Testing Copositivity in Pentadiagonal Matrices

Sjoerd van Bree, s.vanbree@student.utwente.nl, University of Twente, The Netherlands

ABSTRACT

Testing a matrix for copositivity can in general not efficiently be done. Therefore, it is of interest to find as many classes of matrices as possible for which there exists a better method. We propose an algorithm based on eliminating block diagonal principal submatrices to check copositivity for pentadiagonal matrices. We found this algorithm runs in $O(\varphi^n * n^4)$, where φ is the golden ratio. Tests were performed on the algorithm to compare it to the generic method.

Additional Key Words and Phrases: copositive matrix, pentadiagonal matrix, computational complexity.

1 INTRODUCTION

The complexity of an algorithm determines the time or space in memory it takes to execute with regard to the input size. Different algorithms have different complexities, and it can determine how useful an algorithm is, since a high complexity can cause a program to take an unworkable amount of time. For any computational problem, it is desiarable to find the most efficient algorithm possible.

Copositive matrices have been a subject of research for some time and have applications in many fields of science, examples can be found in [2]. For these reasons it is useful to be able to test whether a matrix is copositive or not. However, testing for copositivity is an NP-hard problem, and for a general matrix, the best current algorithm has a computational complexity of $O(n^4 * 2^n)$ [5].

There has already been research to find classes of matrices in which copositivity can be tested more efficiently. So far three cases have been found. The first case is the acyclic matrix, which can be tested in linear time [4]. The second case is the matrix in which all off diagonal entries are nonpositive, such a matrix is copositive if and only if it is semidefinite [3]. This can be tested in $O(n^3)$ via Cholesky factorization. The third case is the tridiagonal matrix, which can also be tested in linear time [1].

In [6], an algorithm is provided to test copositivity of tridiagonal matrices. The algorithm is based on a characterization of Väliaho and has a complexity of $O(n^6)$. We would like to investigate how the argument given in [6] can be extended for pentadiagonal matrices, and if the resulting algorithm is polynomial time or not.

TScIT 37, July 8, 2022, Enschede, The Netherlands

Our conjecture is that the extention to the pentadiagonal case must depend on the number of diagonals. In this paper, we focus on characterization and development of an algorithm for pentadiagonal matrices. We will provide the computational complexity of this algorithm.

The structure of the paper is as follows: first, we will introduce pentadiagonal matrices and their relevancy to this problem. Then, an algorithm will be propesed for testing copositivity in this matrix, and the complexity of that algorithm will be shown. The algorithm will be implemented and tests will be performed to compare it to the generic method. Finally, the results as well as future work will be discussed.

1.1 Preliminaries

The following definitions will be used throughout the paper:

Principal submatrix. A principal submatrix of matrix A is a submatrix of A for which the row with index i is removed if and only if the column with index i is removed.

Tridiagonal matrix. Matrix $A \in \mathbb{R}^{nxn}$ with elements $a_{i,j}$, 1 ≤ i,j ≤ n is tridiagonal if $a_{i,j} = 0$ for all i,j such that |i-j| ≥ 2. In other words a tridiagonal matrix is a square matrix for which all entries except those on the main diagonal and the two diagonals next to the main diagonal are zeros. Such a matrix has the following general form.

$$M = \begin{pmatrix} a_{1,1} & a_{1,2} & 0 & \dots & 0 \\ a_{2,1} & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & a_{n-1,n} \\ 0 & \dots & 0 & a_{n,n-1} & a_{n,n} \end{pmatrix}$$

Pentadiagonal matrix. Matrix $A \in \mathbb{R}^{nxn}$ with elements $a_{i,j}$, 1 ≤ i,j ≤ n is pentadiagonal if $a_{i,j} = 0$ for all i,j such that |i-j| ≥ 3. A pentadiagonal square matrix is a matrix for which all entries except those on the main diagonal, the two diagonals above the main diagonal, and the two diagonals below the main diagonal are zeros. Such a matrix has the following general form.

$$M = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & 0 & \dots & 0 \\ a_{2,1} & \ddots & \ddots & \ddots & \ddots & \vdots \\ a_{3,1} & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & a_{n-2,n} \\ \vdots & \ddots & \ddots & \ddots & \ddots & a_{n-1,n} \\ 0 & \dots & 0 & a_{n,n-2} & a_{n,n-1} & a_{n,n} \end{pmatrix}$$

Block diagonal matrix. A block diagonal matrix A is a square n×n matrix for which there exists an i < n for which $a_{i,i} \neq 0$, and $a_{j,k} = 0$ for j ≤ i < k and k ≤ i < j, and there is at least one $a_{r,r} \neq 0$ with r > i.

Copositive matrix. A symetric square matrix A ∈ $\mathbb{R}^{n\times n}$ is copositive if $x^T Ax \geq 0$ for every nonnegative vector x ∈ $\mathbb{R}^n_+$. The following is an example of a copositive matrix

$$M = \begin{pmatrix} 1 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Binary string. A binary string of length n is a sequence of n ones and zeros.

2 PROBLEM STATEMENT

In general, testing whether a matrix is copositive takes exponential time. But we also know that a tridiagonal matrix can be tested in linear time. It is interesting to find out what happens when we increase the number of diagonals in the matrix. The first case that needs to be investigated is the matrix with two more diagonal added, which is a pentadiagonal matrix. This leads to the following questions:

(1) What is an efficient algorithm for testing copositivity in a pentadiagonal matrix?

(2) What is the computational complexity of this algorithm?

3 METHODLOLOGY

To test copositivity in a pentadiagonal matrix, we are going to use a characterization by Keller, found in the paper by Väliaho [7]:

Theorem. A matrix is copositive if and only if each principal submatrix for which the cofactors of its last row are nonnegative, has a nonnegative determinant.

However, since the number of principal submatrices of a matrix is 2^n, this would result in an exponential algorithm. Quist [6] has shown that testing the complexity of tridiagonal matrices can be done in $O(n^6)$. They used the fact that principal submatrices which can be seen as block diagonal don't have to be tested. By testing all pricipal submatrices that are not block diagonal, we are also testing all block from making up all block diagonal principal submatrices. Their algorithm exploits the idea of connected indices to compute the number of block diagonal matrices. Howerver, in the case of pentadiagonal matrices connected indices are not sufficient. Therefore, we will use a different method.

We can represent a principal submatrix as a binary sting. We do this in the following way: for a n×n matrix, the i-th entry in the binary string represents the presence or absence of the i-th row and column in the matrix. A 1 includes the row and column in the principal submatrix, and a 0 excludes it. In the following theorem we provide a connection between a binary string and block diagonal martrices:

Theorem. If a binary string has two or more zeros between any pair of consecutive ones, the string represents a block diagonal matrix.

Proof: For a n×n pentadiagonal matrix A and an i < n-2, removing the row and column with index i+1and i+2 from A while not removing row and column i+3 will remove all of the following entries: $a_{i,i+1}$, $a_{i,i+2}$, $a_{i+1,i}$, $a_{i+2,i}$, $a_{i-1,i+1}$, and $a_{i+1,i-1}$. Since for elements $a_{j,k}$ with 1 ≤ j,k ≤ n we have $a_{i,j} = 0$ for all j,k such that |j-k| ≥ 3 since the matrix is pentadiagonal, the principal submatrix now satisfies our condition for a block diagonal matrix.

We can show that the inverse is also true in case the pentadiagonal matric has no zero entires on its five main diagonals. This shows us that in those cases we have found all blockdiagonal principal submatrices.

Proof. For a n×n pentadiagonal matrix A and an i < n-1, removing the row and column with index i+1 from A while not removing row and column i and i+2 will cause $a_{i,i+2}$ to become right adjacent to $a_{i,i}$ in the submatrix. Since $a_{i,i+2}$ is nonzero, it does not satisfy our condition for a blockdiagonal matrix.

Since we don't have to test the block diagonal submatrices, we want to find the complement to the set of submatrices decscribed in the theorem above. Specifically, we want to find all binary strings with fewer than two zeros between any pair of consecutive ones.

3.1 Algorithm

To find these binary strings, first we will make an algorithm that returns all binary strings starting with a one and ending with a one that meet this condition. The algorithm takes as input an integer n for the length of the binary string. This algorithm is given below.

```
 1: findPartialStrings(n)
 2: if n = 0
 3:    return []
 4: if n = 1
 5:    return ["1"]
 6: if n = 2
 7:    return ["11"]
 8: else
 9:    S = []
10:    for element in (findPartialStrings(n-1)) do
11:        add ("1" + element) to S
12:    for element in (findPartialStrings(n-2)) do
13:        add ("10" + element) to S
14:    return S
```

Step 2-7 : For the cases of n=0, n=1 and n=2, the algorithm returns the only possible string**.**

Step 9-13: First, we create an empty list S in which we can store all found binary strings. We split the binary strings in 2 different cases: the case where the string starts with "11", and the case where the string starts with "10".
In the case where our string starts with "11", we know that the second entry is a "1". Since the last entry of the string is also a one, we can find all possible binay strings that can follow our initial "1" by recursively calling our algorithm on length n-1. We can append all found stings to our initial "1" and add them to list S.
In the case where the string starts with "10", the next entry needs to be another "1", otherwise there will be more than two zeros between two consecutive ones. Because the remaining string still needs to end with a one, we can find all possible binary strings that can follow our initial "10" by recursively calling our algorithm on n-2. We can append all found stings to our initial "10" and add them to list S.

Step 14: We return the list S with all found strings.

The number of binary strings for any length n is the number of binary strings for length n-1 plus the number of binary strings for n-2. If we say R(n) = { binary strings starting with a one and ending with a one with fewer than two zeros between any pair of consecutive ones} then |R(n)| = |(R(n-1)| + |R(n-2)| with |R(1)| = 1 and |R2)| = 1. This clearly shows |R(n)| = F_n being the Fibonacci sequence. The complexity of our algorithm is equal to the growth of the number of binary strings. The numbers in the fibonacci sequence grow in $O(\varphi^n)$, with φ being golden ratio. Since the number of binary strings grows at the same speed as the numbers in the Fibonaccy sequence, this is also our algorithm's complexity.

Now that we have all the binary strings of length n starting and ending with a one, we need to find the remaining cases. If the binary sting doesn't start with a one, it starts with any number of zeros, and if it doesn't end with a one, it ends with any number of zeros. Therefore, we find all remianing strings by taking a string we found with the previous algorithm that are smaller than our length n and adding zeros to the start and end.

The algorithm to find these is given below.

```
1: findBinaryStrings(n)
2: S = []
3: add (n zeros) to S
4: for c = 0,...,n-1 do
5:    middleSection = findPartialStrings(n-c)
6:    for b = 0,...,c do
7:        for each element in middleSection do
8:            add ((b zeros) + element + (c-b zeros)) to S
9: return S
```

The algorithm requires an integer for the length of the binary string. It returns a list with all the binary strings of length n that satisfy our condition.

Step 2: Create an empty list to hold the strings.

Step 3: Because we are going to assume that there is at least one "1" in our string, we manually add the string consisting of all zeros.

Step 4 and 5: For a binary string of length n, we are going to create sections starting and ending with a one via the findPartialStrings algorithm with length n-c. Variable c is going to determine the total amount of zeros we are going to add at the beginning and at the end of the section those sections going from 1 to n-1.

Step 6-8: variable b determines the division of the c zeros at the front and at the back of every section created by findPartialStrings, going from 0 zeros in the front, to c zeros in the front. We add all binary strings created this way to list S.

Step 9: We return the list S with all found strings.

W have two for loops with a linear complexity, followed by a for loop dependent on the number of binary strings created by findPartialStrings, which is $O(\varphi^n)$. The complexity of the algoritm is the largest of these complexities, which is $O(\varphi^n)$. By computing a few large n, we found that the complexity of the algoirthm is $\approx 1.894*\varphi^n$

As shown by Quist [6], testing a single principal submatrix takes the computation of at most n determinants, with an order of at most n. Since calculating a single determinant can be done in $O(n^3)$, testing a single principal submatrix can be done in $O(n^4)$. Doing this for every principal submatrix gives us a time complexity of testing a pentadiagonal matrix for copositivity of $O(n^4 * \varphi^n)$.

3.2 Experiments

To measure the performance of the algorithms, they were both implemented in python version 3.7.2, making use of the numpy library. All tests were ran on a system using Intel Core i7-7700HQ CPU @ 2.80GHz and windows 10.

For the first test that was performed we compared the number of binary strings generated by the findBinaryStrings algorithm to the number of principal submatrices that would be used by the generic algorithm, which is 2^n. The test was performed for matrices of size 1 to 15. The results of this test can be found in the table below.

Matrix Size	findBinaryStrings	2^n
1	2	2
2	4	4
3	8	8
4	15	16
5	27	32
6	47	64
7	80	128
8	134	256
9	222	512
10	365	1024
11	597	2048
12	973	4096
13	1582	8192
14	2568	16384
15	4164	32768

The number of principal submatrices that needs to be tested is the same for a matrix of size 1,2 and 3, but starting from matrix size 4 the number of principal submatrices found by the findBinaryStrings algorithm is smaller than in the general case. The difference between the two methods keeps increasing, at matrix size 15 the number of strings of the genetic algorithm is 8 times larger than the number of strings in our method. This shows us that we can expect testing copostitvity to be faster as well.

For the second test we compared the number of binary strings found by the findBinaryStrings algorithm to the number of binary strings with fewer than two zeros between any pair of consecutive ones, to give us more reason to believe that our algorithm does indeed create such strings.
This was done by creating an algorithm to generate all different binary strings of length n. These strings where then tested on containing two or more zeros between any two consecutive ones, and those that did were removed from the set. The set was then compared to the set of binary strings generated by the findBinaryStrings algorithm. The test was again performed on matrices of size 1 to 15. The results can be found below.

Matrix Size	Equivalent Sets
1	YES
2	YES
3	YES
4	YES
5	YES
6	YES
7	YES
8	YES
9	YES
10	YES
11	YES
12	YES
13	YES
14	YES
15	YES

The set of strings generated by the findBinaryStrings algorithm is the same as the true set of binary strings that represent principal submatrices that are not block diagonal. This shows us that we can expext the algorithm to correctly test for copositivity, since it generates all principal submatrices that need to be tested.

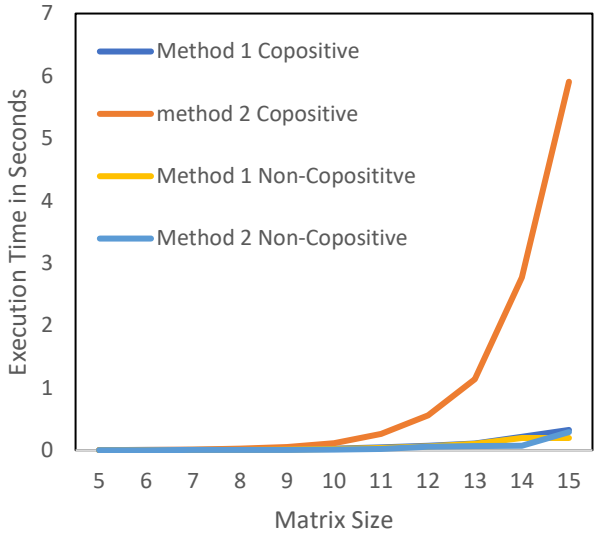
For the third test we tested several matrices for copositivity using the generic method as well as our proposed method. We recorded the outcomes of the tests as well as the time it took both algorithms to run the tests.
The matrices for this test were randomly generated. The matrices were generated in the following way: the matrix is pentadiagonal, the matrix is symetrical, all entries on the main diagonal are between 0 and 1, and the entries on the four main off diagonals are between -0.5 and 0.5. The test for copositivity described by Väliaho [7] was also implemented.
For matrices of size 5 to 15 we each generated 100 random matrices and applied both methods to them.
First, we compare the matrices found to be copositive by the findBinaryStrings algorithm with those found to be opositive by the generic method to see whether the algorithms perform the same. The results are shown in the table below.

Matrix Size	Equivalent Result
5	YES
6	YES
7	YES
8	YES
9	YES
10	YES
11	YES
12	YES
13	YES
14	YES
15	YES

Both algorithm gave the same results for the copositivity of the matrices. This shows that the proposed method is equivalent in

functionality to the generic method, at least for the set of matrices this test was performed on.

Finally, the avarage execution time of both methods is compared in regards to the matrix size. The results are seperated in the matices that were found copositive and those that were found non copositive because those execution times greatly differed. The results are show in the following graph, with Method 1 being our proposed method and method 2 being the generic method.



While the execution times of testing a non copositive matrix are similar, the exection times of testing a copositive matrix are far appart. This is because when testing a non copositive matrix, both algorithms will teminate when only finding a single principal submatrix that doesn't pass the test. However, testing a copositive matrix forces the algorithm to test every principal submatrix. This makes our proposed mehtod more efficient since there are less principal submatrices that need to be tested.

4 CONCLUSION AND DISCUSSION

While testing copositivity in a tridiagonal matrix can be done in linear time, testing copositivity in a pentadiagonal matrix requires $O(\ n^4 * \varphi^n\ )$. However, for the general case the complexity is $O(n^4 * 2^n)$, which makes the method proposed in this paper still an improvement for these matrices.

While this paper assumes that every entry on the five main diagonals of pentadiagonal matrices is nonzero, this is of course not necessarily the case. An optimization could be written for the proposed algorithm which tests for the existence of more block diagonal principal submatrices that exist because some of the five main diagonal entries are zeros.
In the current algorithm, a very crude estimation is used for calculating determinants once the binary strings have been found. This is because we had already found an exponential factor in the computational complexity. However, there might be a more efficient way to do this part of the calculation, which would lower the complexity of the algorithm.
Some of the principal submatrices of pentadiagonal matrices are tridiagonal, if a different way of calculating determinates would be implemented it might be worthwhile to find these tridiagonal cases since calculating the determinant of a tridiagonal matrix can be done more efficiently.
When testing the algorithm, the implementation of the test for copositivity was not checked for its computational complexity. Because of this the test results could have misrepresented the actual time it would take if this method was implemented in a more efficient way. However, since both our proposed method and the generic method were tested on the same implementation, the expectation is that the relative results are accurate.
The tests in the experiments section used randomly generated matrices with specific properties. Because of this, the algorithms could behave differently on matrices generated with other properties.

For future work, it would be interesting to see how this complexity increases as the number of diagonals increase. The expectation is that instead of finding binary strings that avoid two zeros between consecutive ones, for a heptadiagonal matrix, all binary string would need to be found that avoid three zeros between every pair consecutive ones. This would add a third recursive call to the findPartialStrings algorithm with the string "100". Instead of the Fibonacci sequence, this would lead to the Tribonacci sequence i.e., adding the previous three elements of the series to get the next one. We expect a pattern will exist for increasing the number of diagonals in this way.

5 REFERENCES

[1] Bomze, I. M. (2000). Linear-time copositivity detection for tridiagonal matrices and extension to block-tridiagonality. SIAM Journal on Matrix Analysis and Applications, 21(3), 840-848.
[2] Bomze, I. M. (2012). Copositive optimization–recent developments and applications. European Journal of Operational Research, 216(3), 509-520.
[3] Ikramov, K. D., & Savel'Eva, N. V. (2000). Conditionally definite matrices. Journal of Mathematical Sciences, 98(1), 1-50.
[4] Ikramov, K. D. (2002). Linear-time algorithm for verifying the copositivity of an acyclic matrix. Computational mathematics and mathematical physics, 42(12), 1701-1703.
[5] Kaplan, W. (2000). A test for copositive matrices. Linear Algebra and its Applications, 313(1-3), 203-206.
[6] Quist, A. J., de Klerk, E., Roos, C., & Terlaky, T. (1998). Copositive realxation for genera quadratic programming. Optimization methods and software, 9(1-3), 185-208.
[7] Väliaho, H. (1986). Criteria for copositive matrices. Linear Algebra and its applications, 81, 19-34.