Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)

Dancing with Decision Diagrams: A Combined Approach to Exact Cover

Masaaki Nishino,[1] **Norihito Yasuda,**[1] **Shin-ichi Minato,**[2] **Masaaki Nagata**[1]
[1]NTT Communication Science Laboratories, NTT Corporation
[2]Graduate School of Information Science and Technology, Hokkaido University
nishino.masaaki@lab.ntt.co.jp

Abstract

Exact cover is the problem of finding subfamilies, S^*, of a family of sets, $\mathcal{S}$, over universe U, where S^* forms a partition of U. It is a popular NP-hard problem appearing in a wide range of computer science studies. Knuth's algorithm DLX, a backtracking-based depth-first search implemented with the data structure called dancing links, is known as state-of-the-art for finding all exact covers. We propose a method to accelerate DLX. Our method constructs a Zero-suppressed Binary Decision Diagram (ZDD) that represents the set of solutions while running depth-first search in DLX. Constructing ZDDs enables the efficient use of memo cache to speed up the search. Moreover, our method has a virtue that it outputs ZDDs; we can perform several useful operations with them. Experiments confirm that the proposed method is up to several orders of magnitude faster than DLX.

Introduction

The exact cover problem is the problem of finding subfamily $\mathcal{S}^*$ of a family of sets, $\mathcal{S}$, over universe U, that forms a partition of U. The exact cover problem is a popular NP-hard problem that appears in a wide range of computer science studies. For example, it is known that some puzzles including pentominoes, n-queen problems (Knuth 2000), and Sudoku (Gunther and Moon 2012) can be formulated and solved as exact cover problems or slightly generalized forms. Moreover the perfect matching problem (Korte and Vygen 2012) is a special case of exact cover, and the graph coloring problem can be solved by converting it into an exact cover problem (Koivisto 2006).

Knuth's algorithm X (Knuth 2000) is a dedicated algorithm that can find all exact covers. It is a simple depth-first backtracking-based search algorithm, but it runs efficiently when implemented with a data structure called *dancing links*. Dancing links represent a binary matrix as doubly-linked lists. Knuth calls algorithm X implemented with dancing links algorithm DLX. DLX has been empirically confirmed to be the fastest for solving the problem of finding all exact covers (Junttila and Kaski 2010).

Finding all exact covers are beneficial for several reasons: we can find exact covers which allows maximization of an objective function even if the objective is complex, e.g., non-linear or multiple objectives. The exact cover problem appears in designing combinatorial objects like electric circuits (Chang and Jiang 2016) and 3D shapes (Hu et al. 2014), that made from small fragments. In these situations, it is beneficial to output many candidates and allow designers to interactively select from them.

Since DLX is a depth-first-search-based method, it may encounter the same sub-problems several times. We can accelerate DLX if we can avoid solving such duplicate sub-problems. A naive approach is to use memo cache to store the solutions of sub-problems. However, since the number of solutions to sub-problems can be exponentially many, storing them into memo cache requires a huge amount of memory. Moreover, we have to combine solutions of sub-problems to construct the solution of the main-problem. This procedure incurs computation complexity proportional to the size of the solution. These points make it inefficient to naively introduce memo cache into DLX.

In this paper, we enable the efficient use of memo caching on DLX by using *Zero-suppressed Binary Decision Diagrams* (ZDDs) (Minato 1993) to represent the set of exact covers. ZDDs, a variant of Binary Decision Diagrams (BDDs) (Akers 1978; Bryant 1986), represent a family of sets as a directed acyclic graph (DAG). We modify DLX to output the ZDD representing the set of exact covers. Since we can make ZDDs that contain solutions to sub-problems, sub-ZDDs (subgraphs), memo cache only requires constant memory for each problem by storing the address of the root node of the sub-ZDD representing the set of solutions. Moreover, combining solutions of sub-problems can be done in constant time if they are represented as ZDDs. Thus using ZDD with DLX enables the effective use of memo caching and can accelerate the search. To our knowledge, this is the first work to improve DLX. We conduct experiments with a wide range of benchmark instances of exact cover problems, and confirm that our proposal is up to several orders of magnitude faster than DLX.

In addition to accelerating DLX, our proposal has the virtue that it outputs ZDDs. ZDDs support useful operations that can run in time proportional to ZDD size. For example, ZDDs support model counting, enumeration of subsets, finding the best subset that maximizes an objective function (Darwiche and Marquis 2002; Knuth 2011), and efficient binary operations over families of sets, called *family*

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Figure 1: An instance of an exact cover problem represented as a binary matrix.

algebra (Knuth 2011; Minato 1994). These operations make it easy to manipulate a set of exact covers. For example, using family algebra makes it easy to find exact covers that satisfy additional constraints.

Exact Cover Problem and Algorithm DLX

Given universe $U = \{1, \ldots, M\}$ and family of sets $\mathcal{S}$ over U, the exact cover problem is finding $\mathcal{S}^* \subseteq \mathcal{S}$ that is a partition of U, i.e., every $i \in U$ is contained in exactly one subset in $\mathcal{S}^*$. An instance of an exact cover problem can be represented as a 0-1 binary $N \times M$ matrix, where M is the cardinality of universe U, and N is the number of subsets contained in $\mathcal{S}$. Every column corresponds to an element in the universe, and every row corresponds to a set contained in $\mathcal{S}$. With this representation, the exact cover problem corresponds to finding a set of rows that contains exactly one 1 for each column. Fig. 1 is an example of a binary matrix that represents an exact cover problem, where $N = 5$ and $M = 6$. In this example, the set of rows $\{1, 3\}$ and $\{2, 3, 5\}$ forms an exact cover.

Knuth's algorithm X (Knuth 2000) finds all exact covers by performing depth-first backtracking-based search. We detail the procedure in Alg.1. The main procedure of the algorithm is $\mathsf{Search}(A, R)$ which recursively searches for exact covers of input binary matrix X. Procedure $\mathsf{Search}(A, R)$ first checks whether A is an empty matrix or not. If A is empty, the algorithm determines that we have found a new exact cover, and so outputs solution R and returns (line 2-4). Otherwise, it first selects column c by using some criterion (line 5)[1], and then row r such that $A[r, c] = 1$. If r is selected, we include r into R (line 7), and then for every column, j, satisfying $A[r, j] = 1$, delete every row, i, satisfying $A[i, j] = 1$ and column j from A to make a submatrix of A (line 8-11). We recursively apply $\mathsf{Search}(A, R)$ to the obtained submatrix. After subprocedure $\mathsf{Search}(A, R)$ finishes, it recovers the rows and columns deleted after row r was selected (line 14). It then repeats the same procedure for all different r such that $A[r, c] = 1$, which completes the procedure.

Example 1. Suppose that the matrix shown in Fig. 1 is given as the input of algorithm X. It first calls $\mathsf{Search}(X, \emptyset)$. Suppose that $c = 1$ is selected at line 5. Then, the rows satisfying $A[r, 1] = 1$ are $r = 1, 2$. We process the loop from line 6-14 with every r. If we select $r = 2$, then columns 1, 2 are deleted since $A[2, 1] = 1$ and $A[2, 2] = 1$. After that, rows that have 1 in a deleted column are also deleted. In this case, rows $1, 2$ are deleted. The resulting submatrix obtained by deleting rows and columns is

$$\begin{array}{c} \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccc} 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \end{array}. \tag{1}$$

We recursively call $\mathsf{Search}(A, R)$ by setting the submatrix as its argument. Suppose that column $c = 5$ is selected. Then there is only one row $r = 5$ that satisfies $A[r, 5] = 1$. Thus columns $3, 5$ and rows $4, 5$ are deleted, which yields submatrix

$$\begin{array}{c} \\ 3 \end{array} \begin{array}{c} \begin{array}{cc} 4 & 6 \end{array} \\ \begin{pmatrix} 1 & 1 \end{pmatrix} \end{array}$$

Next, the only choice, selecting $r = 3$, makes the submatrix empty, thus selected rows $\{2, 5, 3\}$ form an exact cover. After a solution has been found, the algorithm continues to search for the next cover by backtracking to recover deleted rows and columns. If we select $c = 4$ and $r = 4$ on Search whose input is (1), the resulting matrix is

$$\begin{array}{c} 5 \\ \begin{pmatrix} \ \end{pmatrix} \end{array}.$$

Since this matrix has an undeleted column but no row remains, we cannot delete the column and the search fails.

The running time of algorithm X strongly depends on how fast it can move forward (i.e., selecting a row and deleting rows and columns) and backward (i.e., recovering deleted rows and columns). The data structure called dancing links is based on doubly-linked lists. It offers high efficiency on removing and restoring rows and columns of matrices through the following two operations. Let $L[x]$ and $R[x]$ point to the predecessor and successor of element x in a doubly-linked list. Then the operations

$$L[R[x]] \leftarrow L[x], \quad R[L[x]] \leftarrow R[x]$$

Algorithm 1: Knuth's algorithm DLX.

```
   Input: Binary Matrix X
1  function Search(A, R):
2      if A is empty then
3          Output R
4          return
5      Select a column, c
6      for r such that A[r, c] = 1 do
7          Include r into R
8          for j such that A[r, j] = 1 do
9              Delete column j from A
10             for i such that A[i, j] = 1 do
11                 Delete row i from A
12         Search(A, R)
13         Delete r from R
14         Recover rows & columns deleted in lines 9, 11
15     return
16 Search(X, ∅)
```

[1] Knuth recommends the simple heuristic of selecting the column with minimum number of 1s.

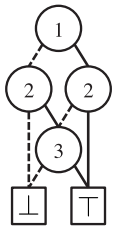

Figure 2: A ZDD representing $\{\{1,2\},\{1,3\},\{2,3\}\}$
.

remove x from the list. Suppose that we do not change values $L[x]$ and $R[x]$ after x is removed, then the following operations

$$L[R[x]] \leftarrow x, \quad R[L[x]] \leftarrow x$$

puts x back into the list. Algorithm X implemented with dancing links, called *DLX*, represents an input binary matrix by using dancing links, and it extensively uses the above two operations to efficiently remove and restore rows and columns of matrices.

Zero-suppressed Binary Decision Diagrams

The data structure Zero-suppressed Binary Decision Diagram (ZDD) represents a family of sets as a directed acyclic graph (DAG). Fig. 2 is an example ZDD that represents the family of sets $\{\{1,2\},\{1,3\},\{2,3\}\}$ over universe $\{1,2,3\}$. ZDDs have two types of nodes: terminal nodes and branch nodes. A terminal node has no outgoing edges. A ZDD has exactly two terminal nodes that have labels $\top$ and $\bot$. Terminal nodes are represented as rectangles in the figure. Branch nodes are non-terminal nodes, and are represented as circles in the figure. Every branch node has a label representing the element the node corresponds to, and two outgoing edges, called lo-edge and hi-edge. Child ZDD nodes that are pointed to by lo- and hi- edges are called lo-child and hi-child, respectively. Every branch node is represented by a tuple (i,l,h), where i is the label, l is the lo-child, and h is the hi-child of the node. In the figure, the label of a branch node is represented as a symbol appearing in the circle, and hi and lo edges are represented by solid lines and dashed lines, respectively. A branch node that has no ancestor node is the root node, and a ZDD always has exactly one root node. Every path from the root to $\top$ terminal node corresponds to subset S contained in the family of sets that the ZDD represents, where S is made from the path created by selecting the labels of the branch nodes whose hi-edges lie on the path. The ZDD in Fig. 2 has three such paths:$1 \rightarrow 2 \rightarrow \top$, $1 \rightarrow 2 \dashrightarrow 3 \rightarrow \top$, and $1 \dashrightarrow 2 \rightarrow 3 \rightarrow \top$. They corresponds to subsets $\{1,2\}$, $\{1,3\}$, and $\{2,3\}$.

A ZDD is *ordered* if labels of visited branch nodes in a path from the root to a terminal node always follow an order over the elements. The ZDD in Figure 2 is an ordered ZDD since every path from the root to a terminal node follows the order $1,2,3$. For ordered ZDDs (OZDDs), there are reduction rules that can remove redundant substructures from an OZDD to reduce its size. We say an OZDD is *reduced* if no further application of the reduction rule is successful. It is known that a reduced OZDD is canonical, i.e., if there are two reduced OZDDs α and β that follow the same ordering and represent set families f and g, then α and β are isomorphic if and only if $f = g$. The canonicity of reduced OZDDs is quite useful since it enables constant time equivalency checking, and efficient family algebra operations. We call non-ordered ZDDs Free ZDDs (FZDDs). Both OZDDs and FZDDs support several useful operations that run in polynomial time. Operations supported by ZDDs are found in (Knuth 2011) and (Darwiche and Marquis 2002) [2].

OZDDs are closely related to OBDDs. The main difference is the reduction rule called the zero-suppression rule, that deletes an OZDD node whose hi-edge is connected to $\bot$. Due to this rule, OZDDs tend to become more compact than OBDDs when representing a family of sets that consists of small numbers of subsets.

Algorithm 2: Algorithm DXZ.

```
   Input: Binary Matrix X
1  C ← empty memo cache
2  Z ← empty ZDD node table
3  function Search(A):
4      if A is empty then
5          return ⊤
6      if col(A) found in C then return C[col(A)]
7      Select a column, c
8      x ← ⊥
9      for r such that A[r, c] = 1 do
10         for j such that A[r, j] = 1 do
11             Delete column j from A
12             for i such that A[i, j] = 1 do
13                 Delete row i from A
14         y ← Search(A)
15         if y ≠ ⊥ then
16             x ← Unique(r, x, y, Z)
17         Recover rows & columns deleted in lines 11, 13
18     Store pair (col(A), x)) into C. return x
19 z ← Search(X)
20 return root ZDD node z
```

Algorithm DXZ

We show our proposed algorithm, called *DXZ (DLX with ZDD)* in Alg. 2. The algorithm is only slightly different from DLX. The main difference is the existence of some additional operations for constructing ZDDs (lines 5, 8, 14, 16) and for referring to memo caches (lines 6, 18). We first show how ZDDs are constructed in DXZ.

Similar to DLX, DXZ performs depth-first recursive backtracking search. The main difference is that $\mathsf{Search}(A)$ returns the ZDD representing the set of all exact covers of problem A. DXZ uses the following recursive relationships of exact covers. Let $\mathsf{Cover}(A)$ be the family of sets

[2]Precisely, operations supported by BDDs are shown in (Darwiche and Marquis 2002). However, operations supported by BDDs in polytime are also supported by ZDDs in polytime

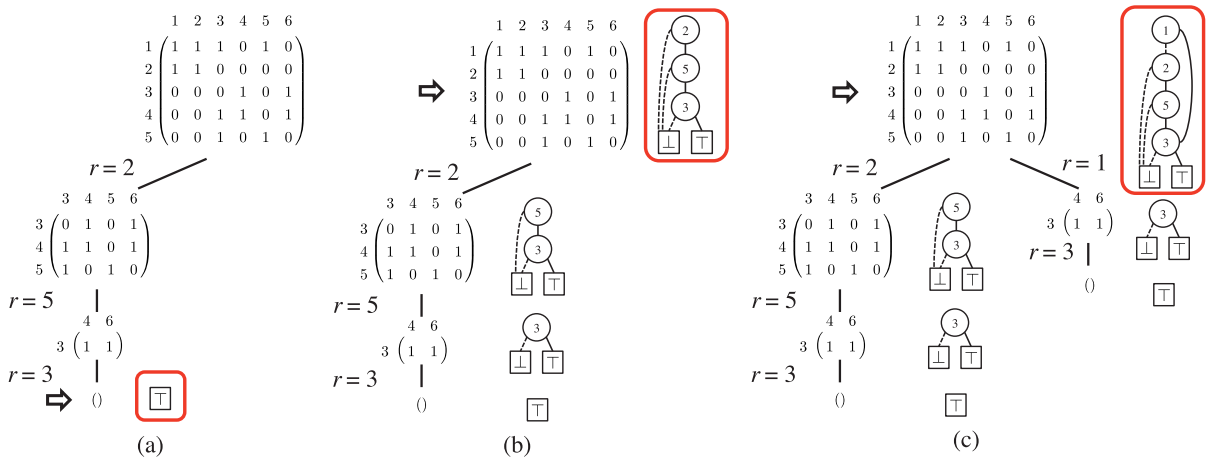

Figure 3: An example of the running procedure of DXZ. Arrows represent current state, and ZDDs in red rectangles represent the latest ZDDs constructed by DXZ.

representing the set of all exact covers of matrix A. Let $i_1, i_2, \ldots, i_n$ be the rows that have a non-zero element in column c, and let $A_1, A_2, \ldots A_n$ be the submatrices obtained by selecting rows $i_1, i_2, \ldots, i_n$ and deleting all corresponding rows and columns from A. By using $\mathsf{Cover}(A_j)$, $\mathsf{Cover}(A)$ is recursively defined as

$$\mathsf{Cover}(A) = \bigcup_{j=1}^{n} (\{\{i_j\}\} \sqcup \mathsf{Cover}(A_j)) \tag{2}$$

where $\sqcup$ represents the join of set families and is defined as $f \sqcup g = \{a \cup b \mid a \in f \text{ and } b \in g\}$.

DXZ uses the above recursion to combine sub-ZDDs representing $\mathsf{Cover}(A_i)$. At line 8, it introduces ZDD $x = \bot$ to represent the set of all exact covers of input matrix A. The algorithm updates x by $x \leftarrow \mathsf{Unique}(r, x, y, Z)$, where $\mathsf{Unique}(i, l, h, Z)$ returns the address of the ZDD node that has label i and the addresses of its lo and hi child nodes are l and h, respectively. It checks whether there already exists a node in Z that has the same triple (i, l, h) . If such node is found, it returns the address of the node. Otherwise it creates a new ZDD node (i, l, h), inserts it into Z, and returns the address of the node. ZDD node (i, l, h) represents the family of sets

$$\langle l \rangle \cup (\{\{i\}\} \sqcup \langle h \rangle) \ ,$$

where $\langle z \rangle$ is the family of sets that ZDD node z represents. Since $\langle y \rangle$ (line 14) corresponds to $\mathsf{Cover}(A_j)$ for some i_j, updating $x \leftarrow \mathsf{Unique}(r, x, y, Z)$ with some r satisfying $A[r, c] = 1$ corresponds to updating x to ZDD node representing $\langle x \rangle \cup \{\{i_j\}\} \cup \mathsf{Cover}(A_j)$. Therefore, repeating this procedure for all r satisfying $A[r, c] = 1$ corresponds to constructing a ZDD that represents $\mathsf{Cover}(A)$ defined in (2).

We show some theoretical results.

Proposition 1. *The number of ZDD nodes stored in table Z does not exceed the number of ZDD nodes required to represent the set of exact covers of the input matrix.*

Proof. Every sub-ZDD made in the child procedures always appears in the ZDD made by the parent procedure. It means that there are no branch nodes that appear in sub-ZDDs but do not appear in parent ZDDs. Therefore, the number of ZDD nodes stored in the table is bounded by the size of the finally obtained ZDD. □

Proposition 2. *The running time overhead of DXZ is always constant.*

Proof. Proving this fact is easy since it only involves computing Unique for every row selection in Search. Since we can implement the procedure by using a hash table whose size is at most equal to the size of the ZDD representing the set of all exact covers, Unique can be performed in constant time. □

We also note that the ZDD generated by DXZ is a FZDD. However, if we use a consistent rule on selecting column c (Alg. 2 line 7), we can make the algorithm generate an OZDD. We show details in a later section.

Example 2. Here we show an example of how DXZ constructs ZDDs. Fig. 3 shows the procedure of constructing the ZDD representing the set of all exact covers of the input matrix of Fig. 1. In this figure, the depth-first search procedure of DXZ is represented by a search tree, where every matrix at a node represents the input matrix A of $\mathsf{search}(A)$, and every child node represents a sub-matrix of A which is used as the input of the recursive call. ZDDs constructed in search procedure appear at the right of each search tree node.

We first select $c = 1$, $r = 2$ to obtain the subproblem represented as the left child of the root node. Then we select $c = 5$, $r = 5$, and then $r = 3$ to reach an empty matrix. It means $\{2, 3, 5\}$ forms an exact cover. When finding an empty matrix, the procedure search returns terminal ZDD $\top$ (Fig.3 (a)). In the backtracking procedure, the returned ZDD is then combined with branch ZDD nodes corresponding to selected rows and the ZDD representing $\{\{2, 3, 5\}\}$ is

made at the top level (Fig.3 (b)). Then, $c = 1$, $r = 1$ and $c = 1, r = 3$ are selected and we find another exact cover $\{\{1, 3\}\}$. The ZDD representing $\{1, 3\}$ is recursively constructed in the backtracking, and finally combined with the ZDD representing $\{\{2, 3, 5\}\}$, and the solution ZDD representing $\{\{1, 3\}, \{2, 3, 5\}\}$ is obtained (Fig.3 (c)). Combining two ZDDs representing $\{\{1, 3\}\}$ with $\{\{2, 3, 5\}\}$ is easily performed by adding the root node in Alg.2 line 16.

Memo Caching

In the example of Fig. 3 (c), the same subproblem whose input is $(1\ \ 1)$ appears in both left and right subtrees. If we store the solution of the problem, we do not need to solve the same problem many times. We use memo cache to store the address of ZDD nodes returned by the recursive Search procedures.

If submatrices B equal B', then $\mathsf{Cover}(B) = \mathsf{Cover}(B')$. Therefore, the naive memo caching technique of using submatrix as a key to memorize/access its exact covers can reduce the amount of computation. Actually, we do not need to store the whole matrix as a key as it is sufficient to use the set of column IDs of the matrices as the key. We use the following fact.

Proposition 3. *Suppose that an exact cover problem is represented as binary matrix A. Let B, B' be binary matrices obtained by running DXZ and selecting some rows from A and deleting corresponding columns and rows. Then, if the set of remaining columns of B and B' are equivalent, then $B = B'$.*

Proof. After selecting row r, DXZ deletes every column, j, that satisfies $A[r, j] = 1$, and every row, i, that satisfies $A[i, j] = 1$. It means deleted rows are those that have at least one 1 in a deleted column. Therefore, after selecting some rows, the remaining submatrices have rows that have no 1 in any of the deleted columns. Therefore, the set of deleted columns uniquely determines the set of remaining rows, and the submatrices are identical. □

This fact suggests that the set of remaining columns are enough to judge the equivalence of submatrices. We thus use an M-bit sequence, which represents the set of remaining column IDs, as the key to the memo cache.

Memo caching is used in lines 6 and 18 in Alg. 2. C is memo cache, and it maps an M-bit vector to a ZDD node pointer. In line 6, it checks if the current $\mathsf{Search}(A)$ was previously solved. Here $\mathsf{col}(A)$ represents the set of column IDs. If $\mathsf{col}(A)$ is found in C, it returns the associated ZDD $C[\mathsf{col}(A)]$. At line 18, the algorithm stores solution ZDD x into memo cache by associating it with $\mathsf{col}(A)$.

Note that the memo caching technique is independent from the ZDD-based representation of exact covers, and we can use memo cache with the original DLX. However, ZDD-based representation is suitable for memo caching because a ZDD can be represented as a pointer to a branch node stored in table Z. Therefore, every cache entry is represented by a pair of M-bit array and a branch node pointer, and the memory overhead of using memo cache is small.

Fixed Ordering and OZDD

In general, the ZDD obtained by DXZ is not ordered. This is because we dynamically select column c by using some criterion that reflects the values of input matrix A. We can modify DXZ to construct reduced OZDDs. Modifications for constructing ordered ZDDs are simple; we alter the way of selecting column c and the order of examining row r such that $A[r, c] = 1$. Specifically, we select the leftmost column c, and select rows r such that $A[r, c] = 1$ in descending order of r. The resulting ZDD is an ordered and reduced ZDD.

Proposition 4. *ZDDs constructed by DXZ with the above criteria of selecting c and r are ordered and reduced, where the order of rows follows descending order of pairs (c_r, r), where c_r is the smallest column, c, that $X[r, c] = 1$ for given input matrix X.*

Proof. We first show that output ZDDs are ordered. Since we employ fixed ordering in selecting left-most columns in DXZ, if column c is selected, then current submatrix does not have columns $1, \ldots, c-1$ nor rows, r, for which $c_r < c$. It means the ZDD representing exact covers of the current submatrix does not contain nodes whose label r satisfies $c_r < c$. Thus ZDD nodes whose label is r never appear before nodes with label r' such that $c'_r < c_r$. If there are two rows $r < r'$ such that $c_r = c_{r'}$, then, DXZ always selects r' before r at line 9 of Alg. 2. This means the ZDD node whose label is r' is never an ancestor of the node with label r in the updating procedure of x. Therefore, constructed ZDDs follow the descending order of pairs (c_r, r).

Next we show that the OZDDs output by DXZ are reduced. Reduced OZDDs satisfy the following two conditions: (1) do not have branch nodes whose hi-child is $\perp$, and (2) do not have branch nodes whose label, lo- and hi- child nodes are equivalent. The OZDDs made by the above ordering satisfies (1) because the algorithm does not create ZDD nodes whose hi-child is $\perp$, and it also satisfies (2) because $\mathsf{Unique}(i, l, h)$ checks that no duplicated branch nodes are created. Therefore, the constructed OZDDs are reduced. □

Reduced OZDDs are strict subsets of FZDDs, and support more polytime operations than FZDDs (Darwiche and Marquis 2002). One important operation specific to OZDDs is family algebra, which combines two OZDDs to make a new OZDD. Using family algebra makes it easy to find exact covers that satisfy one or more external constraints, or combining exact covers of different problems.

Experiments

Dataset

We use three different datasets: polyominoes, exact cover benchmarks, and set partitioning benchmarks. A polyomino is a plane geometric figure formed by joining one or more equal squares edge to edge, and polyomino tiling is the problem of locating sets of polyominoes on a given field made of squares. This problem can be seen as an exact cover problem (Knuth 2000). We use tiling problems of dominoes (polyominoes with two cells) with 8×8 and 10×10 fields and tetrominoes (four cells) with 6×6 and 8×8 fields as

Instance	# col	# row	Computation time (seconds) D	DLX	Z	DXZ	F	c2d	SS	# covers	DXZ statistics |ZDD|	Memory
domino_8_8	64	112	1.23	5.21	1.74	**0.001**	**0.001**	1.26	0.08	12988816	2316	32.4
domino_10_10	100	180	24282	> 600	> 600	**0.003**	**0.003**	109.45	6.94	258584046368	13550	33.4
tetromino_6_6	36	381	0.15	0.19	0.17	0.023	**0.013**	> 600	155.74	178939	25819	34.7
tetromino_8_8	64	800	16475	> 600	> 600	42.62	**0.88**	> 600	> 600	19077209438	10387777	1187.3
bell-09	9	511	0.002	0.006	0.003	**0.002**	**0.002**	39.38	1.20	21147	3356	32.5
bell-10	10	1023	0.017	0.034	0.023	0.012	**0.010**	440.23	19.70	115975	10353	33.5
doublefact-17	18	153	2.00	10.19	2.95	**0.007**	0.008	> 600	> 600	34459425	30510	34.9
doublefact-19	20	190	37.89	194.43	60.68	**0.022**	**0.022**	> 600	> 600	654729075	89665	41.5
kts15-ptpt-0020	132	391	**0.003**	**0.003**	**0.003**	**0.003**	0.73	5.04	0.15	8	171	32.2
kts15-ptpt-0021	132	388	**0.002**	0.003	0.003	0.003	0.83	14.06	0.46	72	1139	32.3
latin7-blk-00008	90	125	**0.017**	0.028	0.042	0.039	0.22	47.77	2.14	27072	122203	43.5
latin7-blk-00009	90	125	**0.018**	0.028	0.040	0.037	0.15	45.67	2.37	27776	118214	43.3
delta	126	1142	0.35	0.32	0.34	**0.31**	> 600	> 600	43.08	27	299	32.7
exotic_fives	72	2440	96.50	97.30	97.38	**92.87**	> 600	> 600	> 600	31520	184656	52.1
heart	180	900	8.58	8.91	8.62	**7.70**	> 600	> 600	> 600	3082	28898	35.3
meteor	60	1690	0.40	0.40	0.39	**0.34**	> 600	> 600	135.61	300	1569	32.7
sppnw07	36	3108	285.27	> 600	371.66	**9.96**	31.47	> 600	> 600	1993195009	2037119	221.1
sppnw08	24	356	0.038	0.080	0.049	**0.008**	0.015	57.19	1.90	168255	10426	33.3
sppnw09	40	2305	1469.18	> 600	> 600	**19.18**	67.76	> 600	> 600	8785116944	5308978	616.4
sppnw10	24	659	0.13	0.18	0.16	**0.034**	0.035	481.73	15.01	193640	28735	35.0
sppnw12	27	454	48.48	107.59	60.61	**0.008**	0.39	33.14	0.61	226514057	3997	32.6
sppnw15	31	465	0.004	0.004	0.004	**0.002**	0.024	15.71	0.70	40	58	32.2
sppnw19	40	2145	393.56	> 600	485.27	**3.01**	18.96	> 600	> 600	2128793883	540714	105.1
sppnw20	22	566	0.061	0.103	0.069	**0.018**	0.023	> 600	22.00	216884	15890	33.8
sppnw21	25	426	0.031	0.052	0.036	**0.010**	0.012	409.13	13.82	110458	11350	33.3
sppnw22	23	531	0.039	0.056	0.047	**0.017**	0.020	> 600	23.87	88884	20446	34.5

Table 1: Experimental results on benchmark problems. D: DLX that does not store any solution, DLX: DLX that stores every exact cover as an array, Z: DXZ without memo cache, DXZ: DXZ, F: DXZ with fixed ordering, c2d: c2d compiler, and sharpSAT: sharpSAT solver. # covers: number of exact covers, $|\mathrm{ZDD}|$: number of ZDD nodes constructed by DXZ, and Memory: maximum amount of memory used by DXZ (MByte). The best results are shown in bold fonts.

benchmark problems. Exact cover benchmark dataset used in (Junttila and Kaski 2010)[3] contains several artificial exact cover problems. The dataset contains 217 instances of 6 categories. Since similar results were obtained for instances of the same category, we show only the results for typical instances of each category. We also used the set partitioning problem benchmark instances available at OR-LIBRARY[4]. It contains several real-world set partitioning problem instances. We omit instances for which none of the methods finished within 600 seconds and instances for which both DXZ and DLX finished within 1 milliseconds.

Methods and Settings

We compared DXZ with DLX, as well as c2d and SharpSAT. c2d (Darwiche 2004) is a system that compiles a CNF into a d-DNNF (deterministic, decomposable negation normal form). An exact cover problem can be formulated and solved as the problem of enumerating all models of a CNF, and the problem can be solved by compiling a CNF into d-DNNF. SharpSAT (Thurley 2006) is a solver of #SAT problems. Comparison was also made to SharpSAT since DXZ can be used to counting the numbers of solutions. There are several CNFs that represent the same exact cover problem. We employed the ladder CNF encoding used in (Junttila and Kaski 2010). We also compared two variants of DXZ. The first one does not use memo cache and the second one builds OZDDs by employing a fixed order in selecting columns. Since it is a difficult problem to find an appropriate ordering of columns, we did not apply any preprocessing for finding good orderings except for pentominoes tiling. In pentominoes, the order of columns that starts from a corner field and ends at the opposite corner field works well.

Algorithms were implemented in C++. Experiments were conducted on a Linux machine with a Xeon E5-2687W 3.10 GHz CPU and 128 GB RAM. We set the sizes of memo cache used in DXZ and its fixed ordered version to 32 MBytes, where every memo cache entry is represented by a pair of a 192-bit (24 Bytes) column ID vector and an 8 Byte pointer indicating a ZDD node (total 32 Bytes) and the memo cache table has $2^{20} = 1M$ entries.

Results

We show the results in Tab. 1. The compared methods are "D": DLX without storing found covers[5], "DLX": DLX with storing of found covers into an array, "Z": DXZ without

[3]The dataset is available at http://www.tcs.hut.fi/~tjunttil/experiments/CP2010/

[4]http://people.brunel.ac.uk/~mastjjb/jeb/orlib/

[5]Strictly speaking, "D" does not solve exact cover problems since it does not store solutions. We show the results of "D" to confirm that DXZ is nearly always faster than DLX when we omit time for storing solutions.

memo cache, "DXZ": DXZ with memo cache, "F": DXZ with memo cache where the order of columns is fixed, "c2d": results of c2d solver, and "SS": results of sharpSAT solver.

Comparing DLX with DXZ, we can see DXZ nearly always yields equivalent or much better results. It tends to be much faster than the other methods with instances that have huge numbers of covers. Comparing DXZ with Z, we find that the improvements come from the use of memo cache. For example, DXZ and F run much faster than other methods on the domino_10_10 instance. This result comes from their efficient use of memo cache. Memo cache tends to works well if the same $\mathsf{col}(A)$ appears frequently on the search. In the domino tiling problem, $\mathsf{col}(A)$ corresponds to the set of fields without domino cells. Since there are many possible ways to tile cells that makes the same patterns of unfilled fields, the same $\mathsf{col}(A)$ frequently appears in the problem.

The fixed order variant F is generally slower than original DXZ, but it tends to be faster than the other baselines when instances have huge numbers of solutions. An interesting exception is tetromino, where F is much faster than normal DXZ. This is because there is a reasonable ordering of columns (cell order that starts from a corner cell and ends at the opposite corner cell) in the problem. If such good ordering of columns exists, fixed ordering can be fast since similar columns will be deleted in Search procedure, and memo cache works well.

We can see that the number of ZDD nodes is much smaller than the number of exact covers in some instances. These results indicate the superiority of ZDDs as condensed representations of sets of exact covers. Once represented as ZDDs, we can efficiently select, enumerate, and sample exact covers using them. The amount of memory consumed by algorithm DXZ can be estimated as the size of memo cache (32 MByte) plus that needed to represent the ZDD. If ZDD size is large, the amount of used memory also becomes large. However, we should remember that in such cases ZDDs are much smaller than the number of covers.

Conclusion

We proposed a new algorithm, DXZ, by extending DLX, the state-of-the-art method for finding all exact covers. Our innovation lies in combining DLX with Zero-suppressed Binary Decision Diagrams. DXZ can fully exploit memo caching and so offers significant speed enhancement. The algorithm outputs valid ZDDs representing the set of all exact covers, thus the set of exact covers can be easily manipulated with ZDD operations.

References

Akers, S. B. 1978. Binary decision diagrams. *Computers, IEEE Trans. on* 100(6):509–516.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Trans. on* C-35(8):677–691.

Chang, H.-Y., and Jiang, I. H.-R. 2016. Multiple patterning layout decomposition considering complex coloring rules. In *DAC*.

Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *JAIR* 17(1):229–264.

Darwiche, A. 2004. New advances in compiling cnf to decomposable negation normal form. In *ECAI*, 328–332.

Gunther, J., and Moon, T. 2012. Entropy minimization for solving sudoku. *Signal Processing, IEEE Trans. on* 60(1):508–513.

Hu, R.; Li, H.; Zhang, H.; and Cohen-Or, D. 2014. Approximate pyramidal shape decomposition. *ACM Trans. Graph.* 33(6):213:1–213:12.

Junttila, T., and Kaski, P. 2010. Exact cover via satisfiability: An empirical study. In *CP*, 297–304.

Knuth, D. E. 2000. Dancing links. In *Millenial Perspectives in Computer Science*, 187–214.

Knuth, D. E. 2011. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley.

Koivisto, M. 2006. An $O(2^n)$ algorithm for graph coloring and other partitioning problems via inclusion–exclusion. In *FOCS*, 583–590.

Korte, B., and Vygen, J. 2012. *Combinatorial optimization*. Springer.

Minato, S. 1993. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC*, 272–277.

Minato, S. 1994. Calculation of unate cube set algebra using zero-suppressed BDDs. In *DAC*, 420–424.

Thurley, M. 2006. sharpSAT – counting models with advanced component caching and implicit BCP. In *SAT*, 424–429.