Informatik-Studienarbeit 20. März 2002

Erweiterung des VIQTORYA-Systems um Disjunktionen

Thomas Schurtz

Zusammenfassung: *Bislang konnte VIQTORYA, das Visual Query Tool For Syntactically Annotated Corpora, Suchabfragen an linguistische Korpora nur mittels eines logischen UND verknüpfen, obwohl die eigens definierte Query-Sprache auch ODER-Verknüpfungen vorsieht. Aufgabe der hier vorgestellten Studienarbeit war es, Suchabfragen, die nun solche ODER-Verknüpfungen enthalten dürfen, automatisch in disjunktive Normalform (DNF) umzuformen, da so die einzelnen Disjunkte wie bisher durchgearbeitet werden können. Anschließend müssen dann lediglich die Resultate vereinigt werden. Das zu diesem Zweck entwickelte Java-Programm wird im Folgenden näher erläutert. Dabei werden die einzelnen Verarbeitungsschritte immer anhand eines Beispiels verdeutlicht. Einleitend wird außerdem die Query-Sprache kurz definiert, die im Zuge der Studienarbeit leicht erweitert worden ist.*

1 Einleitung

VIQTORYA ist ein System, mit dem sich syntaktisch annotierte Korpora einfach auf spezielle linguistische Eigenschaften hin durchsuchen lassen können. Die Korpora liegen dabei in Form einer SQL-Datenbank vor. Der Aufbau dieser Datenbank und des gesamten VIQTORYA-Systems wird detailliert in den von Laura Kallmeyer [Kallmeyer2000] und von Ilona Steiner zusammen mit Laura Kallmeyer [SteinerKallm2002] verfassten Papers vorgestellt. Kern von VIQTORYA ist ein Java-Tool, das Ausdrücke in einer eigenen, speziell entwickelten Query-Sprache in komplexe SQL-Abfragen übersetzt, die dann mittels der Java-SQL-Schnittstelle JDBC an die Datenbank übergeben und ausgewertet werden. Diese Query-Sprache ist auf die besonderen Anforderungen von Linguisten zugeschnitten und sehr klar und einfach strukturiert, weshalb sie zwar nicht alle Möglichkeiten einer direkten SQL-Abfrage bietet, dafür aber wesentlich einfacher zu handhaben und zu erlernen ist.

Bislang wurde jedoch nicht die komplette Query-Sprache von VIQTORYA unterstützt, denn es fehlte die Möglichkeit der Verwendung von Disjunktionen, also mit ODER verknüpften Suchbedingungen. Dies schränkte die Komplexität möglicher Abfragen stark ein, was den Anstoß für die vorliegende Studienarbeit lieferte. Von Ilona Steiner durchgeführte Tests ergaben, dass eine direkte Übersetzung von Suchabfragen, die ODER-Verknüpfungen enthalten, nach SQL nicht in Betracht kommt, denn die Datenbank benötigte unzumutbar viel Zeit bei der Bearbeitung solcher Abfragen. Deshalb wurde beschlossen, die Suchabfragen zunächst in disjunktive Normalform (DNF) zu bringen. Die DNF ist so definiert, dass ODER-Verknüpfungen nur auf der obersten Ebene vorkommen und die durch ODER verknüpften sogenannten Disjunkte ausschließlich UND-Verknüpfungen enthalten, sie sieht also schematisch folgendermaßen aus (die eingeklammerten Teilterme sind die Disjunkte):

```
(...UND...UND...UND...) ODER (...UND...UND...) ODER
 (...UND...UND...)  ...
```

Eine so formulierte Suchabfrage kann dann gestückelt, also Disjunkt für Disjunkt, an die Datenbank gestellt werden, wozu das bereits vorhandene Tool ohne Modifikationen dienen kann. Anschließend müssen lediglich die Ergebnisse der einzelnen Abfragen vereinigt werden. Vorteil dieser Methode ist, dass die Datenbank selbst keine ODER-Verknüpfungen abarbeiten muss. Die einzelnen Disjunkte werden wie bisher sehr schnell verarbeitet und auch die Transformation der Suchabfrage in DNF benötigt nicht viel Zeit, so dass insgesamt ein erheblicher Geschwindigkeitsgewinn entsteht.

Ziel dieser Studienarbeit war die Entwicklung eines Programms, das diese Transformation von Suchabfragen in DNF bewerkstelligt. Es wurde die Programmiersprache Java verwendet, da auch die anderen Bestandteile von VIQTORYA – speziell das eigentliche Query-Tool – in Java programmiert wurden. Wie in Java üblich wurde eine eigene Klasse erstellt, die über später im Text erläuterte Schnittstellen vom Query-Tool aufgerufen werden kann. Die Klasse trägt den Namen `QueryDNF` und zu ihr gehört außerdem die Klasse `QueryException`, die zur Fehlerbehandlung dient. Die Dateien `QueryDNF.class` und `QueryException.class` beinhalten diese Klassen und müssen dem aufrufenden Programm zur Verfügung stehen. Lauffähig sind sie ab der Java-Version 1.1.

In den folgenden Kapiteln werden die Funktionsweise des Programms und seine Schnittstellen detailliert erläutert, zunächst wird jedoch eine kurze Definition der Query-Sprache geliefert. Die ursprüngliche Definition von Laura Kallmeyer [Kallmeyer2000] wurde nämlich im Zuge der Studienarbeit erweitert, um komplexe Klammerungen zu ermöglichen.

2 Die Query-Sprache

Die linguistischen Korpora, die sich mittels VIQTORYA durchsuchen lassen können, bestehen aus komplexen Baumstrukturen, die den Aufbau des jeweiligen Satzes modellieren. Diese Baumstrukturen beinhalten Knoten mit ihren Labels (*category* und möglicherweise *token*) und zugehörigen Kanten, die mit einem *function*-Label versehen sind. Nach Knoten mit einer gewissen *category* lässt sich über die Query-Sprache mittels des Ausdrucks `cat(i)=c` suchen. Hier steht `c` für die gesuchte *category*, während `i` die Nummer des Knotens bezeichnet, mit der sich in möglichen weiteren Bedingungen auf diesen Knoten Bezug nehmen lässt. Genauso lässt sich mittels `fct(i)=e` nach Knoten mit einer *function* `e` suchen und mit `token(i)=t` nach Knoten mit einem *token* `t`. Eine Negation dieser Bedingungen ist ebenfalls möglich. Sucht man beispielsweise nach einem Knoten, der die *category* `c` nicht haben darf, bringt man dies durch `cat(i)!=c` zum Ausdruck.

Es lässt sich auch nach Knoten suchen, die in bestimmter Beziehung zu einem anderen Knoten stehen. `i>>j` sucht zum Beispiel nach einem Knoten `i`, der den Knoten `j` dominiert. `i!>>j` sucht nach einem Knoten `i`, der den Knoten `j` nicht dominiert. Analog zu `>>` (für *dominates*) gibt es die Relationen `>` (für *immediately dominates*) und `..` (für *linearly precedes*) und ihre Negationen.

Wie sich einzelne Suchbedingungen mit einem logischen UND verknüpfen lassen, ist induktiv folgendermaßen definiert worden: Sind q_1 und q_2 Suchabfragen, so sind auch q_1 `&` q_2 Suchabfragen. Das logische ODER wurde bislang hingegen so definiert: Sind q_1 und q_2 Suchabfragen, so sind auch (`(`q_1 `|` q_2`)`) Suchabfragen. Es wurde also eine Klammer zwingend vorgeschrieben. Es sollte so deutlich gemacht werden, dass UND (`&`) stärker bindet als ODER (`|`). Es verhinderte allerdings die einfache Verkettung mehrerer durch ODER verknüpfter Bedingungen, da beispielsweise (`(`q_1 `|` q_2 `|` q_3`)`) keine korrekte Query wäre.

Sie müsste stattdessen zu `((`q_1 `|` q_2`)` `|` q_3`)` umformuliert werden. Außerdem sah diese Definition keine Möglichkeit zur Klammerung von UND-Verknüpfungen vor. Es wurde daher beschlossen, die Query-Sprache an dieser Stelle zu erweitern. Die Klammerung um ODER-Verknüpfungen ist nicht mehr zwingend vorgeschrieben, stattdessen gilt die zusätzliche Definition, dass UND stärker bindet, als ODER. Es ist nun also auch q_1 `|` q_2 eine korrekte Query und q_1 `|` q_2 `&` q_3 ist äquivalent zu q_1 `|` `(`q_2 `&` q_3`)`. Klammerungen sind jetzt beliebig gestattet, d.h. wenn `q` eine Suchabfrage ist, so ist auch `(q)` eine Suchabfrage. Auch eine Negation ist möglich: `!(q)` ist ebenfalls eine korrekte Query und besagt, dass die Bedingung(en) `q` nicht gelten dürfen.

Erweitert um diese Möglichkeiten lautet die formelle Definition der Query-Sprache nun folgendermaßen:

Definition (*(C, E, T)*-queries)
(C, E, T)-queries are inductively defined:

(a) for all i∈ℕ, t∈*T:*
`token(i)=t` *and* `token(i)!=t` *are queries,*
(b) for all i∈ℕ, c∈*C:*
`cat(i)=c` *and* `cat(i)!=c` *are queries,*
(c) for all i∈ℕ, e∈*E:*
`fct(i)=e` *and* `fct(i)!=e` *are queries,*
(d) for all i, j∈ℕ*:*
`i>>j` *and* `i!>>j` *are queries,*
`i>j` *and* `i!>j` *are queries,*
`i..j` *and* `i!..j` *are queries,*
(e) for all queries q_1, q_2*:*
q_1 `&` q_2 *and* q_1 `|` q_2 *are queries,*
`(`q_1`)` *is a query,*
`!(`q_1`)` *is a query.*
(f) `&` *takes precedence over* `|`

In dieser Definition bezeichnen *C*, *E* und *T* die Mengen der möglichen *categories*, *functions* und *token*. Sie basiert auf der Definition 1 aus [Kallmeyer2000].

3 Das Programm

Die ganze Funktionalität des erarbeiteten Java-Programms steckt in der Klasse `QueryDNF`. Sie enthält alle Funktionen, mit denen sich aus einem Eingabestring, der die ursprüngliche Suchabfrage enthält, eine Suchabfrage in DNF erstellen lässt. Diese Funktionen werden in diesem Kapitel in der Reihenfolge erläutert, in der sie arbeiten. Zur Verdeutlichung wird dabei eine Beispielabfrage Schritt für Schritt in DNF umgewandelt. Wie es in Java üblich ist, werden bei einem solchen Transformationsprozess eventuell auftretende Fehler mit so genannten Ausnahmen oder *Exceptions* verarbeitet.

3.1 `QueryException`

Die Ausnahmen, die in QueryDNF auftreten können, sind Objekte vom Typ `QueryException`. Die zugehörige Klasse leitet sich ab von der Standardklasse `Exception` und überschreibt lediglich die beiden Konstruktoren. Wenn während der

Umwandlung einer Suchabfrage in DNF nun beispielsweise eine Klammer fehlt, erzeugt die Funktion, die diesen Fehler bemerkt hat, ein `QueryException`-Objekt und bricht seine Programmausführung ab. Zusätzlich wird dem Objekt ein String mit einer kurzen Erklärung des Fehlers übergeben. Eine so erzeugte Ausnahme muss in Java zwingend von einer aufrufenden Funktion abgefangen werden. In `QueryDNF` geschieht dies innerhalb der Schnittstellenfunktionen, die schließlich die Fehlermeldung ausgeben und als Resultat den Wert `null` zurückliefern. Sie werden später noch genauer erläutert.

3.2 Instanzvariablen, Konstruktor und Hilfsfunktionen von `QueryDNF`

Die Klasse `QueryDNF` besitzt drei Instanzvariablen. In `originalQueryString` wird als Referenz die ursprüngliche Suchabfrage abgelegt, während mittels `queryString` die Vorarbeiten zur Umformung in DNF vorgenommen werden. Die DNF wird schließlich in Form einer `ArrayList` namens `disjunctList` gespeichert. In diese Liste werden die erzeugten Disjunkte einzeln als String eingetragen. Die Schnittstellenfunktionen liefern entweder diese Liste zurück oder einen String, der die komplette Query in DNF enthält.

Der Konstruktor von `QueryDNF` verlangt als Argument einen String mit der umzuformenden Suchabfrage. Dieser wird in `originalQueryString` abgelegt. Mit der Funktion `setQueryString` kann dem Objekt eine neue Suchabfrage übergeben werden, die dann wiederum in `originalQueryString` gespeichert wird. In diesem Fall wird auch die Ergebnisliste `disjunctList` geleert, damit sie die neuen Disjunkte aufnehmen kann. Mit `getQueryString` kann man die ursprüngliche Query abfragen.

Zwei wichtige Hilfsfunktionen für die DNF-Transformation sind `getOpeningBracket` und `getClosingBracket`. Als Argumente verlangen sie einen String, der eine Suchabfrage enthält, sowie die Position der schließenden beziehungsweise der öffnenden Klammer. Als Resultat liefern die Funktionen die zugehörige öffnende beziehungsweise schließende Klammer. Beide arbeiten analog, weshalb es genügt, nur eine Funktion explizit zu erklären, beispielsweise `getClosingBracket`. Zunächst wird eine Zählvariable auf die Position des ersten Zeichens nach der öffnenden Klammer gesetzt. Die Zahl der geöffneten Klammern wird auf 1 gesetzt, die Rückgabevariable auf –1. Nun wird innerhalb einer Schleife jedes Zeichen der Query mittels der Zählvariablen durchlaufen. Wird eine weitere öffnende Klammer gefunden, erhöht sich die Zahl der geöffneten Klammern um 1, wird eine schließende Klammer gefunden, verringert sie sich um 1. So werden innere Klammern übersprungen. Ist die Zahl der geöffneten Klammern 0, so wurde die gesuchte schließende Klammer gefunden und die Schleife bricht ab. Resultat ist die aktuelle Position im String, also die Zählvariable. Wird keine schließende Klammer gefunden, liefert die Funktion –1 zurück, denn mit diesem Wert wurde die Rückgabevariable ja vor der Schleife belegt.

Schließlich enthält `QueryDNF` noch zwei Hilfsfunktionen zur Erleichterung der Arbeit mit Strings. Die Funktion `insertString` fügt einen String in einen anderen String an einer bestimmten Position ein und liefert den neuen String als Resultat zurück, im Fehlerfall `null`. Ein Fehler tritt beispielsweise auf, wenn die Position, an die der neue String eingefügt werden soll, im alten String gar nicht vorhanden ist. Die Funktion `deleteChar` löscht ein einzelnes Zeichen an einer bestimmten Position aus einem String und liefert den neuen String, beziehungsweise ebenfalls `null` im Fehlerfall, zurück.

3.3 Transformation einer Suchabfrage in DNF

Die zentrale Funktion von `QueryDNF` ist `getDNF`. Dort wird die DNF-Transformation durchgeführt, die sich in drei große Schritte aufteilen lässt. Im ersten, vorbereitenden Schritt entfernt `getDNF` zuerst Leerzeichen und doppelte Negationen aus dem Abfragestring und klammert ihn ein. Dann werden in einer Unterfunktion namens `clarifyBrackets` zur einfacheren Unterscheidung zwischen Paramter-Klammern und syntaktischen Klammern letztere durch eckige Klammern ersetzt. Schließlich folgt noch eine Unterfunktion, in der die Suchabfrage auf korrekte Klammersetzung überprüft wird, nämlich `checkBrackets`. Für den zweiten großen Schritt wird die Unterfunktion `eliminateNotBrackets` aufgerufen. In ihr werden die NICHT-Klammern durch hereinziehen der Negation in die Klammer aufgelöst. Schließlich erfolgt in `createDNF` die eigentliche DNF-Transformation durch rekursive Auflösung der ODER-Verknüpfungen.

In den folgenden Unterkapiteln wird beschrieben, wie die genannten Schritte im Detail arbeiten. Als Fallbeispiel dient folgende Suchabfrage, die in DNF umgeformt werden soll:

```
cat(1)=NX & (cat(2)=PX & fct(2)=FOPP | cat(2)=SIMPX |
 cat(2)=VF & !(1!>>2 & 2!..1 | fct(1)=FOPP))
```

3.3.1 Vorbereitende Schritte

In `getDNF` wird zunächst die originale Suchabfrage in die Arbeitsvariable `queryString` kopiert und die Ergebnisliste `disjunctList` wird vorsorglich gelöscht. Ist der Suchabfragestring `null`, so wird `null` auch als Resultat der DNF-Transformation zurückgeliefert und die Funktion wird beendet. Andernfalls werden nun als vorbereitende Maßnahme sämtliche Leerzeichen aus dem String entfernt, ebenso alle eventuell vorhandenen doppelten Negationen (`!!`). Schließlich wird der String eingeklammert. Grund dafür ist der Verzicht auf die Forderung, dass alle ODER-Verknüpfungen geklammert sein müssen (siehe Kapitel 2). Wegen der Festlegung, dass ODER schwächer bindet als UND, liegen alle ungeklammerten ODER-Verknüpfungen auf der obersten Ebene der Suchabfrage, fasst man sie als Baumstruktur auf. Daher kann der komplette Query-String als oberste ODER-Klammer angesehen werden. Mit den gesetzten Klammern kann nun zu jeder ODER-Verknüpfung ein Paar umgebender Klammern gefunden werden und der Transformationsprozess kann vereinheitlicht werden. Ansonsten hätten die ungeklammerten ODER-Verknüpfungen separat behandelt werden müssen. Nach diesen Vorarbeiten sieht die Beispielabfrage folgendermaßen aus:

```
[cat(1)=NX&(cat(2)=PX&fct(2)=FOPP|cat(2)=SIMPX|cat(2)=VF&
 !(1!>>2&2!..1|fct(1)=FOPP))]
```

Es ist hier zu sehen, dass die neu gesetzten äußeren Klammern eckig sind. Im ersten großen Bearbeitungsschritt, der in der nun aufgerufenen Funktion `clarifyBrackets` durchgeführt wird, werden auch die inneren Klammern in eckige umgewandelt. Ausgenommen davon sind die Klammern, in denen die Knotennummern stehen, denn von diesen sollen sich die anderen Klammern abheben. Diese Methode arbeitet der Einfachheit halber nicht mit einem String, sondern kopiert die Suchabfrage in ein `char`-Array (ein Feld von Zeichen). Das Ersetzen eines einzelnen Zeichens durch ein anderes ist damit leichter. In einer Schleife werden zunächst alle runden öffnenden Klammern der Query durchlaufen.

Dabei wird jeweils in einer inneren Schleife geprüft, ob jedes Zeichen bis zur schließenden Klammer eine Zahl ist. Ist dies nicht der Fall, so handelt es sich nicht um eine Knotennummern-Klammer und sie muss in eine eckige umgewandelt werden. Analog wird mit den schließenden Klammern verfahren, bevor das `char`-Array als Resultat zurück auf `queryString` kopiert wird. Die Beispiel-Query sieht nach dieser Bearbeitung so aus:

```
[cat(1)=NX&[cat(2)=PX&fct(2)=FOPP|cat(2)=SIMPX|cat(2)=VF&
 ![1!>>2&2!..1|fct(1)=FOPP]]]
```

Im zweiten Bearbeitungsschritt prüft die als nächstes aufgerufene Methode `checkBrackets`, ob zu jeder öffnenden Klammer eine schließende vorhanden ist und umgekehrt. Dazu kopiert sie den Query-String in eine Hilfsvariable. Dann wird in einer Schleife jede öffnende eckige Klammer angesteuert. Ihre Position wird vermerkt und mittels der Hilfsfunktion `getClosingBracket` ihre zugehörige schließende Klammer ermittelt. Ist diese gefunden worden, so wird das Klammernpaar einfach aus dem Arbeitsstring gelöscht und die nächste öffnende Klammer wird angesteuert. Ist sie nicht gefunden worden, wird eine Ausnahme ausgelöst. Wenn keine öffnende Klammer mehr vorhanden ist, testet die Methode, ob trotzdem noch eine schließende Klammer im String enthalten ist. Falls ja, dann hat sie keine zugehörige öffnende Klammer und es wird wiederum eine Ausnahme ausgelöst. Ansonsten sind alle Klammern in Ordnung.

Es kann mit dem dritten Schritt weitergemacht werden, `getDNF` ruft dazu die Funktion `eliminateNotBrackets` auf. Der Name sagt es schon, es sollen die NICHT-Klammern aufgelöst werden.

3.3.2 Auflösung der NICHT-Klammern

Im Zuge der Einführung der Klammersetzung in die Query-Sprache wurde auch die Möglichkeit hinzugefügt, ganze geklammerte Ausdrücke zu negieren. Diese NICHT-Klammern müssen aufgelöst werden, das heißt die Negation muss in die Klammer hineingezogen werden. Dies geschieht mittels folgender logischer Äquivalenzen:

```
NICHT(a ODER b) = (NICHT a UND NICHT b)
NICHT(a UND b) = (NICHT a ODER NICHT b)
```

Es ist deshalb notwendig, die NICHT-Klammern in dieser Weise aufzulösen, weil die einzelnen Disjunkte einer DNF überhaupt keine Klammern enthalten sollten. Außerdem ist jede UND-Verknüpfung innerhalb einer NICHT-Klammer nach obigen Gesetzen eigentlich eine ODER-Verknüpfung, muss also als eine solche im DNF-Transformationsprozess behandelt werden. Dort würde die Klammer dann sowieso aufgelöst, was aber wegen möglicher Verschachtelungen komplizierter wäre, als einfach vorher sämtliche NICHT-Klammern aufzulösen. Es blieben nur NICHT-Klammern übrig, die ausschließlich ODER-Verknüpfungen enthalten. Neben der schon erwähnten Tatsache, dass selbst solche Klammern in DNF-Disjunkten nicht erlaubt sind, könnte auch das vorhandene Query-Tool damit nichts anfangen, denn es kennt überhaupt keine Klammern.

In der Methode `eliminateNotBrackets` wird mittels einer Schleife jeweils die erste öffnende NICHT-Klammer angesteuert. Dazu wird nach der Zeichenkette `![` gesucht. Zu dieser Klammer wird mit `getClosingBracket` die zugehörige schließende Klammer gesucht. Dann wird die komplette NICHT-Klammer in eine Hilfsvariable kopiert. Im Beispiel würde diese Hilfsvariable folgenden String beinhalten:

```
![1!>>2&2!..1|fct(1)=FOPP]
```

Diese NICHT-Klammer enthält eine UND- und eine ODER-Verknüpfung, und wegen der stärkeren Bindung des UNDs besteht eine implizite Klammerung um `1!>>2&2!..1`. Diese Klammerung muss beim Hereinziehen der Negation beachtet werden, denn sonst ergäbe sich diese nicht äquivalente Query:

```
[1>>2|2..1&fct(1)!=FOPP]
```

Also muss die implizite Klammer gesetzt werden, was die Funktion `addConjunctionBrackets` übernimmt, die als Parameter eine NICHT-Klammer fordert und diese mit den neu gesetzten Klammern um die UND-Verknüpfungen zurückliefert. Mit einer Schleife wird in dieser Funktion jedes UND (`&`) angesteuert. Dann wird von diesem UND aus der String nach links und rechts durchlaufen, bis entweder eine Klammer oder ein ODER (`|`) auftaucht. Innere Klammern werden bei diesem Vorgang übersprungen. In unserem Beispiel findet die Funktion zwar eine öffnende Klammer auf dem Weg nach links, doch rechts stößt sie zuvor auf das ODER. Also fehlt eine Klammer um diese UND-Verknüpfung und sie wird gesetzt, bevor nach dem nächsten UND gesucht wird. Es ist im Beispiel keines mehr vorhanden und die Funktion endet mit folgendem String als Resultat:

```
![[1!>>2&2!..1]|fct(1)=FOPP]
```

Dieser nun korrekt geklammerte String wird im nächsten Schritt der Funktion `eliminateSingleNotBracket` übergeben. Diese löscht zunächst das Negationszeichen (`!`) vor der NICHT-Klammer und geht dann Zeichen für Zeichen durch den String. Die erste öffnende Klammer wird dabei genauso übersprungen, wie die letzte schließende. Das erste Zeichen, das die Funktion im Beispiel bearbeitet, ist also die neue öffnende Klammer um die UND-Verknüpfung. Da kein Negationszeichen links neben dieser Klammer steht, muss sie negiert werden. Es wird ein Ausrufezeichen gesetzt und zum Ende der Klammer gesprungen. Nächstes Zeichen ist dann das ODER (`|`), das in ein UND (`&`) umgewandelt wird. Nun läuft die Schleife weiter bis zum Gleichheitszeichen von `fct(1)=FOPP`. Links davon steht kein Ausrufezeichen, also muss es gesetzt werden. Danach findet die Schleife keine relevanten Zeichen (`[`, `>`, `&`, `|`, `=`, `.`) mehr und endet mit folgendem Resultat:

```
[![1!>>2&2!..1]&fct(1)!=FOPP]
```

Die Programmausführung kehrt damit zurück zur Funktion `eliminateNotBrackets`. Dort wird die aufgelöste NICHT-Klammer in den Query-String eingebaut, was zu diesem String führt:

```
[cat(1)=NX&[cat(2)=PX&fct(2)=FOPP|cat(2)=SIMPX|cat(2)=VF&
  [![1!>>2&2!..1]&fct(1)!=FOPP]]
```

Nun wird die nächste NICHT-Klammer aus dem String herausgelöst:

```
![1!>>2&2!..1]
```

Diese wird wiederum der Funktion `addConjunctionBrackets` übergeben, die diesmal jedoch bereits ein Klammerpaar um die UND-Verknüpfung vorfindet und nichts unternehmen muss. Weiter geht es mit `eliminateSingleNotBracket`. Diese Funktion durchläuft

wieder Zeichen für Zeichen das Innere der NICHT-Klammer und entfernt diesmal die Negationszeichen und wandelt das UND in ein ODER um. Resultat dieses Vorgangs ist der String `[1>>2|2..1]`, der abschließend wieder in `queryString` eingebaut wird:

```
[cat(1)=NX&[cat(2)=PX&fct(2)=FOPP|cat(2)=SIMPX|cat(2)=VF&
 [[1>>2|2..1]&fct(1)!=FOPP]]]
```

Da keine weiteren NICHT-Klammern vorhanden sind, endet `eliminateNotBrackets` schließlich mit obigem Resultat.

3.3.3 Die DNF-Transformation

Nach der Auflösung der NICHT-Klammern steht der eigentlichen DNF-Transformation nichts mehr im Wege. Dazu ruft `getDNF` die Funktion `createDNF` auf. Diese Funktion ist rekursiv, als Startwert wird ihr der Query-String übergeben.

Die Transformation geschieht nach folgendem logischen Gesetz:

```
a UND (b ODER c) UND d = (a UND b UND d) ODER (a UND c UND d)
```

Die ODER-Klammer wird also aufgelöst, indem an ihre Stelle einmal das `b` und einmal das `c` in die umgebende Formel eingesetzt wird, wodurch zwei Disjunkte entstehen. Genau nach diesem Schema arbeitet auch `createDNF`.

Erster Schritt ist das Aufsuchen der ersten ODER-Verknüpfung und der zugehörigen umgebenden Klammern, die wie in Kapitel 3.3.1 gesehen vorhanden sein müssen. Um diese Klammern zu finden, durchläuft eine Schleife den zu bearbeitenden String vom ODER aus nach links zur öffnenden Klammer und nach rechts zur schließenden Klammer. Eventuelle innere Klammern werden dabei übersprungen. Die Positionen der Klammern werden vermerkt und die schließende Klammer wird in ein ODER (`|`) umgewandelt. Dies dient zur Vereinfachung der jetzt folgenden Schleife. Diese durchläuft die gesamte Klammer Zeichen für Zeichen. Trifft sie auf ein ODER, so leitet sie den Rekursionsvorgang ein. Im Beispiel ist dies die ODER-Klammer (fett markiert ist die veränderte schließende Klammer und das erste nun angesteuerte ODER):

```
[cat(2)=PX&fct(2)=FOPP|cat(2)=SIMPX|cat(2)=VF&
 [[1>>2|2..1]&fct(1)!=FOPP]|
```

Jetzt wird ein neuer Query-String zusammengebaut. Erster Bestandteil ist der Teil des alten Strings, der vor der ODER-Klammer steht:

```
[cat(1)=NX&
```

Es folgt alles vom letzten bearbeiteten ODER bis zum aktuellen ODER beziehungsweise von der öffnenden Klammer bis zum ersten ODER:

```
[cat(1)=NX&cat(2)=PX&fct(2)=FOPP
```

Schließlich wird der Rest des alten Query-Strings nach der ODER-Klammer hinzugefügt:

```
[cat(1)=NX&cat(2)=PX&fct(2)=FOPP]
```

Mit diesem String wird nun rekursiv wieder `createDNF` aufgerufen. Da er keine ODER-Verknüpfungen mehr enthält, ist der Trivialfall der Rekursion eingetreten und der String wird als Resultat in die Liste der Disjunkte, `disjunctList`, eingetragen.

Analog verläuft der nächste Schleifendurchlauf. Statt `cat(2)=PX&fct(2)=FOPP` wird nun einfach `cat(2)=SIMPX` in die umgebende Query eingebaut, was zu folgendem String führt:

```
[cat(1)=NX&cat(2)=SIMPX]
```

Auch darin ist keine ODER-Verknüpfung mehr enthalten, die damit rekursiv aufgerufene Funktion `createDNF` landet also wieder im Trivialfall und trägt den String in die `disjunctList` ein.

Danach geht die Schleife weiter bis zum letzten ODER, das ja eigentlich die schließende Klammer war. Nun wird folgender String zusammengebaut:

```
[cat(1)=NX&cat(2)=VF&[[1>>2|2..1]&fct(1)!=FOPP]]
```

Zu beachten ist, dass die ODER-Verknüpfung zwischen `1>>2` und `2..1` übersprungen worden ist, da sie in einer inneren Klammer liegt. Mit diesem String landet `createDNF` nicht im Trivialfall. Stattdessen findet sie die vorhin übersprungene ODER-Klammer vor, bei der wiederum die schließende Klammer der Einfachheit halber in ein ODER umgewandelt wird:

```
[1>>2|2..1|
```

Deshalb startet jetzt wieder eine Schleife, welche diese ODER-Klammer durchläuft und die einzelnen Disjunkte in die umgebende Query einbaut. Es ergeben sich diese beiden Strings:

```
[cat(1)=NX&cat(2)=VF&[1>>2&fct(1)!=FOPP]]
[cat(1)=NX&cat(2)=VF&[2..1&fct(1)!=FOPP]]
```

Hier sind keine weiteren ODERs mehr enthalten, die erzeugten Rekursionen landen im Trivialfall und die zwei Strings werden jeweils der `disjunctList` hinzugefügt. Die Programmausführung kehrt zurück zur ersten Rekursionsebene, doch hier war die Schleife ja bereits am letzten ODER angelangt und `createDNF` endet.

Es bleibt noch zu erwähnen, dass in den Trivialfällen sämtliche eckige Klammern aus den jeweiligen Strings gelöscht werden, bevor sie in die Ergebnisliste eingetragen werden. Das ist möglich, da nur noch UND-Verknüpfungen in den Disjunkten enthalten sind.

Die Programmausführung kehrt nun in die Funktion `getDNF` zurück. Dort wird noch eine kosmetische Korrektur an allen Disjunkten vorgenommen. Das bereits vorhandene Query-Tool erwartet nämlich Leerzeichen vor und nach den ODER-Symbolen. Endgültiges Resultat der Funktion `getDNF` ist im Beispiel schließlich eine Liste aus diesen vier Strings:

```
cat(1)=NX & cat(2)=PX & fct(2)=FOPP
cat(1)=NX & cat(2)=SIMPX
```

```
cat(1)=NX & cat(2)=VF & 1>>2 & fct(1)!=FOPP
cat(1)=NX & cat(2)=VF & 2..1 & fct(1)!=FOPP
```

3.3.4 Laufzeit des Algorithmus

Die Bearbeitungszeit des Beispiels beträgt je nach Rechner und Betriebssystem in etwa eine Millisekunde. Die allgemeine Laufzeit ist abhängig von der Anzahl der resultierenden Disjunkte. Sucht man beispielsweise nach einem Knoten, der eine aus zehn möglichen *categories* und eine aus zehn möglichen *functions* haben soll, so ergeben sich schon 10*10=100 mögliche Kombinationen und somit auch hundert Disjunkte. Fügt man einen zweiten Knoten hinzu, der wiederum hundert mögliche Kombinationen aufweist, kommt man schon auf 100*100=10000 Disjunkte. Betrachtet man die Disjunkte als Blätter eines Baumes, den der rekursive Algorithmus erzeugt, so verhält sich die Laufzeit proportional zur Anzahl aller Knoten des Baumes, wobei die Knoten die rekursiven Verzweigungen darstellen. Diese Anzahl ist natürlich nochmals höher als die Anzahl der Disjunkte. Eine Suchabfrage nach obigem Muster mit hundert möglichen Kombinationen dauert in etwa zwischen 1000 und 2000 Millisekunden, eine mit 10000 möglichen Kombinationen nimmt bereits zwei bis drei Minuten in Anspruch. Allerdings ist es sehr unwahrscheinlich, dass solche Abfragen in der Praxis auftreten, denn im Regelfall würde die Anzahl der Resultate dabei noch wesentlich höher liegen, als die Anzahl der Disjunkte. Jedes Disjunkt liefert normalerweise nämlich mehrere Ergebnisse, also erhielte man bei einer Suche mit 10000 Disjunkten wahrscheinlich auch weit mehr als 10000 Resultate.

3.4 Zusammensetzung der Liste der Disjunkte zu einem Query-String

Die Funktion `getDNF` lieferte als Resultat eine `ArrayList` aus einzelnen Strings, also ein Feld aus Disjunkten. Möglicherweise möchte man aber einen einzigen String als Resultat erhalten, der die komplette Suchabfrage in DNF enthält und in der Query-Sprache gültig ist. Diese Aufgabe übernimmt die Funktion `getDNFString`.

Sie ruft zunächst `getDNF` auf, um die DNF-Transformation durchzuführen. Dann baut sie in einer Schleife aus den resultierenden Disjunkten, die in der `disjunctList` abgelegt wurden, einen einzigen Query-String zusammen. Dabei wird der Übersichtlichkeit halber jedes Disjunkt eingeklammert. Abschließend wird eine Klammer um den ganzen String gesetzt. Die in Kapitel 3.3 verwendete Suchabfrage würde zu folgendem Resultat führen:

```
((cat(1)=NX & cat(2)=PX & fct(2)=FOPP) | (cat(1)=NX &
 cat(2)=SIMPX) | (cat(1)=NX & cat(2)=VF & 1>>2 &
 fct(1)!=FOPP) | (cat(1)=NX & cat(2)=VF & 2..1 &
 fct(1)!=FOPP))
```

3.5 Schnittstellenfunktionen

Die Klasse `QueryDNF` bietet zwei statische Schnittstellenfunktionen. Diese ersparen den alternativen Weg, sich selbst eine Objektinstanz von `QueryDNF` anzulegen und darauf `getDNF` oder `getDNFString` anzuwenden. Bei solch einer Vorgehensweise müssten auch die durch mögliche Fehler erzeugten Ausnahmeobjekte (Kapitel 3.1) in einem so genannten *try-catch-Block* abgefangen werden.

Die Schnittstellenfunktionen heißen `queryToDNF` und `queryToDNFString`. Als Argumente benötigen beide nur einen String, der die umzuwandelnde Suchabfrage enthält.

Intern erzeugen sie damit ein `QueryDNF`-Objekt und rufen darauf die Funktion `getDNF` (für `queryToDNF`) beziehungsweise `getDNFString` (für `queryToDNFString`) auf. Die jeweiligen Resultate werden schließlich zurückgeliefert, wobei auch mögliche Ausnahmen abgefangen werden. Tritt eine Ausnahme auf, liefern beide Funktionen `null` als Resultat zurück und geben eine Fehlermeldung aus.

Die Funktion `queryToDNF` liefert die einzelnen Disjunkte als Strings in einer `ArrayList` zurück, während `queryToDNFString` einen einzelnen Ergebnisstring wie in Kapitel 3.4 erzeugt.

Schließlich enthält `QueryDNF` noch eine `main`-Funktion, womit sich eine DNF-Transformation auch über die Kommandozeile durchführen lässt. Dies ist allerdings eher zu Testzwecken gedacht. Der Befehl `java QueryDNF "query"` startet das Programm, wobei `query` die umzuwandelnde Suchabfrage enthalten sollte. Ausgegeben wird die resultierende Query in DNF als einzelner String wie in Kapitel 3.4.

4 Zusammenfassung

In dieser Ausarbeitung wurde die Java-Klasse `QueryDNF` vorgestellt, die eine komplexe Suchabfrage der erweiterten VIQTORYA-Query-Sprache in disjunktive Normalform umwandelt. Dazu wurde einleitend die Query-Sprache vorgestellt und definiert, bevor dann im Einzelnen auf die verschiedenen Arbeitsschritte eingegangen wurde, die bei einer DNF-Transformation durchlaufen werden. Untermalt wurden sämtliche Schritte durch die tatsächliche Umwandlung einer Beispiel-Suchabfrage. Schließlich wurden die Schnittstellen vorgestellt, über die speziell das bisherige Query-Tool nun eine DNF-Umwandlung starten kann.

Literatur

[Kallmeyer2000] Laura Kallmeyer: A query tool for syntactically annotated corpora. In Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora, Hong Kong, October 2000.

[SteinerKallm2002] Ilona Steiner und Laura Kallmeyer: VIQTORYA – A Visual Query Tool for Syntactically Annotated Corpora. Proceedings of the Third International Conference on Language Resources and Evaluation, Las Palmas, Canary Islands, Spain, May 2002.