Available online at www.sciencedirect.com
ScienceDirect
Expert Systems with Applications 34 (2008) 2858–2869
Expert Systems with Applications
www.elsevier.com/locate/eswa

An efficient bit-based feature selection method

Wei-Chou Chen [a], Shian-Shyong Tseng [a], Tzung-Pei Hong [b,*]

[a] Department of Computer and Information Science, National Chiao Tung University, Hsinchu 300, Taiwan, ROC
[b] Department of Electrical Engineering, National University of Kaohsiung, Kaohsiung 811, Taiwan, ROC

Abstract

Feature selection is about finding useful (relevant) features to describe an application domain. Selecting relevant and enough features to effectively represent and index the given dataset is an important task to solve the classification and clustering problems intelligently. This task is, however, quite difficult to carry out since it usually needs a very time-consuming search to get the features desired. This paper proposes a bit-based feature selection method to find the smallest feature set to represent the indexes of a given dataset. The proposed approach originates from the bitmap indexing and rough set techniques. It consists of two-phases. In the first phase, the given dataset is transformed into a bitmap indexing matrix with some additional data information. In the second phase, a set of relevant and enough features are selected and used to represent the classification indexes of the given dataset. After the relevant and enough features are selected, they can be judged by the domain expertise and the final feature set of the given dataset is thus proposed. Finally, the experimental results on different data sets also show the efficiency and accuracy of the proposed approach.

Keywords: Feature selection; Bitmap indexing; Rough set; Classification; Clustering

1. Introduction

Feature selection is about finding useful (relevant) features to describe an application domain. Selecting relevant and enough features to effectively represent and index the given dataset is an important task to solve the classification and clustering problems intelligently. This task is, however, quite difficult to carry out since it usually needs an exhaustive search to get the features desired. In the past, some approaches have been proposed to solve the feature selection problem (Almullim et al., 1991; Chen, Tseng, Chen, & Jiang, 2000, 2002, 2003; Doak, 1992; Gonzalez & Perez, 2001; Huang & Tseng, 2004; John et al., 1994; Lee, Chen, Chen, & Jou, 1997; Liu et al., 1996; Liu & Setiono, 1998; Quinlan, 1986; Yu, 2001). These approaches can roughly be classified into the following two strategies:

1. Optimal strategy: This kind of approaches considers all the subsets of a given feature set (Almullim et al., 1991; Schlimmer et al., 1993; Wu, 1999). Some searching techniques, such as branch and bound, may be adopted to reduce the search space. For example, Liu et al. proposed a special feature selector (Liu et al., 1996), which randomly produced feature subsets according to the Las Vegas algorithm (Brassard, 1996). It thus searched the entire solution spaces and guaranteed to get an optimal feature set.
2. Heuristic strategy: This kind of approaches prunes search spaces according to some heuristics. The results obtained by these approaches are usually not optimal, but within a short time (Zhong, Dong, & Ohsuga, 2001). There are three typical heuristic approaches for feature selection, including *forward selection, backward selection* and *bi-directional selection*. The forward selection approach initializes the desired feature set as null and then adds features into it until the results are satisfactory (Miao, 1999; Skowron & Rauszer, 1992; Yu, 2001). The backward selection approach initializes the

* Corresponding author.
E-mail addresses: sstseng@cis.nctu.edu.tw (S.-S. Tseng), tphong@nuk.edu.tw (T.-P. Hong).

0957-4174/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2007.05.037

desired feature set as all the given features and then removes unnecessary features from it (Choubey, 1998; Yu, 2001). The bi-directional selection approach initializes the desired feature set as a partial feature set, and then either puts good features into it or eliminates bad features from it (Doak, 1992).

In the past, we proposed a bitwise indexing method based on a given feature set to accelerate case matching in CBR (Chen, Tseng, Chang, & Jiang, 2001, 2002). In this paper, we further investigate the determination of the appropriate feature set. We propose a two-phase feature selection approach to discover significant feature sets from a given database table, and use them to further investigation. The proposed feature selection approach originates from the bitmap indexing (O'Neil & Quass, 1997; Wu et al., 1998) and rough set techniques (Pawlak, 1982, 1991). Naturally, it is designed to discover optimal feature sets for a given dataset since the proposed method is originated from the rough set theory. The Experimental results also show the efficiency and accuracy of the proposed approach.

This paper is organized as follows. Some related works are reviewed in Section 2. The proposed feature selection method and some corresponding definitions and algorithms are stated in Section 3. The time and space complexities of the proposed algorithms are analyzed in Section 4. Experimental results are shown in Section 5. Conclusions are finally given in Section 6.

2. Review of feature selection and rough sets

Feature selection is about finding useful (relevant) features to describe an application domain. The problem of feature selection can formally be defined as selecting minimum features M' from original M features where $M' \subseteqq M$ such that the class distribution of M' features is as similar as possible to M features (Last, Kandel, & Maimon, 2001). Generally speaking, the function of feature selection is divided into three parts: (1) simplifying data description, (2) reducing the task of data collection, and (3) improving the quality of problem solving. The benefits of having a simple representation are abundant such as easier understanding of problems, and better and faster decision making. In the field of data collection, having less features means that less data need to be collected. As we know, collecting data is never an easy job in many applications because it could be time-consuming and costly. Regarding the quality of problem solving, the more complex the problem is if it has more features to be processed. It can be improved by filtering out the irrelevant features which may confuse the original problem, and it will win the better performance. There are many discussions about feature selection, and many existing methods to assist it, such as GA technology (Raymer, Punch, Goodman, & Kuhn, 2000), entropy measure (Huang & Tseng, 2004), and rough set theory (Tseng, Jothishankar, & Wu, 2004; Yu, 2001).

Next, the rough set theory is briefly reviewed. The rough set theory, proposed by Pawlak in 1982 (Pawlak, 1982), can serve as a new mathematical tool for dealing with data classification problems. It adopts the concept of equivalence classes to partition training instances according to some criteria. Two kinds of partitions are formed in the mining process: lower approximations and upper approximations. Rough sets can also be used for feature reduction. The features that do not contribute to the classification of the given training data are removed. The concepts of equivalence classes and approximations are quite suitable to generate the bit-based class vectors and record vectors, which can then be directly and efficiently transformed to the bitwise indexing matrixes for CBR systems (Yang et al., 2000). This paper thus adopts these concepts to solve the feature selection problem.

3. The proposed bitmap-based feature selection method

As we mentioned above, we proposed a heuristic feature selection approach, called the *bitmap-based feature selection method with discernibility matrix* (Chen, Yang, & Tseng, 2002), to find a nearly optimal feature set. However, finding the optimal solutions of feature selection is still needed in some applications. Although, some exhaustive search methods can guarantee the optimality of selected feature sets, the computation cost may be very high.

In this section, we thus consider finding an optimal solution via the rough set techniques and the bit-based indexing method for the feature selection. The proposed approach encodes a given data set into a bit vector matrix and uses bit-processing operations on them to reduce the computation time. The proposed approach consists of several main steps, as shown in Fig. 1.

There are two-phases in the proposed algorithm – bitmap indexing phase and feature selection phase. In the bitmap indexing phase, the given dataset is transformed into a bitmap indexing matrix with some additional data information. In the feature selection phase, a set of relevant and enough features are selected and used to represent the dataset. The details of the two-phases are described in following sub-sections.

3.1. Problem definitions

Let T denote a target table in a database, R denote the set of n records in T, and C denote the set of m features in T, R can then be represented as $\{R_1, R_2, \ldots, R_n\}$, where R_i is the ith record. C can be represented as $\{C_1, C_2, \ldots, C_m\}$, where C_j is the jth feature. The first $m-1$ elements in C are condition features and the last one, C_m, is a decision feature. Let V_j denote the domain of C_j. V_j can then be represented as $\{V_{j1}, V_{j2}, \ldots, V_j\sigma_j\}$, where each element is a possible value of C_j and σ_j is the number of possible values of C_j. Let $V_j(i)$ denote the value of C_j in record R_i, $V_j(i) \neq null$. Table 1 shows an example of a target table T with ten records $R = \{R_1, R_2, \ldots, R_{10}\}$ and five features

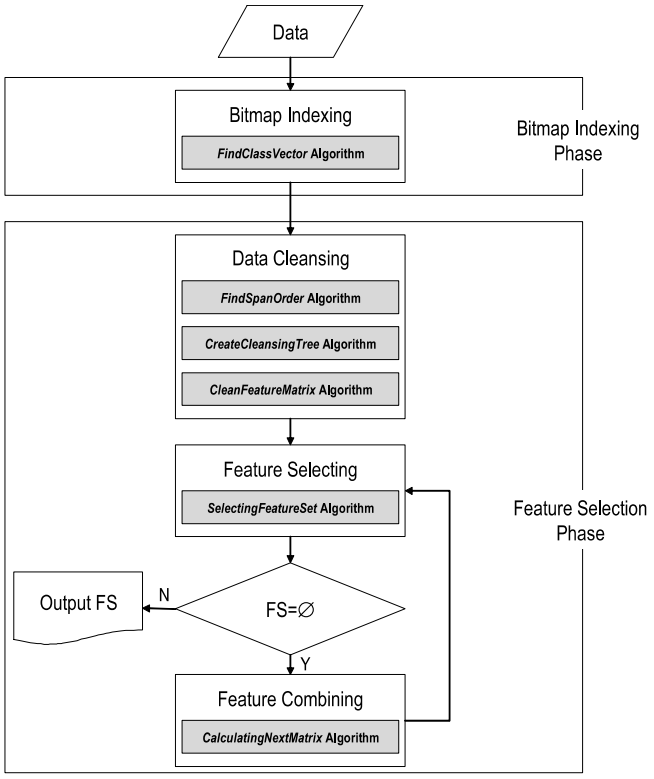

Fig. 1. The flowchart of the proposed feature selection approach.

Table 1
An example of a target table

	C_1	C_2	C_3	C_4	C_5
R_1	M	L	3	M	1
R_2	M	L	1	H	1
R_3	L	L	1	M	1
R_4	L	R	3	M	2
R_5	M	R	2	M	2
R_6	L	R	3	L	3
R_7	H	R	3	L	3
R_8	H	N	3	L	3
R_9	H	N	2	H	2
R_{10}	H	N	2	H	1

$C = \{C_1, C_2, C_3, C_4, C_5\}$. C_5 is a decision feature and the others are condition features.

The purpose of this paper is to find the one of the smallest feature set to effectively index the given table. The definitions and algorithms used in the bitmap indexing phase and in the feature selection phase are described below.

3.2. Bitmap indexing phase

In this phase, the target table is first transformed into a bitmap indexing matrix with some additional classification information. Let b_i is a bit in a bit vector. Let ONE_k denote the bit string of length k, with all the bits set to 1, $ZERO_k$ denote the one with all the bits set to 0, and $UNIQUE_k$ denote the one, with only one bit set to 1 and the others set to 0. A record vector, which is used to keep the information of the records with a specific value of a feature, is defined below.

Definition 1 (*Record vector*). A record vector RV_{jk} is a bit string $b_1, b_2, \ldots, b_n$, with b_i set to 1 for $V_j(i) = V_{jk}$ and set to 0 otherwise, where $1 \leqslant j \leqslant m$, $1 \leqslant k \leqslant \sigma_j$, and $1 \leqslant i \leqslant n$.

RV_{jk} thus keeps the information of the records with the kth possible value of the feature C_j. For example in Table 1, C_1 has three possible values $\{M, L, H\}$. The record vector for $C_1 = M$ is 1100100000 since the first, second and fifth records have this feature value. Similarly, the record vector for $C_1 = L$ is 0011010000 and for $C_1 = H$ is 0000001111. All the record vectors are shown in the third column of Table 2.

A class vector, which is used to keep the information of the classes (values of the decision feature) with a specific value of a feature, is defined below.

Definition 2 (*Class vector*). A class vector CV_{jk} is a bit string $b_1, b_2, \ldots, b\sigma_m$, with b_i set to 1 if $RV_{jk} \cap RV_{mi} \neq ZERO_n$, and set to 0 otherwise, where σ_m is the number of possible values of C_m and n is the number of records in R.

Here, the "AND" bitwise operator is used for the intersection in Definition 2. CV_{jk} thus keeps the information of the classes related to the kth possible value of the feature C_j. For example in Table 2, the record vector (RV_{11}) for $C_1 = M$ is 1100100000 and the one (RV_{51}) for $C_5 = 1$ is 1110000001. Since, the bitwise intersection of 1100100000 and 1110000001 is 1100000000, not equal to $ZERO_{10}$, the first bit in RV_{11} is thus 1. Similarly, the second and third bits in RV_{11} are 1 and 0 from the intersection results of RV_{11} with RV_{52}, and with RV_{53}. the class vector CV_{11} is thus 110. All the class vectors are shown in the fourth column of Table 2. Formally, a class vector CV_{jk} can be obtained by the following *FindClassVector algorithm*.

Table 2
The record vectors and class vectors from Table 1

Feature	Feature-value	Record vector	Class vector
C_1	V_{11}	1100100000	110
	V_{12}	0011010000	111
	V_{13}	0000001111	111
C_2	V_{21}	1110000000	100
	V_{22}	0001111000	011
	V_{23}	0000000111	111
C_3	V_{31}	1001011100	111
	V_{32}	0110000000	100
	V_{33}	0000100011	110
C_4	V_{41}	1011100000	110
	V_{42}	0100000011	110
	V_{43}	0000011100	001
C_5	V_{51}	1110000001	100
	V_{52}	0001100010	010
	V_{53}	0000011100	001

Algorithm 1 (*FindClassVector*).

Input: Record vector RV_{jk}.
Output: Class vector CV_{jk}.
Step 1: Set CV_{jk} to $ZERO\sigma_m$.
Step 1: For each i, $1 \leqslant i \leqslant \sigma_m$, set the ith bit of CV_{jk} to 1 if $RV_{jk} \cap RV_{mi} \neq ZERO_n$; otherwise, set it to 0.
Step 1: Return CV_{jk}.

Definition 3 (*Feature-value vector*). A feature-value vector F_{jk} is the concatenation of RV_{jk} and CV_{jk}.

For example, the feature-value vector F_{11} in Table 2 is 1100100000110, which is RV_{11} concatenated with CV_{11}. All the feature-value vectors for a feature are then collected together as a feature matrix. This is defined below.

Definition 4 (*A feature matrix for a feature*). A feature matrix M_j for the feature C_j is denoted $\begin{bmatrix} F_{j1} \\ F_{j2} \\ \vdots \\ F_{j\sigma_j} \end{bmatrix}$, where σ_j is the number of possible values in C_j.

For example, the feature matrix M_1 in Table 2 is show as follows:

$$M_1 = \begin{bmatrix} 1100100000\underline{110} \\ 0011010000\underline{111} \\ 0000001111\underline{111} \end{bmatrix}$$

The bits with underlines are class vectors. From the definition of the feature matrix, it is easily derived that applying the bitwise operator "OR" on all the record vectors in a feature matrix will get the ONE_n vector, and applying the bitwise operator "AND" on any two record vectors in a feature matrix will get the $ZERO_n$ vector. Note that, the "OR" and "AND" operators are defined for executing the "OR" and "AND" operations on all corresponding bits of the given two bit vectors. Thus, if we apply the bitwise operator "XOR" on all the record vectors in a feature matrix, we will also get the $ZERO_n$ vector. Take M_1 as an example. The result for 1100100000 OR 0011010000 OR 0000001111 is 1111111111. The result for 1100100000 AND 0011010000 is 0000000000. The result for 1100100000 XOR 0011010000 XOR 0000001111 is 0000000000.

Definition 5 (*A feature matrix for a table T*). A feature matrix M for a table T is denoted $\begin{bmatrix} M_1 \\ M_2 \\ \vdots \\ M_m \end{bmatrix}$, where m is the number of features in T.

For example, the matrix composed of the bit strings from columns 3 and 4 of Table 2 is the feature matrix for the data given in Table 1. The feature matrix for a table is then input to the feature selection phase to find relevant and enough features.

3.3. Feature selection phase

In this phase, we want to find a set of relevant and enough features to represent the given dataset. It is further divided into several stages. First, a feature-based spanning tree is built for cleansing the bitmap indexing matrix. The dataset with noisy information is thus judged and filtered out according to the spanning tree. The cleansed, noisy-free bitmap indexing matrix can then be used to determine the optimal feature set for some classification and clustering problems.

3.3.1. Creating cleansing tree

Before the feature selection phase is executed, the correctness of the target table needs to be verified. If there are some records in the target table with the same values of all condition features, but with different ones of the decision feature, they are treated as noise records and are filtered out from the target table. Intuitively, every two records can be compared to find out the inconsistent records in the target table. Its time complexity is $O(n^2m)$, where n is the number of records and m is the number of features. Below, we propose the concept of a cleansing tree to decrease the time complexity to $O(nmj)$, where j is the maximum number of possible feature values of a feature and *n is usually much larger than j* in general classification and clustering problems. The formation of a cleaning tree depends on the given feature order. We thus have the following definition.

Definition 6 (*Spanned feature order*). A spanned feature order O is a permutation consisting of all the condition features in a target table T.

For example in Table 1, $\langle C_1, C_2, C_3, C_4 \rangle$ can be a spanned feature order. When a spanned feature order is given, a cleansing tree can then be built according to it. The definition of a cleansing tree is first given below.

Definition 7 (*Cleansing tree*). A cleansing tree *Ctree* is a tree with a root denoted *root*[*Ctree*]. Every node x in the tree corresponds to a feature value. A node y is the parent of a node x if the feature of y precedes the feature of x in the given spanned feature order. A node z is the sibling of a node x if they have the same feature, but different values.

A structure of a cleansing tree is shown in Fig. 2. Its maximum height is $m-1$, where m is the number of features in a decision table T. Each node x has three pointers, which are $p[x]$, *left-child*$[x]$ and *right-sibling*$[x]$, respectively pointing to its parent node, its leftmost child node and its first right-sibling node. It also contains two additional information, *record*$[x]$ and *class*$[x]$, which indicate the associated record and class vectors of x. If node x has no child, then *left-child*$[x] =$ NIL; if node x is the rightmost child of its parent, then *right-sibling*$[x] =$ NIL.

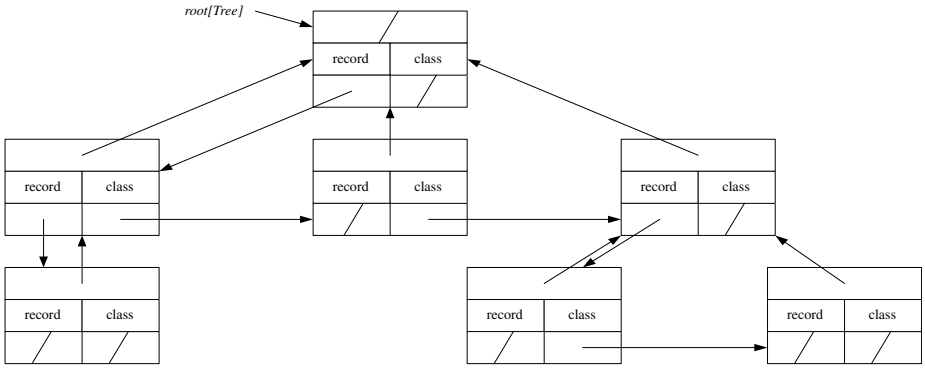

Fig. 2. The structure of a cleansing tree.

As mentioned above, records may have the same values of all condition features, but different value of the decision feature. These records are called inconsistent. Inconsistent records can also be found out when the cleansing tree is built. The building algorithm uses the valid mask vector to find the consistent records. The valid mask vector is defined as follows.

Definition 8 (*Valid mask vector*). A valid mask vector *ValidMask* for a target table T a bit string $b_1, b_2, \ldots, b_n$, with b_i set to 1 if the ith record R_i is not inconsistent with other records, and set to 0 otherwise.

The cleansing tree for a given spanned feature order can be built by the following *CreateCleansingTree* algorithm. The *ValidMask* is initially set to ONE_n, and will be modified along with the execution of the *CreateCleansingTree* algorithm.

Algorithm 2 (*CreateCleansingTree*).

Input: A feature matrix M, the valid mask *ValidMask* and a spanned feature order O.
Output: The valid mask *ValidMask*.
Step 1: Create an empty node x and set it as the root node.
Step 2: Initialize $record[x] = ONE_n$, $class[x] = ONE\sigma_m$ and $depth = 0$, where the variable *depth* is used to represent the depth of the node x in the cleansing tree.
Step 3: Set $px = x$, where px is used to keep the current parent node.
Step 4: If $class[x]$ is not equal to $UNIQUE_{\sigma_m}$ and *depth* is not equal to $m-1$, do Step 5 to build the child nodes of node x; otherwise, go to Step 7.
Step 5: Let C_j be the current feature in the spanned feature order to be considered. For each feature-value vector F_{jk} in a feature matrix M_j for C_j, if $(record[px]$ AND $RV_{jk}) \neq ZERO_n$, do the following sub-steps:
Step 5.1: Create an empty node y.
Step 5.2: If $left\_child[x] =$ NIL, consider y as a child node of x and set $p[y] = x$ and $left\_child[x] = y$; otherwise, consider y as a sibling node of x and set $p[y] = p[x]$ and $right\_sibiling[x] = y$.
Step 5.3: Set $record[y] = (record[p[y]]$ AND $RV_{jk})$ and $class[y] = (class[p[y]]$ AND $CV_{jk})$.
Step 5.4: If $depth = m-1$ and $class[y] \neq UNIQUE_{\sigma_m}$, set $ValidMask = (record[y]$ XOR $ValidMask)$.
Step 5.5: Set $x = y$.
Step 6: If $left\_child[px] \neq$ NIL, set $x = left\_child[px]$, $depth = depth + 1$ and go to Step 3. Otherwise, do the next step.
Step 7: If $right\_sibiling[x] \neq$ NIL, $x = right\_sibiling[x]$ and go to Step 3; otherwise, set $x = p[x]$ and do the next step.
Step 8: If $x \neq Tree[root]$, go to Step 7; otherwise, return *ValidMask* and stop the algorithm.

For example, the cleansing tree for the data in Table 1 with the spanned feature order $\langle C_1, C_2, C_3, C_4 \rangle$ will be built as shown in Fig. 3. At first, the root node is generated and all the bits in *record*[root] and *class*[root] are set to 1. Since, *class*[root] is not equal to $UNIQUE_3$, and the current depth is 0, not equal to $m-1$, the next step is executed to build the child nodes of the root. The first feature C_1 in the spanned feature order is considered. Since, it has three possible values and $(record$[root] AND $RV_{1k})$, $k = 1$ to 3, is not equal to $ZERO_{10}$, three nodes, represented as nodes 1, 2 and 3, are created as the children of the root. Since, node 1, the left-child node of the root, is not NIL, it is then processed to generate its child nodes in the same way. Nodes 4 and 5 are then created for the second feature C_2 in the spanned feature order. Since, *class*[node 4] has been equal to ONE_{10}, the sibling of node 4, which is node 5, is then considered. Since, *class*[node 5] has also been equal to ONE_{10}, the sibling of node 5, is then considered. But since node 5 has no sibling, its parent node, node 1 is considered. The sibling of node 1, which is node 2 is then processed. The same procedure is then executed until the whole cleansing tree is generated.

The numbers at the left of the nodes in Fig. 3 indicate the order built. In node 15, the second and third bits of the class vector are both "1". It means that the corresponding record vectors will have more than one "1". The corresponding records with bit "1" are then inconsistent since

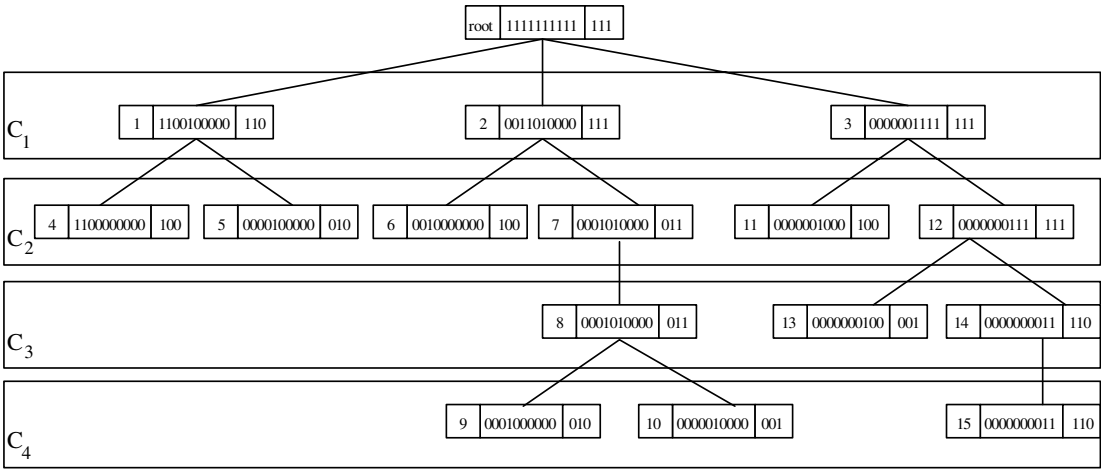

Fig. 3. Cleansing tree with feature spanned order $\langle C_1, C_2, C_3, C_4\rangle$.

their values of all condition features are the same, but their values of the decision feature are different. In this example, the ninth and tenth records are inconsistent. The *ValidMask* are thus modified from "1111111111" to "1111111100".

3.3.2. Finding appropriate spanned order

In the above example, the spanned feature order O is set as $\langle C_1, C_2, C_3, C_4\rangle$. Different orders will apparently affect the performance of the cleansing spanning trees built. A cleansing spanning tree with a better spanned feature order can reduce the space and time complexities. In the past, there were some famous tree structures for classification, such as the decision-tree approach (Quinlan, 1986), which was based on the entropy theory to select the next best feature. In order to reduce the computational complexity for evaluating the spanning order of features, the following heuristics are thus proposed.

H1: The more '1' bits a record vector for a feature value has, the more weight the feature value has.
H2: The more '1' bit the class vector for a feature value has, the less weight the feature value has.

These two heuristics show the relationship between feature values and classes. If a feature value appears in most records with a single class, the weight of this feature value is relatively high. These heuristics can be used to save the computation time when compared to using the entropy theory. The following *FindSpanOrder algorithm* is thus proposed to determine the spanned feature sequence O of all condition features by evaluating the feature weights according to the above heuristics.

Algorithm 3 (*FindSpanOrder*).

Input: A feature matrix M for a table T
Output: A spanned feature order O.
Step 1: Initialize $weight_j = 0$, where $1 \leqslant j \leqslant m-1$.
Step 2: For each M_j in M, set:

$$weight_j \leftarrow \sum_{k=1}^{\sigma_j} \frac{Count(RV_{jk})}{[Count(CV_{jk})]^2},$$

where the function $Count(x)$ is used to count the number of '1' bits in x.
Step 3: Order the features in O in the descendent order of the *weight* values.
Step 4: Return O.

For example, according to the feature matrix in Table 2, the weight of each feature is calculated as shown in Table 3.

The new spanned feature sequence O determined by the above algorithm is thus $\langle C_4, C_2, C_3, C_1\rangle$, instead of the original order $\langle C_1, C_2, C_3, C_4\rangle$. The cleansing tree generated on the new order is shown in Fig. 4.

As we can see, the cleansing tree with new feature order $O = \langle C_4, C_2, C_3, C_1\rangle$ in Fig. 4 is much smaller than that in Fig. 3. The number of nodes has decreased from 15 to 9. Therefore, the computational time of generating and traversing the spanning tree can be greatly reduced.

3.3.3. Cleansing feature matrix

After the cleansing tree is built, the *ValidMask* may not be ONE_n since inconsistent records may exist. The *ValidMask* is then used by the following *CleansingFeatureMatrix* algorithm to remove the inconsistent records from the feature matrix.

Algorithm 4 (*CleansingFeatureMatrix*).

Input: A feature matrix M for a table T and a valid mask vector *ValidMask*.
Output: A cleansed feature matrix M.
Step 1: For each feature-value vector F_{ij} in M, do following sub-steps:
Step 1.1: $RV_{ij} = RV_{ij}$ AND *ValidMask*.
Step 1.2: $CV_{ij} = FindClassVector\ (RV_{ij})$.
Step 2: Return M.

For example, the *ValidMask* is set to "1111111100" after the cleansing tree for Table 1 is built. Since, the ninth and tenth bits of the *ValidMask* are 0, the *CleansingFeatureMatrix* algorithm will set these two bits of all the record vectors in Table 2 to 0. The class vector of each feature value is then recalculated by the *FindClassVector algorithm* according to its new record vector. The revised feature matrix is shown in Table 4.

3.3.4. Some definitions on feature sets

For effectively distinguishing the classes from the feature values, we must extend the concepts related to a single feature to a feature sets. The following definitions are thus needed.

Definition 9 (Power of a feature set). C^s is called the s-power of a feature set C, if each element in C^s is composed of s distinct condition features from C, $1 \leqslant s \leqslant m-1$.

Thus, we have $C^1 = C$. For example, the power set C^1 for the data in Table 1 is $\{\{C_1\}, \{C_2\}, \{C_3\}, \{C_4\}\}$. The power set C^2 is $\{\{C_1,C_2\}, \{C_1,C_3\}, \{C_1,C_4\}, \{C_2,C_3\}, \{C_2,C_4\}, \{C_3,C_4\}\}$. Let $|C^s|$ denote the cardinality of C^s. Then:

$$|C^s| = \binom{m-1}{s}.$$

Let C^s_j denote the jth element in C^s, $1 \leqslant j \leqslant |C^s|$. C^s_j is then a feature set. Also let V^s_j denote the domain of C^s_j, σ^s_j denote the number of possible values in V^s_j, and V^s_{jk} denote the kth feature value of C^s_j. Each feature set can be represented by a name vector, defined below.

Definition 10 (*Name vector of a feature set*). The name vector NV^s_j of a feature set C^s_j is a bit string $b_1, b_2, \ldots, b_{m-1}$, with b_i set to 1 if featureC_i is included in C^s_j and set to 0 otherwise.

For the above example, C^1_1 denotes the first element in C^1, which is $\{C_1\}$. The name vector NV^1_1 is then 1000 since only C_1 is included in C^1_1. For another example, $C^2 = \{\{C_1, C_2\}, \{C_1,C_3\}, \{C_1,C_4\}, \{C_2,C_3\}, \{C_2,C_4\}, \{C_3,C_4\}\}$. C^2_1 denotes the first element in C^2, which is $\{C_1,C_2\}$. The name vector NV^2_1 is then 1100 since C_1 and C_2 are included in C^2_1. Similar to a single feature, some terms related to a feature set is defined below.

Definition 11 (*Record vector of a feature set*). A record vector RV^s_{jk} of a feature set value C^s_{jk} is a bit string $b_1, b_2, \ldots, b_n$, with b_i set to 1 for $V^s_j(i) = V^s_{jk}$ and set to 0 otherwise, where $1 \leqslant j \leqslant |C^s|$ and $1 \leqslant k \leqslant \sigma^s_j$.

Table 3
Calculating the weight of each feature

Feature	Weight	Old order	New order
C_1	$3/4 + 3/9 + 4/9 = 1.53$	1	4
C_2	$3/1 + 4/4 + 3/9 = 4.33$	2	2
C_3	$5/9 + 2/1 + 3/4 = 3.31$	3	3
C_4	$4/4 + 3/4 + 3/1 = 4.75$	4	1

RV^s_{jk} thus keeps the information of the records with the kth possible value of the feature set C^s_j. A class vector, which is used to keep the information of the classes (values of the decision feature) with a specific value of a feature set, is defined below.

Definition 12 (*Class vector of a feature set*). A class vector of CV^s_{jk} of a feature set value C^s_{jk} is a bit string $b_1, b_2, \ldots, b$, with b_i set to 1 if $RV^s_{jk} \cap RV_{mi} \neq ZERO_n$, and set to 0 otherwise, where σ_m is the number of possible values of C_m and n is the number of records in R.

CV^s_{jk} thus keeps the information of the classes related to the kth possible value of the feature set C^s_j. A feature-value vector of a feature set is defined below.

Definition 13 (*Feature-value vector of a feature set*). A feature-value vector F^s_{jk} is composed of RV^s_{jk} and CV^s_{jk}.

Definition 14 (*A feature matrix for a feature set*). A feature matrix M^s_j for the feature set C^s_j is denoted $\begin{bmatrix} F^s_{j1} \\ F^s_{j2} \\ \vdots \\ F^s_{j\sigma_j} \end{bmatrix}$, where $1 \leqslant j \leqslant |C^s|$ and σ^s_j is the number of possible values in C^s_j.

Definition 15 (s-feature matrix for a table T). An s-feature matrix M^s for a table T is denoted $\begin{bmatrix} M^s_1 \\ M^s_2 \\ \vdots \\ M^s_{|C^s|} \end{bmatrix}$, where $1 \leqslant s \leqslant m-1$.

3.3.5. Selecting a feature set

In this sub-section, two algorithms are proposed to find the desired feature set. The first algorithm, named the *SelectingFeatureSet* algorithm, is used to find a feature set from a given s-feature matrix. If there exists a feature set which is sufficient to decide all the records in the given dataset, the feature set will be returned and the feature selection procedure stops. Otherwise, s is incremented and the *SelectingFeatureSet* algorithm is executed again. The second algorithm, named the *CalculatingNextMatrix* algorithm, derives the new feature matrix from the previous feature matrix. The *SelectingFeatureSet* algorithm is described as follows.

Algorithm 5 (*SelectingFeatureSet*).

Input: An s-feature matrix M^s for a table T.
Output: A selected feature set FS.
Step 1: Initialize $FS = \varnothing$, $j = 1$.
Step 2: If $j \leqslant |C^s|$, do the next step; otherwise go to Step 7.
Step 3: Set $k = 1$, where k is used to keep the number of the value currently processed in a feature set C^s_j.

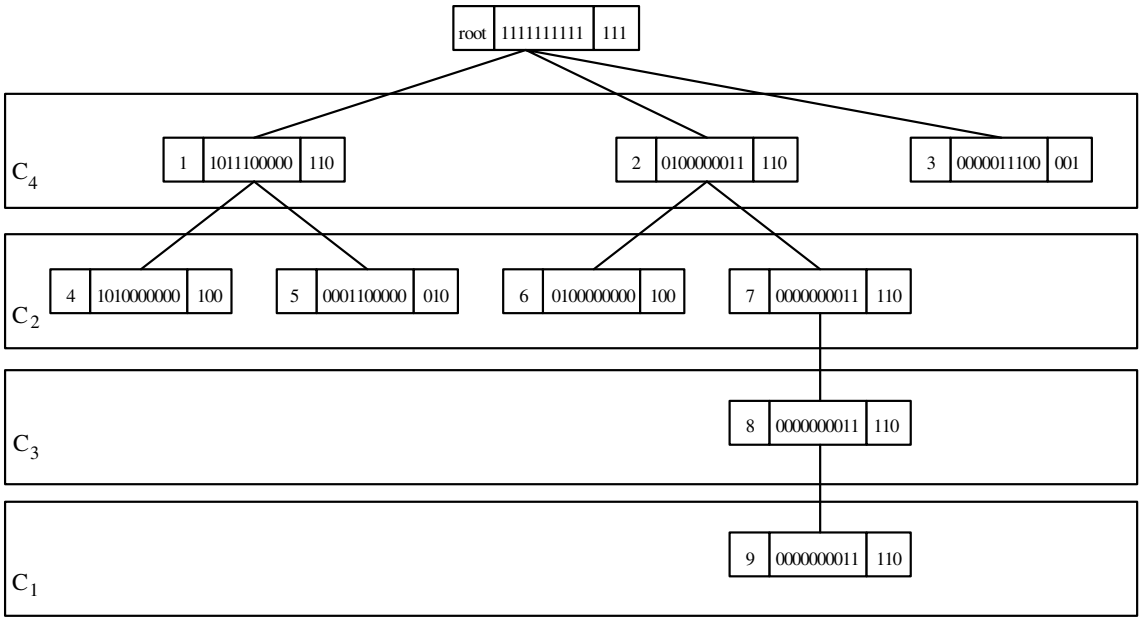

Fig. 4. The cleansing tree generated on the new order $\langle C_4, C_2, C_3, C_1\rangle$.

Step 4: If $k \leqslant \sigma_j^s$, do the next step; otherwise go to Step 6.
Step 5: If $CV_{jk}^s \neq UNIQUE_{\sigma_m}$, set $j = j + 1$ and go to Step 2; otherwise set $k = k + 1$ and go to Step 4.
Step 6: Set $FS = C_j^s$; That is, for each i from 1 to $m-1$, set $FS = FS \cup \{C_i\}$ if the ith bit of the name vector NV_j^s for feature set C_j^s is equal to 1.
Step 7: Return FS.

Take the data in Table 1 as an example to illustrate the above algorithm. s is set at 1 at the beginning. The 1-feature matrix M^1 for the data is the same as the feature matrix M found before. The *SelectingFeatureSet* algorithm will examine the 1-feature sets one by one. The first element M_1^1, which is $\{C_1\}$, is then processed. The class vector CV_{11}^1 for the first feature value C_{11}^1 is 110, which is not equal to $UNIQUE_3$. Using the feature set $\{C_1\}$ can thus not completely distinguish the classes. The other elements in the 1-feature matrix M^1 are then processed in a similar way. In this example, no element is chosen. Thus $\varnothing$ is returned. It means no single feature can completely distinguish the classes s is then incremented, and the *SelectingFeatureSet* algorithm is then executed from the new s-feature matrix. The new feature matrix can be easily derived from the previous feature matrix by the following *CalculatingNextMatrix* algorithm.

Table 4
The cleansed feature matrix of Table 2

Feature	Feature-value	Record vector	Class vector
C_1	V_{11}	1100100000	110
	V_{12}	0011010000	111
	V_{13}	0000001100	001
C_2	V_{21}	1110000000	100
	V_{22}	0001111000	011
	V_{23}	0000000100	001
C_3	V_{31}	1001011100	111
	V_{32}	0110000000	100
	V_{33}	0000100000	010
C_4	V_{41}	1011100000	110
	V_{42}	0100000000	100
	V_{43}	0000011100	001
C_5	V_{51}	1110000000	100
	V_{52}	0001100000	010
	V_{53}	0000011100	001

Algorithm 6 (*CalculatingNextMatrix*).

Input: An s-feature matrix M^s for a table T.
Output: An $(s+1)$-feature matrix M^{s+1} for a table T.
Step 1: For each j, $j = 1$ to $|C^s| - 1$, do the following steps.
Step 2: For each l, $l = (j \bmod m) + 1$ to m, do the following sub-steps.
Step 2.1: Set $NV_j^{s+1} = NV_j^s$ OR NV_l^1.
Step 2.2: Set the temporary counter k to 1.
Step 2.3: For each feature-value vector F_{jx}^s in M_j^s, $1 \leqslant x \leqslant |C_j^s|$, do the following sub-steps:
Step 2.3.1: For each feature-value vector F_{ly}^1 in M_l^1, $1 \leqslant y \leqslant |C_l^1|$, do the following sub-steps:
Step 2.3.1.1: Set $RV_{jk}^{s+1} = RV_{jx}$ AND RV_{ly}^1.
Step 2.3.1.2: Set $CV_{jk}^{s+1} = CV_{jx}^s$ AND CV_{ly}^1.
Step 2.3.1.3: IF $CV_{jk}^{s+1} \neq UNIQUE_{\sigma_m}$, set $CV_{jk}^{s+1} = FindClassVector(RV_{jk}^{s+1})$.
Step 2.3.1.4: Set $k = k+1$.
Step 3: Return the $(s+1)$-feature matrix M^{s+1}.

For example, the 2-feature matrix M^2 for the data in Table 1 is generated from the 1-feature matrix M^1 as follows. The name vector for feature C_1^2 is first calculated. Thus:

$$NV_1^2 = NV_1^1 \text{ OR } NV_1^1 = 1000 \text{ OR } 0100 = 1100.$$

The feature-value vector F_{11}^2 in M_1^2 is then calculated. The record vector is found as follows:

$$RV_{11}^2 = RV_{11}^1 \text{ AND } RV_{21}^1 = 1100100000 \text{ AND } 1110000000 = 1100000000.$$

The class vector is found as follows:

$$CV_{11}^2 = CV_{11}^1 \text{ AND } CV_{21}^1 = 110 \textit{ AND } 100 = 100.$$

In a similar way, all the feature-value vectors in the 2-feature matrix M^2 can be found. The results are shown in Table 5.

Note that in Step 2.2.1.2, the *class* vector derived by the bitwise "AND" operator denotes only the "possible" class distribution. For example, the feature-value vector F_{21}^2 consists of $RV_{21}^2 = 1000000000$ and $CV_{21}^2 = 110$ after Step 2.2.1.2. Since, each record belongs to only one class, the above results are not correct. In fact, the class vector $CV_{21}^2 = 100$. Step 2.2.1.2 is used as a quick check. If $CV_{jk}^{s+1} \neq UNIQUE_{\sigma_m}$, then the *FindClassVector algorithm is run in* Step 2.2.1.3 *to find the correct class vector*.

After the new feature matrix is derived, the *SelectingFeatureSet* algorithm is then executed again to find an appropriate feature set. For the above example, the 2-feature matrix M^2 is then input to the *SelectingFeatureSet algorithm* and the feature set $FS = \{C_2, C_4\}$ are found and returned as the solution.

After the above method is executed, the feature set *FS* to classify the given data set *T* is generated. *FS* may be over-fitting or under-fitting for the problem since they are derived only according to the current data set. These features are then evaluated and modified by domain experts. They thus serve as the candidates for the experts to have a good initial standpoint.

Table 5
The 2-feature matrix M^2 found by the *CalculatingNextMatrix* algorithm

Feature set	Feature set value	Name vector	Record vector	Class vector
C_1^2	V_{11}^2	1100	1100000000	100
	V_{12}^2	1100	0000100000	010
	V_{13}^2	1100	0010000000	100
	V_{14}^2	1100	0001010000	011
	V_{15}^2	1100	0000001000	001
	V_{16}^2	1100	0000000100	001
C_2^2	V_{21}^2	1010	1000000000	100
	V_{22}^2	1010	0100000000	100
	V_{23}^2	1010	0000100000	010
	V_{24}^2	1010	0001010000	011
	V_{25}^2	1010	0010000000	100
	V_{26}^2	1010	0000001100	001
C_3^2	V_{31}^2	1001	1000100000	110
	V_{32}^2	1001	0100000000	100
	V_{33}^2	1001	0011000000	110
	V_{34}^2	1001	0000010000	001
	V_{35}^2	1001	0000001100	001
C_4^2	V_{41}^2	0110	1000000000	100
	V_{42}^2	0110	0110000000	100
	V_{43}^2	0110	0001011000	011
	V_{44}^2	0110	0000100000	010
	V_{45}^2	0110	0000000100	001
C_5^2	V_{51}^2	0101	1010000000	100
	V_{52}^2	0101	0100000000	100
	V_{53}^2	0101	0001100000	010
	V_{54}^2	0101	0000011000	001
	V_{55}^2	0101	0000000100	001
C_6^2	V_{61}^2	0011	1001000000	110
	V_{62}^2	0011	0000011100	001
	V_{63}^2	0011	0010000000	100
	V_{64}^2	0011	0100000000	100
	V_{65}^2	0011	0000100000	010

4. Time and space complexity analysis

The time and space complexities of the proposed algorithms are analyzed in this section. Let n be the number of records, m be the number of features and c be the number of classes. Also define i as the maximum possible number of features in a feature set, j as the maximum number of possible values of a feature, and s as the number of iterations. The time complexity and space complexity of each step in the *FindClassVector* algorithm is shown in Table 6.

The time and space complexities of each step in the *CreateCleansingTree* algorithm is shown in Table 7. Note that the maximum amount of nodes within a *Ctree* is n.

The time and space complexities of each step in the *FindSpanOrder* algorithm is shown in Table 8.

The time and space complexities of each step in the *CleansingFeatureMatrix* algorithm is shown in Table 9.

Table 6
The time and space complexities of the *FindClassVector* algorithm

Step no.	Time complexity	Space complexity
Step 1	$O(1)$	$O(c)$
Step 2	$O(jc)$	$O(jc)$
Step 3	$O(1)$	$O(c)$
Total	$O(jc)$	$O(jc)$

Table 7
The time and space complexities of the *CreateCleansingTree* algorithm

Step no.	Time complexity	Space complexity
Step 1	$O(1)$	$O(1)$
Step 2	$O(1)$	$O(1)$
Step 3	$O(1)$	$O(1)$
Step 4	$O(nmj)$	$O(n)$
Step 5	$O(mj)$	$O(n)$
Step 6	$O(1)$	$O(1)$
Step 7	$O(1)$	$O(1)$
Total	$O(nmj)$	$O(n)^*$

Table 8
The time and space complexities of the *FindSpanOrder* algorithm

Step no.	Time complexity	Space complexity
Step 1	$O(m)$	$O(m)$
Step 2	$O(cm)$	$O(cm)$
Step 3	$O(c\lg c)$	$O(c)$
Step 4	$O(1)$	$O(1)$
Total	$O(Max(cm, clgc))$	$O(cm)$

Table 9
The time and space complexities of the *CleansingFeatureMatrix* algorithm

Step no.	Time complexity	Space complexity
Step 1	$O(mj)$	$O(mj)$
Step 2	$O(1)$	$O(1)$
Total	$O(mj)$	$O(mj)$

Table 10
The time and space complexities of the *SelectingFeatureSet* algorithm

Step no.	Time complexity	Space complexity
Step 1	$O(1)$	$O(1)$
Step 2	$O(m^s j^s)$	$O(1)$
Step 3	$O(1)$	$O(1)$
Step 4	$O(j^s)$	$O(1)$
Step 5	$O(1)$	$O(1)$
Step 6	$O(c)$	$O(c)$
Step 7	$O(1)$	$O(1)$
Total	$O(m^s j^s)$	$O(c)$

Table 11
The time and space complexities of the *CalculatingNextMatrix* algorithm

Step no.	Time complexity	Space complexity
Step 1	$O(m^s j^s)$	$O(m^s j^s)$
Step 2	$O(mj)$	$O(mj)$
Step 3	$O(1)$	$O(j)$
Total	$O(m^s j^s)$	$O(m^s j^s)$

The time and space complexities of each step in the *SelectingFeatureSet* algorithm is shown in Table 10.

The time and space complexities of each step in the *CalculatingNextMatrix* algorithm is shown in Table 11.

5. Experiments

To evaluate the performance of the proposed method, we compare it with other feature selection methods. Our target machine is a Pentium III 1G Mhz processor system, running on the Microsoft Windows 2000 multithreaded OS. The system includes 512 K L2 cache and 256 MB shared-memory.

Several datasets from the UCI Repository (Quinlan, 1986) are used for the experiments. These datasets have different characteristics. Some have known relevant features (such as Monks), some have many classes (such as SoybeanL), and some have many instances (such Mushroom). In addition, a large real data set about endowment insurances from a world-wide financial group is used to examine the usability of the proposed method. Experimental results show the proposed method can discover the desired feature sets and can thus help the enterprise to build a CBR system for their loan promotion function of customer relationship management system. The data set of insurance data uses 27 condition features to describe the states of 3 different insurance types. Different types of attribute values including date/time, numeric and symbolic data exist. They are all transformed into the symbolic type by some clustering methods. Six of them have missing values.

The characteristics of the above datasets are summarized in Table 12.

In the experiments, the accuracy, the number of selected features, and the time will be compared between our method and the traditional rough set method. The accuracy is measured by the classification results of the target table. If the selected feature set can solve the problem without any error, 100% accuracy is reached; otherwise the accuracy is calculated by the number of correctly classified records over the total number of records. Experimental results show both methods can reach 100% accuracy. We then compare the feature sets found by these two approaches. The results are shown in Table 13. Obviously, the accuracy of all datasets is 100% since both of these two methods discover the minimal feature sets.

Note that there may be more then one solution for the selected features. In Table 13, only the first selected feature set (in the alphabetical order) is listed. It is easily seen that the selected feature sets of our proposed approach and the traditional rough set approach are the same except for the SoybeanL problem. The SoybeanL problem needs too much computation time by the traditional rough set approach.

The numbers of the selected features by the two approaches are shown in Table 14. Both methods get the same numbers for all problems except for SoybeanL.

At last, the computation time is compared. The data sets are first loaded into the memory from the hard disk and the processing times are measured. The time is rounded to 0 if the real time is less than 0.001 seconds. The results are shown in Table 15.

Consistent with our expectation, the proposed approach is much faster than the traditional rough set approach. Especially for the Insurance data, our approach needs only

Table 12
The datasets used in the experiments

Database name	Class no.	Condition feature no.	Record no.	Missing features
Monk1	2	6	124	No
Monk2	2	6	169	No
Monk3	2	6	122	No
Vote	2	16	300	No
Mushroom	2	22	8124	Yes
SoybeanL	19	35	683	Yes
Insurance	3	27	35000	Yes

Table 13
The selected feature sets found by the two approaches

Dataset	Feature set		Accuracy (%)
	Traditional rough set approach	Bitmap-based approach	
Monk1	C1, C2, C5	C1, C2, C5	100
Monk2	C1–C6	C1–C6	100
Monk3	C1, C2, C4, C5	C1, C2, C4, C5	100
Vote	C1–C4, C9, C11, C13, C16	C1–C4, C9, C11, C13, C16	100
Mushroom	C3, C4, C11, C20	C3, C4, C11, C20	100
SoybeanL	Need too much computation time	C14, C20, C26, C27, C29, C30, C31, C32, C33, C34, C35	100
Insurance	C4, C15, C17, C20, C22, C25	C4, C15, C17, C20, C22, C25	100

Table 14
The number of the selected features found by the two approaches

Dataset	Traditional RS	Bitmap-based
Monk1	3	3
Monk2	6	6
Monk3	4	4
Vote	8	8
Mushroom	4	4
SoybeanL	11	11
Insurance	6	6

Table 15
The CPU times needed by the two approaches

Dataset	Traditional RS	Bitmap-based
Monk1	0.07	0
Monk2	0.351	0.01
Monk3	0.141	0
Vote	428.19	1.923
Mushroom	4911.32	27.91
SoybeanL	>1000000	247805
Insurance	468656	2435.66

about 40 min, but the traditional rough set approach needs much more computation time.

6. Conclusion and future work

In this paper, we have proposed a bit-based feature selection approach to discover optimal feature sets for the given table(dataset). In this approach, the feature values are first encoded into bitmap indices for searching the optimal solutions efficiently. Also, the corresponding indexing and selecting algorithms are described in details for implementing the proposed approach. Experimental results on different data sets have also shown the efficiency and accuracy of the proposed approach.

The traditional rough set approach has two very time-consuming parts, combination of features and comparison of upper/lower approximations. In this paper, we use the single-time-clock bitwise operations to shorten the computation time of the comparison part. Moreover, the workload in the combination part is highly reduced since the new levels of combination can be generated via the pervious ones. The bitwise operations are also used to speed up the combination generation. The proposed feature selection approach also adopts appropriate meta-data structures to take advantages of the computational power of the bitwise operations.

The feature selection problem is generally an NP-complete problem. Although, the proposed approach can process a larger amount of features than the traditional rough set approach, it still becomes unmanageable especially when the number of features is huge or when the number of possible values of features is large. In the future, we will continuously investigate and design efficient heuristic approaches to manage huge amounts of features and possible values. We will also attempt to integrate different feature selection approaches to automatically select an appropriate one for optimal or near-optimal solutions according to the characteristics of given data sets.

References

Almullim, H. et al. (1991). Learning with many irrelevant features. In *Proceedings of ninth national conference on artificial intelligent*, pp. 547–552.

Brassard, G. et al. (1996). Fundamentals of Algorithm. New Jersey: Prentice Hall.

Chen, W. C., Tseng, S. S., Chen, J. H., & Jiang, M. F. (2000). A framework of feature selection for the case-based reasoning. In *Proceeding of IEEE international conference on systems, man, and cybernetics*, CD-ROM.

Chen, W. C., Tseng, S. S., Chang, L. P., & Jiang, M. F. (2001). A similarity indexing Method for the data warehousing – bit-wise indexing method. In *Lecture notes in artificial intelligent* (Vol. 2035), pp. 525–537.

Chen, W. C., Tseng, S. S., Chang, L. P., & Hong, T. P. (2002). A parallelized indexing method for large-scale case-based reasoning. *Expert System with Applications, 23*(2), 95–102.

Chen, W. C., Yang, M. C., & Tseng, S. S. (2002). A high-speed feature selection method for large dimensional data set. In *Proceeding of international computer symposium*, CD-ROM.

Chen, W. C., Yang, M. C., & Tseng, S. S. (2003). The bitmap-based feature selection method. In *18th ACM symposium on applied computing (SAC), data mining track, CD-ROM*.

Choubey, S. K. et al. (1998). On feature selection and effective classifiers. *Journal of ASIS, 49*(5), 423–434.

Doak, J. (1992). An evaluation of feature selection methods and their application to computer security, Technical Report, University of California.

Gonzalez, A., & Perez, R. (2001). Selection of relevant features in a fuzzy genetic learning. *IEEE Transaction on SMC-Part B, 31*(3), 417–425.

Huang, C. C., & Tseng, B. (2004). Rough set approach to case-based reasoning. *Expert Systems with Applications, 26*(3), 369–385. April.

John, G. H. et al. (1994). Irrelevant feature and the subset selection problem. In *Proceedings of 11th international conference on machine learning*, pp. 121–129.

Last, M., Kandel, A., & Maimon, O. (2001). Information theoretic algorithm for feature selection. *Pattern Recognition Letter, 22*, 799–811.

Lee, H. M., Chen, C. M., Chen, J. M., & Jou, Y. L. (1997). An efficient fuzzy classifier with feature selection based on fuzzy entropy. *IEEE Transaction on SMC-Part B, 27*(2), 426–432.

Liu, H. et al. (1996). A probabilistic approach to feature selection – a filter solution. In *Proceedings of 13th international conference on machine learning*, pp. 319–327.

Liu, H., & Setiono, R. (1998). Incremental feature selection. *Applied Intelligence, 9*, 217–230.

Miao, D. Q. et al. (1999). A heuristic algorithm of reduction for knowledge. *Journal of Computer Research and Development, 36*(6), 681–684.

O'Neil, P., & Quass, D. (1997). Improved query performance with variant indexes. *ACM SIGMOD Record, 26*(2), 38–49.

Pawlak, Z. (1982). Rough set. *International Journal of Computer and Information Sciences*, 341–356.

Pawlak, Z. (1991). *Rough Sets. Theoretical Aspects of Reasoning about Data*. Boston: Kluwer Academic Publishers.

Quinlan, J. (1986). Introduction of decision trees. *Machine Learning, 1*(1), 81–106.

Raymer, M. L., Punch, W. F., Goodman, E. D., & Kuhn, L. A. (2000). Dimensionality reduction using genetic algorithm. *IEEE Transaction on Evolutionary Computation, 4*(2), 164–171.

Schlimmer, J. C. et al. (1993). Efficiently inducing determinations: A complete and systematic search algorithm that uses optimal pruning. In *Proceedings of 10th international conference on machine learning*, pp. 284–290.

Skowron, A., & Rauszer, C. (1992). The discernibility matrices and functions in information systems. *Intelligent Decision Support*, 331–362.

Tseng, B., Jothishankar, M. C., & Wu, T. (2004). Quality control problem in printed circuit board manufacturing – An extended rough set theory approach. *Journal of Manufacturing Systems, 23*(1), 56–72.

Wu, F. B. et al. (1999). *Control and Decision, 14*(3), 206–211.

Wu, M. C., & Alejandro, P. B. (1998). Encoded bitmap indexing for data warehouses. In *Proceedings of IEEE data engineering*, pp. 220–230.

Yang, Y., & Chiam, T. C. (2000). Rule discovery based on rough set theory. In *Proceedings of the third international conference on FUSION* (Vol. 1), pp. TuC4_11–TuC4_16.

Yu, H. et al. (2001). Rough set based knowledge reduction algorithms. *Computer Science, 28*(5), 31–34.

Zhong, N., Dong, J., & Ohsuga, S. (2001). Using rough sets with heuristics for feature selection. *Journal of Intelligent Systems, 16*, 199–214.