Datenbankpraktikum WS 2004/2005

Sven Helmer

8. November 2004

Inhaltsverzeichnis

1 Einführung

Eine alte chinesische Weisheit besagt: „Du hörst etwas und Du vergißt es, Du siehst etwas und Du kannst Dich erinnern, Du tust etwas und Du kannst es“. Da die meisten Lehrveranstaltungen nur audiovisuell ablaufen, ist es immer fraglich wieviel davon hängen bleibt. Da es in diesem Praktikum doch eher um den dritten Punkt geht, hoffe ich, daß doch einiges hängen bleiben wird. Ziel des Praktikums ist es ein relationales Datenbanksystem, in unserem Fall DB2 von IBM, näher kennenzulernen.

Dieses Praktikum ist in fünf größere Teile gegliedert:

- (fortgeschrittenes) Update und Retrieval in SQL
- Embedded SQL
- Dynamic SQL
- Benutzerdefinierte Typen und Funktionen
- E-R-Modellierungsversuch

Weitere wichtige Dinge sind in der Einführung eigentlich nicht mehr zu sagen, bleibt mir nur noch allen Teilnehmern viel Spaß zu wünschen.

2 Fortgeschrittenes SQL

2.1 Theorie

2.1.1 Der CAST-Operator

Wie in Programmiersprachen können auch in SQL Werte von einem Datentyp in einen anderen umgewandelt werden. Abbildung 1 zeigt die Syntax des CAST-Operators.

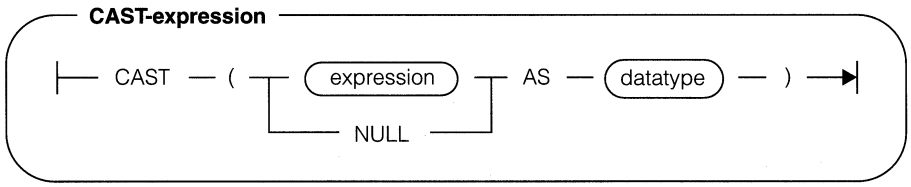

Abbildung 1: Syntax des CAST-Operators

Um den Erfolg der CAST-Operation zu gewährleisten, muß der Zieldatentyp (in den "gecastet" wird) wohldefiniert sein, d.h. die Länge, die Präzision, etc. muß angegeben werden. Man ist auf der sicheren Seite, wenn man alle Eigenschaften explizit angibt, wie z.B. (der Operator || konkateniert zwei Strings):

```
CAST (c1+c2 AS Decimal(8,2))
CAST (name || address AS Varchar(255))
```

Beim Weglassen der Parameter für Decimal wird Decimal(5,0) angenommen, bei Char Char(1) und bei Graphic Graphic(1). Das Weglassen von Parametern bei anderen Datentypen resultiert in Fehlermeldungen. Bei verschiedener Länge (wie z.B. bei Strings) wird ein String mit Leerzeichen aufgefüllt bzw. abgeschnitten (mit einer Warnung an den Benutzer).

Casting ist nützlich beim Aufrufen von Funktionen, die bestimmte Datentypen bei ihren Parametern verlangen. Die Funktion `substr` z.B. verlangt Ganzzahlen als zweiten und dritten Parameter. Wenn x und y Fließkommazahlen sind, dann würde folgender Aufruf Sinn machen:

```
substr(string1, CAST(x as Integer), CAST(y as Integer))
```

Abbildung 2 gibt an, welche CAST-Operationen von DB2 unterstützt werden.

2.1.2 Die CASE-Anweisung

Wenn Daten abgerufen werden, ist es oft wünschenswert die Bedeutung der Werte und nicht irgendwelche kryptischen Abkürzungen zu verwenden. Zu diesem Zweck gibt es die CASE-Anweisung.

TABLE 3-1: Valid Uses of the CAST Expression

Source Datatype	Target Datatype
Smallint, Integer, Decimal, Double	Smallint, Integer, Decimal, Double
Char, Varchar, Long Varchar, Clob	Char, Varchar, Long Varchar, Clob, Blob
Graphic, Vargraphic, Long Vargraphic, Dbclob	Graphic, Vargraphic, Long Vargraphic, Dbclob, Blob
Char, Varchar	Smallint, Integer, Decimal, Date, Time, Timestamp, Vargraphic
Smallint, Integer, Decimal	Char
Date, Time, Timestamp	Char, Varchar
Date	Date
Time	Time
Timestamp	Date, Time, Timestamp
Blob	Blob

Abbildung 2: Erlaubte CAST-Operationen

In folgendem Beispiel wird eine Relation mit den Daten von Offizieren verwendet:

Offiziere(Name, Status, Dienstgrad)

In *Status* wird vermerkt, ob der betreffende Soldat aktiv, Reservist oder außer Dienst ist. Folgende Anfrage wäre vorstellbar:

```
SELECT Name,
   CASE Status
      WHEN 1 THEN 'Aktiv'
      WHEN 2 THEN 'Reservist'
      WHEN 3 THEN 'Ausser Dienst'
      ELSE 'unbekannt'
   END AS Status
FROM Offiziere;
```

Abbildung 3 zeigt die Syntax der CASE-Anweisung.

Im folgenden Beispiel soll eine Gebühr für Fahrzeuge berechnet werden. Die Gebühr richtet sich nach dem Typ des Fahrzeugs. Bei einem PKW hängt die Gebühr vom Gewicht ab, bei einem LKW von der Anzahl der Räder, bei einem Motorrad wird eine Pauschale erhoben.

Fahrzeuge(Kennzeichen, Typ, Gewicht, AnzahlRaeder)

```
SELECT Kennzeichen,
   CASE Typ
      WHEN 'PKW' THEN 0.05 * Gewicht
      WHEN 'LKW' THEN 25.00 * AnzahlRaeder
      WHEN 'Motorrad' THEN 35.00
      ELSE NULL
```

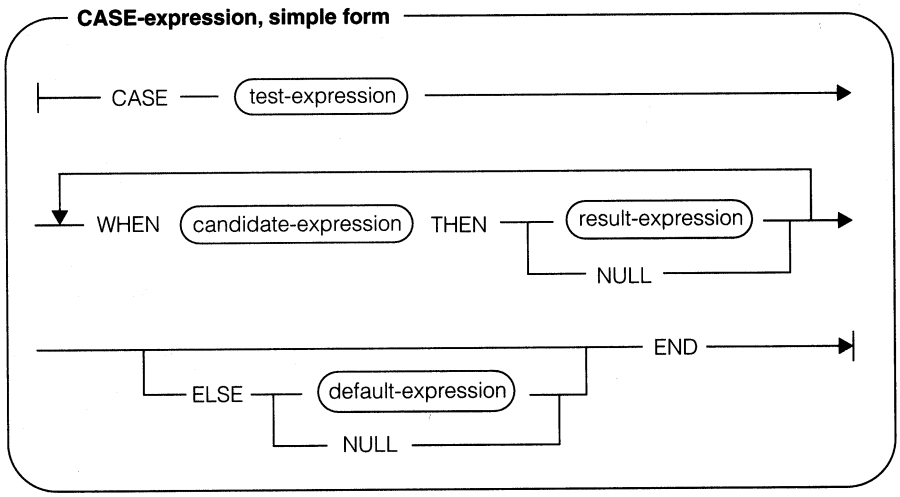

Abbildung 3: Syntax der CASE-Anweisung

```
   END AS Gebuehr
FROM Fahrzeuge;
```

Es gibt auch noch eine allgemeinere Form der CASE-Anweisung bei der eine Suchbedingung angegeben werden kann. Abbildung 4 zeigt die Syntax der allgemeinen CASE-Anweisung.

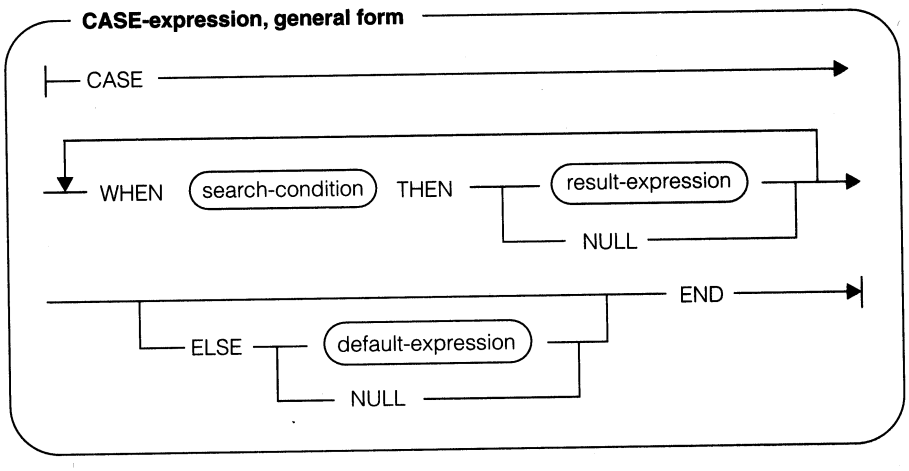

Abbildung 4: Syntax der allgemeinen CASE-Anweisung

Angenommen die Steuer auf Grundbesitz hängt von der Größe des Besitzes ab. Dann kann der Steuersatz für die Relation

Grundbesitz(ParzellenNr, Stadt, Flaeche, Steuersatz)

folgendermaßen ausgerechnet werden:

```
UPDATE Grundbesitz
SET    Steuersatz =
   CASE
      WHEN Flaeche < 10000 THEN 0.05
      WHEN Flaeche < 20000 THEN 0.07
      ELSE 0.09
   END;
```

2.1.3 Die WITH-Anweisung

Wenn in SQL mit mehreren (geschachtelten) Aggregatfunktionen in einer Anfrage umgegangen werden muß, dann werden die SQL-Ausdrücke sehr schnell sehr komplex. Nehmen wir z.B. die Relation

Angestellter(PersonalNr, Name, AbteilungsNr, Gehalt, Vorgesetzter)

Wenn die Abteilung gefunden werden soll, die die größte Summe an Gehältern auszahlt, müssen zuerst die Gehälter aufsummiert werden und danach muß das Maximum gefunden werden.

Man könnte eine Sicht anlegen, um die Anfrage zu beantworten:

```
CREATE VIEW Lohnliste(AbteilungsNr, Summe) AS
   SELECT AbteilungsNr, sum(Gehalt)
   FROM   Angestellter
   GROUP BY AbteilungsNr;

SELECT AbteilungsNr
FROM   Lohnliste
WHERE  Summe =
   (SELECT max(Summe)
    FROM   Lohnliste);
```

Eine Sicht anzulegen, um nur eine Anfrage zu stellen, ist etwas umständlich. Es muß ein Name gewählt werden, der zu keinen Namenskonflikten führt. Nach Beendigung der Anfrage sollte die Sicht wieder gelöscht werden. All diese Operationen müssen in die Systemtabellen eingetragen und wieder gelöscht werden.

Alles in einer Anfrage unterzubringen führt zu folgendem Aussehen:

```
SELECT AbteilungsNr,
FROM   (SELECT AbteilungsNr, sum(Gehalt) AS Summe
        FROM   Angestellter
        GROUP BY AbteilungsNr) AS Lohnliste
WHERE  Summe =
       (SELECT max(Summe)
        FROM   (SELECT AbteilungsNr, sum(Gehalt) AS Summe
                FROM   Angestellter
                GROUP BY AbteilungsNr) AS Lohnliste2);
```

Dieser Ansatz hat zwei entscheidene Nachteile. Erstens ist er nicht besonders elegant und übersichtlich, und zweitens wird das System gezwungen die gleich Unteranfrage zweimal auszuwerten.

Es wäre schön, eine Art nicht-dauerhafter Sicht aufzubauen. Genau dies wird durch die WITH-Anweisung bewerkstelligt:

```
WITH Lohnliste(AbteilungsNr, Summe) AS
   (SELECT AbteilungsNr, sum(Gehalt)
    FROM   Angestellter
    GROUP BY AbteilungsNr)
SELECT AbteilungsNr
FROM   Lohnliste
WHERE  Summe =
   (SELECT max(Summe)
    FROM   Lohnliste);
```

Die Syntax der WITH-Anweisung ist in Abbildung 5 zu sehen.

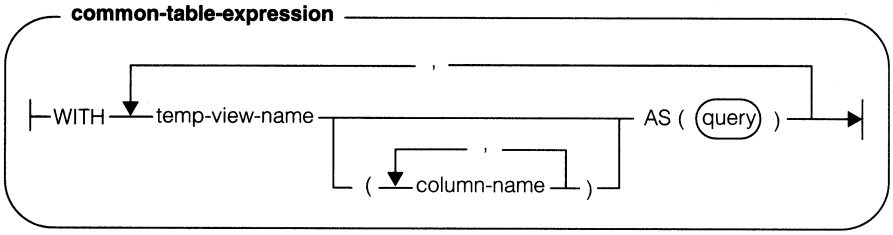

Abbildung 5: Syntax der WITH-Anweisung

2.1.4 Rekursion in SQL

Die WITH-Anweisung kann nicht nur zur Vereinfachung von Anfragen eingesetzt werden. Rekursive Anfragen in DB2 werden mit Hilfe der WITH-Anweisung realisiert.

Betrachten wir wieder die Relation Angestellter von vorhin. Gesucht sind alle Angestellten, die mehr als 100.000 DM verdienen und die Untergebene von Lambsdorff sind. Für direkte Untergebene ist dies einfach:

```
SELECT Name, Gehalt
FROM   Angestellter
WHERE  Vorgesetzter = 'Lambsdorff'
AND    Gehalt > 100000;
```

Für alle indirekten Untergebenen wird dies um einiges schwieriger. In diesem Fall benötigen wir rekursive Anfragen. Eine Anfrage, die das Problem löst, ist z.B. folgende:

```
WITH Untergebene(Name, Gehalt) AS
   ((SELECT Name, Gehalt                          -- Basisanfrage
     FROM   Angestellter
     WHERE  Vorgesetzter = 'Lambsdorff')
    UNION ALL
    (SELECT A.Name, A.Gehalt                      -- rekursive Unteranfrage
     FROM   Untergebene U, Angestellter A
     WHERE  A.Vorgesetzter = U.Name))
SELECT Name, Gehalt                               -- endgueltige Anfrage
FROM   Untergebene
WHERE  Gehalt > 100000;
```

Was passiert in rekursiven Anfragen genau? Eine rekursive Anfrage wird nach folgendem Schema aufgebaut:

1. Am Anfang steht eine WITH-Anweisung, die eine temporäre Sicht aufbaut. Die temporäre Sicht besteht aus zwei Teilen, die mit **UNION ALL** verbunden werden **müssen**.

   (a) Der erste Teil, die "Basisanfrage", ist eine konventionelle Anfrage, die keinerlei rekursive Aufrufe enthält. Dieser Teil wird immer zuerst ausgeführt (in unserem Beispiel werden die direkten Untergebenen von Lambsdorff bestimmt).

   (b) Im zweiten Teil, der "rekursiven Unteranfrage", werden weitere Tupel zur temporären Sicht hinzugefügt. Es muß genau angegeben werden, wie diese neuen Tupel in Beziehung zu den bisher enthaltenen stehen. Außerdem muß sichergestellt werden, daß die rekursive Unteranfrage nicht in alle Ewigkeit weiterläuft, sondern irgendwann abbricht. In unserem Beispiel werden Angestellte in *Untergebene* aufgenommen, wenn sie Untergebene von Angestellten sind, die bereits in *Untergebene* zu finden sind. Das System bricht ab, sobald nur noch Angestellte gefunden werden, die keine Vorgesetzten mehr sind. Im Prinzip wird bei rekursiven Anfragen in SQL die transitive Hülle berechnet. Beim Formulieren der rekursiven Unteranfrage müssen einige Regeln beachtet werden:

      - Es dürfen keine Aggregatfunktionen, SELECT DISTINCT, GROUP BY und HAVING Klauseln verwendet werden.
      - Die rekursive Unteranfrage darf Attribute der temporären Sicht referenzieren, darf aber **keine** Unteranfragen enthalten, die dies tun.

2. Nachdem in der WITH-Anweisung eine temporäre Sicht definiert wurde, kann diese Sicht dann in einer SELECT-Anweisung benutzt werden, um die ursprüngliche Frage zu beantworten (hier Gehalt $>$ 100000).

2.2 Einige nützliche Funktionen

2.2.1 Aggregatfunktionen

stdev(<*numerischer Datentyp*> **)** $\rightarrow$ Double
gibt die Standardabweichung der Werte der Spalte zurück.

variance(<*numerischer Datentyp*> **)** → Double
gibt die Varianz der Werte der Spalte zurück.

2.2.2 Skalare Funktionen

abs(<*numerischer Datentyp*> **)** → *gleicher numerischer Datentyp*
Betragsfunktion

acos(Double **)** → Double
gibt den Arcuscosinus zurück.

ascii(<*String Datentyp*> **)** → Integer
gibt den ASCII-Code des ersten Buchstaben des Strings an.

asin(Double **)** → Double
gibt den Arcussinus zurück.

atan(Double **)** → Double
gibt den Arcustangens zurück.

ceil(<*numerischer Datentyp*> **)** → *gleicher numerischer Datentyp*
rundet auf.

concat(<*String Datentyp*>**,**<*String Datentyp*> **)** → *String Datentyp*
konkateniert die beiden Strings, Kurzform || als Infix-Operator.

cos(Double **)** → Double
gibt den Cosinus zurück.

cot(Double **)** → Double
gibt den Cotangens zurück.

day(Date **)** → Integer
gibt den Tag aus einem Datum zurück.

dayname(Date **)** → Varchar(100)
gibt den Namen des Wochentags aus einem Datum zurück.

dayofyear(Date **)** → Integer
Tag des Jahres, gibt einen Wert zwischen 1 und 366 zurück.

difference(Varchar(4), Varchar(4) **)** → Integer
gibt die Differenz zweier Soundexklangcodes zurück (siehe auch *soundex*).

exp(Double x **)** → Double
gibt e^x zurück.

floor(<*numerischer Datentyp*> **)** → *gleicher numerischer Datentyp*
rundet ab.

hex(<*beliebiger Datentyp*> **)** → Varchar
wandelt Argument in Hexadezimaldarstellung um.

lcase(Varchar **)** → Varchar(4000)
wandelt Argument in Kleinbuchstaben um.

length(<*beliebiger Datentyp*> **)** → Integer
gibt die Länge in Bytes zurück.

ln(Double **)** → Double
gibt den natürlichen Logarithmus zurück.

locate(<*String Datentyp*> **s1,**<*String Datentyp*> **s2)** → Integer
gibt die Position des ersten Auftauchens von String s2 in String s1 an.

log(Double **)** → Double
gibt den natürlichen Logarithmus zurück.

log10(Double **)** → Double
gibt den Zehnerlogarithmus zurück.

mod(Integer m, Integer n **)** → Double
gibt den Rest von m geteilt durch n zurück.

month(Date **)** → Integer
gibt den Monat aus einem Datum zurück.

monthname(Date **)** → Varchar(100)
gibt den Namen des Monats aus einem Datum zurück.

power(Integer x, Integer n **)** → Integer

power(Double x, Double n **)** → Double
gibt x^n zurück.

round(Integer x, Integer n **)** → Integer

round(Double x, Double n **)** → Double
rundet auf n Stellen genau.

sin(Double **)** → Double
gibt den Sinus zurück.

sqrt(Double **)** → Double
gibt die Wurzel zurück.

soundex(Varchar **)** → Char(4)
gibt einen Vierzeichenklangcode eines Wortes zurück (siehe auch *difference*).

substr(<*String Datentyp*> **s,** Integer m**,** Integer n **)** → <*String Datentyp*>
gibt einen Teilstring von s zurück, beginnend mit dem Zeichen an Stelle m und der Länge n.

tan(Double **)** → Double
gibt den Tangens zurück.

ucase(Varchar **)** → Varchar(4000)
wandelt Argument in Großbuchstaben um.

week(Date **)** → Integer
gibt die Jahreswoche aus einem Datum zurück.

year(Date **)** → Integer
gibt das Jahr aus einem Datum zurück.

2.3 Versuch1: SQL Retrieval und Update über CLP

2.3.1 Handhabung des Command Line Processors (CLP)

Der Command Line Processor (CLP) ist die grundlegende Art in DB2, interaktiv SQL-Anweisungen abzusetzen. CLP wird mit dem Kommando `db2` aufgerufen. CLP kann auf drei verschiedene Arten benutzt werden:

1. Die SQL-Anweisung wird direkt nach dem Kommando `db2` angegeben. Um sich z.B. mit der Datenbank `unidb` zu verbinden, muß

   ```
   db2 connect to unidb
   ```

   angegeben werden. Dabei muß beachtet werden, daß diese Zeile zuerst von der Shell bearbeitet wird. D.h. falls irgendwelche Sonderzeichen verwendet werden (wie z.B. *,>,<) sollte die Eingabe in einfachen oder doppelten Anführungszeichen eingefaßt werden:

   ```
   db2 "select * from studenten"
   ```

2. Wenn die Option `-f <filename>` benutzt wird, holt sich CLP die Eingabe aus der angegebenen Datei. Die Eingabe

   ```
   db2 -f stud.clp
   ```

   holt sich die auszuführenen Anweisungen aus der Datei `stud.clp`.

3. Wenn das Kommando `db2` ohne Anweisung aufgerufen wird, gelangt man in den interaktiven Modus. CLP wartet dann mit dem Prompt

   ```
   db2 =>
   ```

   auf Eingaben. Shell Kommandos können aus CLP heraus durch voranstellen eines Ausrufezeichens aufgerufen werden:

```
!ls
```

Bei Verwendung der Option `-t` muß jede Anweisung mit einem Semikolon (;) abgeschlossen werden. Beenden kann man eine CLP Session auf zwei Arten:

1. Man meldet sich von den angemeldeten Datenbanken mittels des Disconnect-Befehls ab und setzt dann die Quit-Anweisung ab, also z.B.:

   ```
   db2 "disconnect unidb"
   db2 "quit"
   ```

2. Empfohlen wird aber eine Session mit der Terminate-Anweisung zu beenden. Das hat den Vorteil, daß alle noch laufenden Hintergrundprozesse der Session beendet werden und alle noch eventuell bestehenden Datenbankverbindungen abgebaut werden.

   ```
   db2 "terminate"
   ```

2.3.2 Laden der Datenbank

Die erste Aufgabe des Praktikums ist es, die Datenbank in der später gearbeitet wird, aufzubauen. Die entspechenden Skripte werden bereitgestellt.

`buildbase` mit diesem Shellskript wird die Datenbank aufgebaut

`deletebase` mit diesem Shellskript wird die Datenbank gelöscht. Falls irgendwann etwas schiefgehen sollte, kann die Datenbank somit komplett gelöscht und wieder neu aufgebaut werden.

2.3.3 Einfachere Anfragen

1. Geben Sie alle Städte an, die ungefähr 100.000 Einwohner haben. Sortieren Sie die Städte nach der Abweichung von dieser Einwohnerzahl (Städte mit kleinerer Abweichung stehen weiter vorne in der Liste). Geben Sie alle Staedte an, die um maximal 10.000 Einwohner von 100.000 Einwohnern abweichen.

2. Geben Sie alle Länder an, deren Hauptstadtname Ähnlichkeit mit dem Landesnamen hat.

3. Erzeugen Sie mit einer Sicht einen Auschnitt der Relation Stadt, in der alle Städte mit mehr als fünf Millionen Einwohnern eingetragen sind. Übernehmen Sie in die Sicht die Attribute *S_ID*, *Name* und *Einwohner*. Beschreiben Sie für die folgenden Operationen jeweils die Reaktionen des Datenbanksystems:

   (a) Fügen Sie eine Stadt mit mehr als fünf Millionen Einwohnern in die Sicht ein. Lassen Sie sich die Sicht ausgeben.

   (b) Löschen Sie die Stadt aus der Sicht. Lassen Sie sich die Sicht ausgeben.

(c) Fügen Sie eine Stadt mit weniger als fünf Millionen Einwohnern in die Sicht ein. Lassen Sie sich die Sicht und die Relation *Stadt* ausgeben.

(d) Fügen Sie die Stadt erneut mit mehr als fünf Millionen Einwohnern in die Sicht ein. Lassen Sie sich die Sicht ausgeben.

(e) Löschen Sie die Stadt aus der Sicht.

(f) Löschen Sie die Stadt aus der Relation *Stadt*.

2.3.4 Fortgeschrittenes SQL

1. (a) Bestimmen Sie die Bevölkerungsdichte der Region, die die Länder Algerien, Libyen und sämtliche Nachbarn dieser Länder umfaßt.

   (b) Vergleichen Sie das Ergebnis mit der Bevölkerungsdichte die man erhält, wenn man die Wüsten als unbewohnbar berücksichtigt.

2. Geben Sie alle Paare von Ländern mit Meeresküste in Europa aus, die an die gleiche Menge von Meeren angrenzen.

   Beispiel:

   [Belgien, Niederlande]
   da $M_{\text{Belgien}} = M_{\text{Niederlande}} = \{\text{Nordsee}\}$

   [Belgien, Spanien] sollte nicht auftauchen,
   da $M_{\text{Spanien}} = \{\text{Atlantik, Mittelmeer}\}$

3. (a) Finden Sie die günstigste Flugverbindung von Kapstadt nach New York.

   (b) Finden Sie die Verbindung von Frankfurt nach Los Angeles mit den wenigsten Umstiegen. Falls mehrere Alternativen bestehen, wählen Sie die günstigste.