International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 1
DOI: https://doi.org/10.17762/ijritcc.v11i1.6054
***Article Received**: 14 November 2022 **Revised**: 16 December 2022 **Accepted**: 30 December 2022*

XDLX: A Memory-Efficient Solution for Backtracking Applications in Big Data Environment using XOR-based Dancing Links

[1*]Varalakshmi M, [2]Peer Mohideen P U, [3]Thilagavathi M, [4]Daphne Lopez
[1]Asst. Prof. Sr., SCOPE
VIT, Vellore
email: mvaralakshmi@vit.ac.in
[2]Senior Director
eTouch Systems Corp
email: pupeer@gmail.com
[3]Asst. Prof. Sr., SITE
VIT, Vellore
email: mthilagavathi@vit.ac.in
[4]Professor, SITE
VIT, Vellore
email: daphnelopez.vit.ac.in

Abstract — Interactive and Backtracking applications often require undoing certain recent operations and updates made to the underlying data structures. The concept of dancing links has made such reverting operations easier and efficient by repeatedly performing unlinking and re-linking of pointers in complex data structures involving circular multiply linked lists. This paper extends the idea of dancing links to XOR linked lists, the memory efficient counterpart of doubly linked lists to develop XDLX, a more space efficient algorithm than DLX to solve exact cover problems without compromising the timing efficiency. Owing to the NP-Complete nature of the exact cover problem, any NP-complete problem can be reduced to it and solved using the proposed memory-efficient dancing links based algorithm, XDLX. The algorithm can be effectively used to solve any backtracking application and will prove to be a significant contribution towards the programming of memory-constrained environments such as embedded systems.

Keywords – Backtracking; Dancing links; DLX; Exact Cover; NP-Complete; Space Efficiency; XOR linked lists.

I. INTRODUCTION

Interactive and Backtracking applications often require undoing certain recent operations and updates made to the underlying data structures. Any technique that can efficiently perform such revert operations has been a topic of interest for researchers. An exact cover problem is a right choice for those intended to address such issues in backtracking applications. An exact cover problem is a decision problem that determines if there exists a subcollection of the subsets of a Universal set such that each element in the Universal set appears in exactly one subset of the subcollection. It is NP-Complete [1] and hence any NP-Complete problem can be reduced to exact cover problem [2] and solved using the same techniques used for solving exact cover problem.

Efficiency of an algorithm to solve a backtracking problem depends on how the search space is narrowed down and how the decision controlling data is maintained internally [3]. Maintaining a stack to store the state information for every call, is time consuming as it requires copying the entire state at each stage.

Hence it is better to employ global data structures that are updated for every recursive call and restored to a former state after returning from the corresponding recursive call. This paper focuses on developing a more efficient strategy than what is available till date, to employ and update such global data structures to solve exact cover problem. A new algorithm XDLX has been devised which is an enhancement of the renowned algorithm DLX that applies the concept of dancing links to solve exact cover problem. The rest of the paper is organized as follows. Section 2 discusses related works in this area. Section 3 expounds the implementation of proposed algorithm, XDLX and presents the experimental results. Section 4 draws conclusions.

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 1
DOI: https://doi.org/10.17762/ijritcc.v11i1.6054
***Article Received**: 14 November 2022 **Revised**: 16 December 2022 **Accepted**: 30 December 2022*

II. RELATED WORK

A. *Algorithm X*

Algorithm X proposed by Knuth [4] uses a matrix to represent the exact cover problem. Elements of the Universal set form the columns and the individual subsets constitute the rows of the matrix. [Fig. 1] shows a matrix representation of an exact cover problem with 6 subsets and 7 elements in the Universal set.If a subset contains a few elements of the Universal set, only the corresponding column entries of that row are marked 1 and others are marked 0. The objective of the algorithm is to find out a set of rows that contribute to exactly one 1 in each column.

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

Figure 1 Matrix Representation of a sample exact cover problem

Algorithm X includes the following steps.

1. If the matrix A has no more columns, the current partial solution is the valid solution and hence terminate
2. Else select a column, c (deterministically).
3. Select a row, r, such that A[r, c] = 1 (nondeterministically).
4. Include r in the partial solution.
5. for each column j where A[r, j] = 1,
   for each i where A[i, j] = 1,
   delete row i from matrix A.
   delete column j from matrix A;
6. Recursively repeat this algorithm on the reduced matrix A.

It is clear that the algorithm performs deletion of rows and columns upon choosing a particular row for its inclusion in the partial solution. All these deletion operations should be reverted or backtracked, if the choice is found to be a wrong guess and the matrix has to be brought back to its previous state.

B. *Dancing Links*

As linked lists are a better choice for frequent insertions and deletions as opposed to arrays, a linked representation, with as many number of nodes as there are 1s in the matrix, is more efficient than an array representation in terms of both space and time complexity, for exact cover problems which are usually sparse. Hence Knuth employs circular multiply linked lists to implement the algorithm and calls it DLX. In his algorithm, he uses dancing links to efficiently solve exact cover problem.

[Fig.2a] represents a node x in a doubly linked list with L(x) and R(x) as its predecessor and successor respectively. Deletion of the node x can be done using the following two steps.

$$R(L(x)) \leftarrow R(x) \text{ and } L(R(x)) \leftarrow L(x) \quad (1)$$

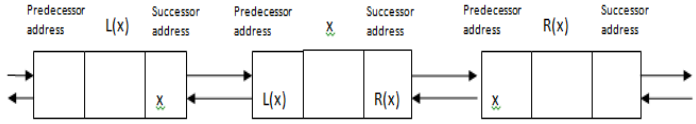

Figure 2a Node x, in a Doubly Linked List with its Predecessor L(x) and Successor R(x)

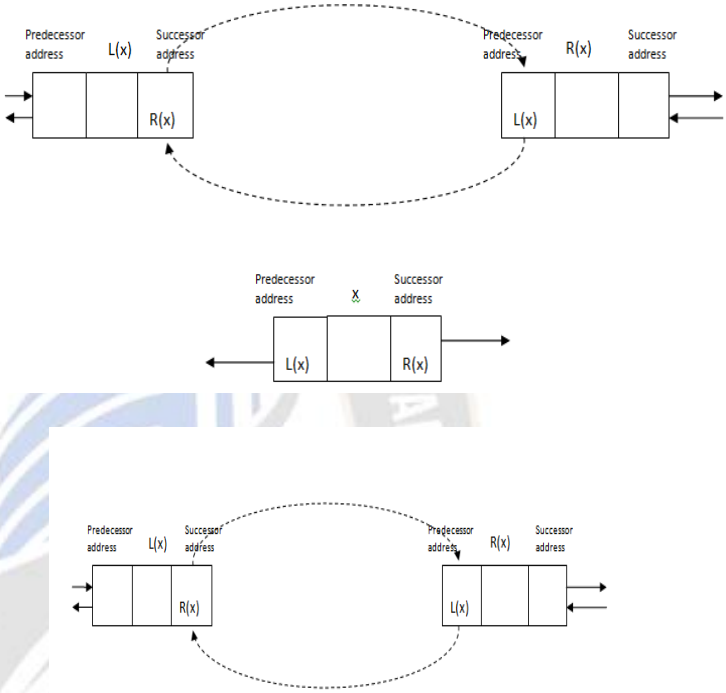

Figure 2b Retainment of address of node x's predecessor and successor, even after its deletion

From [Fig. 2b], it is clear that the node x even after getting detached from the list retains the address of its predecessor and successor nodes, unless it is made to change explicitly. This property helps to reinsert a deleted node in its original position in the list using the operations mentioned in (2).

$$R(L(x)) \leftarrow x \text{ and } L(R(x)) \leftarrow x \quad (2)$$

For this to happen, only the value of x should be kept in track and backtracking applications do this with no additional efforts. Steps 1 and 2 can be applied repeatedly to backtrack until a state is reached from where the algorithm can proceed forward again in another path. This kind of repeated unlinking and relinking of the nodes gives a feel as though the links in the global data structure are dancing gracefully; hence the two steps are identified as the technique of dancing links. The same technique was followed [5] to solve Dijkstra's algorithm [6] for N-Queens

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 1
DOI: https://doi.org/10.17762/ijritcc.v11i1.6054
***Article Received**: 14 November 2022 **Revised**: 16 December 2022 **Accepted**: 30 December 2022*

problem which was twice faster than when solved with previous techniques.

C. Algorithm DLX

Algorithm Knuth names the algorithm X implemented using dancing links as algorithm DLX [4]. An individual node x is created for each 1 in the matrix and consists of five members – L(x), R(x), U(x), D(x) and C(x). L(x) and R(x) point to the row wise predecessor and successor of the node x; U(x) and D(x) point to the column wise predecessor and successor of x. Such an interlinking makes the rows and columns appear to be individual circular multiply linked lists forming a torus. Apart from these data nodes, each column wise multiply linked list also contains another node called the column node, y with the following four members similar to a data node – L(y), R(y), U(y), D(y) and C(y) and two additional members – S(y) ("size" storing the number of 1s in that column) and N(y) ("Name" for identifying the individual columns). The member C of each node points to the column node of the corresponding column. A special node namely the head node serves as the root node for the entire structure and using its L(h) and R(h) it is linked to multiply linked list containing the column nodes. Other members of head node are not used.

Algorithm DLX is designed as a recursive function search(k), which is called initially by passing 0 for k.

If R[h] = h, print the current solution and return.
Otherwise choose a column node, c.
Cover column c
For each r ← D(c), D(D(c)) , . . . , while r ≠ c,
set O[k] ← r;
for each j ← R(r), R(R(r)) , . . . , while j ≠ r,
cover column j ;
search(k + 1);
set r ← O[k] and c ← C(r);
for each j ← L(r), L (L(r)) , . . . , while j ≠r,
uncover column j .
Uncover column c and return

Only those refinements to the algorithm that bring notable improvement in speed are worth the effort [7]. Hence at each stage of the recursive function, a column node c, with the least number of 1s is chosen as it may lead to the fewest branches for exploration. This is according to the suggestion by Golomb and Baumert [8] to select a subproblem that results in the least number of branches, at every stage of the recursive call.

Final solution is displayed by printing the rows stored in O[0], O[1], . . . , O[k−1], where for each row O, N(C(O)) , N(C(R(O))) , N(C(R(R(O)))) ,….. are also printed.

Module: Cover(c)

It first unlinks c from the column header list and then unlinks all the rows of c from the other column wise lists they are in.

Set L (R(c)) ← L(c) and R (L(c)) ← R(c).
for each i ← D(c), D(D(c)) , . . . , while i ≠ c,
for each j ← R(i), R(R(i)) , . . . , while j ≠ i,
set U(D(j)) ← U(j), D(U(j)) ← D(j),
and
set S(C(j)) ← S(C(j)) − 1

Module: UnCover(c)

Uncovering should take place in the reverse order of the cover operation.

for each i = U(c), U(U(c)) , . . . , while i ≠ c,
for each j ← L(i), L(L(i)), . . . , while j ≠ i,
set S(C(j)) ← S(C(j)) + 1, and
set U(D(j)) ← j, D(U(j))← j.
set L(R(c)) ← c and R(L(c)) ← c

The concept of dancing links can also be applied to obtain all possible solutions for a wide range of backtracking applications including Sudoku [9], N-Queens problem, Pentomino puzzle solving and so on. Application of dancing links to N-Queens problem has been visualized to promote its widespread usage [10]. Moreover the algorithm runs much faster for large problems. But the drawback of algorithm DLX is that it uses circular multiply linked lists throughout in which every node stores 2 addresses each (of predecessor and successor) for the horizontal and vertical directions. A new data structure tailored to suit the problem to be solved will increase computational speed [11]. Building on this idea, DLX algorithm can be made more space efficient if the number of addresses (pointers) stored for each node is reduced.

III. PROPOSED ALGORITHM

As a step towards formulating a memory-efficient solution, an XOR list is used in which each node x, instead of storing the four pointers L(x), R(x), U(x) and D(x), stores only 2 pointers namely H(x) and V(x), the pointer difference of its predecessor and successor addresses in the horizontal and vertical directions respectively, where the pointer difference is calculated using XOR operator.

A. Algorithm XDLX

1) Revisiting the concept of Xor Linked Lists

Properties of an exclusive-OR (in short, XOR) operator [12] denoted by '^' are as follows.

i) A^A=0
ii) A^0=A
iii) A^B=B^A
iv) (A^B)^C=A^(B^C)

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 1
DOI: https://doi.org/10.17762/ijritcc.v11i1.6054
***Article Received**: 14 November 2022 **Revised**: 16 December 2022 **Accepted**: 30 December 2022*

This paper focuses on constructing dancing links using XOR lists and as specified each node x, stores only two pointers, H(x) and V(x) in additionto the usual members C(x), N(x), S(x), and an extra member 'num' to hold the row number of the node.

H(x)=ptr to predecessor ^ ptr to successor in the horizontal direction

V(x)=ptr to predecessor ^ ptr to successor in the vertical direction

With this, the space requirement of each node is reduced to half of what is required for a node defined in DLX algorithm. A sample representation of XOR list with nodes connected both vertically and horizontally is given in [Fig. 3]. Consider node B_2. It stores the XOR result of B_1 and B_3 in $V(B_2)$ and that of A_2 and C_2 in $H(B_2)$. If the list traversal should start from node B_2, movement in all four directions is as follows.

Upward:

$B_3 \wedge V(B_2) = B_3 \wedge (B_1 \wedge B_3) = B_1$

Downward:

$B_1 \wedge V(B_2) = B_1 \wedge (B_1 \wedge B_3) = B_3$

Rightward:

$A_2 \wedge H(B_2) = A_2 \wedge (A_2 \wedge C_2) = C_2$

Leftward:

$C_2 \wedge H(B_2) = C_2 \wedge (A_2 \wedge C_2) = A_2$

Similarly if node B2 is deleted, the H and V pointers of its 4 neighbouring nodes B_1, B_3, A_2 and C_2 are modified accordingly.

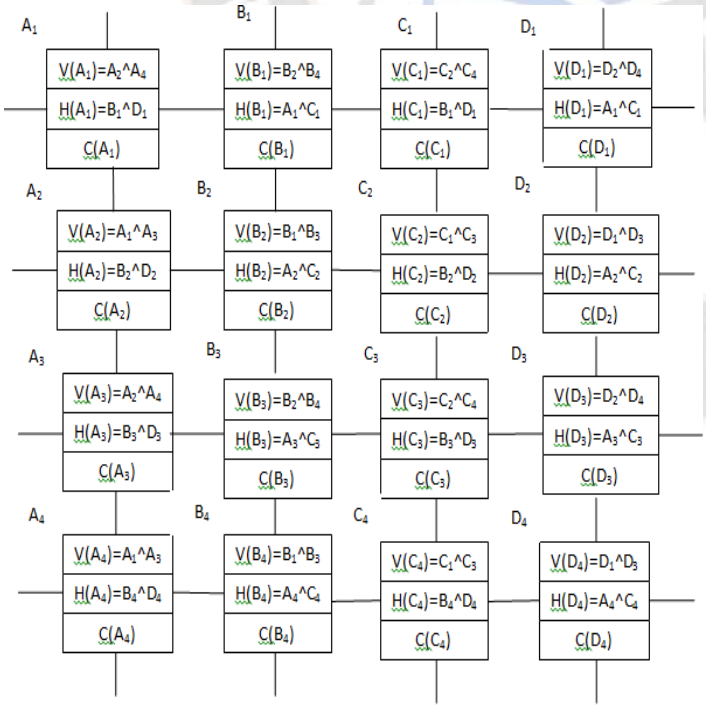

Figure 3 A sample XOR Linked List with nodes connected both vertically and horizontally

It is apparent that these operations require the addresses of the predecessor nodes also to be kept in track along with the XOR results stored in H and V pointers. This will make the space requirement of the XOR list to be equivalent to that of a multiply linked list and hence there will not be any additional benefit by employing an XOR list. Hence the proposed algorithm aims to retrieve, for any node, its predecessor's address but without having to store it.

2) Modules Description

The idea is to use the following pointers in addition to the 2 XOR differences stored at each node - a pointer named 'last' that points to the last column node; 3 arrays of pointers namely rows[ ] pointing to the first node of each row, last_in_row[ ] pointing to the last data node of each row and vlast[ ] pointing to the last data node of each column. [Fig. 4] depicts the resulting data structure for the sample matrix given in [Fig. 1]. With such a data structure, the DLX Algorithm has to be redefined to what can be called as XDLX (Algorithm X implemented using XOR-based dancing links) and its various modules are as follows.

Traverse nodes in the Horizontal direction

If *current* points to the node where we are at present and *prev*points to the node to its left, then *prev* and *current* can be made to move one step ahead in the horizontal direction as follows.

Module: move_horizontal(prev,current)
next=H(current) ^ prev
prev=current
current=next

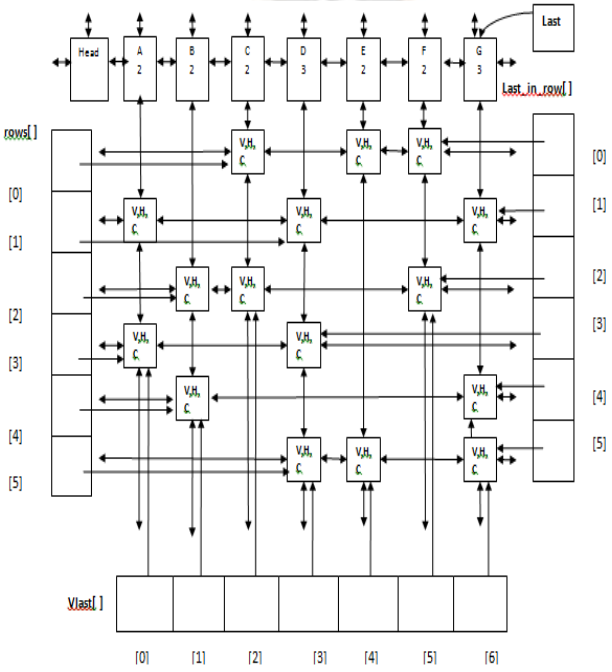

Figure 4 XOR Representation of the matrix given in Figure 1

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 1
DOI: https://doi.org/10.17762/ijritcc.v11i1.6054
***Article Received**: 14 November 2022 **Revised**: 16 December 2022 **Accepted**: 30 December 2022*

Traverse nodes in the Vertical direction

If *current* points to the node where we are at present and *prev*points to the node above it, then *prev* and *current* can be made to move one step ahead in the vertical direction as follows.

```
Module:   move_vertical(prev,current)
          next = V(current) ^ prev
          prev = current
          current = next
```

Retrieve the last node in a particular column

Address of the last node of the column represented by column node c can be retrieved by traversing through the vlast[ ] array.

```
Module: get_vlast_of_c(c)
  i=0
  while (N(C(vlast[i])) ≠ N(c) and i<colindex)
       i=i+1
return (i)
```

Cover

It unlinks c from the column header list and then unlinks all the rows of c from the other column wise lists they are in. Unlinking happens by modifying the XOR differences stored in the predecessor and successor of the node to be unlinked. Let prev_prev, prev, x, succ, succ_succ be few consecutive nodes connected horizontally in an XOR list. The XDLX algorithm works to manipulate addresses in H(prev) and H(succ) before and after deletion of x as given in (3) and (4).

Before deletion of x:

H(prev)=prev_prev ^ x and
H(succ)=succ_succ ^ x (3)

After deletion of x:

H(prev)=prev_prev ^ succand
H(succ)=succ_succ ^ prev (4)

```
Module: Cover(c)
begin
prev=last, current=head
while(current ≠ c)
move_horizontal(prev,current)
prev_prev=H(prev) ^ c
succ=H(c) ^ prev
succ_succ=H(succ) ^ c
H(prev) = prev_prev ^ succ
H(succ) = succ_succ ^ prev
if c==last
last=prev;
   i=get_vlast_of_c(c)
   prev1=c
   current1 = V(prev1) ^ vlast [i]
while (current1 ≠ c)
begin
rowprev=last_in_row[Num(current1)]
rowcurrent=rows[Num(current1)]
while(rowcurrent ≠ current1)
move_horizontal(rowprev, rowcurrent)
     current1right= H(rowcurrent) ^ rowprev
while (current1right ≠ current1)
begin
vcurrent= C(current1right)
        i=get_vlast_of_c(c)
vprev=vlast[i]
while(vcurrent ≠ current1)
move_vertical(vprev,vcurrent)
vsucc= vprev ^ V(current1right)
vprevprev= V(vprev) ^ current1right
vsuccsucc=V(vsucc) ^ current1right
V(vprev) = vprevprev ^ vsucc
V(vsucc)= vprev ^ vsuccsucc
S(C(current1right)) = S(C(current1right)) - 1
if(current1right==vlast[i])
vlast[i]=vprev
move_horizontal(rowcurrent,current1right)
end
move_vertical(prev1,current1)
end
end
```

UnCover

The deleted node retains the XOR difference of the addresses of its predecessor and successor. To reinsert the node in its original position during the uncover operation, the list has to be traversed till two adjacent nodes are identified whose XOR difference is equal to the one stored in the deleted node. Then the deleted node is reinserted in between those two nodes. Reinsertion can be simply done by modifying the XOR differences stored in the predecessor and successor of node x.

```
Module: uncover (c)
   begin
   get_vlast_of_c(c)
   current=vlast[i]
   succ=c
   while(current ≠ c)
   begin
   rowcurrent=last_in_row[Num(vlast[i])]
   rowsucc=rows[Num(vlast[i])]
   move_horizontal(rowsucc,rowcurrent)
   while(rowcurrent ≠ current)
   begin
           i=get_vlast_of_c(c)
   colcurrent=vlast[i],colsucc=C(rowcurrent)
```

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 1
DOI: https://doi.org/10.17762/ijritcc.v11i1.6054
***Article Received**: 14 November 2022 **Revised**: 16 December 2022 **Accepted**: 30 December 2022*

```
Xor_diff= colcurrent ^ colsucc
while(V(rowcurrent) ≠ Xor_diff)
begin
move_vertical (colcurrent, colsucc)
Xor_diff=colcurrent ^ colsucc
end
prevprev=V(colcurrent) ^ colsucc
succsucc= colcurrent ^ V(colsucc)
V(colcurrent)= rowcurrent ^ prevprev
V(colsucc)=succsucc ^ rowcurrent
if(Num(colsucc) <Num(rowcurrent))
      vlast[i]=rowcurrent
        S(C(rowcurrent))=S(C(rowcurrent))+1
move_horizontal(rowsucc, rowcurrent)
end
move_vertical(succ, current)
end
prevheader=last,currentheader=head
Xor_header=prevheader^currentheader
while(H(c) ≠ Xor_header)
begin
move_horizontal (prevheader, currentheader)
Xor_header= prevheader ^ currentheader
end
prevprevheader = H(prevheader) ^ currentheader
succsuccheader = prevheader ^ H(currentheader)
H(currentheader) = c ^ succsuccheader
H(prevheader) = prevprevheader^c
if(H(c) ^ prevheader==head)
last=c
end
```

Search

It is a recursive procedure which repeatedly calls itself till the list becomes empty.

```
Module: search(k)
begin
if(head = last)
print the solution containing all the rows
starting with
       O[m]  where m=0,1,…k and return.
choose a column object c, which is usually the
column with minimum number of ones
cover(c)
    i=get_vlast_of(c)
prev=vlast[i]
    r = V(c) ^ vlast[i]
prev=c
while (r ≠ c)
begin
        O[k]=r;
      rowcurrent=rows[Num(r)]
rowprev=last_in_row[Num(r)]
      while(rowcurrent ≠  r)
move_horizontal(rowprev, rowcurrent)
        j = H(rowcurrent) ^ rowprev
      rowprev=rowcurrent
while (j ≠ r)
begin
              cover(C(j))
move_horizontal(rowprev,j)
end
search(k + 1)
        r = O[k]
        c = C(r)
      backcurrent = last_in_row[Num(r)]
      backprev=rows[Num(r)]
      while(backcurrent ≠ r)
move_horizontal (backprev, backcurrent)
        j= H(backcurrent) ^ backprev)
backprev=backcurrent
while (j ≠ r)
begin
      uncover(C(j))
      move_horizontal(backprev,j)
end
move_vertical(prev,r)
end
uncover(c) and return
end
```

IV. RESULTS AND DISCUSSION

The proposed XDLX Algorithm has been implemented in C++ and compared with the performance of DLX algorithm for sample exact cover problems of varying sizes. For solving an exact cover problem with *n* elements in the Universal set and *m* individual subsets, DLX algorithm requires *4(mn/2)(=2mn)* pointers assuming that each subset contains on an average *n/2* elements of the Universal set. The same problem when implemented with the proposed XDLX algorithm uses a single pointer *last* and the following array of pointers namely *row[m], last_in_row[m], vlast[n]* in addition to the single pointer needed for each node. Thus it can be observed that the total number of pointers required is only *(2mn/2)+2m+n+1 (=mn+2m+n+1)* which has effectively reduced the memory utilization by about 50% with only a slight modification in the program logic and with no notable increase in processing time. Table 1 presents the reduction in space complexity achieved using XDLX as compared to DLX for different values of *m* and *n* and figures 5 to 7 provide graphical representation of the table data. It can be seen that the effect of XDLX on the memory requirement is more for cases with fixed values of *m* and

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 1
DOI: https://doi.org/10.17762/ijritcc.v11i1.6054
***Article Received**: 14 November 2022 **Revised**: 16 December 2022 **Accepted**: 30 December 2022*

varying values of *n* than for cases with fixed values of *n* and varying values of *m*.

TABLE I. SPACE COMPLEXITY OF DLX AND XDLX FOR VARIOUS VALUES OF *m* AND *n*

No. of subsets, *m*	No. of elements in the universal set, *n*	Space complexity of DLX (2mn)	Space complexity of XDLX (mn+2m+n+1)	DLX vs XDLX
100	100	20000	10301	1.941
100	10000	2000000	1010201	1.980
100	100000	20000000	10100201	1.980
10000	100	2000000	1020101	1.960
100000	100	20000000	10200101	1.961

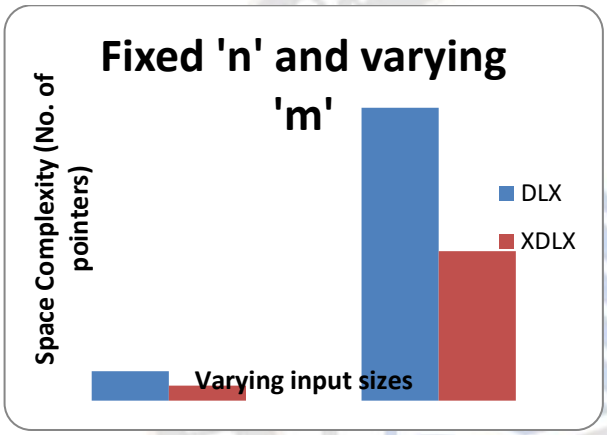

Figure 5 Space Complexity of DLX and XDLX for fixed *n* and varying *m*

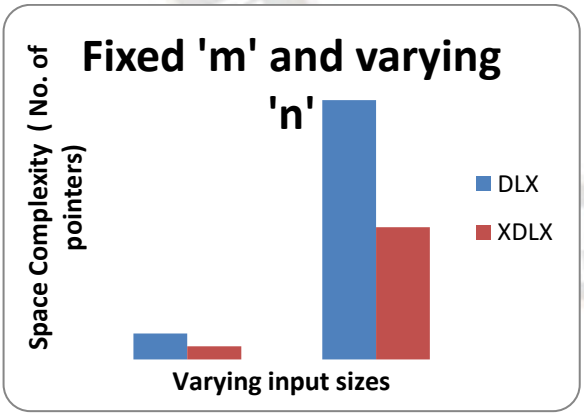

Figure 6 Space Complexity of DLX and XDLX for fixed m and varying n

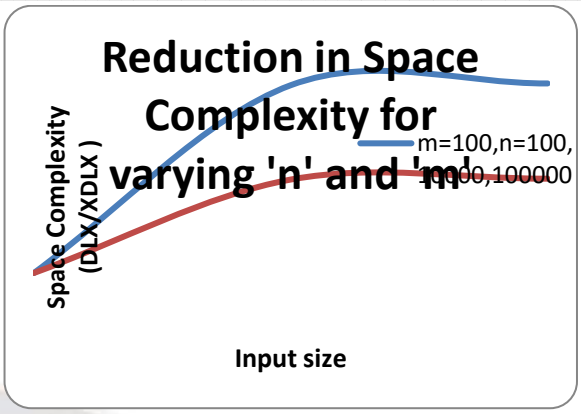

Figure 7 Reduction in Space Complexity

V. CONCLUSION

XDLX Algorithm thus developed using XOR linked lists proves that dancing links can be used to solve backtracking applications such as exact cover problem with a space efficiency better than its DLX counterpart yet without having to compromise the timing efficiency. The efficacy of the algorithm can be well understood when applied in environments with limited memory capacity such as embedded systems. Any NP-Complete problem can be reduced to exact cover problem and solved efficiently using XDLX.

REFERENCES

[1] Garey MR., Johnson DS. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman. 1979; 187-285.
[2] Karp RM. Reducibility among combinatorial problems. Complexity of Computer Computations, Springer US. 1972; 85–103.
[3] Floyd RW. Nondeterministic algorithms. Journal of the ACM (JACM). 1967; 14(4):636–644, 1967
[4] Knuth DE. Dancing Links. arXiv preprint cs/0011047. 2000.
[5] Hitotumatu H, Noshita K. A technique for implementing backtrack algorithms and its application. Information Processing Letters. 1979; 8(4):174–175.
[6] Dahl OJ, Dijkstra EW, Hoare CAR. Structured Programming. Academic Press Ltd.,. 1972.
[7] Knuth DE. Estimating the efficiency of backtrack programs. Mathematics of Computation. 1975; 29(129):122–136.
[8] Golomb SW, Baumart LD. Backtrack programming. Journal of the ACM. 1965; 12(4):516-524.
[9] Harrysson M, Laestander H. Solving Sudoku efficiently with Dancing Links. Bachelor Degree Thesis, 2014.
[10] Doyle M, Rawe B, Rogers A. JDLX: visualization of dancing links. Journal of Computing Sciences in Colleges. 2008; 24(1):9-15.
[11] Barua A, Patra A and Sinha S. PMD—An Algorithm for Finding Determinant of a Polynomial Matrix with New Data Structure. IETE Journal of Research. 2015; 39(1):51-53.
[12] Sinha P. A memory-efficient doubly linked list. Linux Journal. 2005; 2005(129);10.