1. Grundlagen, Verfahren und Strategien.", Springer-Verlag, Studienausgabe der 4. Auflage, Berlin, Heidelberg, 2012.

4. Gudehus, T.: "Logistik 2. Netzwerke, Systeme und Lieferketten.", Springer-Verlag, Studienausgabe der 4. Auflage, Berlin, Heidelberg, 2012.

5. Ten Hompel, M; Schmidt, T.: "Warehouse Management. Organisation und Steuerung von Lager- und Kommissioniersystemen.", Springer-Verlag, 4. neu bearbeitete Auflage, Berlin, Heidelberg, 2010.