LISBON
SCHOOL OF
ECONOMICS &
MANAGEMENT
UNIVERSIDADE DE LISBOA

MASTER

MATHEMATICAL FINANCE

MASTER'S FINAL WORK

INTERNSHIP REPORT

AN ANALYSIS OF BANDWIDTH REDUCTION

LEONARDO LIMA

SUPERVISION:

CLAUDE COCHET
JOÃO JANELA

11 - 2020

GLOSSARY

ACM Adapted Cuthill–McKee.

BW Bandwidth.

PMU Peak memory usage.

Abstract (Master's Thesis)

The matrix bandwidth minimization problem consists in finding a permutation of rows and columns such that non-zero elements are kept in a band as close as possible to the main diagonal. This is a long-established NP-complete problem, that can also be formulated as a vertex labeling problem in a graph. Moreover, reordering instructions in computer programs may reduce peak memory usage by deallocating resources at optimal points of execution. This leads to the peak memory minimization problem, an extension of the bandwidth minimization problem, since it can also be formulated as a vertex labeling problem, where instructions and input/output dependency are translated into vertices and edges, respectively. Fortunately, for these graphs, low bandwidth implies low peak memory usage. In this report, the impact of bandwidth reduction is analyzed when numerically solving the heat equation and reducing peak memory usage of computer programs. The problems are carefully described and a variety of algorithms are implemented in C++, aiming to fully take advantage of bandwidth reduction.

Keywords: Bandwidth minimization; Finite difference scheme; Peak memory usage; Execution time.

TABLE OF CONTENTS

Glossary I

Abstract II

Table of Contents III

List of Figures IV

List of Tables V

Acknowledgements VI

1 Introduction 1

2 The Bandwidth Minimization Problem 3
- 2.1 The Matrix Bandwidth Minimization Problem . . . 3
- 2.2 The Graph Bandwidth Minimization Problem . . . 3
- 2.3 Equivalence and Complexity . . . 4
- 2.4 The Cuthill–McKee algorithm . . . 4

3 The Crank–Nicolson Method 6
- 3.1 The Heat Equation . . . 6
- 3.2 Solution of the Heat Equation in one dimension . . . 7
- 3.3 Solution of the Heat Equation in two dimensions . . . 9
- 3.4 Bandwidth Reduction Impact Analysis . . . 11

4 The Peak Memory Minimization Problem 14
- 4.1 Illustrating the problem . . . 14
- 4.2 Definition . . . 18
- 4.3 Datasets . . . 19
- 4.4 Topological Sorting Approach . . . 20
- 4.5 A Level-based Approach . . . 21
- 4.6 Adapted Cuthill–McKee . . . 22
- 4.7 Analysis . . . 23

5 Conclusion 25

Bibliography 26

List of Figures

1 Crank–Nicolson discretization in 1D 7
2 10
  2a Crank–Nicolson discretization in 2D 10
  2b Linearized order of unknown points on a squared plate 10
3 Linearized order of unknown points on a mirrored C-shaped plate 11
4 12
  4a Bandwidth comparison of Crank–Nicolson scheme 12
  4b Execution time comparison of Crank–Nicolson scheme 12
5 $\mathcal{G}$ under labeling r, a graph representation of algorithm 2 and algorithm 3 15
6 Memory usage when variables are deallocated at the end 16
7 $\mathcal{G}$ under labeling s, a graph representation of algorithm 4 16
8 17
  8a Memory usage when variables are deallocated right after last use . 17
  8b Memory usage when variables are deallocated right after last use and instructions are reordered 17
9 Graph representation of `DateTimeNow_yxmc.txt` 20
10 24
  10a Bandwidth when applying level-based approach 24
  10b Peak memory usage when applying level-based approach 24
11 24
  11a Bandwidth when applying adapted Cuthill–McKee 24
  11b Peak memory usage when applying adapted Cuthill–McKee . . . 24

List of Tables

I Topological sorting results 23

ACKNOWLEDGEMENTS

First of all, I would like to thank Claude Cochet and Professor João Paulo Vicente Janela for their attention, encouragement and patience. I will always be grateful for your guidance.

I would also like to thank my parents and my brothers for their everlasting support. You are my inspiration.

Lastly, I would like to thank Laryssa for brightening my days.

An analysis of bandwidth reduction

By Leonardo Lima

The matrix bandwidth minimization problem consists in finding a permutation such that non-zero elements stay as close as possible to the main diagonal. This can also be formulated as a vertex labeling problem in a graph. Peak memory usage may be reduced by deallocating resources at optimal points of execution. This leads to the peak memory minimization problem, since it can also be formulated as a vertex labeling problem in a graph. For these graphs, low bandwidth implies low peak memory usage. In this report, the impact of bandwidth reduction is analyzed in two applications.

1 Introduction

The matrix bandwidth minimization problem consists in finding a permutation of rows and columns such that non-zero elements of a matrix are kept in a band as close as possible to the main diagonal. Another formulation of the problem is the bandwidth minimization problem on graphs, which consists in numbering vertices such that maximum absolute difference between adjacent vertices is as low as possible. Both formulations are equivalent. The matrix version of the bandwidth minimization problem originated in the 1950s (Wang, Xu & Lisser 2014), whereas the bandwidth minimization problem on graphs dates from 1963, where Harper (Harper 1964) tackled the problem of numbering vertices on the unit n-cube.

A fundamental application of the matrix bandwidth minimization problem is to solve large linear systems. For instance, Gaussian elimination can be performed in time complexity $\mathcal{O}(nb^2)$, considering a matrix with dimension $n \times n$ and bandwidth b (Tarjan 1976). This is a great improvement from the usual $\mathcal{O}(n^3)$, when b is significantly smaller than n. Furthermore, minimizing bandwidth of graphs is useful in a wide range of applications, *e.g.*, minimizing the layout area of a circuit on a chip (Bhatt & Leighton 1984) and navigating large distributed databases with hypertext links (Berry, Hendrickson & Raghavan 1996).

Unfortunately, the bandwidth minimization problem is NP-complete (Papadimitriou 1976). The Cuthill–McKee algorithm (Cuthill & McKee 1969) is an established approach to solve it based on breadth-first search. Even though more sophisticated approaches have been created since then, as in (George 1971) and (Gibbs, Poole & Stockmeyer 1976), the Cuthill–McKee algorithm is simple and straightforward to implement.

This report analyzes the impact of bandwidth reduction when solving the heat equation and reducing peak memory usage of computer programs. Both experiments were

implemented in C++[1], using Eigen[2], a template library for linear algebra. The heat equation (Cebeci & Bradshaw 2012) has been extensively investigated in many areas throughout the years (Thambynayagam 2011). Specifically in finance, its main application is the Black-Scholes formula (Black & Scholes 1973), widely used to price options, a financial derivative. It turns out that the most popular finite difference scheme for approximating the solution of the Black-Scholes equation is the numerically stable Crank–Nicolson method (Duffy 2004). Solving a partial differential equation using the Crank–Nicolson method essentially consists in solving large linear systems. Overall, these linear systems have compact form, but when considering an irregular shaped object, they become sparse. Thus, bandwidth reduction can be used before the solving in order to reduce its execution time. In fact, the experiment shows that execution time largely decreases after applying bandwidth reduction.

Moreover, reordering instructions in computer programs may reduce peak memory usage by deallocating resources at optimal points of execution. This leads to the peak memory usage minimization problem, which is an extension of the bandwidth minimization problem, since it can also be formulated as a vertex labeling problem, where instructions and input/output dependency are translated into vertices and edges, respectively. In addition, the graph is assumed to be weighted, directed and acyclic. Hence, the following constraint must be satisfied: if instruction B depends on the output of instruction A, then label of A must be lower than label of B. Evidently, an algorithm for finding an optimal execution sequence such that peak memory usage is minimized is intractable for graphs of practical size. Fortunately, for these graphs, low bandwidth implies low peak memory usage. Hence, three different approaches for minimizing bandwidth are implemented and analyzed. Even though results are underwhelming, following discussions justify the reasons for it and indicate possible improvements.

This report is organized as follows. Both formulations of the bandwidth minimization problem alongside with the Cuthill–McKee algorithm are presented in section 2. Then, in section 3, the heat equation is derived, discretized and solved numerically using the Crank–Nicolson scheme. Subsequently, an experiment analyzing bandwidth reduction is performed. The peak memory minimization problem is introduced in section 4. Three different approaches tackling the problem are described and analyzed using real datasets. Results are then plotted and discussed. Finally, section 5 concludes the report by discussing achievements and pointing out some further directions.

[1]https://github.com/leonardolima/bandwidth-reduction-analysis

[2]http://eigen.tuxfamily.org/

2 THE BANDWIDTH MINIMIZATION PROBLEM

In this section, the matrix bandwidth minimization problem and the bandwidth minimization problem on graphs are introduced. Initially, formal definitions of both formulations are presented. Then, an implementation of the Cuthill–McKee algorithm, a didactic and straightforward solution to the problem is described and discussed.

2.1 The Matrix Bandwidth Minimization Problem

Let A be a matrix. The bandwidth of A is defined as follows:

Definition 1. *Matrix Bandwidth*

$$B(A) = \max\{|i - j| : a_{ij} \neq 0\}$$

Using words, the bandwidth of A is defined as the maximum distance of a nonzero element from the main diagonal. The goal when tackling the matrix bandwidth minimization problem is to minimize this distance, *i.e.*, all nonzero elements of the matrix must be brought as close as possible to the main diagonal.

2.2 The Graph Bandwidth Minimization Problem

The bandwidth minimization problem has a second formulation, related to graphs. Let G be a graph, specified by a set V of vertices and a set E of edges. The graph is assumed to be finite and undirected. Let $f : V \rightarrow \{1, 2, \ldots, n\}$ be a bijective function, which goes from the set V of vertices to the set of numbers from 1 up to n, where n is the total number of vertices of the graph. This function f is responsible for labeling vertices on the graph. Thus, for each vertex we assign an unique number from 1 up to n. In this case, bandwidth is defined as follows:

Definition 2. *Vertex Bandwidth*

$$B_f(v) = \max_{i:(i,j) \in E} \{|f(i) - f(j)|\}$$

Definition 3. *Graph Bandwidth*

$$B_f(G) = \max_{v \in V} B_f(v)$$

The bandwidth of a particular vertex is defined as the maximum absolute difference between its label and the label of its adjacent vertices. Similarly, the bandwidth of a graph is defined as the maximum bandwidth among all vertices in the graph. Analogous to the

matrix formulation, the goal in this case consists in finding a labeling f such that the bandwidth of the graph is minimized.

2.3 Equivalence and Complexity

Both formulations previously presented are equivalent. In fact, they are interconvertible, which means that it does not matter whether one chooses to work with the matrix or the graph approach. They lead to the exact same results.

Unfortunately, as many interesting computational problems, the bandwidth minimization problem is known to be NP-complete (Papadimitriou 1976). This means that, in practice, finding an optimal solution is not feasible. Hence, approximate solutions are used instead. For instance, considering a graph with n vertices, when trying the brute-force approach, there are $n!$ labeling possibilities. Throughout this report $\mathcal{O}(f(n))$ denotes an asymptotic upper bound on $f(n)$. This is used to describe the worst-case running time of an algorithm, regardless of the input (Cormen, Leiserson, Rivest & Stein 2001). The brute-force approach for the bandwidth minimization problem yields a running time of $\mathcal{O}(n!)$, which turns out to be impractical even if n is small. Thus, a crucial aspect of any algorithm tackling the problem is not only to produce good results, but also to do it in a reasonable amount of time.

2.4 The Cuthill–McKee algorithm

A simple yet robust approach to solving the bandwidth minimization problem is the Cuthill–McKee algorithm (Cuthill & McKee 1969). It traverses the graph in a breadth-first search manner, numbering vertices such that vertices with less adjacent vertices are kept closer to each other. An implementation of the algorithm is described in algorithm 1.

Let A be a sparse and symmetric matrix. The former means that most of its elements are equal to zero, while the latter means that $A = A^T$, *i.e.*, the reflected elements through the main diagonal are equal. When permuting rows and columns, instead of modifying A, a permutation matrix P is used. Of course, P has the exact same dimensions of A, and each one of its rows and columns has exactly one non-zero unit element. Considering this non-zero element to be p_{ij}, i corresponds to the new label and j to the original one. Subsequently, in order to apply permutations to A, one just need to compute PAP^T. Let $G(A)$ denote the graph generated by A. Besides label changes, the resulting graph, $G(PAP^T)$, is structurally indistinguishable from $G(A)$. Clearly, a lower bound for the bandwidth of matrix PAP^T for any of the possible permutation matrices P is given by $floor(\frac{D}{2})$, where D corresponds to the bandwidth of $G(A)$ (Cuthill & McKee 1969).

Algorithm 1 The Cuthill–McKee algorithm

```
 1: function CUTHILL_MCKEE(G(A))
 2:     P ← zero matrix
 3:     new_label ← 0
 4:     for v in G(A) do
 5:         if DEGREE(v) = 0 then                              ▷ Disconnected vertices
 6:             P(new_label, v) ← 1
 7:             new_label ← new_label + 1
 8:             seen ← seen + 1
 9:     starting_vertex ← SELECT_STARTING_V(G(A))
10:     P(0, starting_vertex) ← new_label
11:     new_label ← new_label + 1
12:     seen ← 1
13:     while seen < number of vertices of G(A) do
14:         for v in G(A) do
15:             if DEGREE(v) > 0 then
16:                 adj_vertices ← GET_ADJ_U_VERTICES_SORTED(P, v)    ▷ Ascending order
17:                 for v' in adj_vertices do
18:                     P(new_label, v') ← 1
19:                     new_label ← new_label + 1
20:                     seen ← seen + 1
```

Before starting to unfold the algorithm, the degree of a vertex v, *i.e.*, DEGREE(v), is defined as the number of edges meeting at vertex v. Similarly, when considering a matrix, it is defined as the number of non-zero elements in the row or column corresponding to vertex v, outside of the main diagonal.

Coming back to the algorithm, let G(A) be the graph specified by adjacency matrix A. First, P and new_label are initialized with zeros. From lines 4 to 8, all unconnected vertices are labelled and variables are updated accordingly. Next, SELECT_STARTING_V at line 9 selects (and later at line 10 assigns a label to) a single vertex among all connected vertices with degree equal to the lowest degree found in the graph. Choosing a good starting vertex is crucial to improve the algorithm's performance and in most cases this is a good heuristic to follow. Lastly, for each connected vertex v, function GET_ADJ_U_VERTICES_SORTED lists all its unlabelled adjacent vertices sorted in ascending order at line 16, which is followed by assigning a label to each one of them at line 18, taking into account their order in the list.

3 THE CRANK–NICOLSON METHOD

In this section, the heat equation, a partial differential equation extensively investigated in many areas both in pure and applied mathematics, is derived in one and two dimensions (Cebeci & Bradshaw 2012). Furthermore, the heat equation can be solved numerically using the Crank–Nicolson method (Crank & Nicolson 1947), a stable finite difference method used for efficiently solving numerically partial differential equations. The Crank–Nicolson method essentially consists in solving large linear systems, which, as it has already been mentioned before, can have great performance improvement with bandwidth reduction (Tarjan 1976).

3.1 The Heat Equation

Let $u(x,t)$, $0 \leq x \leq 1$, $t \geq 0$ denote the temperature in a long, thin and insulated bar of uniform material with length 1. Considering a small portion of this bar, exactly between x and $x+h$, it is known that the rate at which heat flows from x to $x+h$, is proportional to $\frac{u(x,t)-u(x+h,t)}{h}$, assuming a small h. Hence, if $u(x,t) > u(x+h,t)$, then heat flows from x to $x+h$. Considering the rate at which heat flows from points $x-h$ and $x+h$ to point x, and dividing it by h, the equation becomes:

$$\frac{\partial u(x,t)}{\partial t} = C\frac{\frac{u(x-h,t)-u(x,t)}{h} - \frac{u(x,t)-u(x+h,t)}{h}}{h}$$

where C is the heat transfer coefficient. Since h is small, let's assume:

$$\frac{u(x-h,t)-u(x,t)}{h} \approx \frac{\partial u(x-\frac{h}{2},t)}{\partial x}$$

and

$$\frac{u(x,t)-u(x+h,t)}{h} \approx \frac{\partial u(x+\frac{h}{2},t)}{\partial x}$$

Hence,

$$\frac{\frac{u(x-h,t)-u(x,t)}{h} - \frac{u(x,t)-u(x+h,t)}{h}}{h} \approx \frac{\partial^2 u(x,t)}{\partial x^2}$$

Taking the limit when $h \to 0$:

$$\lim_{h\to 0} \frac{\frac{u(x-h,t)-u(x,t)}{h} - \frac{u(x,t)-u(x+h,t)}{h}}{h} = \frac{\partial u(x,t)}{\partial t}$$

Results in the one dimensional heat equation:

$$\frac{\partial u(x,t)}{\partial t} = C\frac{\partial^2 u(x,t)}{\partial x^2} \tag{1}$$

Similarly to any partial differential equation, its solution requires an initial condition, that is, when $t = 0$, $u(x,0), \forall x, 0 \leq x \leq 1$ and boundary conditions, that is, $u(0,t)$ and $u(1,t), \forall t \geq 0$. Considering the two dimensional case instead, the heat equation becomes:

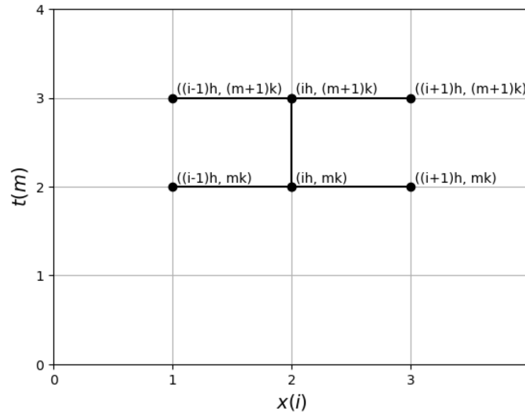
$$\frac{\partial u(x,y,t)}{\partial t} = C\left(\frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2}\right) \tag{2}$$

Initial and boundary conditions in this case correspond to the temperature throughout the object at $t = 0$ and among the boundaries of the object, respectively.

3.2 Solution of the Heat Equation in one dimension

To simplify calculations, the heat transfer coefficient is assumed to be $C = 1$ from now on. The first step to numerically solve the heat equation is to discretize it using finite differences. Let $(x(i), t(m))$ be the points in the grid represented by fig. 1, where $x(i) = ih$, $0 \leq i \leq n = \frac{1}{h}$ and h corresponds to the grid spacing in the x direction. Analogously, $t(m) = mk$ where k corresponds to the time step.

FIGURE 1: Crank–Nicolson discretization in 1D



Let $U_i(m) := u(x(i), t(m))$, where u denotes the continuous function that is approximated by U, the discretized function computed on the grid. The initial condition, $u(x,0)$ gives the value of $U_i(0)$, and the boundary conditions, $u(0,t)$ and $u(1,t)$, give the values at all grid points along the left and right boundaries, respectively, $U_0(m)$ and $U_n(m)$.

Next, the heat equation itself needs to be discretized. In particular, an implicit method has been chosen because it requires solving a linear system at each step. Nevertheless, it is

also relevant to mention that an explicit solution makes it necessary to consider extremely tiny steps, specifically $k \leq \frac{1}{2}h^2$, which makes it expensive computationally (Duffy 2013). Both methods differ in the way they compute the discretized function U. The explicit method computes the value of $U_i(m+1)$ in terms of values at time step m, whereas on the implicit case, values at time steps m and $m+1$ are both necessary. A reasonable implicit method is the Crank–Nicolson (Crank & Nicolson 1947), which is a combination of the well-known forward and backward Euler methods. The discretization of eq. (1) develops into:

$$\frac{U_i(m+1) - U_i(m)}{k} = \frac{1}{2} \cdot \left[\frac{U_{i-1}(m) - 2U_i(m) + U_{i+1}(m)}{h^2} + \frac{U_{i-1}(m+1) - 2U_i(m+1) + U_{i+1}(m+1)}{h^2} \right]$$

Let $z = \frac{k}{h^2}$. Rearranging terms properly, *i.e.*, isolating $m+1$ and m variables on the left and right side of the equation, respectively, the result is:

$$-\frac{z}{2}U_{i-1}(m+1)+(1+z)U_i(m+1)-\frac{z}{2}U_{i+1}(m+1) = \frac{z}{2}U_{i-1}(m)+(1-z)U_i(m)+\frac{z}{2}U_{i+1}(m)$$

Then, the following system of linear equations is obtained:

$$\begin{pmatrix} 1+z & -\frac{z}{2} & & & & & \\ -\frac{z}{2} & 1+z & -\frac{z}{2} & & & & \\ & \ddots & & & & & \\ & & -\frac{z}{2} & 1+z & -\frac{z}{2} & & \\ & & & & \ddots & & \\ & & & & -\frac{z}{2} & 1+z & -\frac{z}{2} \\ & & & & & -\frac{z}{2} & 1+z \end{pmatrix} \cdot \begin{pmatrix} U_1(m+1) \\ U_2(m+1) \\ \dots \\ U_i(m+1) \\ \dots \\ U_{n-2}(m+1) \\ U_{n-1}(m+1) \end{pmatrix}$$

$$= \begin{pmatrix} (1-z) \cdot U_1(m) + \frac{z}{2} \cdot U_2(m) \\ \frac{z}{2} \cdot U_1(m) + (1-z) \cdot U_2(m) + \frac{z}{2} \cdot U_3(m) \\ \dots \\ \frac{z}{2} \cdot U_{i-1}(m) + (1-z) \cdot U_i(m) + \frac{z}{2} \cdot U_{i+1}(m) \\ \dots \\ \frac{z}{2} \cdot U_{n-3}(m) + (1-z) \cdot U_{n-2}(m) + \frac{z}{2} \cdot U_{n-1}(m) \\ \frac{z}{2} \cdot U_{n-2}(m) + (1-z) \cdot U_{n-1}(m) \end{pmatrix}$$

This is a special case of a system of linear equations, named *tridiagonal*, because non-zero elements only appear on the main diagonal plus or minus one column. Due to this fact, considering a matrix of size $n \times n$, running time of solving procedures such as LU decomposition, forward- and back- substitution for this particular case becomes $\mathcal{O}(n)$ (Press, Teukolsky, Vetterling & Flannery 2007), which is a big improvement com-

pared to the usual $\mathcal{O}(n^3)$ when generally solving linear systems.

3.3 Solution of the Heat Equation in two dimensions

The heat flow on a squared plate is considered from this point on. Analogously to the one-dimensional case, the object is discretized in a grid. However, instead of considering only one dimension, in this case there are two. Let $U_i^j(m) := u(x(i), y(j), t(m))$, where $x(i) = ih$, $y(j) = jh$ and $t(m) = mk$. With time, there are three dimensions to take into account, which can be visualized in fig. 2a. Similarly as before, considering eq. (2), the two second-derivatives on the right-hand side of the equation become, respectively:

$$\frac{\partial^2 u(x(i), y(j), t(m))}{\partial x^2} \approx \frac{U_{i-1}^j(m) - 2U_i^j(m) + U_{i+1}^j(m)}{h^2}$$
$$\frac{\partial^2 u(x(i), y(j), t(m))}{\partial y^2} \approx \frac{U_i^{j-1}(m) - 2U_i^j(m) + U_i^{j+1}(m)}{h^2}$$

The result after summing them up is a discretized version of the Laplace operator (or Laplacian) in two dimensions:

$$\frac{U_{i-1}^j(m) + U_{i+1}^j(m) + U_i^{j-1}(m) + U_i^{j+1}(m) - 4U_i^j(m)}{h^2} = \text{Laplacian}(U_i^j(m)) \tag{3}$$

Then, considering the Crank–Nicolson scheme, which is basically the average of the discretized Laplace operator in time steps m and $m+1$:

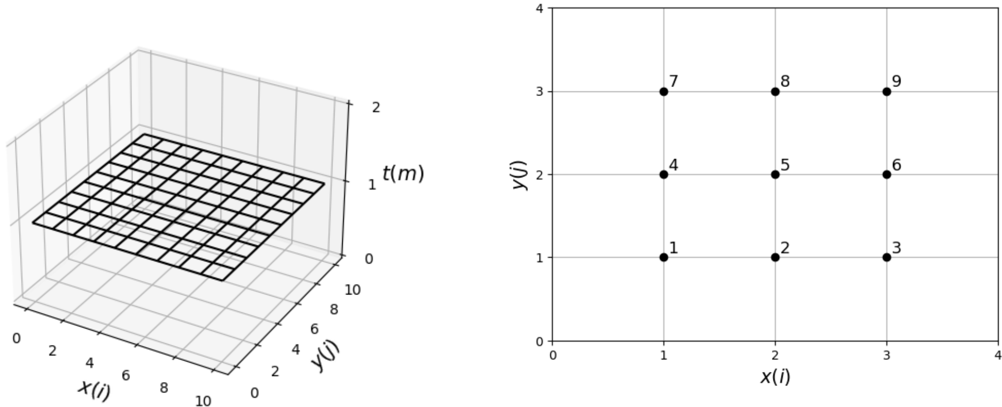
$$\frac{U_i^j(m+1) - U_i^j(m)}{k} = \frac{1}{2}\big[\text{Laplacian}(U_i^j(m)) + \text{Laplacian}(U_i^j)(m+1)\big] \tag{4}$$

Lastly, substituting eq. (3) in eq. (4):

$$\left(1 + \frac{2k}{h^2}\right)U_i^j(m+1) - \frac{k}{2h^2}\big[U_{i-1}^j(m+1) + U_{i+1}^j(m+1) + U_i^{j-1}(m+1) + U_i^{j+1}(m+1)\big]$$
$$= \underbrace{\left(1 - \frac{2k}{h^2}\right)U_i^j(m) - \frac{k}{2h^2}\big[U_{i-1}^j(m) + U_{i+1}^j(m) + U_i^{j-1}(m) + U_i^{j+1}(m)\big]}_{b_i^j(m)}$$

$U_i^j(m+1)$ are the unknown points on the grid, *i.e.*, these are the variables for which the equation must be solved. Considering the specific case of a squared plate discretized by $n = 16$ points, 9 of them are unknown, since they are internal grid points, as shown in fig. 2b. The unknown points are ordered in such a fashion so they can be organized in a vector U, resulting in the linear system $H \cdot U = b$, where H is a matrix and U and b are vectors. Let $a = 1 + \frac{2k}{h^2}$ and $c = -\frac{k}{2h^2}$. A single step of the Crank–Nicolson method for

FIGURE 2



(a) Crank–Nicolson discretization in 2D

(b) Linearized order of unknown points on a squared plate

this specific case can be written as shown in eq. (5).

In particular, H is constituted by $(n-1)^2$ blocks of size $(n-1) \times (n-1)$ and it remains exactly the same at every time step. The main diagonal blocks are tridiagonal matrices with a on the main diagonal and c on the offdiagonals, whereas the offdiagonal blocks all have c on the main diagonal and nothing else, which consists in a band diagonal system. These systems are more general than tridiagonal systems in the sense that $m_1 \geq 0$ and $m_2 \geq 0$ nonzero elements are immediately to the left and to the right, of the main diagonal, respectively. Assuming H has size $n \times n$, m_1 and m_2 are surely both $\ll n$. The running time for solving band diagonal systems still is $\mathcal{O}(n)$ (Press et al. 2007). Most of all, it is important to notice how the order of unknown grid points chosen in fig. 2b affects the form of H. If the order changes, the shape of H also changes, which can result in a bandwidth increase.

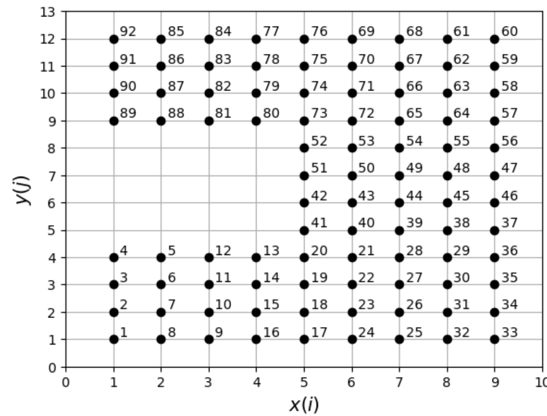
$$
\left(\begin{array}{ccc|ccc|ccc}
a & c & & c & & & & & \\
c & a & c & & c & & & & \\
& c & a & & & c & & & \\
\hline
c & & & a & c & & c & & \\
& c & & c & a & c & & c & \\
& & c & & c & a & & & c \\
\hline
& & & c & & & a & c & \\
& & & & c & & c & a & c \\
& & & & & c & & c & a
\end{array}\right) \cdot
\begin{pmatrix}
U_1^1(m+1) \\ U_2^1(m+1) \\ U_3^1(m+1) \\ U_1^2(m+1) \\ U_2^2(m+1) \\ U_3^2(m+1) \\ U_1^3(m+1) \\ U_2^3(m+1) \\ U_3^3(m+1)
\end{pmatrix}
=
\begin{pmatrix}
b_1^1(m) \\ b_2^1(m) \\ b_3^1(m) \\ b_1^2(m) \\ b_2^2(m) \\ b_3^2(m) \\ b_1^3(m) \\ b_2^3(m) \\ b_3^3(m)
\end{pmatrix}
\tag{5}
$$

3.4 Bandwidth Reduction Impact Analysis

The Crank–Nicolson method for solving the heat equation and the Cuthill–McKee algorithm specified in algorithm 1 were both implemented. With this machinery working, it is possible to apply and analyze the impact of bandwidth reduction in real-world problems. As mentioned earlier, the Crank–Nicolson method is widely used in finance, mainly for approximating option prices with the Black–Scholes formula (Black & Scholes 1973). Furthermore, using the Crank–Nicolson method is convenient because H remains the same through all time step computations, making the impact of reducing its bandwidth immense. Thus, an algorithm for reducing the bandwidth of H can be applied once, in the beginning, and all subsequent computations will benefit from it.

The example illustrated in section 3.3 is not interesting for showing the impact of bandwidth reduction. Despite the fact that H is sparse, it has already a pretty narrow bandwidth. Hence, instead of considering a square plate, consider from now on a mirrored C-shaped plate. The order of unknown grid points is directly related to the shape of H. Thus, the specific order presented in fig. 3 has been chosen.

FIGURE 3: Linearized order of unknown points on a mirrored C-shaped plate



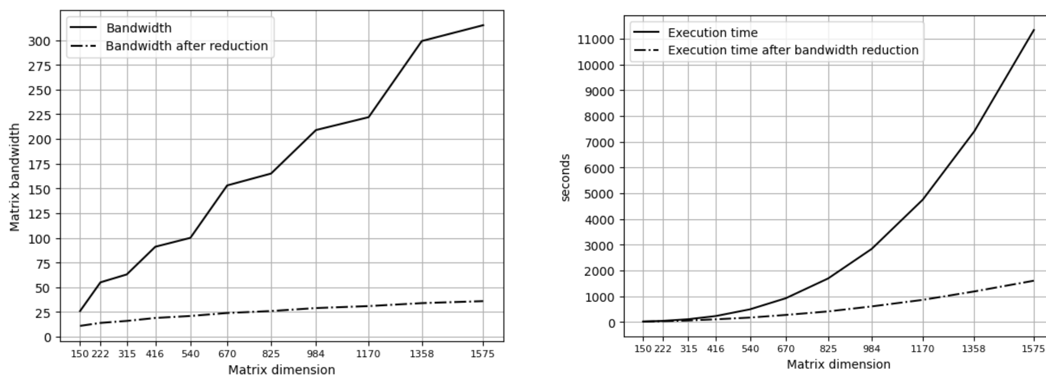
The experiment consists in analyzing the impact of bandwidth reduction on a mirrored C-shaped plate. Specifically, the impact is measured by computing the difference in execution time before and after applying bandwidth reduction to matrix H. First of all, parameters need to be fixed. Let X and Y be the number of grid points in the x and y directions, respectively, with the constraints that $Y = X + 3$ and X and consequently Y are both multiples of 3. Then, $N = \frac{2XY}{3} + \frac{X}{2} \cdot \frac{Y}{3}$ is the total number of grid points. In addition, let $L = 1$ be the length in both directions, which makes $dx = dy = \frac{L}{X}$ be the grid spacing, *i.e.*, the length between two grid points. Lastly, let $dt = 0.0005$ be the time step size and $nsteps = 1000$ be the number of steps to consider. For this particular experiment, C-shaped plates with $X = \{12, 15, \ldots, 42\}$ and $Y = \{15, 18, \ldots, 45\}$ are considered.

Not only these are enough for the current purpose of the experiment, but execution time for greater values also start to become incredibly long from that point onwards.

Before going into the experiment itself, it is important to realize that after reducing the bandwidth of H, an appropriate method for solving band diagonal systems must be considered instead of a general method. Otherwise, the bandwidth reduction impact becomes negligible. A solving method for band diagonal systems from (Press et al. 2007) is used, which consists of first applying LU decomposition to H, and then, once matrix H has been decomposed, given any right-hand side vector, the linear system of equations can be solved with a backsubstitution routine.

Considering all parameters discussed above, for every mirrored C-shaped plate, a Crank–Nicolson scheme consisting of a linear system $H \cdot U = b$ is solved for a vector U. Bandwidth comparison of before and after applying the Cuthill–McKee algorithm to matrix H is shown in fig. 4a. Then, for each mirrored C-shaped plate, execution time (in seconds) for 1000 time steps of the Crank–Nicolson scheme is plotted in fig. 4b. The point is again to compare results before and after applying the Cuthill–McKee algorithm to matrix H. Even though the Cuthill–McKee approach is elementary, the algorithm still performs incredibly well, considering bandwidth reduction for this experiment. Despite the quickly increasing matrix dimension, bandwidth increases surprisingly slowly when the algorithm is applied, reaching a maximum of only 36, instead of an original 315.

FIGURE 4



(a) Bandwidth comparison of Crank–Nicolson scheme

(b) Execution time comparison of Crank–Nicolson scheme

Furthermore, both curves in fig. 4b increasingly differ the greater matrix dimension becomes. The solid line corresponds to execution time before bandwidth reduction, whereas the dashdot line corresponds to execution time after bandwidth reduction. The former displays a tendency for a running time of $\mathcal{O}(n \log n)$, increasing much quicker than the latter, which displays a tendency for a linear running time, or $\mathcal{O}(n)$. For instance, when H has size 1575×1575, if bandwidth reduction is not applied, execution

time is $11338\text{s} \approx 189\text{min}$, but if bandwidth reduction is applied, execution time becomes $1602\text{s} \approx 27\text{min}$.

Summarising, the heat equation has been numerically solved by the Crank–Nicolson method considering a mirrored C-shaped plate. As pointed out before, the linearized order of unknown grid points directly changes the shape of H. Hence, a specific order has been chosen in order to make the bandwidth of matrix H increase. However, applying a bandwidth reduction algorithm to H results in a band diagonal matrix that can be solved in a much faster fashion. Thus, a parallel between bandwidth reduction and faster execution time has been clearly established.

4 THE PEAK MEMORY MINIMIZATION PROBLEM

In this section, the peak memory minimization problem, an extension of the bandwidth minimization problem, is introduced. Memory is a fundamental and scarce resource in a computer (Jones, Hosking & Moss 2016). Hence, memory management is a matter of crucial importance in any relevant computer system, from operational systems to compilers and virtual machines. The discussion in this context is restrained to memory management regarding variables in a program. Memory is allocated to a variable when a block of memory is reserved to store that particular variable. Analogously, a variable is deallocated when that block is no longer used, so it can be freed and used subsequently.

In general, variables can be statically or dynamically allocated. They are allocated during compile time and run time, respectively. Static variables are allocated during compile time because the compiler already knows the amount of memory to allocate before the program has even started. On the other hand, dynamically allocated variables are allocated as required, which is only known at run time, *i.e.*, when the program is being executed (Aho, Lam, Sethi & Ullman 2007). Additionally, a memory leak occurs when a program does not manage memory correctly, keeping parts of memory reserved for variables that are no longer used.

For instance, when a C++ program is executed, dynamically allocated variables are stored in the heap, a specific portion of memory reserved for this use (Stroustrup 2014). Variables allocated in the heap remain allocated until: (i) they are manually deallocated by the user; or (ii) the program terminates. In the following sections the problem of minimizing peak memory usage is tackled by adapting solutions to the bandwidth minimization problem. Evidently, peak memory usage is achieved when memory usage reaches a maximum, and therefore the goal is to minimize this peak as much as possible.

4.1 Illustrating the problem

Instead of starting with the formal definition of the problem, consider the program described in algorithm 2. Implementation details of functions F, G and H are not relevant. However, there is an interesting relationship between variables.

In order to make dependences easier to visualize, the program can also be seen as a graph $\mathcal{G}$, as in fig. 5, where each vertex represents a variable and its subscript corresponds to its label. Also, assume that each vertex is assigned an unique label, which corresponds to its execution order in the program. For example, $\mathtt{A}_1$ denotes that variable $\mathtt{A}$ has label 1. Additionally, edges translate dependences between variables to the graph and their respective weights correspond to the output size each variable produces. For instance, variable $\mathtt{A}$ produces an output of size 2, which is then used as input for instantiating

Algorithm 2 Variables are deallocated at the end of the program

```
1: INIT(A)
2: B ← F(A)
3: C ← G(A)
4: D ← F(B)
5: E ← G(C)
6: F ← H(D, E)
7: FREE(A, B, C, D, E)
8: return F
```

variables B and C.

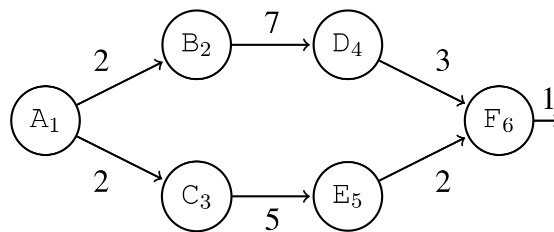


FIGURE 5: $\mathcal{G}$ under labeling r, a graph representation of algorithm 2 and algorithm 3

In this particular example, variables can either be deallocated at the end of the program, as in algorithm 2, or, considering a more efficient approach, as soon as they are not used for further computations, as in algorithm 3. In the former case, from line 7 onwards, the only variable that is going to be used is F, so A, B, C, D and E can be deallocated. Thus, at that point, the total memory usage is also the peak memory usage of the whole program, which is 20, as shown in fig. 6.

Algorithm 3 Variables are deallocated right after last use

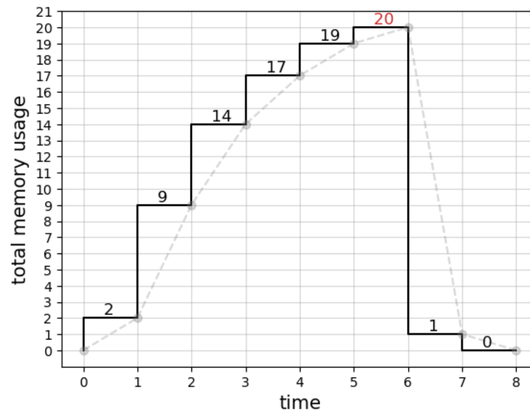
```
1: INIT(A)
2: B ← F(A)
3: C ← G(A)
4: FREE(A)
5: D ← F(B)
6: FREE(B)
7: E ← G(C)
8: FREE(C)
9: F ← H(D, E)
10: FREE(D)
11: FREE(E)
12: return F
```

Algorithm 4 Variables are deallocated right after last use and order is changed

```
1: INIT(A)
2: B ← F(A)
3: D ← F(B)
4: FREE(B)
5: C ← G(A)
6: FREE(A)
7: E ← G(C)
8: FREE(C)
9: F ← H(D, E)
10: FREE(D)
11: FREE(E)
12: return F
```

Alternatively, variables can be deallocated as soon as they are no longer needed, which is the case shown in algorithm 3 and fig. 8a. Considering this approach, after executing line 3, variable A is no longer used, so it can be deallocated. Similarly, after executing lines 5 and 7, variables B and C, respectively, are no longer used. Hence, they are deallocated in lines 6 and 8. To conclude, after executing line 9, variables D and E can also be deallocated. As can be seen, a peak memory usage of 15 is reached, which is a considerable improvement when compared to the first approach. However, one can go a step further and not only deallocate variables when they are no longer necessary, but also reorder instructions in order to minimize peak memory usage. In this way, consider algorithm 4, where the order of instructions $\mathtt{D} \leftarrow \mathtt{F(B)}$ and $\mathtt{C} \leftarrow \mathtt{G(A)}$ is switched.

FIGURE 6: Memory usage when variables are deallocated at the end



Given that dependences between variables have stayed the same, the structure of the graph remains unchanged, as can be seen in fig. 7. However, there is a single label switch between C and D. Evidently, if D is executed before C, D must have lower label than C.

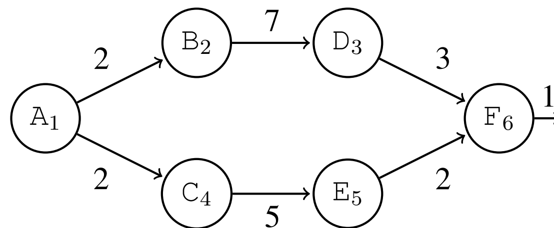
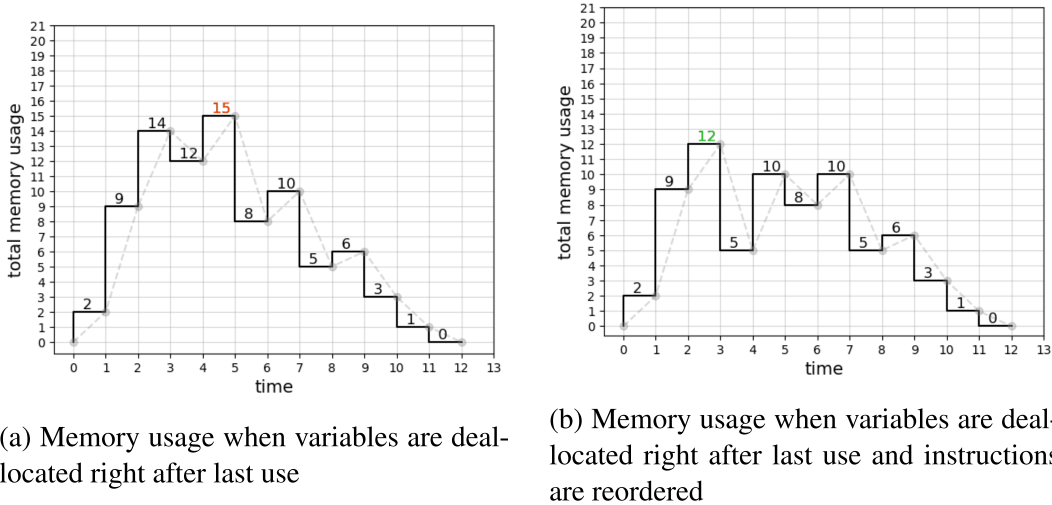


FIGURE 7: $\mathcal{G}$ under labeling s, a graph representation of algorithm 4

Variables are still assumed to be deallocated as soon as they are no longer needed. Despite A only being deallocated at line 6, peak memory usage still decreases to 12, since B could be deallocated first. This is indeed the minimum peak memory usage for this particular program, and the execution of each instruction is depicted in fig. 8b.

FIGURE 8



(a) Memory usage when variables are deallocated right after last use

(b) Memory usage when variables are deallocated right after last use and instructions are reordered

Above all, it is essential to understand that bandwidth minimization and peak memory minimization are directly related to each other. When bandwidth of a graph is low, it means that variables are being deallocated soon or right after they are no longer necessary. However, this does not apply to small programs, including the previous one, since bandwidth of $\mathcal{G}$ under labeling s (algorithm 4) is greater than bandwidth of $\mathcal{G}$ under r (algorithm 2 and algorithm 3), as computed below. Despite that, when considering larger programs, it is clear that a lower bandwidth implies a lower peak memory and vice versa.

$$B_r(\mathcal{G}) = \max\{|1-3|, |1-2|, |3-5|, |2-4|, |4-6|, |5-6|\} = 2$$
$$B_s(\mathcal{G}) = \max\{|1-4|, |1-2|, |4-5|, |2-3|, |3-6|, |5-6|\} = 3$$

An implementation of the procedure described above is detailed in algorithm 5. Initially, two basic defintions about directed graphs should be recalled. A vertex v is the parent of a vertex v' if there exists an edge from v to v'. In addition, v' is said to be a child of v. Starting with function COMPUTE_PEAK_MEM, lists `children`, `parents` and `mem` are used to specify the graph. Evidently, the first two correspond to a list of lists, where each vertex has an entry associated to its list of children and parents, respectively. The list `mem` is used to store the output produced by each vertex, and `path` stores a particular order of execution. In order to keep track of which vertices have already been processed, variables `indeg` and `outdeg` are used, where the former corresponds to the number of unprocessed parents and the latter the number of unprocessed children of a particular vertex. The procedure relies on the queue `to_free` to store vertices that could not be processed at some point, which happens when at least one of its parents has yet to be processed. Thus, at each iteration, when processing the vertices in `path`, in line

Algorithm 5 Computing peak memory usage

```
 1: function FREE_PARENTS(parents, outdeg, mem, cur_mem, v)
 2:     for v' in parents[v] do
 3:         if outdeg[v'] = 0 then
 4:             cur_mem ← cur_mem − mem[v']
 5: function PROC_VERTEX(children, parents, mem, indeg, outdeg, peak, cur_mem, v)
 6:     if indeg[v] = 0 then
 7:         cur_mem ← cur_mem + mem[v]
 8:         if cur_mem > peak then
 9:             peak ← cur_mem
10:         for v' in children[v] do
11:             indeg[v'] ← indeg[v'] − 1
12:         for v' in parents[v] do
13:             outdeg[v'] ← outdeg[v'] − 1
14:         FREE_PARENTS(parents, outdeg, mem, cur_mem, v)
15:         return true
16:     else
17:         FREE_PARENTS(parents, outdeg, mem, cur_mem, v)
18:         return false
19: function COMPUTE_PEAK_MEM(children, parents, mem, path, indeg, outdeg)
20:     cur_mem, peak ← 0
21:     to_free ← empty queue
22:     for v in path do
23:         for v' in to_free do
24:             if PROC_VERTEX(children, . . . , v') then
25:                 remove v' from to_free
26:         if not PROC_VERTEX(children, . . . , v) then
27:             add v to to_free
```

22, first it is necessary to process vertices on the queue. Of course, to_free is a queue because of its first-in-first-out (FIFO) behavior, meaning that oldest vertices are processed first, and if they cannot be processed, they are added to the end of the queue.

Overall, the procedure is very simple. When processing a vertex, PROC_VERTEX first checks if that particular vertex has no parents yet to be processed. Then, if that is the case, cur_mem increases according to the output produced by that particular vertex. Lists indeg and outdeg are also updated accordingly, *i.e.*, when a vertex is processed it has to be removed from the graph, so the indeg of each one of its children and outdeg of each one of its parents have to be decreased by one. When finishing to process a vertex, even if that vertex could not be processed, it is crucial to free its parents if possible. When a vertex is freed, cur_mem decreases accordingly.

4.2 Definition

A more rigorous formulation of the problem is now presented. Let G be a graph, specified by a set V of vertices and a set E of weighted edges. Besides having weights associated to them, in this case edges are also directed. As a consequence of G being

directed, it is also assumed to be acyclic. Again, let a bijective function $f : V \to 1, \dots, n$ be responsible for labeling vertices on G.

As observed in section 4.1, the best solution for this problem is to find an optimal sequence in the graph such that peak memory usage is minimized. Nevertheless, in the next subsection it will be detailed that in practical terms finding this optimal sequence is not feasible, so another approach has to be considered. A reasonable alternative is to work with the bandwidth of the graph instead, since a low bandwidth implies a low peak memory usage, if the graph is large enough. Of course, finding a low bandwidth graph does not necessarily imply that an optimal solution has been achieved, but given the problem's intractability, it is reasonable to consider approximate solutions instead.

Moreover, both definitions for the bandwidth of a vertex and for the bandwidth of a graph are identical to definition 2 and definition 3, respectively. The goal remains to be finding a labeling f such that the bandwidth of the graph is minimized. However, there is an added constraint to the problem. Graph G is now directed, so a particular ordering must be respected when labeling vertices. Specifically, if there exists an edge from v to v', then $f(v) < f(v')$.

4.3 Datasets

A reliable experiment must verify how different approaches perform when a variety of datasets is considered. In this report, datasets[3] were generated and provided by BNP Paribas. They follow a specific format, where the first line corresponds to the number of vertices, N. Then, each line specifies a vertex, that is, $v_0, \dots, v_{N-1}$, where its subscript corresponds to its label. In addition, the first number on the line is the output size produced by that vertex. The subsequent numbers correspond to the list of vertices that are used as input for that vertex, *i.e.*, its list of parents. For instance, consider file `DateTimeNow_yxmc.txt`:

```
7
1
3
66 1 6
77 2
69 3
3
9 5
```

In this case, the number of vertices, or N, is 7. Then, v_0 and v_1 produce output sizes of 1 and 3, respectively, and both do not have parents. Similarly, v_2 produces an output of

[3]https://github.com/leonardolima/bandwidth-reduction-analysis/tree/master/src/datasets

size 66 and has v_1 and v_6 as parents. Additionally, v_3, v_4 and v_5 produce output sizes of 77, 69 and 3, respectively. Also, v_3 and v_4 have v_2 and v_3 as parents, respectively. Finally, the output size of v_6 is 9 and v_5 is its only parent. The graph described is shown in fig. 9. It is important to notice that these graphs can be disconnected, that is, it might not always be possible to find a path between two vertices. For instance, there is no path between v_0 and any vertex in the current example.

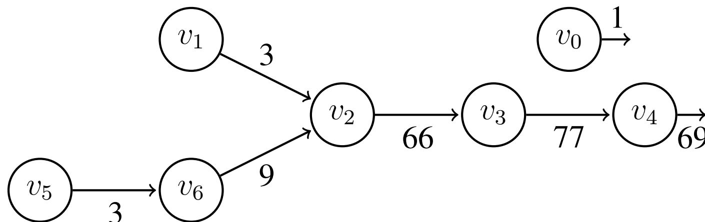


FIGURE 9: Graph representation of `DateTimeNow_yxmc.txt`

4.4 Topological Sorting Approach

As it has already been mentioned, an obvious solution to the peak memory minimization problem is to find all possible sequences in the graph and pick one that minimizes peak memory usage. It surely is tempting to follow this path. Given a directed acyclic graph, a topological sort consists in a linear ordering of all its vertices, such that v appears before v' in the ordering if there exists an edge from v to v' (Cormen et al. 2001). Topological sorting is the problem of finding topological sorts. Clearly, in order to find a minimum, all topological sorts must be found and compared. Implementing an algorithm to find all topological sorts is quite straightforward, as described in algorithm 6.

A list of lists `children` specifies the graph, where each vertex has an entry associated to a list of its children. Again, a list `indeg` is used to keep track of which vertices can be added to `sort`, which is the case when their indegree is equal to 0, or in other words, when all their parents have already been processed. Additionally, a boolean list `marked` initialized with $false$ at every position is maintained to keep track of vertices that have already been processed. When a vertex satisfies the conditions in line 3, the next step is to "remove" it from the graph, which consists in decreasing the indegree of each one of its children. Then, it can be added to the end of `sort` and list `marked` can be modified accordingly. At this point, TOPOLOGICAL_SORTING is called recursively.

Backtracking, a technique widely applied in combinatorial optimization problems is then applied. The modifications made to `indeg`, `sort` and `marked` are all reverted, such that in the next iteration of the *for* loop in line 3 another possible topological sort is considered. Even though this approach is very simple to grasp and implement, it cannot be applied in practice. In fact, the only reason it has been presented is to motivate

Algorithm 6 All topological sorting arrangements

```
 1: function TOPOLOGICAL_SORTING(children, possible_sorts, sort, indeg, marked)
 2:     for v in children do
 3:         if indeg[v] = 0 and not marked[v] then
 4:             for v' in children[v] do
 5:                 indeg[v'] ← indeg[v'] − 1
 6:             add v to the end of sort
 7:             marked[v] = true
 8:             TOPOLOGICAL_SORTING(children, possible_sorts, sort, indeg, marked)
 9:             for v' in children[v] do
10:                 indeg[v'] ← indeg[v'] + 1
11:             remove last element of sort
12:             marked[v] = false
13:     if SIZE(sort) = SIZE(children) then                ▷ Path is completed
14:         add sort to possible_sorts
```

the demand for better solutions. Assuming $|V|$ to be the number of vertices of a specific graph, at each recursive call, all vertices are traversed, resulting in a running time complexity of $\mathcal{O}(|V|^{|V|})$. Of course, a lot of optimizations can be made to improve its overall performance (Varol & Rotem 1981), but this goes beyond the current scope of this work.

The peak memory minimization problem is tackled by other approaches in further sections. An obvious way of analyzing the performance of a particular approach is to compare it with others. Thus, in a later subsection each approach is compared with solutions of the topological sorting algorithm described in (Kahn 1962), which will act as a benchmark. It traverses the graph in a depth-first search manner, trying to find a topological sort. Unlike algorithm 6, it does not try to enumerate all possibilities, finding only a single solution among possibly many.

4.5 A Level-based Approach

Another intuitive solution to the problem is to use "level" as a measure for labeling. The word level is inside quotation marks because there is no formal definition of level for graphs, only for trees. Nonetheless, graphs are restricted to be directed acyclic graphs, hence a notion of level can still be applied in order to develop a solution. This approach is not supposed to be efficient nor particularly useful, but rather important to illustrate how one can solve the problem utilizing information of the graph's structure. It can be seen as a first step towards more robust solutions. Thereafter, an informal notion of level can be defined recursively, as follows:

Definition 4. *Vertex level*

$$level(v) = \begin{cases} 0 & \textit{if } v \textit{ does not have parents} \\ \max\limits_{\{v' \; in \, parents(v)\}} level(v') + 1 & \textit{otherwise} \end{cases}$$

Starting with function APPLY_LEVEL, information about parents and children are stored in two distinct lists of lists. Next, a recursive implementation of the breadth-first search method is used for traversing the graph. Function ASSIGN_LEVEL basically implements definition 4. There are only two possibilities, either the vertex does not have parents and its level is assigned to 0, or it does have parents and its level is assigned to the maximum level among its parents plus one. It is important to notice that multiple vertices might have the same level by the end of the procedure. Given how the definition was designed, this is unavoidable. Hence, it is enough to sort the list `levels` in a way that its original ordering takes precedence if two vertices happen to have the same level.

Algorithm 7 Level-based approach

```
1: function ASSIGN_LEVEL(parents, children, levels, v)
2:     if parents[v] is empty then
3:         levels[v] ← 0
4:     else
5:         for p in parents[v] do
6:             if levels[v] < levels[p] + 1 then
7:                 levels[v] ← levels[p] + 1
8:     for c in children[v] do
9:         ASSIGN_LEVEL(parents, children, levels, c)
10: function APPLY_LEVEL(G)
11:     parents ← PARENTS_IN_GRAPH(G)
12:     children ← CHILDREN_IN_GRAPH(G)
13:     levels ← {·}
14:     for v in parents do
15:         if parents[v] is empty then
16:             ASSIGN_LEVEL(parents, children, levels, v)
```

4.6 Adapted Cuthill–McKee

A slightly more robust approach is an adapted version of the Cuthill–McKee algorithm (ACM), considering that graphs are now directed and acyclic. The basic idea of the Cuthill–McKee approach is to traverse the graph in a breadth-first search manner, assigning labels according to the degree of its vertices, in ascending order. That is, from lowest to highest degree.

In this case, disconnected vertices are processed first. Processing a vertex means "removing" it from the graph and adding it to the end of the linear ordering `path`. After that, `starting_v` is processed, where again a vertex with minimum degree is chosen. Next, each vertex is processed until `path` is complete, that is, all vertices have been processed. When processing a vertex, the indegree of each one of its children is decreased. Then, the vertex is added to the end of `path` and it is marked as visited.

Algorithm 8 Adapted Cuthill–McKee

```
1: function PROCESS_VERTEX(children, indeg, path, marked, seen, n_vertices, v)
2:     for v′ in children[v] do
3:         indeg[v′] ← indeg[v′] − 1
4:     add v to the end of path
5:     marked[v] ← true
6:     seen ← seen + 1
7: function NODAL_NUMBERING(parents, children, indeg, starting_v, n_vertices)
8:     marked ← {false, . . . , false}
9:     path ← {}
10:    seen ← 0
11:    disconnected_vertices ← LIST_DISCONNECTED_V(parents, children)
12:    for v in disconnected_vertices do
13:        PROCESS_VERTEX(children, indeg, path, marked, seen, n_vertices, v)
14:    PROCESS_VERTEX(children, indeg, path, marked, seen, n_vertices, starting_v)
15:    while seen < n_vertices do
16:        for v in children do
17:            if indeg[v] = 0 and not marked[v] then
18:                PROCESS_VERTEX(children, indeg, path, marked, seen, n_vertices, v)
```

4.7 Analysis

To conclude, every approach described above was implemented and tested with all datasets provided. Surely, listing all topological sorts is the best approach in terms of finding the correct solution, since it checks every possible case and finds a minimum. On the other hand, running time complexity makes the algorithm unfeasible for practical purposes. When a graph has number of vertices greater than 9, the number of possibilities explode. Nevertheless, the algorithm was implemented and it can actually be checked that minimum peak memory usage for the example presented in section 4.1, specified in file `example.txt` is indeed 12. Results are presented in the table below, where BW stands for bandwidth and PMU for peak memory usage.

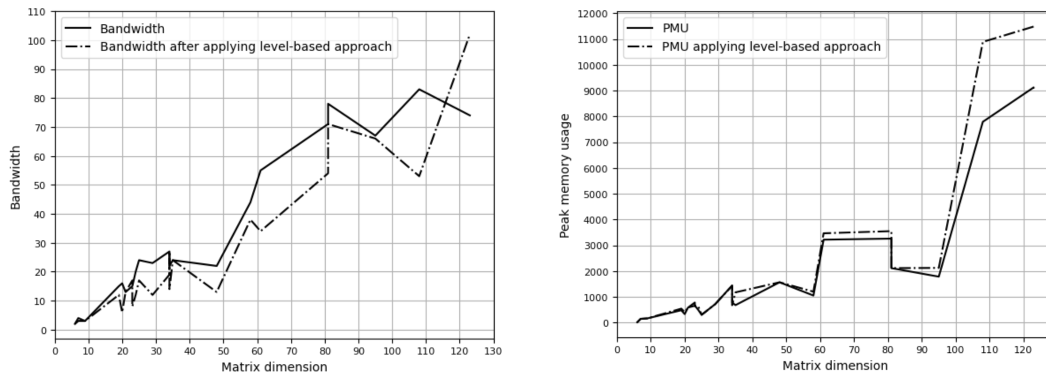
TABLE I: TOPOLOGICAL SORTING RESULTS

File	$\lvert V \rvert$	BW (before)	BW (after)	PMU	PMU (benchmark)
example.txt	6	2	3	12	15
DateTimeNow_yxmc.txt	7	4	3	146	147
CountRecords_yxmc.txt	9	3	4	160	166

Next, the level-based approach seems impractical, which is is not very far from the truth. However, the algorithm performed surprisingly well when it comes to bandwidth

reduction. As expected, results for peak memory usage reduction are disappointing. At most points, values overlap with the benchmark, implying that they are almost or exactly the same. When they differ, the level-based approach performs much worse compared to the benchmark, as can be seen in fig. 10a and fig. 10b.

FIGURE 10

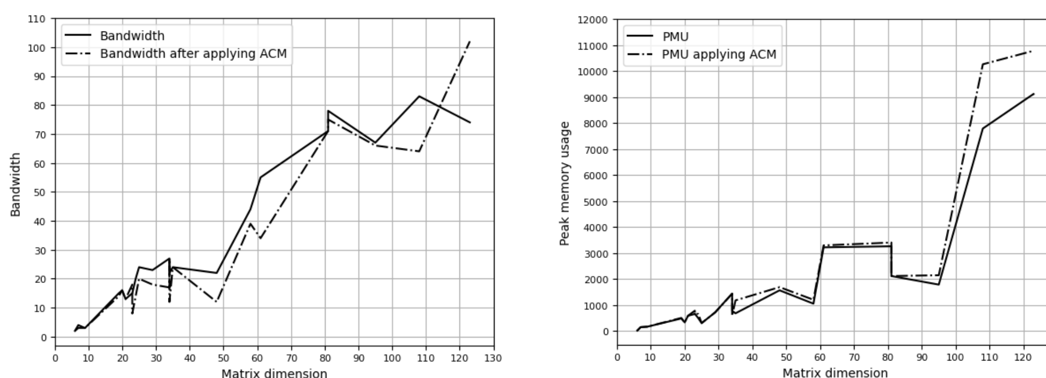


(a) Bandwidth when applying level-based approach

(b) Peak memory usage when applying level-based approach

Finally, when considering the adapted Cuthill–McKee approach results are underwhelming. They are plotted in fig. 11a and fig. 11b. Bandwidth reduction performance is similar to the level-based approach. When it comes to peak memory usage, results are also similar but looking at the exact numbers there is a slight improvement. Even so, it still did not perform better than the benchmark. Presumably, similarities between the level-based approach and the ACM should have been expected. Both approaches tend to process vertices in a breadth-first search fashion, whereas the topological sorting algorithm used as benchmark finds a topological sort traversing the graph using depth-first search. Thus, an algorithm based on depth-first search suit better these specific datasets.

FIGURE 11



(a) Bandwidth when applying adapted Cuthill–McKee

(b) Peak memory usage when applying adapted Cuthill–McKee

5 CONCLUSION

Throughout this report the impact of bandwidth reduction in real-world applications has been analyzed. Initially, the Crank–Nicolson method is used for numerically solving the heat equation on mirrored C-shaped plates. This method consists in solving large linear systems of the form $H \cdot U = b$ at each time step, where H is a sparse matrix that remains the same through all computations. Hence, applying the Cuthill–McKee algorithm to H results in a linear system that can be solved in a much faster fashion. The experiment is run for each mirrored C-shaped plate. For each case, bandwidth reduction is successfully achieved and consequently a much faster execution time.

Next, the peak memory minimization problem is tackled by three different approaches. First, a brute-force method that checks all possibilities using topological sorting produces the best results but only for graphs of size smaller than 10. An alternative implementation of topological sorting that quickly tries to find an arbitrary solution is then used as benchmark. Then, datasets are tested using the level-based approach and the adapted Cuthill–McKee approach. Even though the latter is more refined, results for both of them are underwhelming when tested against the benchmark. Undoubtedly, they are in fact very similar to each other, specifically in the way they process vertices, in a breadth-first search manner. Contrarily to the topological sorting used as benchmark, which process vertices in a depth-first search manner. This seems to suit datasets considerably better. Thus, results can surely improve when other approaches are used.

As of now, bandwidth reduction approaches implemented clearly are not robust enough to provide good solutions in some cases. Expanding the collection of algorithms would definitely be a great improvement. For instance, an ad-hoc approach can be developed using the level-based approach as a starting point. Lastly, metaheuristic-based algorithms such as FNC-HC and VNS-*band* (Chagas & de Oliveira 2015) can also be considered.

REFERENCES

Aho, A., Lam, M., Sethi, R. & Ullman, J. (2007), *Compilers: Principles, Techniques and Tools, 2nd Edition*, Pearson Higher Education.

Berry, M. W., Hendrickson, B. & Raghavan, P. (1996), 'Sparse matrix reordering schemes for browsing hypertext', *Lectures in applied mathematics - American Mathematical Society* **32**, 99–124.

Bhatt, S. N. & Leighton, F. T. (1984), 'A framework for solving vlsi graph layout problems', *Journal of Computer and System Sciences* **28**(2), 300–343.

Black, F. & Scholes, M. (1973), 'The pricing of options and corporate liabilities', *Journal of political economy* **81**(3), 637–654.

Cebeci, T. & Bradshaw, P. (2012), *Physical and computational aspects of convective heat transfer*, Springer Science & Business Media.

Chagas, G. O. & de Oliveira, S. L. G. (2015), Metaheuristic-based heuristics for symmetric-matrix bandwidth reduction: A systematic review., *in* 'ICCS', pp. 211–220.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2001), *Introduction to algorithms*, Vol. 5, MIT press Cambridge.

Crank, J. & Nicolson, P. (1947), 'A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type', *Mathematical Proceedings of the Cambridge Philosophical Society* **43**(1), 50–67.

Cuthill, E. & McKee, J. (1969), Reducing the bandwidth of sparse symmetric matrices, *in* 'Proceedings of the 1969 24th national conference', pp. 157–172.

Duffy, D. J. (2004), 'A critique of the crank nicolson scheme strengths and weaknesses for financial instrument pricing', *The Best of Wilmott* p. 333.

Duffy, D. J. (2013), *Finite Difference methods in financial engineering: a Partial Differential Equation approach*, John Wiley & Sons.

George, A. (1971), Computer implementation of the finite element method, PhD thesis, Ph. D. Dissertation, Computer Science Department, Stanford University.

Gibbs, N. E., Poole, Jr, W. G. & Stockmeyer, P. K. (1976), 'An algorithm for reducing the bandwidth and profile of a sparse matrix', *SIAM Journal on Numerical Analysis* **13**(2), 236–250.

Harper, L. H. (1964), 'Optimal assignments of numbers to vertices', *Journal of the Society for Industrial and Applied Mathematics* **12**(1), 131–135.

Jones, R., Hosking, A. & Moss, E. (2016), *The garbage collection handbook: the art of automatic memory management*, CRC Press.

Kahn, A. B. (1962), 'Topological sorting of large networks', *Communications of the ACM* **5**(11), 558–562.

Papadimitriou, C. H. (1976), 'The np-completeness of the bandwidth minimization problem', *Computing* **16**(3), 263–270.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P. (2007), *Numerical recipes 3rd edition: The art of scientific computing*, Cambridge university press.

Stroustrup, B. (2014), *Programming: principles and practice using C++*, Pearson Education.

Tarjan, R. E. (1976), Graph theory and gaussian elimination, *in* 'Sparse Matrix Computations', Elsevier, pp. 3–22.

Thambynayagam, R. M. (2011), *The diffusion handbook: applied solutions for engineers*, McGraw Hill Professional.

Varol, Y. L. & Rotem, D. (1981), 'An algorithm to generate all topological sorting arrangements', *The Computer Journal* **24**(1), 83–84.

Wang, C., Xu, C. & Lisser, A. (2014), Bandwidth Minimization Problem, *in* 'MOSIM 2014, 10ème Conférence Francophone de Modélisation, Optimisation et Simulation', Nancy, France.