Instructions for MIE457F Project 2: A Data Mining System

Checkpoint Due: Oct. 27, 2003, midnight
Final Submission: Nov. 3, 2003, midnight

Questions to Scott Sanner

ssanner@cs.toronto.edu

1 Overview

The purpose of this programming project is to have you modify and implement parts of a data mining system that interacts with an actual database in real-time while working in a structured Java programming environment.

This project makes use of a DMQL [1] inspired language that we will dub DMQL-457. Rather than being a pure subset of DMQL, this query language has been simplified and altered slightly to facilitate implementation. Nonetheless, upon completion of this project, you will have built a non-trivial and reasonably efficient data mining system that will work with *any* database system and *many* Star-structured data schemas.

For this project, you will be provided with a working command shell interface, a DMQL-parser, and an implementation of the basic system architecture. Your task will be to complete the following three items:

1. *Checkpoint Requirement*: Implement the backend JDBC/SQL database queries to populate a data cube according to the specifications of a DMQL command.
2. *Final Submission Requirement*: Implement the basic roll-up/drill-down operations (add/drop attribute only) for a data cube according to the specification of a DMQL command.
3. *Final Submission Requirement*: Modify a data mining algorithm to use the information gain metric for induction of classification rules according to the specification of a DMQL command.

The programming portion of this project should not be too difficult but it will require working knowledge of a number of Java topics including package structure, the JavaDoc utility, the Java Foundation Classes (JFC) – especially *java.lang* and *java.util*.

Of primary importance for this project is an understanding of efficient data cube implementation using the *java.util* classes.[1] This document will cover the details of this implementation but it is assumed that the student has a working knowledge of the *java.util* classes.

[1] Populating and working with the data cube is the main focus of this project!

2 Background reading

In order to understand the content of this document, you should first understand the lecture slides for weeks 5, 6 and 8 on the topics of data and knowledge mining [2].

Additional helpful material can be found in Han et al. [3]. This is a reasonably comprehensive reference for the data mining content covered in this course.

3 Getting started

All of the information to compile, run the code, and view the documentation for this project is described in the *README* file in the root directory of the project.

To get started with the project, download the archive from the following webpage (right click and *Save As* from a web browser):

```
http://www.cs.toronto.edu/~ssanner/Projects/index.html
```

To unpack the archive into a subdirectory, use a Windows compression/decompression utility such as *WinZip*. Or, from the UNIX command line, you can type:

```
tar -xvzf P2.tar.gz
```

From this point, open a command prompt (UNIX: any shell, WINDOWS: Start->MIE Software->gams) and change into the *Project1* subdirectory. On a Windows system, use the command `type␣README` or on a UNIX system, use the command `cat␣README` to view the *README* file.

The *README* file should have all of the information required to compile, run the command-line interface, and access the JavaDoc documentation in the *javadoc* subdirectory. The LaTeXversion of this file along with its corresponding compiled *ps* and *pdf* versions is in the *docs* subdirectory.

Note: A useful text editor for Windows is *TextPad* (Start->TextPad). For UNIX, useful text editors are *emacs*, *xemacs*, *vi*, and *pico*.

4 Existing code structure

Before starting to modify the current data mining system, it is first important to understand its structure.

4.1 Package structure

The package structure for this project is given in Figure 1. Details of the packages and the classes within them can best be explored through the JavaDoc documentation. However, a quick overview of each package is given below.

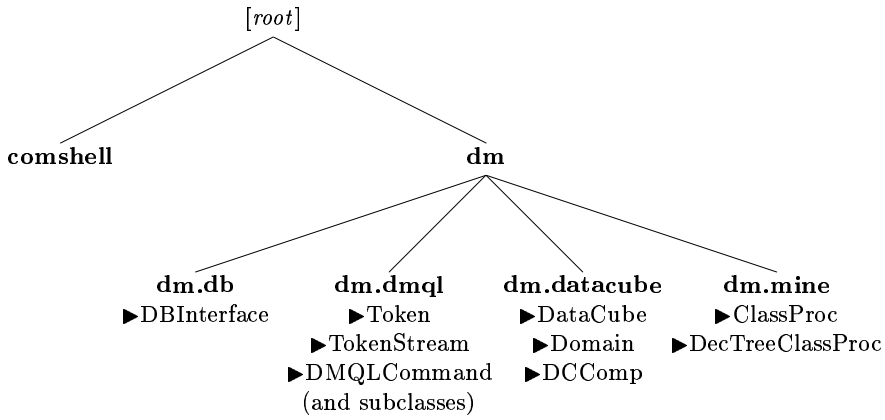

Fig. 1. Package structure and major classes for the MIE457F Project 2: Data Mining.

[***root***] This is the root of the *dm* and *comshell* packages and is the default package if no package is named. As in most projects, the root package simply exists to provide a top level of organization for all of the subpackages. It contains no classes.

comshell This is an auxiliary package that provides support for a command line interface. Other packages such as *ir* will extend the *ComShell* class in the *comshell* package in order to implement their own command shell.

dm This is the root of the *dm* (i.e. Data Mining) package. This package contains one class *DMShell* that extends a *ComShell* class in order to provide a customized command line interface for this project. The command shell makes calls to the subpackages of this package in order to implement the *DMShell* commands.

dm.db This package contains one class, *DBInterface*, which encapsulates all interaction with an underlying database. One only need to provide a driver name, database name, and this class will take care of processing any SQL query and returning the *java.SQL.ResultSet* object containing the table for the query result.

dm.dmql This package contains all classes required to parse DMQL-457 commands. The allowable DMQL-457 commands for this project are outlined in figure 2.

```
DMQL-Command ::= { Def-Dimension | Def-Cube | Drop-Dim |
                   Add-Dim | Display-Cube | Find-Rules }+
Def-Dimension    ::= define dimension ⟨ Dim-Name ⟩
                     from ⟨ DB-Table ⟩
                     as ⟨ Table-Key ⟩ { , ⟨ Table-Col ⟩ }*
Def-Cube         ::= define cube ⟨ Cube-Name ⟩
                     from ⟨ DB-Table ⟩
                     with dimensions ⟨ Table-Key ⟩ for ⟨ Dim-Name ⟩
                                  { , ⟨ Table-Key ⟩ for ⟨ Dim-Name ⟩ }*
                     having content ⟨ Table-Col ⟩
Drop-Dim         ::= drop ⟨ Dim-Name ⟩ from ⟨ Cube-Name ⟩
Add-Dim          ::= add ⟨ Dim-Name ⟩ to ⟨ Cube-Name ⟩
Display-Cube     ::= display ⟨ Cube-Name ⟩
                     [ with width ⟨ Cell-Width ⟩ ]
                     [ using { sum | min | max | avg |
                                 | count-all | count-val ⟨ Value ⟩ } ]
Find-Rules       ::= find classification rules for ⟨ Cube-Name ⟩
                     related to ⟨ Dim-Name ⟩ { , ⟨ Dim-Name ⟩ }*
                     with support threshold ⟨ Support-Threshold ⟩ ,
                          confidence threshold ⟨ Conf-Threshold ⟩
```

Fig. 2. BNF Syntax for the DMQL-457 language. Note that the syntax uses [] for optional items, { } for grouping, * for 0 or more repetitions, + for 1 or more repetitions, and ⟨⟩ for user specified terms. Also note that a parser for this DMQL syntax is already provided. The semantics of this language is explained in detail in section *dm.dmql*, however, the basic semantics should be reasonably apparent from the syntax.

Each of the basic commands in this grammar is represented by a subclass of *java.dm.DMQLCommand*. These subclasses are all internal classes of *java.dm.DMQL* and thus are prefixed by *DMQL*. These subclasses follow:[2]

- *DMQL.DefineDim*: This command class represents the definition of a data cube dimension as a multiattribute table with a single key and any number of attributes relating to that key (all from the same table).[3]
- *DMQL.DefineCube*: This command class represents a data cube definition. It relies on the fact that all required dimensions have already been defined (otherwise this operation cannot be completed). It's main purpose is to generate a SQL query on the database backend to populate the internal data cube data structure.

[2] Although these class names do not exactly match the grammar, the translation should be readily apparent.

[3] Single attribute dimension keys are a restriction of the DMQL-457 language but this is just a restriction on the key – the dimension itself can be multiattribute. However, this restriction could be relaxed without too much difficulty. The restriction that a dimension come from only one table is simply a constraint of Star-structured database schemas.

- *DMQL.Display*: This command represents a display request for a data cube using one of the specified aggregation functions (in the event that the data cube has multiple entries per cube cell, perhaps resulting from a roll-up).[4]
- *DMQL.AddDropDim*: This command represents a request to add or drop a dimension from a specified data cube.
- *DMQL.FindClassRule*: This command represents a request to find classification rules for a data cube.[5]

Note that instances of these *DMQLCommand* subclasses are generated by the command-line interface *dm.DMShell* in response to *dmql-query* commands and passed to *dm.datacube.Domain* where they are actually processed. This code has *already* been written.

dm.datacube This package contains an implementation of the data cube (i.e. *DataCube*) data structure along with a supporting class (i.e. *Domain*) for maintaining dimension and data cube information for a given database domain.

dm.mine This package contains the basic interface for all f̈ind classification rule q̈ueries (i.e., in *ClassProc*) along with a simple implementation based on the Gini metric (i.e., in *DecTreeClassProc*).

4.2 Compilation and Makefiles

How you compile the project will depend on what system you are using. UNIX offers a useful built-in utility for compiling projects called the *Make* utility which is described below.

- *If you are using UNIX Makefiles (optional)*: In every package directory there is a corresponding Makefile (of the same name) that describes how to compile or process all of the files in that directory. For the most part, you should only need to modify this file to add additional classes or subpackages to be compiled. These are defined by the *PACKAGES* and *CLASSES* variables in the Makefile and *you simply need to add the appropriate entry whenever you add a subpackage or class to a package.*

[4] If the data cube contains only one datum per set of dimension attributes, aggregation functions such as sum, min, or max will clearly have no effect on the data displayed.

[5] Since the data cube has only one value item, it is assumed that the goal is to classify this value. Additionally, since we are not handling the capability to aggregate over ranges, this classification rule only makes sense with values that represent classes. That is, it would make sense to call this query if a data cube contained values representing a type of food purchase (i.e., fast-food, eat-in, fine-dining) but it would not make sense for a data cube containing the amount of the purchase (i.e. otherwise we'd be classifying someone who spent 10 dollars and someone who spend 11 dollars differently). We would want to aggregate over value ranges to handle this latter case better, but this is beyond the scope of this course project. It indicates the possible dimensions to include in the rules and a support and confidence threshold hold that must hold for any returned rule.

For compilation on either platform, refer to the *README* file in the root directory (*Project1*) for more information.

4.3 Using JavaDoc documentation

There is a reasonable amount of code already in this project and the JavaDoc documentation will be your most effective tool for getting a high-level overview of the package and class structure. JavaDoc documentation uses a web browser interface and allows you to click on any package, class, or method to explore it further.

For information on building JavaDoc documentation, refer to the *README* file in the root directory (*Project1*). This explains how to build and view the code documentation for this project.

4.4 Running the command-line interface

The command-line interface as implemented in the class *DMShell* in the *ir* package provides a powerful way to interact with the data mining system. The user simply issues commands for indexing documents or URLs and viewing/querying the current index. Furthermore, the user can save commands to a script file and execute this automatically from the command-line or even the UNIX/DOS prompt.

An example script (similar to *test.dmsh*) follows:

```
// A sample DMShell script
echo on
setenv DB-NAME "h:/Project2/StarSchemaEx.mdb"
new-domain MyDomain
timer start

// Define some dimensions
dmql-query "define dimension DimTime from TimeData as TimeKey"
dmql-query "define dimension DimLoc  from LocData  as LocKey"
dmql-query "define dimension DimAge  from AgeData  as AgeKey"

// Define a data cube
dmql-query "define cube MyCube from MainData
            with dimensions Time     for DimTime,
                            Loc      for DimLoc,
                            CustAge  for DimAge
            having content FoodType"

// Roll-up on the location dimension
dmql-query "drop dimension DimLoc from MyCube"

// Display the rolled-up cube without any aggregation function
```

```
dmql-query "display MyCube with width 12"

// Drill down on the location dimension
dmql-query "add dimension DimLoc to MyCube"

// Now display counts for fine-dining (assuming ID=2)...
dmql-query "display MyCube with width 12 using count-val 2"

// Try out a rule-learning algorithm...
dmql-query "find classification rules for MyCube
            related to DimYear, DimLoc, DimAge
            with support threshold 0.10,
                 confidence threshold 0.65"

timer stop
echo off
return
```

When applied to a small database, the output from the above script may look like:

```
...

>  [dmql-query "define dimension DimAge  from AgeData  as AgeID, AgeDesc"]

Processing Dimension Query:

* DMQL.DefineDim
  - Dim:   DimAge
  - Table: AgeData
  - Key:   AgeID
  - Cols:  [AgeDesc]

DMQL command successfully executed.

>  []

>  [// Define a data cube]

>  [dmql-query "define cube MyCube from MainData with dimensions ...

Processing Cube Query:

* DMQL.DefineCube
  - Name:   MyCube
```

```
  - Table:  MainData
  - Keys:   [Year, Location, CustAge]
  - Dims:   [DimYear, DimLoc, DimAge]
  - ValCol: FoodType

DMQL command successfully executed.

>  [dmql-query "drop DimLoc from MyCube"]

Processing Display Query:

* DMQL.AddDropDim
  - Fun:  drop
  - Dim:  DimLoc
  - Cube: MyCube

DMQL command successfully executed.

>  [dmql-query "display MyCube with width 12"]

Processing Display Query:

* DMQL.Display
  - Cube:   MyCube
  - Ag-fun: none (0)
  - Params: []

DimYe\DimAge | [Teens]      [20's]       [30's]       [40's]
-------------+-----------------------------------------------
1999         | [3, 4]       [1]          ~            ~
2000         | ~            [5]          ~            ~
2001         | ~            ~            ~            [2, 2, 1]
2002         | ~            ~            ~            ~

DMQL command successfully executed.

>  [dmql-query "add DimYear to MyCube"]

Processing Display Query:

* DMQL.AddDropDim
  - Fun:  add
  - Dim:  DimYear
  - Cube: MyCube
```

```
DMQL command successfully executed.

>  [dmql-query "display MyCube with width 12 using count-val 2"]

Processing Display Query:

* DMQL.Display
  - Cube:   MyCube
  - Ag-fun: count-val (6)
  - Params: [2]

* DimYear=1999:

   DimLo\DimAg  | [Teens]      [20's]       [30's]       [40's]
   -------------+-----------------------------------------------
   [Canada, Tor | ~            [0]          ~            ~
   [Canada, Ott | [0]          ~            ~            ~
   [USA, New Yo | ~            ~            ~            ~

* DimYear=2000:

   DimLo\DimAg  | [Teens]      [20's]       [30's]       [40's]
   -------------+-----------------------------------------------
   [Canada, Tor | ~            ~            ~            ~
   [Canada, Ott | ~            ~            ~            ~
   [USA, New Yo | ~            [0]          ~            ~

* DimYear=2001:

   DimLo\DimAg  | [Teens]      [20's]       [30's]       [40's]
   -------------+-----------------------------------------------
   [Canada, Tor | ~            ~            ~            ~
   [Canada, Ott | ~            ~            ~            [2]
   [USA, New Yo | ~            ~            ~            ~

* DimYear=2002:

   DimLo\DimAge | [Teens]      [20's]       [30's]       [40's]
   -------------+-----------------------------------------------
   [Canada, Tor | ~            ~            ~            ~
   [Canada, Ott | ~            ~            ~            ~
   [USA, New Yo | ~            ~            ~            ~

DMQL command successfully executed.
```

```
>  [dmql-query "find classification rules for MyCube related to ...

Processing FindClassRule Query:

* DMQL.FindClassRules
  - Cube:     MyCube
  - Dims:     [DimYear, DimLoc, DimAge]
  - Support:  0.1
  - Confid:   0.65
  - ClassAlg: class dm.mine.GiniDTClassProc

The following rules met the query requirements:

  * Rule:       DimYear=1999 ^ DimAge=[20's] => FoodType=1
  - Support:    0.1429
  - Confidence: 1

  * Rule:       DimYear=2001 ^ DimLoc=[Canada, Ottawa] => FoodType=2
  - Support:    0.4286
  - Confidence: 0.6667

  * Rule:       DimYear=2001 => FoodType=2
  - Support:    0.4286
  - Confidence: 0.6667

DMQL command successfully executed.

...
```

For full directions on running the command shell, see the *README* file in the root directory of the project.

5 Your tasks

Following is an outline of the specific tasks you need to perform for this project.

5.1 Checkpoint Requirement: Data-cube and backend JDBC/SQL implementation

For the first checkpoint you will be required to implement the following method in *dm.datacube.Domain* that takes a *DefineCube* DMQL query and populates a data cube using backend JDBC/SQL calls to a database:

```
public boolean processDefineCube(DMQL.DefineCube com);
```

Here are some pointers that may help you with this implementation (the following topics will be covered in more detail during tutorial):

- To populate the *DataCube*, you need to generate a SQL query that populates the *Dimension* data (perhaps doing a join on the table specified by the *DefineCube* query in order to cut down on unused dimension keys). Additionally, you need to generate a SQL query that loads the main *Star Schema* data into the *DataCube* according to the *DefineCube* command.
- Note that the *DefineDim* commands that have been previously executed are cached in the *_hmDimName2DefineDim* field of *Domain*.
- Building the SQL queries is relatively straightforward and can be done simply by processing the *DefineDim* and *DefineCube* command objects and appending an appropriate SQL String (using the + operator) fragment for every element of these commands.[6]
- Querying the database is simple and can be done exactly as demonstrated in lab. For convenience, a database interface class similar to that used in lab (i.e., *dm.db.DBInterface*) is provided as a member of *Domain*. *Domain* automatically takes care of connecting to the database according to the command shell parameters, all you will likely need to do is call the *query(...)* method on the *DBInterface* object to perform a SQL query and obtain the results as a *java.sql.ResultSet*.
- Once you have performed the query and have the results in a *ResultSet*, extract the content row-by-row and put it into the data cube. See the database query example from lab for how to access the contents of a *ResultSet*. (Or look at the *PrintResultSet* method in *dm.db.DBInterface*).
- As an example of loading the *Dimension* and *Domain* data, a few examples loading *dummy* data into these objects have been provided as the current implementation of *processDefineCube(...)*.

Once you have completed this task, verify that your implementation works for a few sample databases and queries and submit the checkpoint as described at the end of this document.

5.2 Final Submission Requirement: Implement data cube roll-up/drill-down

For this task, you will need to implement the basic roll-up/drill-down procedure for data cubes. There are many ways to do this, but for reasons that can make our rule-induction algorithm very efficient, we actually want a lossless scheme for roll-ups. That is, when we drop a dimension, we want to treat all entries in the *DataCube* as equal when they share the same value for all other dimensions, but otherwise retain them in the *DataCube*. Then, we can efficiently drill-down (for this project – just add dimensions) without having to requery the database.[7]

[6] See the class *dm.dmql.DMQL* to view the five different command objects and their field structure.

[7] Note that rolling up still allows us to aggregate over dimensions – we simply have to do the aggregation dynamically as we access the data since we didn't actually

The easiest way to perform drill-downs and roll-ups is to modify the *DCComp* member for the sorted *DataCube* array to ignore certain dimensions when comparing. This can be done through a simple method call to the *DCComp*.

To implement the add and drop dimension commands, you simply need to modify the appropriate methods in *Domain* and *DataCube* to connect the command interface for *AddDropDim* to the implementation of the command. See the comments in these files for direct pointers to the places requiring modification.

To verify that this task is working, the display queries should not reflect the changes made by the add and drop commands.

5.3 Final Submission Requirement: Data mining algorithm modification

Now that we have define procedures for obtaining data cubes, our final task will be to perform efficient data mining with these data cubes.

An example of a decision tree classification rule generator using the Gini metric is defined in *dm.mine.GiniDTClassProc*. For this task, you will want to implement the information gain metric *instead* of the Gini metric.

The information I of a set S of data (where each element has m possible classifications, $C_1 \ldots C_m$) is given by:[8]

$$I(S) = -\sum_{i=1}^{m} \left(\frac{\sharp S_i}{\sharp S} \log \frac{\sharp S_i}{\sharp S} \right) \tag{1}$$

The information *gain* $Gain$ of splitting a set S on dimension d with n possible values is given by:[9]

$$Gain(S, d) = I(S) - \left(\sum_{i=1}^{n} \frac{\sharp S^{d=i}}{\sharp S} I(S^{d=i}) \right) \tag{2}$$

For this task, you will probably want to use *dm.mine.GiniDTClassProc* as a template for implementing the decision tree algorithm using information gain as a splitting metric as opposed to Gini.

This will require you to make your own implementation of interface *dm.mine.ClassificationProc* in a class that should be called *dm.mine.InfoGainDTClassProc*. Note that the current implementation of *dm.mine.GiniDTClassProc* already takes into account the support and confidence thresholds[10] specified by the *FInd-*

collapse it. However, if we ensure that the *DCComp* for dropped dimensions always places these dimensions last in the ordering, then we ensure that all "identical" data from the perspective of the roll-up is contiguous!

[8] $\sharp S_i$ is the number of elements in set S having class C_i. $\sharp S$ is the total count of objects in the set S.

[9] We use $S^{d=i}$ to denote the subset of S having dimension d restricted to a value $i \in 1 \ldots n$.

[10] The support for a rule is simply the fraction of total transaction elements that satisfy a rule and the confidence is simply the fraction of transaction elements satisfying the antecedent that also satisfy the consequent.

ClassRules command. These thresholds are provided in order to avoid generating rules that are of little use to the user and you will want to ensure that all mined rules satisfy these thresholds.

Note that the current implementation of *dm.mine.GiniDTClassProc* is very inefficient in many places and can be optimized considerably. Since efficiency will be a factor in the project grade, it is advised that you attempt to optimize the code for time/memory efficiency when producing the information gain version of the decision tree classification procedure.

Upon completion of this final task, you should test out your mining algorithm on a few example databases to test that everything is working correctly and to experiment with data mining!

6 Submission and marking

6.1 Turning in your project

The checkpoint due date for this project is **Oct. 27, 2003 at midnight**. The final submission due date for this project is **Nov. 3, 2003 at midnight**. To turn in the project, **use either a *tar* or *zip* utility such as *WinZip* to compress the *Project1* directory along with its subdirectories and email this *single* compressed file to Scott Sanner at *ssanner@cs.toronto.edu***. You will be sent notification within 24 hours that the project has been received and that it can be successfully decompressed.

In the event that there are any complications with the submission process (e.g., a corrupted file), we will request to view the directories you are submitting to verify that no file has been modified past the due date/time. Thus, **you must ensure that you have a copy of the code on your local *H* drive on the ECF machines (so we know that the timestamp is valid) and you must *not* modify it beyond the due date (otherwise the timestamp will indicate this and we will not be able to accept your project)**.

6.2 Evaluation

Your program will be evaluated using some command-line scripts that we develop. These will simply be more complex versions of the example script *test.dmsh* and example database *StarSchemaEx.mdb* given in the root directory of the project.

Evaluation will be based on the following criteria:

1. *Completness* – Have you completed all of the requirements?
2. *Clarity* – Is your code well-structured and understandable?
3. *Commenting* – Do you have general as well as JavaDoc-specific comments?
4. *Correctness* – Are there any errors which lead to incorrect program behavior?
5. *Efficiency* – How efficient is your implementation for queries on very large indices? How long does it take to process a query?

6.3 Extra credit

For extra credit, you may consider the following project enhancements:

1. Implement better display methods. You can either enhance the text display with additional features or use a simple Java graph-drawing package to graphically display multidimensional data.
2. Implement a concept hiearchy and allow roll-up/drill-down over this hierarchy rather than restricting these operations to simple dimension add/drops as is currently done.
3. Modify the code to handle multiattribute dimension keys.
4. Modify the code to handle multiattribute data cube content values.
5. Extend the data cube content value to types other than the Integer type implemented here.
6. Allow aggregation over ranges for both dimension and value data.
7. Allow for continuous domains and discretization for both dimension and value data.
8. Implement additional aggregation functions for the display query.
9. Test other decision tree induction metrics.
10. Implement additional rule induction algorithms (e.g., the APRIORI algorithm).

References

1. Han, J., Fu, Y., Wang, W., Koperski, K., Zaiane, O.: DMQL: A data mining query language for relational databases. In: SIGMOD'96 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'96), Montreal, Canada (1996)
2. Fox, M.S.: Lecture Slides, Weeks 5, 6, and 8: Data and Knowledge Mining. On-line: http://www.mie.utoronto.ca/courses/mie457f/ (2003)
3. Han, J., , Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann (2001)