Covering Complete Graphs using the Dancing Links Algorithm[⋆]

Petr Kovář[1,*,§], Yifan Zhang[1,§]

[1] VŠB - Technical University of Ostrava, Department of Applied Mathematics, 17. listopadu 2172/15, 708 00 Ostrava, Czech Republic

Abstract
The dancing links algorithm by Knuth can be used to find decompositions of a complete graph efficiently. In this paper, we generalize it to find covers of a complete graph by cliques as well. This problem arose as one step of a parallel implementation of the bounded element method when solving partial differential equations. The modification brought up an issue of counting certain covers of multiple edges multiple times. Having solved the problem by imposing a natural order, we show the computational results achieved by the modified algorithm.

Keywords
graph covering, combinatorial design, dancing links, complete graph

1. Motivation

Boundary element methods (BEM) have become a useful tool for solving partial differential equations. In comparison to widely used discretization techniques, as smaller systems with fewer degrees of freedom they offer certain advantages. However, when implementing BEM, one of the major difficulties lies in dense matrices. To parallelize the computation, a method of decomposing a dense matrix into (dense) submatrices using cyclic decomposition of complete graphs K_n into n complete subgraphs K_k was introduced in [1]. This translates to a decomposition of an n by n block matrix to n submatrices with k by k blocks each, in which only one diagonal block is occupied. The additional requirement for the decomposition to be cyclic simplified the implementation, but introduced certain restrictions. The parameters n and k are not independent: a necessary condition for a cyclic decomposition to exist is $n = k^2 - k + 1$. Moreover, the existence of a cyclic decomposition follows from the existence of the so called ρ-labeling, for which existence is guaranteed only if $k - 1$ is a prime power.

When n becomes large, the cyclic decomposition is suitable only for parallel systems with shared memory. At the same time, parameter k is preferably small. The architecture of contemporary supercomputers relies rather on fast local memory for each processor or each core, however the core memory is rather limited. While computation is cheap, memory and data transfer are expensive, therefore it is favorable to keep k small.

1.1. Covering of complete graphs

In this paper, we address the hardest part of the BEM implementation - the problem of decomposing K_n into K_k's where k is a small integer while n grows along with the number of available cores. In general, such decomposition cannot be cyclic. For fixed $k = 3$ and $k = 4$ constructions based on Steiner triple or quadruple systems can be used for certain limited values of n. For remaining values of n we use covers in addition to decomposition. We want to guarantee that only a small percentage of computation will be doubled. This can be modeled by graph coverings with a small number of doubly or triply covered edges called *excess*. Constructions of covers are known for fixed values $k = 3$, $k = 4$ as well as for most values $k = 5$ and $k = 6$.

Covering of K_n by K_k for a single fixed k, $k = 3, 4$ with a given excess is based on results by Hanani [2] and Mills [3], [4] and other authors and can be found in [5]. Covers using simultaneously K_3 and K_4 allow an even smaller excess. In Section 4 we provide solutions for small n found by brute force and compare the running times.

To find such covers for small n by brute force, we successfully use a modification of the dancing link algorithm (DLX) by Donald Knuth [6]. Even though the solution of the decomposition problem (exact cover problem in Knuth's terminology) is NP hard in general, the efficient implementation of the DLX algorithm allows to find solutions for not very large n or in some cases even disprove the existence for certain small values of n. In this paper we present two ways of modifying the algorithm for solving a covering problem instead of a decomposition

ITAT'23: Information technologies – Applications and Theory, September 22–26, 2023, Tatranské Matliare, Slovakia

[⋆] 2000 MSC: 05C70, 05C78

[*] Corresponding author.

[§] These authors contributed equally.

petr.kovar@vsb.cz (P. Kovář)
http://homel.vsb.cz/~kov16 (P. Kovář)
0000-0002-7847-5060 (P. Kovář)

problem. One modification is based on extending the data structure, while the other is a modification of the algorithm itself. In the forthcoming sections, we describe the two approaches.

1.2. Decomposition of complete graphs using the dancing links algorithm

The dancing links algorithm, proposed by Donald E. Knuth in [6], has been an efficient[1] approach to solving the exact cover problem, in which a matrix of 0s and 1s is given and we would like to determine whether it admits a set of rows containing exactly one 1 in each column. For example, the following matrix has such a set formed by the rows 2 and 3.

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

A special data structure is suggested by Knuth to implement the algorithm, that each 1 in the matrix be represented by a data object x with five fields including $L[x]$, $R[x]$, $U[x]$, $D[x]$, pointing circularly left, right, up, down to the neighbour of x, as well as $C[x]$ that links x to the column object it belongs to. Situated in the list header, each column object y has two additional fields besides those attributed to a normal data object, its size $S[y]$ which counts the number of 1s in the column as well as its name $N[y]$ that serves as an identifier for printing. An additional root object h is added linking all column objects circularly, i.e. $R[h]$ points to the leftmost column, while $L[h]$ to the rightmost one. The object h does not admit the other fields, i.e. $U[h]$, $D[h]$, $C[h]$, $S[h]$ and $N[h]$ are not used.

The algorithm works as follows. A depth-first search tree is set up by invoking *search*(0). As long as there is at least one column in the list headers, the algorithm chooses the leftmost column, and tries to form a solution by the first row that is able to cover this column. The choice of column we made here left the field S unused throughout the algorithm. Before continuing, the algorithm requires the chosen column to be "removed" by modifying the links of its neighbours in the function *cover_column*. Other columns covered by the same row need to be "removed" in the same way, so that the size of the data matrix will decrease. Then *search*(1) is invoked recursively on the shrunk matrix to include another row in the partial solution. If this partial solution turns out to be a valid solution, it is printed by the function *print_solution* when the corresponding search reaches a point where all columns have been "removed", i.e. $R[h] = h$. Then the algorithm goes back to the last level of depth and attempts to include the next row that covers the same column, as soon as the "removed" columns are "restored" in the matrix by the corresponding modifications of links in the function *uncover_column*. The output of the algorithm would contain all feasible solutions to the exact cover problem, while it also makes sense to stop after finding the first solution when we are only interested in the existence of a solution.

[1]It is a frugal approach to the problem, albeit not polynomial.

```python
def choose_column():
    return R[h]

def cover_column(c):
    L[R[c]] = L[c]
    R[L[c]] = R[c]
    i = D[c]
    while i != c:
        j = R[i]
        while j != i:
            U[D[j]] = U[j]
            D[U[j]] = D[j]
            j = R[j]
        i = D[i]
    return

def uncover_column(c):
    i = U[c]
    while i != c:
        j = L[i]
        while j != i:
            U[D[j]] = j
            D[U[j]] = j
            j = L[j]
        i = U[i]
    L[R[c]] = c
    R[L[c]] = c
    return

def print_solution():
    s = []
    for o in partial_solution:
        row = [N[C[o]]]
        p = R[o]
        while p != o:
            row.append(N[C[p]])
            p = R[p]
        s.append[row]
    print(s)
    return

def search(depth):
    if R[h] == h:
        print_solution()
        return
    else:
        c = choose_column()
        cover_column(c)
        r = D[c]
        while r != c:
            partial_solution.append(r)
            j = R[r]
            while j != r:
                cover_column(C[j])
                j = R[j]
            search(depth + 1)
            r = partial_solution.pop()
            c = C[r]
            j = L[r]
            while j != r:
                uncover_column(C[j])
                j = L[j]
            r = D[r]
        uncover_column(c)
        return
```

The decomposition of a complete graph K_n by copies of a smaller complete graph K_k ($k \leq n$) can be naturally modelled as an exact cover problem, in which the edges of K_n are treated as columns and the subgraphs of K_n isomorphic to K_k serve as the rows: one row for every choice of k vertices from the n with ones denoting the edges contained in the particular choice.

For example, consider the decomposition of K_7 into K_3, where 21 columns are given, each corresponding to an edge of K_7, and there are $\binom{7}{3} = 35$ rows that represent the 35 K_3-subgraphs of K_7. Denote the vertices of K_7 by $1, 2, \cdots, 7$, and let $e_{i,j} := (0, \cdots, 0, \overset{\text{edge } i\text{-}j}{1}, 0, \cdots, 0)$, then, for instance, the row corresponding to the triangle $\Delta 123$ has the form $e_{1,2} + e_{1,3} + e_{2,3}$. Apparently, the rows $e_{1,2} + e_{1,4} + e_{2,4}$, $e_{2,3} + e_{2,5} + e_{3,5}$, $e_{3,4} + e_{3,6} + e_{4,6}$, $e_{4,5} + e_{4,7} + e_{5,7}$, $e_{1,5} + e_{1,6} + e_{5,6}$, $e_{2,6} + e_{2,7} + e_{6,7}$ and $e_{1,7} + e_{1,3} + e_{3,7}$ constitute a solution to the exact cover problem, which corresponds to the decomposition of the K_7 into the following 7 copies of K_3:

$$\{\Delta 124, \Delta 235, \Delta 346, \Delta 457, \Delta 156, \Delta 267, \Delta 137\}.$$

To count the number of feasible coverings from a theoretical perspective, notice that fixing the triangle $\Delta 123$ in the solution is equivalent to choosing 3 as the last element in the triangle covering the edge 1-2 among 5 options: $3, 4, 5, 6, 7$; fixing also $\Delta 145$ is equivalent to choosing 5 as the last element in the triangle covering 1-4 among 3 options: $5, 6, 7$; then fixing $\Delta 246$ is equivalent to choosing 6 as the last element in the triangle covering 2-4 among 2 options: $6, 7$; and it is now apparent that the remaining part of the solution is unique: $\Delta 167$, $\Delta 356$, $\Delta 257$ and $\Delta 347$.

When the structure of 21 columns and 35 rows are inputted, the dancing links algorithm finds 30 solutions for the decomposition of K_7 into K_3, as expected.

2. Covering complete graphs using the dancing links algorithm

The major challenge posed to the original dancing links algorithm when considering the covering rather than the decomposition of a complete graph is that a solution may contain more than one row covering the same column. This section suggests possible modifications to the original algorithm in order to overcome this issue.

2.1. Extra columns

In graph coverings, the excess multigraph is often not determined uniquely. When the excess is fully known, it seems natural to add the edges in the excess as extra columns, so that the covering problem is transformed to a decomposition one.

For example, consider the covering of K_6 by 6 copies of K_3, of which the excess is known ([5], p. 371, Table VI.11.41) to be a perfect matching of K_6. Without loss of generality, assume the doubled edges are 1-2, 3-4 and 5-6, then the dancing links structure would contain $15 + 3 = 18$ columns corresponding to the edges. However, notice that the rows containing one of the three extra edges, such as $e_{1,2} + e_{1,3} + e_{2,3}$ and $e_{4,5} + e_{5,6} + e_{4,6}$, must be doubled in the data structure, resulting in 32 rows instead of 20.

Introducing extra columns does not only bring about bloated data structure, but also redundant solutions. If some solution contains a row that has been doubled when building the data structure, it would appear again as another solution including the clone of the same row. For instance, solving the covering problem of K_6 by K_3 with 3 extra columns as mentioned above using dancing links yields 16 solutions, of which only 2 are genuinely different from each other.

2.2. Revisiting columns

Besides introducing extra columns, the algorithm can be steered to revisit a column as long as it is not “removed" from the structure during the last visit, which motivates us to assign a multiplicity $M[x]$ to each column x in the original data structure that equals 1 by default. When an edge is in the excess of a covering problem, we raise the multiplicity of the corresponding column to match the amount it needs to be covered. The M vector can be implemented as an additional field of the header elements. For example, when covered by triangles, if we order the edges of K_6 lexicographically, then the M vector would be

$$\overset{\text{edge 1-2}}{2}, 1, 1, 1, 1, 1, 1, 1, 1, \overset{\text{edge 3-4}}{2}, 1, 1, 1, 1, \overset{\text{edge 5-6}}{2}.$$

We modify the functions *cover_column* and *uncover_column* accordingly so that the multiplicity is always increased or decreased by 1 while the "removal" or "restoration" of the column only happens when its multiplicity is 1 or 0, respectively.

```
def cover_column(c):
    if M[c] == 1:
        L[R[c]] = L[c]
        R[L[c]] = R[c]
        i = D[c]
        while i != c:
            j = R[i]
            while j != i:
                U[D[j]] = U[j]
                D[U[j]] = D[j]
                j = R[j]
            i = D[i]
    M[c] -= 1
    return

def uncover_column(c):
    if M[c] == 0:
        i = U[c]
        while i != c:
            j = L[i]
            while j != i:
                U[D[j]] = j
                D[U[j]] = j
                j = L[j]
            i = U[i]
        L[R[c]] = c
        R[L[c]] = c
    M[c] += 1
    return
```

This approach avoids the unnecessary expansion of data structure, while continues to generate duplicate solutions. Suppose two rows cover the same column and sit in a feasible solution to a covering problem, the same solution may reappear with the two rows included in the reverse order. If the leftmost column has a high multiplicity, the algorithm yields numerous duplicate solutions; in comparison, if the column with higher multiplicity is placed further to the right in the structure, as the data matrix shrinks during the search, the algorithm tends to generate fewer duplicate solutions.

For example, consider the covering of K_5 by 4 copies of K_3, of which the excess is known ([5], p. 371, Table VI.11.41) to be a double edge. With the edges of K_5 ordered lexicographically, assigning multiplicity 3 to the leftmost column corresponding to the edge 1-2 yields 6 solutions of which all are essentially the same: a permutation of $\Delta 123$, $\Delta 124$, $\Delta 125$ together with $\Delta 345$, while assigning multiplicity 3 to the rightmost column 4-5 yields a unique solution: $\Delta 123$, $\Delta 145$, $\Delta 245$, $\Delta 345$.

3. Efficiency considerations

This section evaluates the dancing links algorithm adapted to the covering problem in Section 2.2. Some improvements to eliminate multiple counts are suggested.

3.1. Ordered search of rows

An apparent shortcoming of our algorithm is the redundant search of rows covering the same column with nontrivial multiplicity, which not only produces an inaccurate number of solutions, if any exist, but also harms the efficiency of the algorithm. To address the issue, when covering such a column, the rows containing it should be processed only in a certain order.

One approach is to introduce an order on the elements in each column. More specifically, we return the currently processed row r in the *search* function, and start the next search from the next valid row once the columns concerned are properly "removed" by the *cover_column* function. This idea is realised by introducing the previously unused field $S[x]$ to each data object x: the objects in a column c are assigned values $1, 2, \cdots, S[c]$ from the top row to the bottom. The modified *search* function (see the lines 8–13 below) follows.

```
def search(depth, prev):
    if R[h] == h:
        print_solution()
        return prev
    else:
        c = choose_column()
        cover_column(c)
        r = D[c]
        if c == C[prev]:
            while S[r] < S[prev]:
                if r == c:
                    break
                r = D[r]
        while r != c:
            partial_solution.append(r)
            j = R[r]
            while j != r:
                cover_column(C[j])
                j = R[j]
            search(depth + 1, r)
            r = partial_solution.pop()
            c = C[r]
            j = L[r]
        while j != r:
            uncover_column(C[j])
            j = L[j]
        r = D[r]
    uncover_column(c)
    return r
```

3.2. Exploiting the symmetry

In addition to the redundant solutions addressed in Section 3.1, another factor that increases the size of output is that some solutions are simply a permutation of vertices applied to one another, such as the following two decompositions of K_7 into K_3:

$$\{\Delta 124, \Delta 235, \Delta 346, \Delta 457, \Delta 156, \Delta 267, \Delta 137\},$$
$$\{\Delta 123, \Delta 245, \Delta 346, \Delta 357, \Delta 156, \Delta 267, \Delta 147\}.$$

If we draw the vertices $1, 2, \cdots, 7$ of K_7 cyclically, the first solution above is a cyclic decomposition into K_3 while the second seems not to be cyclic. However, if we swap the vertices 3 and 4, the second solution becomes cyclic as well, which implies that the two solutions are equivalent.

Multiple counts of equivalent solutions appeared already in graph decompositions. Motivated by such considerations, we can fix certain row(s) when searching solutions using the dancing links algorithm to omit equivalent solutions, provided that merely the existence of any solution interests us. The implementation follows.

```
def utilise_row(r):
    partial_solution.append(r)
    obj = r
    while True:
        cover_column(C[obj])
        obj = R[obj]
        if obj == r:
            break
    return
```

After the data setup, we may execute several lines of *utilise_row* to restrict our attention to the solutions containing certain rows, before invoking *search*(0). For the sake of conservation of the structure, it is advisable to call the following *neutralise_row* function in the opposite order after the search of solutions, when the original data structure is properly restored.

```
def neutralise_row(r):
    if partial_solution.pop() != r:
        return -1
    else:
        obj = L[r]
        while True:
            uncover_column(C[obj])
            obj = L[obj]
            if obj == L[r]:
                break
        return
```

For instance, with the following lines executed, the algorithm finds only 2 decompositions of K_7 into K_3 containing the triangles $\Delta 123$ and $\Delta 145$, as expected.

```
utilise_row('1, 2, 3')
utilise_row('1, 4, 5')
search(0)
neutralise_row('1, 4, 5')
neutralise_row('1, 2, 3')
```

It is worth mentioning that some solutions, even sometimes all, may be omitted when a search bears too many fixed rows. Further research can be carried out on the maximal amount of (independent) rows one is able to fix while preventing the algorithm from missing a solution.

4. Computations

To demonstrate the efficiency of the DLX algorithm for the cover problem, we summarize computation times for finding all solutions when covering K_n by K_k for $k = 3, 4$ along with the count of all possible covers. Confirming the existence by finding the first solution, provided it exists, for n in Tables 2 through 4 took less than 1 second in each of the cases. Table 1 compares running times of the original dancing link algorithm with the modified algorithm where decomposition (not cover) is possible, since the DLX algorithm does not support covers. The times are essentially identical, since the modification did not have much influence on the performance.

Tables 2 through 4 then compare running times of the cover problem for different sizes of the covered graph. Columns 3 and 4 give the counts and times when finding all possible covers using the modification of the DLX algorithm described in Section 2.2, where some equivalent solutions are counted multiple times. Columns 5 and 6 give the counts and times when finding all possible covers eliminating multiple counts (and thus reducing the number of cases to traverse) using the modification described in Section 3.1.

Other approaches have been used to find complete graph decompositions using a brute force search. In [7] were graph decompositions found via a SAT solver. The authors say that to produce a K_6 decomposition of K_{31} took about 100 seconds. Using our implementation of the DLX algorithm it took about 25 seconds to set up the data structure described in Section 1.2 for the K_6 decomposition of K_{31} (465 columns and 736 281 rows). Then however a decomposition was found in less than 1 second.

n	k	count all	time [s]	count unique	time [s]
7	3	30	0	30	0
9	3	840	0	840	0
13	3	1 197 504 000	9141	1 197 504 000	9120
13	4	1 108 800	14	1 108 800	14
16	6	0	72	0	70

Table 1
Comparing original DLX and modified DLX algorithm for decompositions of K_n into K_k.

n	excess	count all	time [s]	count unique	time [s]
5	$(K_2)^2$	6	0	1	0
6	M	2	0	2	0
7	$\emptyset$	30	0	30	0
8	$K_{1,3} \cup M$	276	0	96	0
9	$\emptyset$	840	0	840	0
10	M	36 952	0	12 384	0
11	$(K_2)^2$	1 088 640	11	181 440	1
12	M	25 952 624	486	17 291 520	231
13	$\emptyset$	1 197 504 000	8854	1 197 504 000	9089

Table 2
Single CPU times for computing all coverings of K_n with given excess by K_3.

n	excess	count all	time [s]	count unique	time [s]
6	M	1	0	1	0
7	$2C_3 \cup (K_2)^2$	6	0	0	0
8	$2(K_2)^2 \cup C_4$	24	0	2	0
9	$K_{1,4} \cup M$	0	0	0	0
10	$(K_2)^3$	0	0	0	0
11	C_{11}	4	0	2	0
12	M	240	0	240	0
13	$\emptyset$	1 108 800	14	1 108 800	13
14	$3K_3 \cup X$	22 565 808	1821	4 969 400	554

Table 3
Single CPU times for computing all coverings of K_n with given excess by K_4.

On the other hand, the nonexistence of a K_6 decomposition of K_{16} was shown in less than a second by UNSAT in [7], while our algorithm gave the negative answer only after traversing the whole search tree in 70 seconds. All computations were performed on a notebook with an i5 core at 2.5 GHz.

When running the DLX algorithm for $k = 3$ and $k = 4$ we include the positive answers only (Tables 2 and 3), since the existence is known. When covering K_n simultanously by K_3 and K_4 we provide running times of the DLX algorithm even in some cases when no solution exists (e.g. with excess K_2), which was demonstrated by a brute force search with negative outcome (Table 4).

n	excess	count all	time [s]	count unique	time [s]
6	$\emptyset$	0	0	0	0
6	M	3	0	3	0
7	$\emptyset$	30	0	30	0
8	$\emptyset$	0	0	0	0
8	$(K_2)^2$	180	1	30	0
9	$\emptyset$	840	0	840	0
10	$\emptyset$	33 600	0	33 600	1
11	$\emptyset$	0	9	0	8
11	$(K_2)^2$	1 088 640	41	181 440	8
12	$\emptyset$	4 435 200	807	4 435 200	756
12	M	26 948 244	4337	18 152 400	3852
12	K_2	1 672	0	0	730

Table 4
Single CPU times for computing all coverings of K_n with given excess simultanously by K_3 and K_4.

5. Conclusion

Dancing link algorithm was introduced as an efficient implementation for a set decomposition problem. An easy modification of the data structure used by the algorithm is suitable for certain packing problems of K_k into a supergraph K_n (some edges are not included in the subgraphs and form the *padding*) was already provided by Knuth in the paper [6]. The advantage of the approach is that the padding needs not to be specified ahead.

In this paper we discuss the modification of the original algorithm suitable for covering problems with specified excess. A simple modification of the structure with additional columns is mentioned briefly (Section 2.1). A second approach that modifies the algorithm rather than the structure is described in more detail (Section 2.2). The modification implements a required cover count for every element and covering a column decreases this column count until it can be covered when the count reaches zero.

The efficiency is briefly demonstrated in Section 4. The row $n = 12$ of Table 2 shows the algorithms did find more than 17 million different covers in 231 seconds when distinguishing vertex labels. The solutions were just counted, not stored. Would we store a simple text description of each solution, the output takes 12 GB (!) and the running time would be nearly doubled (additional 220 s when writing on an SSD disk). This suggests a rough estimate for $n = 12$ and $k = 3$: finding a single solution using DLX algorithm is comparable to storing the solution on a media. For $n = 13$ we did find the count of more than a billion solutions in 2.5 hours, but we did not attempt to store the covers.

Acknowledgments

This work is partially supported by Grant of SGS

No. SP2023/011, VŠB–Technical University of Ostrava, Czech Republic.

References

[1] D. Lukáš, P. Kovář, T. Kovářová, M. Merta, A parallel fast boundary element method using cyclic graph decompositions, Numerical Algorithms 70 (2015) 807–824. doi:10.1007/s11075-015-9974-9.

[2] H. Hanani, Balanced incomplete block designs and related designs, Discrete Mathematics 11 (1975) 255–369. doi:10.1016/0012-365X(75)90040-0.

[3] W. Mills, On the covering of pairs by quadruples i, Journal of Combinatorial Theory, Series A 13 (1972) 55–78. doi:10.1016/0097-3165(72)90008-8.

[4] W. Mills, On the covering of pairs by quadruples. ii, Journal of Combinatorial Theory, Series A 15 (1973) 138–166. doi:10.1016/S0097-3165(73)80003-2.

[5] C. J. Colbourn, J. H. Dinitz (Eds.), Handbook of combinatorial designs, 2nd. ed., CRC Press, 2007.

[6] D. E. Knuth, Dancing links, 2000. arXiv:cs/0011047.

[7] W. Zhao, M. Liffiton, P. Jeavons, D. Roberts, Finding graph decompositions via sat, 2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI) (2017) 131–138. doi:10.1109/ICTAI.2017.00031.